x402-Based AI Agent Marketplace

Technical Architecture Whitepaper

Version: 1.0
Date: November 2025
Document Type: Technical Architecture & Design Rationale

Executive Summary

This whitepaper presents the complete technical architecture for an AI agent marketplace built on the x402 payment protocol. Unlike traditional API marketplaces that require subscriptions or pre-funding, this system enables true **pay-per-use commerce** where autonomous agents can programmatically pay for services on a per-request basis.

The Challenge We're Solving

Imagine an AI agent that needs to use various APIs throughout its operation—sentiment analysis, image recognition, data enrichment, translation services. In today's world, this agent faces several problems:

- **Capital Lock-up:** Must pre-fund accounts with each provider, locking up capital
- **Subscription Waste:** Pays monthly fees even when usage is sporadic
- **Manual Setup:** Requires human intervention to set up payment methods
- **Trust Issues:** No cryptographic proof that services were delivered as paid
- **Vendor Lock-in:** Switching providers means new setup and more locked capital

Our Solution

We introduce a marketplace where:

1. **Agents pay per request** — Only pay for what you use, when you use it
2. **Zero pre-funding** — No need to lock capital with providers
3. **Fully automated** — Agents handle payments programmatically without human intervention
4. **Cryptographically verified** — Every payment and delivery has irrefutable proof
5. **Sub-second experience** — Resource delivery happens in ~500ms despite blockchain settlement

How It Works (High Level)

Think of it as a "payment gateway meets API marketplace":

1. **Agent requests resource** → Marketplace returns payment terms
2. **Agent signs payment promise** → Uses its cryptographic wallet
3. **Marketplace validates promise** → Verifies signature and amount
4. **Resource delivered immediately** → Agent gets what it needs (~500ms)
5. **Settlement happens in background** → Blockchain transaction confirms payment (~30s)
6. **Receipt generated** → Cryptographic proof for all parties

The key innovation is **optimistic delivery with asynchronous settlement**. The agent doesn't wait for blockchain confirmation; it gets the resource immediately while settlement happens in the background. This gives us the best of both worlds: blockchain's trust guarantees with HTTP's speed.

Table of Contents

---

1. Foundational Concepts

Before diving into the architecture, let's establish shared understanding of key concepts.

1.1 What is HTTP Status Code 402?

The HTTP protocol includes a status code "402 Payment Required" that was reserved decades ago but never fully standardized. The idea was simple: when a server needs payment to provide a resource, it returns 402 instead of 200 (success).

Traditional HTTP Flow:

Client: GET /resource
Server: 200 OK + resource data

With Payment Required:

Client: GET /resource
Server: 402 Payment Required + payment instructions
Client: GET /resource + payment proof
Server: 200 OK + resource data

This is elegant because it keeps payment logic at the HTTP protocol level, not buried in application code. Any HTTP client can understand "402 = need to pay" without knowing anything about the specific payment method.

1.2 What is x402?

x402 is a specification that defines **how** the 402 payment negotiation should work. It standardizes:

- **Payment Offers:** What information the server provides about payment requirements
- **Payment Intents:** How clients prove they'll pay
- **Payment Responses:** How servers acknowledge payment status
- **Headers:** Specific HTTP headers for exchanging payment information

Think of it as "OAuth for payments" — a standardized flow that any server and client can implement to handle payment negotiation over HTTP.

1.3 What is a Facilitator?

In payment systems, a facilitator is an intermediary that makes transactions easier between parties. In our architecture, the facilitator:

- **Coordinates** payment flows between consumers and providers
- **Abstracts complexity** so consumers don't manage blockchain directly
- **Provides guarantees** by taking responsibility for settlement
- **Generates proof** through cryptographic attestations

Think of it like a payment processor (Stripe, PayPal), but instead of credit cards, it handles blockchain settlements. The key difference: the facilitator doesn't hold funds long-term, only during the brief settlement period.

1.4 What is Blockchain Settlement?

Settlement is the final, irrevocable transfer of value. When we say "blockchain settlement," we mean:

- **Onchain transaction:** A transaction recorded on a public blockchain (Ethereum, Base, etc.)
- **Cryptographic finality:** Once confirmed with enough blocks, the transaction is practically irreversible
- **Transparent verification:** Anyone can verify the transaction happened
- **Programmable money:** Tokens (like USDC stablecoins) are transferred via smart contracts

Why blockchain instead of traditional payment rails? Because:

- **24/7 operation** — No bank hours or weekends
- **Global reach** — No currency conversions or international fees
- **Programmable** — Agents can control their own wallets
- **Verifiable** — Cryptographic proof of every transaction

1.5 What is Optimistic Delivery?

Optimistic delivery means giving the consumer their resource **before** the blockchain settlement confirms. It's called "optimistic" because we're optimistic the settlement will succeed.

Traditional (Pessimistic):

1. Consumer pays
2. Wait for blockchain confirmation (12-30 seconds)
3. Give resource to consumer

Total time: ~30 seconds

Optimistic:

1. Consumer promises to pay (signed intent)
2. Give resource immediately
3. Settle payment in background

Total time: ~500ms for resource

This is similar to how credit cards work — the merchant gives you goods immediately, and actual settlement with the bank happens days later. The difference is our settlement happens in seconds, not days.

1.6 What is an Attestation?

An attestation is a cryptographic statement that something happened. In our system, the facilitator generates attestations to prove:

- "I verified this payment intent was properly signed"
- "I submitted a blockchain transaction for this payment"
- "The settlement completed successfully with this transaction hash"

Think of it as a digital notarization. The facilitator's private key signs a message saying "I attest that X happened," and anyone with the facilitator's public key can verify this signature is authentic.

1.7 What is a Nonce?

A nonce (number used once) is a unique value that prevents replay attacks. In our system, nonces serve multiple purposes:

Payment Nonce:

- Each payment offer includes a unique nonce
- If a consumer reuses the same signed payment intent, we reject it
- This prevents "replay attacks" where someone tries to reuse a valid payment

Blockchain Nonce:

- Each blockchain transaction has a sequential nonce
- This ensures transactions process in order
- If you try to send two transactions with the same nonce, only one confirms

1.8 What is EIP-712?

EIP-712 is an Ethereum standard for signing structured data. Instead of signing a raw string, you sign data with a defined schema.

Without EIP-712:

Sign: "Pay 0.05 USDC to 0x1234..."

Problem: Ambiguous, could be phishing

With EIP-712:

Sign structured data:
{
domain: "marketplace.io",
type: "PaymentIntent",
invoice_id: "inv_123",
amount: "50000",
token: "0xUSDC...",
...
}
Problem solved: Clear structure, domain-bound, type-safe

This prevents phishing because users can see exactly what they're signing, and signatures are bound to specific domains.

1.9 What is KMS/HSM?

KMS (Key Management Service): A cloud service (like AWS KMS) that securely stores cryptographic keys and performs signing operations without ever exposing the private key.

HSM (Hardware Security Module): A physical device that stores keys and performs crypto operations. More secure than software but more expensive.

Why use KMS/HSM instead of storing keys in code?

- **Security:** Keys never leave the secure enclave
- **Compliance:** Meets regulatory requirements (FIPS 140-2)
- **Auditability:** Every signing operation is logged
- **Rotation:** Keys can be rotated without code changes

1.10 What is State Machine Design?

A state machine is a system that can be in exactly one state at a time, with defined transitions between states.

Example: Invoice States

PENDING → VALIDATED → SETTLING → SETTLED
↓
FAILED

Rules:

- An invoice starts in PENDING
- Can only move from PENDING → VALIDATED (not backwards)
- Once SETTLED, never changes to another state
- Clear rules for each transition

This prevents bugs like:

- "What if we try to settle an already-settled invoice?"
- "Can an invoice go from FAILED back to PENDING?"

By making states and transitions explicit, we eliminate ambiguity.

---

2. Problem Space Deep Dive

Let's explore why existing solutions fail for AI agent commerce.

2.1 The AI Agent Economy

We're at the beginning of an "AI agent economy" where autonomous software agents will:

- **Consume services** (APIs, compute, data) to accomplish tasks
- **Provide services** to other agents and humans
- **Exchange value** for these services

These agents operate without human intervention, making thousands of decisions per day. Traditional payment methods fail because they assume a human is in the loop.

Example Scenario:

An AI research assistant agent needs to:

1. Search academic papers (API call: $0.02)
2. Summarize papers (API call: $0.15)
3. Translate to French (API call: $0.05)
4. Generate citation (API call: $0.01)
5. Create visualization (API call: $0.10)

Total: $0.33 across 5 different providers in 2 minutes

Problems with existing solutions:

Credit Cards:

- Agent can't have credit card (requires human KYC)
- Even if we proxy through human's card, fees are 2.9% + $0.30 = $0.60 (more than the actual cost!)
- Risk of card decline, fraud flags, etc.

Pre-funded Accounts:

- Agent must maintain accounts with 5 different providers
- Must pre-deposit funds (maybe $10 each = $50 locked up)
- Must monitor balances, auto-refill
- What if agent only uses each service once per month?

Crypto Payment Gateways:

- Designed for e-commerce checkout (humans clicking "Buy Now")
- No standard for API-level payments
- Agent would need to interact with different payment flows for each provider
- No atomic guarantee that payment = service delivery

Subscriptions:

- Agent pays $20/month to each provider

- If usage is sporadic, most money is wasted
- Can't dynamically choose cheapest provider
- Vendor lock-in

2.2 Requirements for Agent-Native Payments

Based on the problems above, we need:

R1: Pay-per-use economics

- Pay only for actual usage, not monthly subscriptions
- Support micropayments ($0.01-$10 range)
- Transaction fees must be lower than payment amount

R2: Zero pre-funding

- Agent shouldn't lock capital with each provider
- Just-in-time payments
- Capital efficiency

R3: Fully programmable

- No human intervention required
- No web forms, email confirmations, etc.
- Deterministic, API-first flow

R4: Atomic payment-delivery

- Payment and service delivery must be cryptographically linked
- If agent pays but doesn't get service, there's proof
- If agent gets service but doesn't pay, provider has proof

R5: Fast

- Sub-second latency for payment validation
- Resource delivery shouldn't wait for blockchain finality
- Agent can make many calls per second if needed

R6: Verifiable

- Cryptographic proof of every transaction
- Both parties can verify what happened
- Supports dispute resolution

R7: Interoperable

- Standard protocol works across all providers
- Agent learns once, works everywhere
- Providers implement once, accessible to all agents

2.3 Why Blockchain is Necessary

You might ask: "Why use blockchain? Why not just a traditional database?"

Reasons blockchain is essential:

Trust minimization:

- In a traditional system, you trust the platform operator
- Platform could claim "we paid the provider" when they didn't

- With blockchain, every payment is publicly verifiable

Agent autonomy:

- Agents can control their own wallets (self-custody)
- No need for platform to hold agent's funds
- Agent can verify every transaction independently

Global settlement:

- Works across borders instantly
- No currency conversion
- No banking hours (24/7 operation)

Programmability:

- Agents can programmatically sign transactions
- No need for API keys that could be revoked
- Cryptographic identity

Finality:

- Once confirmed, blockchain transactions are irreversible
- Clear definition of "paid" vs "not paid"
- No chargebacks or payment disputes months later

2.4 Why Pure Blockchain Payments Fail

If blockchain is great, why not have agents pay directly onchain without a marketplace?

Problems with direct blockchain payments:

User experience:

- Agent must pay gas fees (extra cost)
- Agent must manage nonces (complex)
- Agent must wait for confirmation (12-30 seconds)
- Agent must handle different chains (Ethereum, Base, Arbitrum each work differently)

Provider burden:

- Provider must run blockchain infrastructure
- Provider must monitor for incoming payments
- Provider must match payments to API requests
- Provider must handle payment failures

No service linkage:

- Blockchain transaction says "0.05 USDC sent to 0x1234"
- Doesn't say what service this was for
- Provider has to figure out which API call this payment corresponds to
- No atomic guarantee

Example failure:

1. Agent calls API
2. Provider returns data
3. Agent submits blockchain payment
4. Payment fails (insufficient gas)
5. Now what? Agent got service but didn't pay. Provider has no recourse.

Or alternatively:

1. Agent submits blockchain payment
2. Waits 30 seconds for confirmation
3. Calls API
4. API is down
5. Now what? Agent paid but didn't get service. Complex refund process.

2.5 The Marketplace Value Proposition

A marketplace solves these problems by being an intermediary that:

For Consumers (Agents):

- Abstracts blockchain complexity (no gas management, nonce handling)
- Enables optimistic delivery (don't wait for blockchain)
- Single integration point (works with all providers)
- Discovery (find providers, compare prices)
- Dispute resolution (marketplace helps resolve issues)

For Providers:

- Zero blockchain knowledge required (just HTTP API)
- Immediate settlement notification (no monitoring blockchain)
- Risk reduction (marketplace handles payment failures)
- Customer acquisition (marketplace brings agents to you)
- Payment guarantee (marketplace ensures they get paid)

For the Ecosystem:

- Standardization (x402 protocol)
- Network effects (more agents → more providers → more agents)
- Trust (marketplace reputation)
- Innovation (focus on services, not payment infrastructure)

---

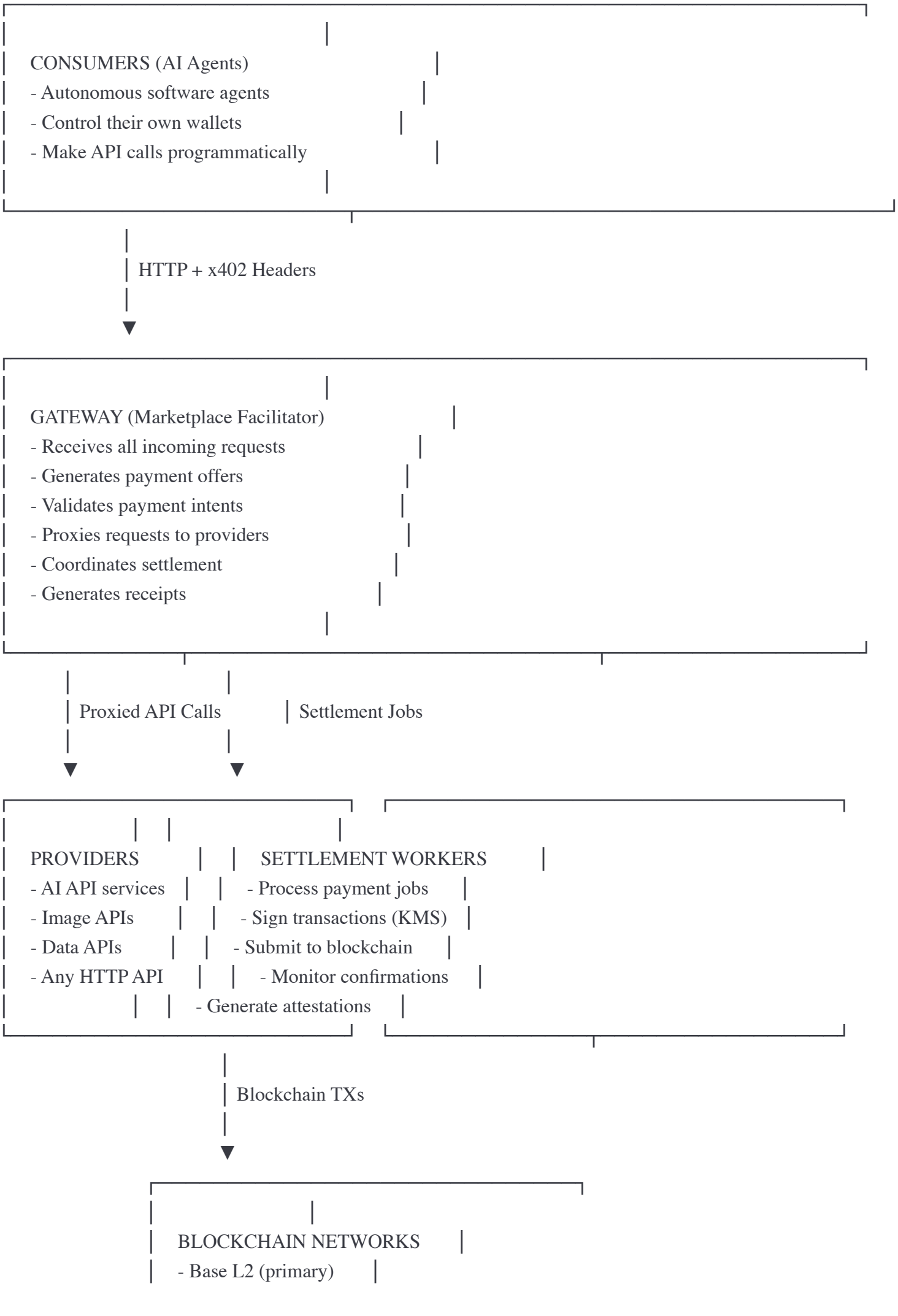
3. Architecture Overview

Now that we understand the problem and requirements, let's look at the high-level architecture.

3.1 System Components

The marketplace consists of four main logical components:

[DIAGRAM_002_COMPONENT_ARCHITECTURE] *Detailed component architecture showing all major system components in a layered view: Consumers, Gateway API, Settlement Workers, Data Layer, and External Integrations.*


CONSUMERS (AI Agents)
- Autonomous software agents
- Control their own wallets
- Make API calls programmatically
HTTP + x402 Headers
GATEWAY (Marketplace Facilitator)
- Receives all incoming requests
- Generates payment offers
- Validates payment intents
- Proxies requests to providers
- Coordinates settlement
- Generates receipts
Proxied API Calls
Settlement Jobs
PROVIDERS
- AI API services
- Image APIs
- Data APIs
- Any HTTP API
SETTLEMENT WORKERS
- Process payment jobs
- Sign transactions (KMS)
- Submit to blockchain
- Monitor confirmations
- Generate attestations
Blockchain TXs
BLOCKCHAIN NETWORKS
- Base L2 (primary)

3.2 Component Responsibilities

Gateway (Facilitator):

- **Entry point** for all consumer requests
- **Payment negotiation** via x402 protocol
- **Signature verification** to ensure payment intent is valid
- **Request routing** to appropriate provider
- **Settlement orchestration** by enqueuing jobs
- **Receipt generation** with cryptographic attestations

Settlement Workers:

- **Job processing** from durable queue
- **Transaction construction** for blockchain
- **Nonce management** to prevent conflicts
- **KMS integration** for secure signing
- **Blockchain submission** via RPC nodes
- **Confirmation monitoring** until finality
- **Attestation signing** for receipts

Database (PostgreSQL):

- **Source of truth** for all system state
- **Invoice management** tracking payment lifecycle
- **Job tracking** for settlement status
- **Receipt storage** for cryptographic proofs
- **Nonce coordination** for blockchain transactions

Job Queue (Redis/RabbitMQ):

- **Durable persistence** surviving restarts
- **At-least-once delivery** guarantees
- **Retry logic** for failed settlements
- **Priority queuing** for urgent settlements

Blockchain Infrastructure:

- **RPC nodes** for submitting transactions
- **Multiple providers** for redundancy (Infura, Alchemy, self-hosted)
- **Network diversity** (L1 and L2 options)

3.3 Data Flow Layers

The system operates across multiple logical layers, each with different latency characteristics:

[DIAGRAM_003_DATA_FLOW_LAYERS] *Four-layer data flow showing progression from Payment Negotiation (<100ms) through Resource Delivery (200-500ms), Settlement Execution (2-30s), to Reconciliation (daily batch), with timing characteristics.*

Layer 1: Payment Negotiation (Synchronous, <100ms)

This is the x402 protocol handshake:

1. Consumer requests resource without payment
2. Gateway generates payment offer
3. Consumer signs payment intent
4. Gateway validates signature and terms

This layer is pure HTTP and involves no blockchain interaction. It must be fast because it's in the critical path of every API call.

Layer 2: Resource Delivery (Synchronous, 200-500ms)

After payment validation:

1. Gateway proxies request to provider
2. Provider processes request (AI inference, data lookup, etc.)
3. Provider returns result
4. Gateway forwards to consumer with payment response header

This layer's latency depends entirely on the provider's API. The marketplace adds minimal overhead (<20ms).

Layer 3: Settlement Execution (Asynchronous, 2-30 seconds)

Happens in the background after consumer receives resource:

1. Worker dequeues settlement job
2. Worker acquires blockchain nonce
3. Worker builds and signs transaction
4. Worker submits to blockchain
5. Worker monitors for confirmations
6. Worker generates final attestation

This layer involves blockchain interaction and takes longer, but consumer doesn't wait for it.

Layer 4: Reconciliation (Batch, daily)

Periodic background process:

1. Compare database state with blockchain state
2. Identify any discrepancies
3. Generate reconciliation report
4. Alert operators if issues found

This layer ensures system integrity over time.

3.4 Trust Model

Understanding who trusts whom and why:

[DIAGRAM_004_TRUST_MODEL] *Trust relationship diagram showing what each party (Consumer, Gateway, Provider) trusts others to do and how trust is enforced through cryptography, blockchain transparency, and economic incentives*.

Consumers trust the Gateway to:

- Generate fair payment terms (correct amount, no price manipulation)
- Validate their payment intent correctly (not claim it's invalid when it's valid)
- Forward their request to the provider (not drop it)
- Actually submit blockchain settlement (not pocket the funds)
- Generate honest attestations (cryptographic proof)

Mitigation: Gateway's reputation is at stake. All settlements are publicly verifiable on blockchain. Consumers can check any transaction hash.

Providers trust the Gateway to:

- Only forward requests with validated payments
- Actually settle payments on blockchain
- Provide them payment even if settlement fails (marketplace guarantees)

Mitigation: Gateway stakes reputation and potentially capital. Providers can verify settlements onchain. SLA agreements can include penalties.

Gateway trusts the Consumer to:

- Not maliciously spam system (mitigated by rate limiting)
- Have sufficient balance for payment (checked at settlement time, not critical)

Gateway trusts the Provider to:

- Actually deliver the service (not critical, consumer can dispute)
- Be available (provider's reputation at stake)

Key insight: Trust is minimized through:

- **Cryptography** (signatures, attestations)
- **Blockchain transparency** (all settlements public)
- **Economic incentives** (reputation, SLAs)
- **Redundancy** (multiple providers for same service)

3.5 Scaling Model

How does the system scale as volume grows?

Horizontal Scaling (Add more instances):

Gateway:

- Stateless design allows infinite horizontal scaling
- Load balancer distributes requests across N gateway instances
- Each instance can handle ~1,000 requests/second
- Bottleneck is database connection pool, not compute

Workers:

- Each network (Base, Arbitrum, Mainnet) has separate worker pool
- Within a network, workers process jobs in parallel
- Nonce coordination happens via database locks (serialized per network)
- Can scale to ~100 workers per network before nonce lock contention

Vertical Scaling (Bigger instances):

Database:

- Primary bottleneck for write-heavy workload
- Can scale up to largest RDS instance (96 vCPU, 768GB RAM)
- Read replicas handle analytical queries
- Partitioning by time (monthly) for invoice/job tables

Blockchain RPC:

- Self-hosted nodes provide unlimited throughput
- Backup commercial providers (Infura, Alchemy) for redundancy
- Private relays (Flashbots) for censorship resistance

Caching:

- Redis caches payment offers, provider metadata, gas prices
- Reduces database load for repeated queries
- Improves latency for hot paths

Expected Scaling Limits:

With moderate hardware:

- **10,000 requests/second** (10 gateway instances)
- **100,000 settlements/day** (5 workers × 20 settlements/min × 1440 min/day)
- **$10M+ GMV/month** at $10 average transaction value

Beyond this, would need:

- Database sharding (by network or provider)
- Multiple settlement queues (priority lanes)
- Geographic distribution (regional deployments)

---

4. The x402 Protocol

Let's dive deep into how x402 payment negotiation works.

4.1 Protocol Philosophy

x402 is designed around these principles:

HTTP-native:

- Uses standard HTTP status codes (402)
- Uses standard HTTP headers (X-PAYMENT-*)
- No custom protocols or WebSockets
- Works with any HTTP client

Stateless:

- Each request is self-contained
- No session management required
- Can be load balanced easily
- Scales horizontally

Cryptographically secure:

- All payment commitments are signed
- Signatures use standard algorithms (ECDSA)
- Structured data signing (EIP-712) prevents phishing
- Each payment has unique nonce (prevents replay)

Extensible:

- Headers use JSON for complex data structures
- Version field allows protocol evolution
- Optional fields for future features

4.2 Protocol Flow

Here's the complete flow with all details:

[DIAGRAM_005_X402_PROTOCOL_FLOW] *Complete x402 protocol sequence diagram showing HTTP requests/responses between Consumer, Gateway, and Provider, with X-PAYMENT headers and timing.*

Request 1: Consumer Requests Resource (No Payment)

```
HTTP Request:
GET /provider/sentiment-api/analyze?text=Hello+World HTTP/1.1
Host: gateway.marketplace.io
Accept: application/json
```

Consumer makes a normal HTTP GET/POST request. No payment information included because consumer doesn't yet know the price or payment terms.

Response 1: Gateway Returns 402 with Payment Offer

```
HTTP Response:
HTTP/1.1 402 Payment Required
Content-Type: application/json
X-PAYMENT-OFFER: eyJpbnZvaWNlX2lkIjoiaW52X2FiYzEyMyIsImFtb3VudCI6IjUwMDAwIiw...

{
  "error": "Payment required",
  "message": "Please provide payment via X-PAYMENT header"
}
```

The gateway returns HTTP 402 status code with X-PAYMENT-OFFER header containing a base64-encoded JSON object.

Decoded X-PAYMENT-OFFER:

[DIAGRAM_006_PAYMENT_OFFER_STRUCTURE] *Visual representation of PaymentOffer JSON structure with annotations explaining each field's purpose, security role, and cryptographic properties.*

json

```
{
  "invoice_id": "inv_abc123",
  "amount": "50000",
  "token": "0x833589fCD6eDb6E08f4c7C32D4f71b54bdA02913",
  "network": "base",
  "payee": "0x742d35Cc6634C0532925a3b844Bc9e7595f0bEb",
  "operation": "sentiment_analysis",
  "nonce": "1699564832000-a3f2d9c8",
  "expires_at": "2025-11-08T10:15:00Z",
  "facilitator": "0x9876abcd1234ef567890...",
  "signature": "0xabcdef123456..."
}
```


Field Explanations:

- **invoice_id**: Unique identifier for this payment request. Used to prevent replay attacks and track the payment through its lifecycle.
- **amount**: Payment amount in token's smallest unit. For USDC (6 decimals), "50000" = 0.05 USDC. Always a string to prevent JavaScript number precision issues.
- **token**: ERC-20 token contract address. Using contract address (not symbol like "USDC") prevents ambiguity across networks.
- **network**: Blockchain network name ("base", "arbitrum", "mainnet"). Tells consumer which network to expect settlement on.
- **payee**: Provider's wallet address. This is where funds will ultimately be sent.
- **operation**: Human-readable operation name. Helps with analytics and debugging. Not cryptographically critical.
- **nonce**: Unique value combining timestamp and randomness. Format: `{timestamp_ms}-{random_hex}`. Prevents two payment offers from ever having same nonce.
- **expires_at**: ISO 8601 timestamp when offer expires (typically 5 minutes). Protects against price changes and stale quotes.
- **facilitator**: Gateway's wallet address. Proves this offer came from the legitimate marketplace, not a phishing attempt.
- **signature**: ECDSA signature over all above fields, signed by facilitator's private key. Consumer can verify this signature to confirm offer authenticity.

Why Sign the Offer?

The facilitator signs the payment offer to prevent:

- **Price manipulation**: Consumer can't change amount and claim "gateway offered this price"
- **Phishing**: Attacker can't generate fake offers pretending to be marketplace
- **Disputes**: If there's a disagreement about terms, signature proves what was actually offered

Request 2: Consumer Resubmits with Signed Payment

Now consumer's agent:

1. Reviews the payment offer
2. Decides to accept the terms
3. Creates a payment intent matching the offer
4. Signs the intent with its private key
5. Resubmits the same request with X-PAYMENT header

```
HTTP Request:
GET /provider/sentiment-api/analyze?text=Hello+World HTTP/1.1
Host: gateway.marketplace.io
Accept: application/json
X-PAYMENT: eyJpbnZvaWNlX2lkIjoiaW52X2FiYzEyMyIsInBheWVyIjoiMHgxMjM0Li4uIiw...
```

Decoded X-PAYMENT (PaymentIntent):

[DIAGRAM_007_PAYMENT_INTENT_STRUCTURE] *PaymentIntent JSON structure showing how consumer signs commitment to payment, with field matching to PaymentOffer and EIP-712 structured signing.*

json


```
{
  "invoice_id": "inv_abc123",
  "payer": "0x1234567890abcdef...",
  "amount": "50000",
  "token": "0x833589fCD6eDb6E08f4c7C32D4f71b54bdA02913",
  "network": "base",
  "nonce": "1699564832000-a3f2d9c8",
  "timestamp": "2025-11-08T10:10:05Z",
  "signature": "0x987654fedcba..."
}
```


Field Explanations:

- **invoice_id**: Must match the invoice_id from payment offer. This links the intent to the specific offer.
- **payer**: Consumer's wallet address. Recovered from signature verification—consumer proves they control this address.
- **amount, token, network, nonce**: Must exactly match payment offer. Gateway verifies these haven't been tampered with.
- **timestamp**: When consumer signed this intent. Must be recent (within offer validity period).
- **signature**: ECDSA signature over all above fields, signed by consumer's private key (payer address). This is what proves consumer authorizes the payment.

EIP-712 Structured Signing:

The signature is created using EIP-712, which means consumer's wallet shows:

Sign Message

Domain: marketplace.io
Type: PaymentIntent

invoice_id: inv_abc123
payer: 0x1234...
amount: 50000 (0.05 USDC)
token: USDC on Base
network: base

[Sign] [Reject]

This structured display prevents phishing because user sees exactly what they're signing.

Response 2: Gateway Returns Resource with Payment Status

HTTP Response:
HTTP/1.1 200 OK
Content-Type: application/json
X-PAYMENT-RESPONSE: eyJpbnZvaWNlX2lkIjoiaW52X2FiYzEyMyIsInN0YXR1cyI6InNldHRsaW5nIiw...


```
{
  "sentiment": "positive",
  "confidence": 0.89,
  "processing_time_ms": 234
}
```


Consumer gets the actual API response (sentiment analysis result) immediately. The X-PAYMENT-RESPONSE header provides payment status.

Decoded X-PAYMENT-RESPONSE:

json

```
{
  "invoice_id": "inv_abc123",
  "status": "settling",
  "settle_job_id": "job_xyz789",
  "estimated_confirmation": "2025-11-08T10:10:35Z"
}
```


Field Explanations:

- **invoice_id**: Same invoice being tracked
- **status**: Current payment status
  - `"settling"`: Settlement job submitted to blockchain (in progress)
  - `"settled"`: Blockchain transaction confirmed
  - `"failed"`: Settlement failed (consumer will get refund)
- **settle_job_id**: Reference ID for the settlement job. Consumer can poll for updates if desired.
- **estimated_confirmation**: When we expect settlement to complete. Typically ~30 seconds for L2 networks.

4.3 Payment Offer Generation Logic

When gateway receives initial request (without payment), it must generate an offer. Here's the decision process:

Step 1: Identify the Provider

From request path `/provider/sentiment-api/analyze`, extract provider ID: `sentiment-api`

Look up provider in database:

```
Provider {
  id: "sentiment-api",
  name: "Sentiment Pro API",
  wallet_address: "0x742d...",
  default_token: "USDC",
  default_network: "base",
  pricing: {
    sentiment_analysis: "50000", // 0.05 USDC
    bulk_sentiment: "200000"     // 0.20 USDC
  }
}
```

Step 2: Determine Operation

From request path `/analyze`, determine operation: `sentiment_analysis`

Look up price for this operation: `50000` (0.05 USDC)

Step 3: Select Network and Token

Use provider's preferences:

- Token: USDC (address: `0x833...` on Base)
- Network: Base L2

Could allow consumer to specify preferences in future (e.g., "I prefer Arbitrum"), but for MVP, use provider defaults.

Step 4: Generate Invoice ID

Create unique invoice ID by hashing:

```
invoice_id = "inv_" + hash(
  provider_id +
  operation +
  amount +
  timestamp +
  random_bytes
)[:20]

Result: "inv_abc123def456ghi789"
```

Step 5: Generate Nonce

Combine timestamp (ensures chronological ordering) with randomness (ensures uniqueness even if two requests same millisecond):

```
nonce = timestamp_ms + "-" + random_hex(8)
Result: "1699564832000-a3f2d9c8"
```

Step 6: Set Expiration

Payment offers expire after configurable time (typically 5 minutes) to prevent:

- Stale price quotes (if provider changes pricing)
- Replay attacks with old offers
- Consumer hoarding offers to use later

```
expires_at = current_time + 5_minutes
Result: "2025-11-08T10:15:00Z"
```

Step 7: Create Offer Object

json

```
{
  "invoice_id": "inv_abc123def456",
  "amount": "50000",
  "token": "0x833589fCD6eDb6E08f4c7C32D4f71b54bdA02913",
  "network": "base",
  "payee": "0x742d35Cc6634C0532925a3b844Bc9e7595f0bEb",
  "operation": "sentiment_analysis",
  "nonce": "1699564832000-a3f2d9c8",
  "expires_at": "2025-11-08T10:15:00Z",
  "facilitator": "0x9876abcd1234ef567890..."
}
```

Step 8: Sign Offer

Sign the offer using facilitator's private key (stored in KMS):

```
message = JSON.stringify(offer) // Deterministic encoding
hash = keccak256(message)
signature = ECDSA_sign(hash, facilitator_private_key)
```

Append signature to offer:

json

```
{
  ...(all fields above),
  "signature": "0xabcdef123456..."
}
```

Step 9: Store Invoice

Before returning to consumer, store invoice in database with status PENDING:

```
INSERT INTO invoices (
  invoice_id, operation, amount, token, network,
  payee, nonce, expires_at, status, created_at
) VALUES (
  'inv_abc123', 'sentiment_analysis', 50000, '0x833...', 'base',
  '0x742d...', '1699564832000-a3f2d9c8', '2025-11-08T10:15:00Z', 'PENDING', NOW()
);
```

This database record is the source of truth. When consumer returns with payment, we'll look up this invoice to validate.

Step 10: Return Offer

Base64-encode the signed offer and return in header:

```
X-PAYMENT-OFFER: eyJpbnZvaWNlX2lkIjoiaW52X2FiYzEyMyIsImFtb3VudCI6IjUwMDAwIiw...
```

4.4 Payment Intent Validation Logic

When gateway receives request with X-PAYMENT header, it must validate the payment intent. This is critical security—we must ensure the payment is legitimate before delivering the resource.

Validation Steps:

[DIAGRAM_008_VALIDATION_FLOWCHART] *Comprehensive flowchart showing all validation steps Gateway performs when receiving signed PaymentIntent: decode, lookup, status check, expiry check, field verification, signature validation, and atomic update.*

Step 1: Decode Payment Intent

Base64-decode the X-PAYMENT header and parse JSON:

json

```
{
  "invoice_id": "inv_abc123",
  "payer": "0x1234567890abcdef...",
  "amount": "50000",
  "token": "0x833589fCD6eDb6E08f4c7C32D4f71b54bdA02913",
  "network": "base",
  "nonce": "1699564832000-a3f2d9c8",
  "timestamp": "2025-11-08T10:10:05Z",
  "signature": "0x987654fedcba..."
}
```

Step 2: Look Up Invoice

Query database for invoice:

```
SELECT * FROM invoices WHERE invoice_id = 'inv_abc123';
```

Check 2a: Invoice Exists If not found → Reject with "Invoice not found"

This could happen if:

- Consumer made up a fake invoice_id
- Invoice was created on different gateway instance and DB replica lag
- Invoice was deleted (shouldn't happen, but defensive coding)

Check 2b: Invoice Status If status != PENDING → Reject with "Invoice already processed"

This prevents replay attacks where consumer tries to reuse same payment intent multiple times. Once an invoice moves past PENDING status, it can never be used again.

Check 2c: Invoice Not Expired If current_time > expires_at → Reject with "Invoice expired"

Offers expire after 5 minutes. Consumer needs to request a new offer with current pricing.

Step 3: Verify Intent Matches Offer

Compare payment intent fields with stored invoice:

```
if (intent.amount != invoice.amount) {
  reject("Amount mismatch");
}
if (intent.token != invoice.token) {
  reject("Token mismatch");
}
if (intent.network != invoice.network) {
  reject("Network mismatch");
}
if (intent.nonce != invoice.nonce) {
  reject("Nonce mismatch");
}
```

This prevents consumer from modifying the terms. For example, consumer can't:

- Change amount from 50000 to 5 (pay less)
- Change token from USDC to a worthless token
- Change network to avoid paying

Step 4: Verify Signature

This is the most critical check. We need to verify:

1. The signature is cryptographically valid
2. The signature was created by the payer's private key

```
// Reconstruct the message that was signed
message = construct_EIP712_message(intent)

// Hash the message
hash = keccak256(message)

// Recover the address that created this signature
recovered_address = ecrecover(hash, intent.signature)

// Verify it matches claimed payer
if (recovered_address != intent.payer) {
  reject("Invalid signature");
}
```

How ECDSA Signature Verification Works:

ECDSA (Elliptic Curve Digital Signature Algorithm) has a special property: from a signature and message, you can recover the public key (and thus address) that created the signature.

```
Private Key (secret)
    ↓ creates
Signature (intent.signature)
    ↓ ecrecover(message, signature)
Public Key
    ↓ hash
Address (intent.payer)
```

So we don't need the private key to verify—we can recover the address from the signature and check if it matches the claimed payer.

Step 5: Check Timestamp Freshness

```
if (intent.timestamp < invoice.created_at - tolerance) {
  reject("Intent timestamp too old");
}
if (intent.timestamp > current_time + tolerance) {
  reject("Intent timestamp in future");
}
```

This prevents:

- Replay of old signed intents
- Clock skew attacks

Tolerance is typically ±5 minutes.

Step 6: Atomic State Update

If all checks pass, we need to atomically:

1. Update invoice status to VALIDATED
2. Record the payer address
3. Create a settlement job

This must be atomic (all-or-nothing) to prevent race conditions where two concurrent validation attempts both succeed.

```
BEGIN TRANSACTION;

-- Lock the invoice row
SELECT * FROM invoices
WHERE invoice_id = 'inv_abc123'
FOR UPDATE;

-- Check status again (double-check pattern)
IF status != 'PENDING' THEN
  ROLLBACK;
  reject("Race condition - invoice already processed");
END IF;

-- Update invoice
UPDATE invoices
SET status = 'VALIDATED',
    payer = '0x1234...',
    validated_at = NOW()
WHERE invoice_id = 'inv_abc123';

-- Create settlement job
INSERT INTO tx_jobs (
  job_id, invoice_id, status, attempts, created_at
) VALUES (
  'job_xyz789', 'inv_abc123', 'QUEUED', 0, NOW()
);

-- Link job to invoice
UPDATE invoices
SET settle_job_id = 'job_xyz789'
WHERE invoice_id = 'inv_abc123';

COMMIT;
```

The `SELECT ... FOR UPDATE` is critical—it acquires a row-level lock, preventing other transactions from modifying this invoice until we commit. This is how we prevent race conditions in a distributed system with multiple gateway instances.

Step 7: Enqueue Settlement Job

After database transaction commits, add job to queue:

```
redis.rpush("settlement_queue", "job_xyz789")
```

This is done AFTER the database transaction to ensure we don't enqueue a job for an invoice that failed validation.

Step 8: Return Success

If validation succeeds, gateway now:

1. Proxies the original request to provider
2. Gets provider response
3. Returns response to consumer with X-PAYMENT-RESPONSE header

4.5 Why This Flow is Secure

Let's consider various attack scenarios:

Attack 1: Replay Attack

Attacker intercepts a valid X-PAYMENT header and tries to reuse it.

Defense:

- Each invoice has unique nonce
- Once validated, invoice moves to VALIDATED status
- Second attempt sees status != PENDING and rejects
- Attacker can't generate new invoice with different nonce because they don't have facilitator's private key to sign offers

Attack 2: Man-in-the-Middle

Attacker intercepts payment offer and modifies amount before consumer sees it.

Defense:

- Payment offer is signed by facilitator
- If attacker modifies amount, signature becomes invalid
- Consumer should verify signature before signing intent (good practice)
- Even if consumer doesn't verify, gateway will reject when fields don't match database

Attack 3: Consumer Lies About Amount

Consumer creates payment intent claiming amount is "5" instead of "50000".

Defense:

- Gateway validates intent.amount == invoice.amount
- Invoice amount is what facilitator originally offered (in database)
- Consumer can't change database without access
- Intent rejected as "amount mismatch"

Attack 4: Consumer Never Completes Payment

Consumer gets payment offer but never returns with signed intent.

Defense:

- Invoice expires after 5 minutes
- No resource delivered unless payment validated
- Invoice remains in PENDING status and eventually cleaned up
- No cost to system except tiny database storage

Attack 5: Gateway is Compromised

Attacker gains access to gateway and tries to steal payments.

Defense:

- Gateway doesn't hold private keys (stored in KMS)
- All settlements are publicly visible on blockchain
- Consumers can audit: "Did my payment actually reach provider?"
- Attestations are signed, proving gateway's claim
- Providers verify they received funds

Attack 6: Consumer Modifies Provider Response

Consumer tries to modify the API response after receiving it.

Defense:

- Not possible—HTTP response is sent directly from gateway
- Consumer can't modify what they already received
- If consumer claims response was different, provider logs show truth

4.6 Protocol Extensions

The x402 protocol is designed to be extensible. Future versions could add:

Dynamic Pricing:

json

```
{
  ...(standard fields),
  "pricing_model": "dynamic",
  "price_factors": {
    "base": "50000",
    "load_multiplier": 1.2,
    "priority_surcharge": "10000"
  }
}
```

Batched Payments:

json

```
{
  "batch_id": "batch_abc",
  "invoices": [
    { "invoice_id": "inv_1", "amount": "50000" },
    { "invoice_id": "inv_2", "amount": "30000" }
  ],
  "total_amount": "80000"
}
```

Subscription Hints:


json

```
{
  ...(standard fields),
  "subscription_available": true,
  "subscription_terms": {
    "calls_per_month": 1000,
    "monthly_price": "40000000"  // 40 USDC
  }
}
```

Service Level Guarantees:


json

```
{
  ...(standard fields),
  "sla": {
    "max_latency_ms": 500,
    "availability": 99.9,
    "penalty_rate": 0.1  // 10% refund if SLA breached
  }
}
```

---

5. Payment Flow Architecture

Now let's dive deep into what happens at each stage of the payment flow.

5.1 The Complete Flow (Detailed)

Here's the end-to-end flow with timing and all system interactions:

T=0ms: Consumer Agent Makes Request

Agent: GET /provider/api/operation

↓

Gateway receives request

Gateway: No X-PAYMENT header present

↓

Gateway identifies provider from path

Gateway queries database for provider pricing

↓

[DATABASE QUERY: ~5ms]

SELECT * FROM providers WHERE id = 'provider-id'

SELECT price FROM provider_pricing WHERE operation = 'operation'

↓

Gateway generates invoice_id, nonce

Gateway creates PaymentOffer object

↓

[DATABASE WRITE: ~10ms]

INSERT INTO invoices (invoice_id, amount, ...) VALUES (...)

↓

Gateway signs offer with KMS

↓

[KMS API CALL: ~20ms]

KMS signs hash of offer

↓

T=35ms: Gateway returns 402 response

Gateway: 402 Payment Required

X-PAYMENT-OFFER: {encoded offer}

↓

Consumer Agent receives 402

T=35ms: Consumer Agent Processes Offer

Agent decodes X-PAYMENT-OFFER

Agent examines terms (amount, token, network)

Agent decides to accept

↓

[AGENT LOCAL OPERATION: ~50ms]

Agent creates PaymentIntent object

Agent signs with EIP-712 (local wallet)

↓

T=85ms: Agent resubmits request

Agent: GET /provider/api/operation

X-PAYMENT: {encoded signed intent}

↓

Gateway receives request with payment

T=85ms: Gateway Validates Payment

Gateway decodes X-PAYMENT header

Gateway extracts invoice_id

↓

[DATABASE QUERY: ~5ms]

SELECT * FROM invoices WHERE invoice_id = 'inv_123'

↓

Gateway validates:

✓ Invoice exists

✓ Status = PENDING

✓ Not expired

✓ Amount matches

✓ Token matches

✓ Network matches

↓

Gateway verifies ECDSA signature

(cryptographic operation, ~5ms)

↓

[DATABASE TRANSACTION: ~15ms]

```
BEGIN;
  SELECT * FROM invoices WHERE invoice_id = 'inv_123' FOR UPDATE;
  UPDATE invoices SET status = 'VALIDATED', payer = '0x...';
  INSERT INTO tx_jobs (invoice_id, status) VALUES ('inv_123', 'QUEUED');
COMMIT;
```

↓

[QUEUE OPERATION: ~5ms]

```
RPUSH settlement_queue "job_xyz789"
```

↓

T=115ms: Validation complete

T=115ms: Gateway Forwards to Provider

Gateway constructs provider request

Gateway adds marketplace headers:

X-Marketplace-Invoice: inv_123

X-Marketplace-Payer: 0x1234...

↓

[PROVIDER API CALL: 200-400ms]
Provider processes request (AI inference, data lookup, etc.)
↓
Provider returns result
↓
T=515ms: Gateway receives provider response

T=515ms: Gateway Returns to Consumer

Gateway constructs response
Gateway adds X-PAYMENT-RESPONSE header:
status: "settling"
settle_job_id: "job_xyz789"
estimated_confirmation: T+30s
↓
T=520ms: Consumer receives response
↓
Consumer Agent now has:
✓ The actual API result (sentiment, image, data, etc.)
✓ Payment status ("settling")
✓ Job ID to track settlement

==========

CONSUMER IS DONE - Got resource in ~500ms
Everything below happens in background

==========

T=520ms: Settlement Worker Picks Up Job

Worker polls settlement queue
↓
[QUEUE OPERATION: ~5ms]
LPOP settlement_queue → "job_xyz789"
↓
Worker queries job details
↓
[DATABASE QUERY: ~5ms]
SELECT j.*, i.*
FROM tx_jobs j
JOIN invoices i ON j.invoice_id = i.invoice_id
WHERE j.job_id = 'job_xyz789'
↓

Worker has:
  invoice_id, amount, token, network, payee, payer
      ↓
T=530ms: Worker starts processing

T=530ms: Worker Builds Transaction

[DATABASE TRANSACTION: ~20ms]
BEGIN;
  -- Lock nonce for this network
  SELECT * FROM relayer_nonce_state
  WHERE network = 'base' FOR UPDATE;

  -- Get current nonce
  current_nonce = last_nonce;

  -- Increment for this transaction
  UPDATE relayer_nonce_state
  SET last_nonce = last_nonce + 1
  WHERE network = 'base';
COMMIT;
      ↓
Worker now has nonce = N
      ↓
Worker constructs blockchain transaction:
  from: RELAYER_WALLET (0xAAAA...)
  to: PAYEE_WALLET (0xBBBB...)
  value: 0 (no ETH transfer)
  data: transfer(to, amount) [ERC-20 transfer]
  nonce: N
  gasPrice: [query current gas price]
  gasLimit: 100000
      ↓
[RPC QUERY: ~100ms]
eth_gasPrice → current_gas_price
eth_getTransactionCount(RELAYER_WALLET) → verify nonce
      ↓
T=650ms: Transaction ready to sign

T=650ms: Worker Signs Transaction

Worker sends transaction to KMS

↓

[KMS API CALL: ~50ms]

KMS signs transaction with relayer key

Returns signed transaction bytes

↓

T=700ms: Signed transaction ready

T=700ms: Worker Submits to Blockchain

Worker calls RPC provider

↓

[RPC CALL: ~100ms]

eth_sendRawTransaction(signed_tx)

↓

RPC returns transaction hash

tx_hash = "0xabcd1234..."

↓

[DATABASE UPDATE: ~10ms]

```
UPDATE tx_jobs
SET status = 'SUBMITTED',
    tx_hash = '0xabcd1234...',
    submitted_at = NOW()
WHERE job_id = 'job_xyz789';
```

↓

```
UPDATE invoices
SET status = 'SETTLING',
    tx_hash = '0xabcd1234...'
WHERE invoice_id = 'inv_123';
```

↓

T=810ms: Transaction submitted to blockchain

T=810ms - T=30s: Waiting for Confirmation

Worker polls for transaction receipt

↓

[LOOP EVERY 2 SECONDS]

```
  eth_getTransactionReceipt(tx_hash)
  if receipt exists:
    if receipt.status == success:
      check confirmations
      if confirmations >= 3:
        DONE!
```

```
    else:
      FAILED (transaction reverted)
  else:
    continue waiting
        ↓
On Base L2: typically 2-5 seconds for inclusion
            + wait for 3 confirmations
            = ~10-15 seconds total
        ↓
T=25s: Transaction confirmed!

═══════════════════════════════════════════════════════════

T=25s: Worker Finalizes Settlement

═══════════════════════════════════════════════════════════

Worker has confirmation receipt
Worker extracts:
  block_number
  transaction_hash
  gas_used
  confirmation_count
        ↓
Worker generates attestation:
  {
    invoice_id,
    payer,
    payee,
    amount,
    token,
    network,
    operation,
    tx_hash,
    block_number,
    settled_at,
    facilitator
  }
        ↓
Worker signs attestation with KMS
        ↓
[KMS API CALL: ~30ms]
KMS signs hash of attestation
        ↓
[DATABASE TRANSACTION: ~20ms]
BEGIN;
  UPDATE tx_jobs
```

```
  SET status = 'CONFIRMED', confirmed_at = NOW()
  WHERE job_id = 'job_xyz789';

  UPDATE invoices
  SET status = 'SETTLED', settled_at = NOW(), block_number = 12847392
  WHERE invoice_id = 'inv_123';

  INSERT INTO receipts (invoice_id, attestation, attestation_signature)
  VALUES ('inv_123', {...}, '0xsig...');
COMMIT;
    ↓
T=25.5s: Settlement complete!

═══════════════════════════════════════════════════
FINAL STATE
═══════════════════════════════════════════════════

✓ Consumer received resource (T=520ms)
✓ Provider will receive funds (T=25.5s)
✓ Blockchain transaction confirmed (T=25s)
✓ Cryptographic receipt stored (T=25.5s)
✓ All parties have proof of payment
```

5.2 Optimistic Delivery Deep Dive

The key innovation is **optimistic delivery**—giving the consumer their resource before blockchain confirms. Let's understand the implications.

Why Optimistic?

[DIAGRAM_010_OPTIMISTIC_VS_PESSIMISTIC] *Side-by-side comparison showing timing difference between optimistic delivery (~500ms) vs pessimistic/traditional approach (~25s), highlighting dramatic UX improvement.*

Without optimistic delivery:

- Consumer requests resource
- Gateway validates payment intent
- Gateway submits blockchain transaction
- Consumer waits 25 seconds for confirmation
- Gateway finally returns resource
- **Total time: ~25 seconds**

This is unacceptable for API calls. Imagine an AI agent that needs to make 100 API calls—that's 2500 seconds = 42 minutes of just waiting for blockchain confirmations!

With optimistic delivery:

- Consumer requests resource
- Gateway validates payment intent
- Gateway returns resource immediately
- Gateway settles payment in background
- **Total time: ~500ms**

Now 100 API calls take 50 seconds (100 × 500ms), with settlements happening in parallel in the background.

What Could Go Wrong?

The risk: Consumer gets resource but settlement fails. What happens then?

Scenario 1: Transaction Reverts

Transaction is submitted but blockchain rejects it (reverts). Possible reasons:

- Insufficient gas provided
- Token contract bug
- Relayer wallet out of USDC balance

Handling:

1. Worker detects revert from transaction receipt
2. Updates invoice status to FAILED
3. Alerts operators
4. Marketplace issues refund to consumer
5. Marketplace compensates provider from reserve fund

Cost: Marketplace absorbs this rare failure

Scenario 2: Transaction Never Confirms (Stuck)

Transaction submitted but never mined. Possible reasons:

- Gas price too low (transaction not attractive to miners)
- Network congestion
- Nonce conflict (very rare with our design)

Handling:

1. Worker waits for timeout (5 minutes)
2. Worker attempts to "bump" transaction (resubmit with higher gas)
3. If still stuck, manual operator intervention
4. Eventually either confirms or is canceled
5. If canceled, marketplace refunds consumer

Cost: Marketplace absorbs gas cost of failed attempts

Scenario 3: Provider Never Delivers Resource

Settlement succeeds but provider's API was down or returned error.

Handling:

1. Gateway detects provider error (500, timeout, etc.)
2. Gateway doesn't forward to consumer
3. Settlement is canceled (or refunded)
4. Consumer's payment intent is marked invalid
5. Consumer can retry with new request

Cost: No cost to any party—payment never settled

Risk Quantification

How risky is optimistic delivery?

Probability of settlement failure:

- Transaction revert: ~0.01% (1 in 10,000)
  - Mostly due to infrastructure issues (out of gas, rate limits)
- Transaction stuck: ~0.05% (5 in 10,000)
  - Usually resolved by gas bump
- Provider failure: ~0.1% (10 in 10,000)
  - Depends on provider reliability

Overall risk: ~0.15% of transactions

Expected cost:

With 100,000 transactions/day:

- 150 failures/day
- Average transaction value: $0.10
- Risk exposure: $15/day
- With refund processing: marketplace loses gas fees only (~$0.01/tx)
- **Total cost: $1.50/day = $550/year**

This is negligible compared to the UX benefit of sub-second responses.

Mitigation Strategies:

Reserve Fund:

- Marketplace maintains reserve of $10,000 USDC
- Used to immediately compensate providers if settlement fails
- Refilled from platform fees
- Sufficient to cover months of failures

Provider SLAs:

- Providers with >99% uptime get lower risk weighting
- Providers with poor reliability may require pre-settlement

Consumer Reputation:

- Track settlement success rate per consumer wallet
- Consumers with high failure rate may be required to pre-fund

Dynamic Risk Assessment:

- If network congestion high, increase gas prices preemptively
- If KMS/RPC issues detected, pause optimistic delivery temporarily
- Graceful degradation: can fall back to pessimistic delivery (wait for confirmation)

5.3 Settlement Guarantees

What guarantees does the system provide?

Guarantee 1: Atomicity

For any given invoice, either:

- **Consumer gets resource AND payment settles**, OR
- **Consumer doesn't get resource AND payment doesn't settle**

We never have a state where:

- Consumer got resource but payment didn't settle (marketplace refunds)
- Payment settled but consumer didn't get resource (provider compensates)

Guarantee 2: Idempotency

If consumer sends the same X-PAYMENT header twice (accidental retry, network issue), the second request:

- Is rejected with "Invoice already processed"
- Does NOT charge consumer twice
- Does NOT create duplicate settlement
- Returns error clearly explaining issue

Guarantee 3: Finality

Once invoice reaches SETTLED status:

- Payment is confirmed onchain with N confirmations (N=3 for L2, N=12 for L1)
- Risk of blockchain reorg is negligible (<0.0001%)
- Cryptographic receipt is generated
- Settlement is irreversible

Guarantee 4: Verifiability

Every settlement produces:

- **Transaction hash**: Unique identifier on blockchain
- **Attestation**: Facilitator's cryptographic signature
- **Block number**: Specific block where transaction included
- **Receipt**: Complete record with all details

Any party can verify:

- "Did this payment actually happen?"
- "Was the correct amount transferred?"
- "Did it go to the right address?"

Guarantee 5: Recoverability

If system crashes mid-settlement:

- All state is persisted in database
- Jobs in queue are durable
- On restart, system automatically:
    - Identifies incomplete settlements
    - Retries failed transactions
    - Reconciles with blockchain state

No manual intervention needed for recovery.

---

6. Settlement Architecture

Let's explore how blockchain settlement actually works.

6.1 Settlement Worker Design

Settlement workers are processes that dequeue jobs and execute blockchain transactions. They operate independently from the gateway.

Why Separate Workers?

Separation of Concerns:

- Gateway handles HTTP (payment validation, request proxying)
- Workers handle blockchain (transaction signing, submission, monitoring)
- Clean abstraction—gateway never touches blockchain

Scalability:

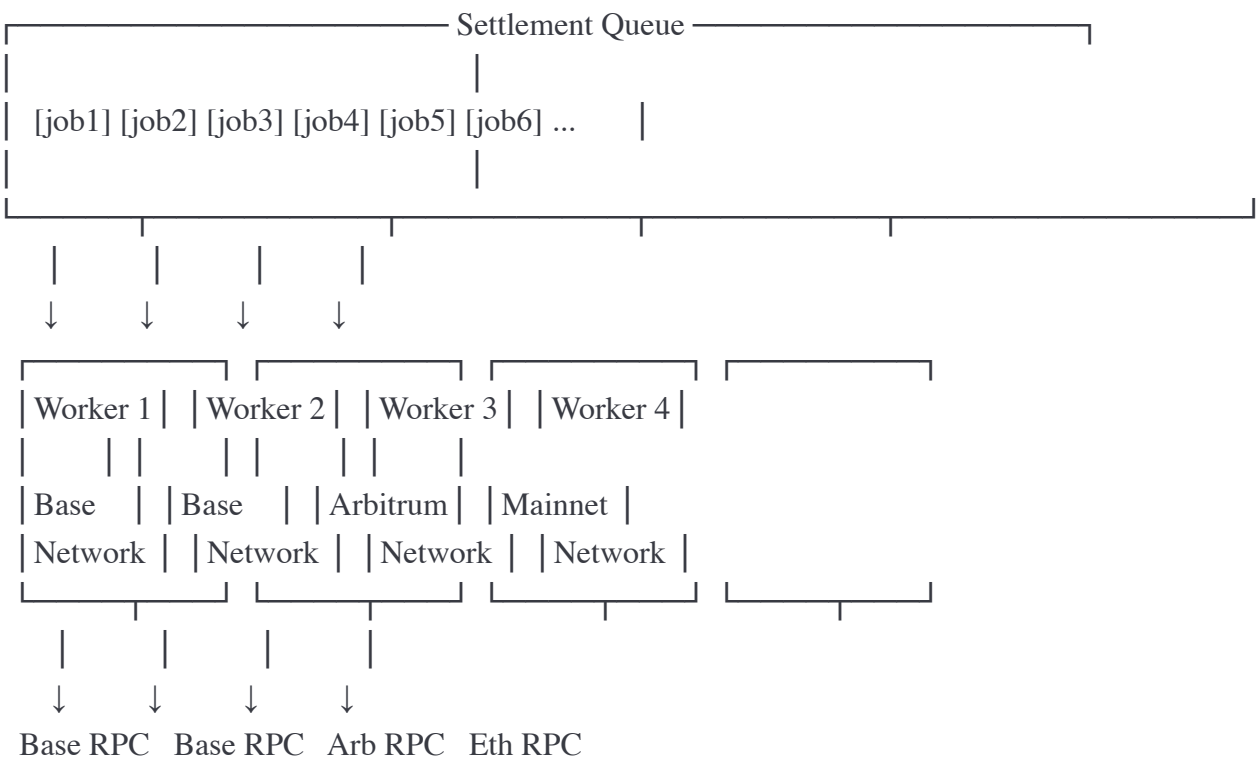
- Can scale gateway and workers independently
- Gateway scales with HTTP request rate
- Workers scale with settlement volume
- Different scaling characteristics

Reliability:

- Gateway must respond quickly (<500ms)
- Workers can take longer (20-30s per settlement)
- Worker crashes don't affect gateway availability
- Can restart workers without disrupting API traffic

Worker Pool Architecture:

[DIAGRAM_011_SETTLEMENT_WORKER_ARCHITECTURE] *Settlement worker pool architecture showing per-network worker instances, job queues, RPC endpoints, and database nonce coordination mechanism.*



Design Decisions:

Per-Network Workers:

- Each blockchain network has dedicated worker pool
- Base workers only handle Base settlements
- Arbitrum workers only handle Arbitrum settlements
- This prevents nonce conflicts across networks

Why? Each blockchain maintains separate nonce sequence for relayer wallet. If Worker 1 processes Base job with nonce 100, and Worker 2 processes Arbitrum job, Arbitrum transaction uses separate nonce sequence (e.g., nonce 50). No coordination needed across networks.

Shared Queue Per Network:

- All Base jobs go to "base_settlement_queue"
- All Arbitrum jobs go to "arbitrum_settlement_queue"
- Workers pull from their respective queues

Why? Simpler than having per-worker queues. Any Base worker can process any Base job. If one worker is slow/crashed, others pick up slack.

Nonce Coordination:

- Within a network, all workers share nonce state in database
- Workers use database locks to serialize nonce allocation
- This ensures no two workers use same nonce

6.2 Blockchain Transaction Construction

When worker processes a job, it must construct a blockchain transaction. Let's understand each field.

Transaction Structure:

[DIAGRAM_012_BLOCKCHAIN_TRANSACTION_STRUCTURE] *Annotated Ethereum transaction structure showing all fields (from, to, value, data, nonce, gasPrice, gasLimit, chainId) with explanations and ABI encoding breakdown for ERC-20 transfer.*

```
{
  from: RELAYER_WALLET_ADDRESS,
  to: TOKEN_CONTRACT_ADDRESS,
  value: 0,
  data: transfer(PAYEE_ADDRESS, AMOUNT),
  nonce: N,
  gasPrice: GP,
  gasLimit: GL,
  chainId: NETWORK_CHAIN_ID
}
```

Field Explanations:

from (Relayer Wallet Address):

- This is the marketplace's wallet that signs transactions
- Example: `0xAAAA1111BBBB2222CCCC3333...`
- This wallet holds USDC and must have ETH for gas

to (Token Contract Address):

- Address of the ERC-20 token contract (USDC, USDT, etc.)
- Example: `0x833589fCD6eDb6E08f4c7C32D4f71b54bdA02913` (USDC on Base)

- NOT the payee address—that goes in the data field

Why? ERC-20 transfers are actually smart contract function calls. The transaction goes to the token contract, and the data field specifies the transfer parameters.

value (ETH Amount):

- Set to 0 because we're not transferring ETH
- We're transferring tokens (USDC), which happens via contract call
- ETH is only used for gas (paid separately, not in 'value' field)

data (Encoded Function Call):

- Encoded call to `transfer(address to, uint256 amount)`
- Example: `0xa9059cbb000000000000000000000000BBBB2222CCCC3333...000000000000000C350`

Encoding Breakdown:

```
0xa9059cbb                                                         // Function selector (keccak256("transfer(address,uint256)")[:4])
000000000000000000000000BBBB2222CCCC3333                           // Payee address (padded to 32 bytes)
000000000000000000000000000000000000000000000000000000000000C350 // Amount (50000 = 0xC350 in hex, padded to 32 bytes)
```

This is ABI encoding—standard way to encode Ethereum function calls.

nonce (Transaction Sequence Number):

- Sequential number for this wallet's transactions
- Each transaction increments nonce
- Transaction with nonce N must be mined before transaction with nonce N+1

Why? Prevents replay attacks and ensures transaction ordering.

Example:

- Transaction 1: nonce=100, transfer 0.05 USDC to Alice
- Transaction 2: nonce=101, transfer 0.10 USDC to Bob
- If transaction 2 arrives first, miners wait for transaction 1
- This ensures correct ordering

gasPrice (Wei per Gas Unit):

- How much we're willing to pay per unit of gas
- Denominated in Wei (1 ETH = 10^18 Wei)
- Example: 50000000 Wei = 0.00000005 ETH = 0.05 Gwei

Why Dynamic? Gas prices fluctuate based on network demand:

- Low congestion: 0.01 Gwei (very cheap on L2s)
- High congestion: 100+ Gwei (expensive on mainnet)

Worker queries current gas price and adds margin:

```
current_gas_price = eth_gasPrice()  // e.g., 50000000 Wei
our_gas_price = current_gas_price * 1.1  // 10% above market for faster inclusion
```

gasLimit (Maximum Gas Units):

- Maximum gas units we're willing to spend
- ERC-20 transfer typically uses ~65,000 gas
- We set limit to 100,000 for safety margin

Why Not Higher? If set too high, we might overpay. If set too low, transaction fails with "out of gas" error.

chainId (Network Identifier):

- Unique ID for each blockchain network
- Base: 8453
- Arbitrum: 42161
- Ethereum Mainnet: 1

Why? Prevents replay attacks across networks. A signed transaction for Base cannot be replayed on Arbitrum because signature includes chainId.

6.3 Nonce Management Deep Dive

Nonce management is critical and tricky. Let's understand why and how.

The Nonce Problem:

[DIAGRAM_013_NONCE_RACE_CONDITION] *Before/after visualization showing race condition when two workers access nonce simultaneously without locking, and how database FOR UPDATE locking prevents collision.*

Imagine two workers processing jobs simultaneously:

```
Worker 1: Processes invoice_A, needs nonce
Worker 2: Processes invoice_B, needs nonce

Database says last_nonce = 100

Worker 1: Reads last_nonce (100), uses nonce 101
Worker 2: Reads last_nonce (100), uses nonce 101  ← PROBLEM!

Both workers submit transactions with nonce 101
Only one gets mined
Other is rejected as "nonce too low"
```

This is a **race condition**—both workers read the same value before either updates it.

Solution: Database Locking

[DIAGRAM_014_NONCE_COORDINATION_FLOW] *Complete nonce acquisition flowchart showing transaction boundaries, SELECT FOR UPDATE lock acquisition, nonce increment, and safe release.*

We use `SELECT ... FOR UPDATE` to acquire an exclusive lock:

```
Worker 1:
  BEGIN TRANSACTION;
  SELECT last_nonce FROM relayer_nonce_state WHERE network='base' FOR UPDATE;
  -- This acquires a lock. Worker 2 now blocks here.
  last_nonce = 100
  new_nonce = 101
  UPDATE relayer_nonce_state SET last_nonce=101 WHERE network='base';
  COMMIT;  -- Lock released

Worker 2:
  BEGIN TRANSACTION;
  SELECT last_nonce FROM relayer_nonce_state WHERE network='base' FOR UPDATE;
  -- Now unblocked, reads updated value
  last_nonce = 101
  new_nonce = 102
  UPDATE relayer_nonce_state SET last_nonce=102 WHERE network='base';
  COMMIT;
```

How FOR UPDATE Works:

When Worker 1 executes `SELECT ... FOR UPDATE`:

- PostgreSQL acquires a row-level lock on that record
- Other transactions trying to `SELECT ... FOR UPDATE` the same row block (wait)
- Lock is held until transaction commits or rolls back
- This serializes access—only one worker at a time

Nonce Reconciliation:

What if database says nonce is 100, but blockchain says nonce is 105?

This could happen if:

- System crashed and some transactions completed without updating database
- Manual transaction was sent outside the system
- Database was restored from backup

Reconciliation Process:

```
Function ReconcileNonce(network):
  db_nonce = SELECT last_nonce FROM relayer_nonce_state WHERE network=network
  chain_nonce = eth_getTransactionCount(RELAYER_ADDRESS, "pending")

  if chain_nonce > db_nonce:
    // Chain is ahead—some transactions we don't know about
    LOG_WARNING("Nonce drift detected: DB has {db_nonce}, chain has {chain_nonce}")

    // Query recent transactions to understand what happened
    recent_txs = eth_getLogs(RELAYER_ADDRESS, last_24_hours)

    // Update database to match chain
    UPDATE relayer_nonce_state SET last_nonce=chain_nonce WHERE network=network

    // Alert operators to investigate
    SEND_ALERT("Nonce reconciliation performed")

  else if db_nonce > chain_nonce:
    // DB is ahead—we have pending transactions not yet mined
    pending_tx_count = db_nonce - chain_nonce
    LOG_INFO("{pending_tx_count} transactions pending confirmation")

    // This is normal—no action needed

  else:
    // Perfect sync
    LOG_INFO("Nonce in sync")
```

Run reconciliation:

- On worker startup (detect issues immediately)
- Every 5 minutes (catch drift early)
- After any transaction submission failure (diagnose issue)

Nonce Recovery Scenarios:

Scenario 1: Transaction Dropped from Mempool

Worker submits transaction with nonce 100
Transaction enters mempool
Network congestion—transaction not mined for 10 minutes
Mempool purges old transactions
Transaction disappears

Issue: Database says nonce 101 (incremented), but nonce 100 never mined

Recovery:

1. Reconciliation detects chain_nonce (100) < db_nonce (101)
2. System realizes nonce 100 never mined
3. Resubmit transaction with nonce 100 but higher gas price
4. Transaction mines successfully
5. Nonce sequences resume

Scenario 2: Worker Crashes After Nonce Assignment

Worker begins transaction:
- Acquires lock, reads nonce 100
- Updates database nonce to 101
- CRASH (before submitting blockchain transaction)

Issue: Database says nonce 101, but nonce 100 was never used

Recovery:

1. Settlement job remains in QUEUED state (transaction never submitted)
2. Another worker eventually picks up the job
3. Worker sees job in QUEUED, attempts processing
4. Worker tries to acquire nonce, gets 101
5. Worker submits with nonce 101
6. Blockchain rejects: "nonce too high, expecting 100"
7. Worker detects error, triggers reconciliation
8. Reconciliation resets nonce to 100
9. Retry succeeds

Key Insight: Nonce management must be robust to crashes, network failures, and race conditions. Database locks + reconciliation provide this robustness.

6.4 Transaction Signing with KMS

Signing transactions is security-critical. We use KMS (Key Management Service) to keep private keys secure.

Why KMS?

Security:

- Private keys never leave KMS hardware

- Keys stored in HSM (Hardware Security Module)—tamper-proof device
- Even with full AWS account access, attacker can't extract keys

Auditability:

- Every signing operation logged
- Can audit: "Who signed what, when?"
- Helps detect unauthorized transactions

Key Rotation:

- Can generate new keys without touching code
- Old keys remain for verification of historical signatures
- Smooth transition from old key to new key

Compliance:

- Meets regulatory requirements (SOC 2, FIPS 140-2)
- Important for enterprise customers

Signing Process:

[DIAGRAM_015_KMS_SIGNING_FLOW] *Transaction signing flow showing security boundaries between Worker and KMS/HSM, with steps: serialize, hash, send to KMS, sign in secure enclave, return signature, construct signed transaction.*

Worker has unsigned transaction:
{
from: "0xAAAA...",
to: "0xBBBB...",
value: 0,
data: "0xa9059cbb...",
nonce: 101,
gasPrice: 50000000,
gasLimit: 100000,
chainId: 8453
}

Step 1: Serialize Transaction
- RLP encode the transaction (Recursive Length Prefix — Ethereum's serialization format)
- Result: byte array like [0xf8, 0x6c, 0x65, ...]

Step 2: Hash Transaction
- tx_hash = keccak256(RLP_encoded_tx)
- Result: 32-byte hash like 0x1234abcd5678ef90...

Step 3: Send to KMS
- KMS_API_CALL: Sign(KeyId="relayer-base-v1", Message=tx_hash, MessageType="DIGEST")
- KMS uses private key stored in HSM to sign
- Returns signature components: { r, s, v }

Step 4: Construct Signed Transaction
- Append signature to transaction
- RLP encode again with signature included
- Result: signed_tx (byte array)

Step 5: Verify (Optional but Recommended)
- Recover address from signature
- recovered_address = ecrecover(tx_hash, signature)
- Verify: recovered_address == from_address
- If mismatch, something went wrong (abort)

Step 6: Submit
- eth_sendRawTransaction(signed_tx)
- Blockchain validates signature and processes transaction

ECDSA Signature Components:

r, s, v: These are mathematical components of the ECDSA signature

- **r**: X-coordinate of elliptic curve point (32 bytes)

- **s**: Signature proof (32 bytes)
- **v**: Recovery ID (1 byte, usually 27 or 28)

Why v? ECDSA signatures are not unique—two possible signatures for same message. The v value indicates which one, allowing address recovery.

KMS Configuration:

```
AWS KMS Key Configuration:
  - KeyType: ECC_SECG_P256K1 (same curve as Ethereum)
  - KeyUsage: SIGN_VERIFY
  - KeyPolicy: Only settlement-worker IAM role can use
  - Logging: All operations logged to CloudWatch
  - Rotation: Manual (create new key, update code to use new KeyId)
```

Key Rotation Process:

[DIAGRAM_026_KEY_ROTATION_PROCESS] *Step-by-step key rotation timeline showing creation of new key, funding new address, configuration update, monitoring transition, and archiving old key.*

```
Current: relayer-base-v1
New: relayer-base-v2

1. Generate new key in KMS (relayer-base-v2)
2. Derive Ethereum address from public key
3. Transfer USDC balance from old address to new address
4. Update worker configuration to use new KeyId
5. Deploy worker update
6. Monitor: new transactions using new key
7. After 90 days, archive old key (disable signing, keep for verification)
```

Why 90 days? Gives time to verify all old settlements, resolve any disputes, etc.

6.5 Confirmation Monitoring

After submitting transaction, worker must wait for confirmation. This is not instant.

Blockchain Confirmation Process:

[DIAGRAM_016_CONFIRMATION_MONITORING] *Timeline showing blockchain block progression with increasing confirmations (1, 2, 3) and decreasing reorg risk (1% → 0.1% → 0.01%), from transaction submission to final confirmation.*

T=0s: Transaction Submitted
- Worker calls eth_sendRawTransaction
- Transaction enters mempool (pool of pending transactions)
- Miners see transaction, decide whether to include

T=2s: Transaction Mined (Block N)
- Miner includes transaction in new block
- Block N is mined and broadcast
- Transaction now has 1 confirmation

T=4s: Block N+1 Mined
- Next block builds on Block N
- Transaction now has 2 confirmations

T=6s: Block N+2 Mined
- Transaction now has 3 confirmations
- On L2 (Base), this is considered final

T=6s: Worker Marks as Confirmed
- Worker queries receipt: eth_getTransactionReceipt(tx_hash)
- Receipt shows: status=success, blockNumber=N
- Worker queries current block: eth_blockNumber → N+3
- Confirmations = (N+3) - N = 3 ✓
- Mark invoice as SETTLED

Why Wait for Multiple Confirmations?

Blockchain Reorgs:

- Blockchains can temporarily "fork"—two miners find blocks simultaneously
- Eventually, one chain becomes longer (canonical)
- Shorter chain is abandoned (reorg)
- Transactions in abandoned blocks become unconfirmed again

Example Reorg:

[DIAGRAM_017_BLOCKCHAIN_REORG] *Fork visualization showing how blockchain reorganization occurs: competing chains from same parent block, longer chain wins, transaction in shorter chain becomes unconfirmed.*

Block N-1
↓
Block N (includes our transaction)
↓
Block N+1

Meanwhile, another miner found different Block N:
Block N-1
↓
Block N' (doesn't include our transaction)
↓
Block N+1'
↓
Block N+2' ← This chain becomes longer

Reorg happens: Network switches to longer chain
Our transaction is now unconfirmed again!

Reorg Probability:

- 1 confirmation: ~1% reorg chance
- 2 confirmations: ~0.1% reorg chance
- 3 confirmations: ~0.01% reorg chance
- 12 confirmations: ~0.0000001% reorg chance (effectively impossible)

Confirmation Requirements:

Base L2: 3 confirmations (~6 seconds)

- L2s have faster block times and less reorg risk
- 3 confirmations gives ~0.01% reorg risk (acceptable)

Arbitrum L2: 3 confirmations (~3 seconds)

- Even faster block times

Ethereum Mainnet: 12 confirmations (~3 minutes)

- Slower block times, more reorg risk
- 12 confirmations is standard for high-value transfers

Monitoring Loop:

```
Function MonitorConfirmation(tx_hash, network):
  required_confirmations = NETWORK_CONFIG[network].confirmations
  poll_interval = NETWORK_CONFIG[network].poll_interval
  timeout = 5 minutes

  start_time = now()

  while true:
    if now() - start_time > timeout:
      RAISE_ERROR("Transaction confirmation timeout")

    receipt = eth_getTransactionReceipt(tx_hash)

    if receipt == null:
      // Transaction not yet mined
      SLEEP(poll_interval)
      continue

    if receipt.status == "reverted":
      RAISE_ERROR("Transaction reverted: " + receipt.revertReason)

    current_block = eth_blockNumber()
    confirmations = current_block - receipt.blockNumber + 1

    if confirmations >= required_confirmations:
      // Success!
      RETURN receipt

    SLEEP(poll_interval)
```

Handling Stuck Transactions:

What if transaction never confirms?

Scenario: Gas Price Too Low

```
Worker submits transaction with gasPrice=1 Gwei
Network congestion—minimum gasPrice now 10 Gwei
Transaction sits in mempool, never mined
```

Solution: Gas Price Bumping

After 2 minutes of no confirmation:

1. Query current gas price: 10 Gwei
2. Create NEW transaction with same nonce but higher gas:
   - nonce: 101 (same)
   - gasPrice: 15 Gwei (50% higher than current)
   - all other fields same
3. Sign and submit
4. Miners see two transactions with nonce 101
5. Miners prefer higher gas price (15 Gwei)
6. New transaction gets mined
7. Old transaction automatically invalid (same nonce)

This is called "transaction replacement" or "gas bumping."

Important: Must increase gas price by at least 10% (most networks enforce this rule). Can't replace with same or lower gas price.

[Document continues with remaining sections...]

7. State Management & Consistency

State management is the backbone of system reliability. Let's understand how we maintain consistency across distributed components.

7.1 The State Machine Approach

A state machine is a formal model where a system can be in exactly one "state" at any time, with well-defined transitions between states.

Why State Machines?

Clarity:

- At any moment, we know exactly what state an invoice is in
- No ambiguity like "is this paid?" "maybe?" "partially?"
- Clear answer: PENDING, VALIDATED, SETTLING, or SETTLED

Correctness:

- Define valid transitions: PENDING → VALIDATED ✓
- Reject invalid transitions: SETTLED → PENDING ✗
- Prevents bugs from unexpected state changes

Recoverability:

- After a crash, read state from database
- Know exactly where we left off
- Resume processing from that point

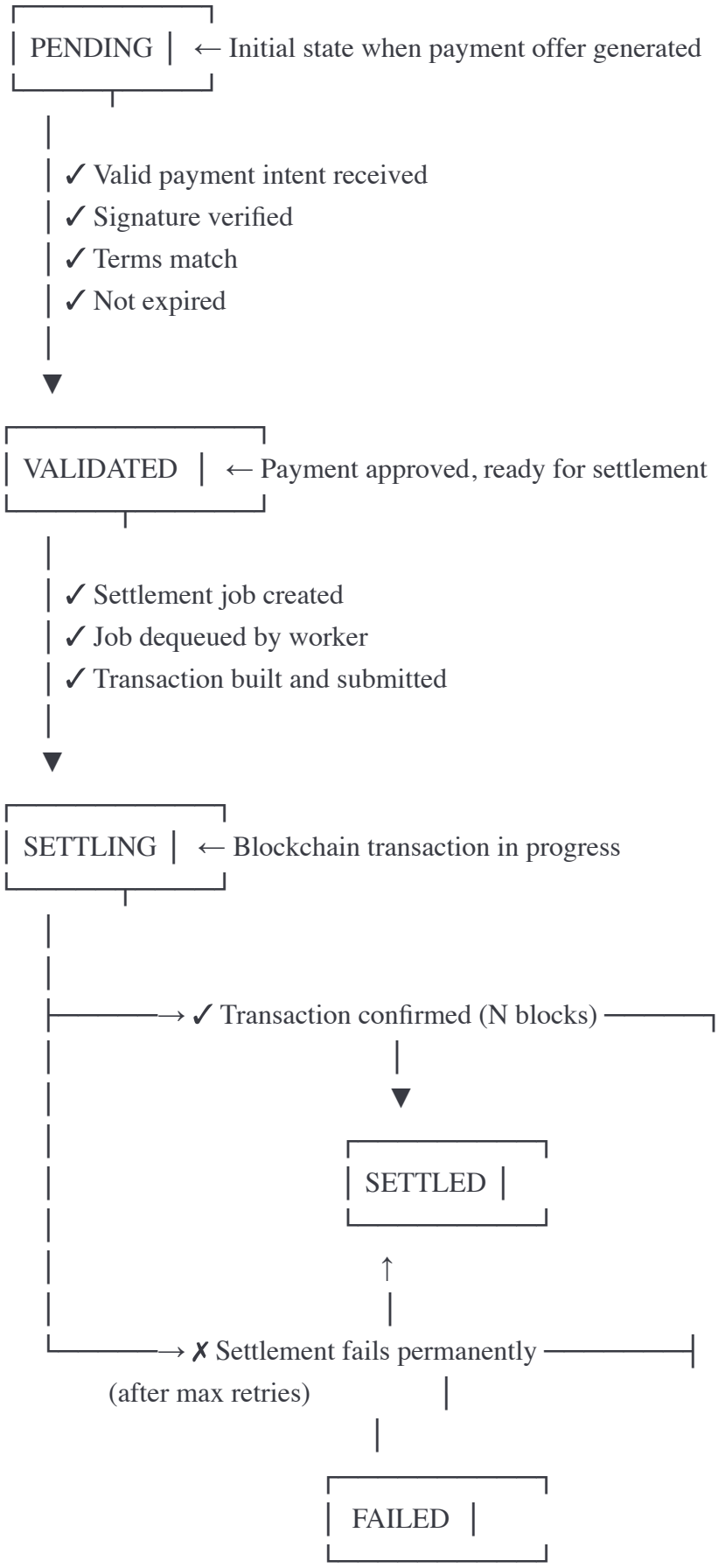
Auditability:

- Track state changes over time
- Answer questions like: "When did this invoice get validated?"
- Compliance and debugging

7.2 Invoice State Machine (Detailed)

Let's examine each state and transition in detail.

[DIAGRAM_018_INVOICE_STATE_MACHINE] *Professional UML state machine diagram showing all invoice states (PENDING, VALIDATED, SETTLING, SETTLED, FAILED), valid transitions with conditions, and terminal states.*



State Definitions:

PENDING:

- **When:** Payment offer generated and returned to consumer
- **Duration:** Typically seconds to minutes (until consumer signs and returns)

- **Can transition to:** VALIDATED (normal flow) or stay PENDING forever (consumer never responds)
- **Database fields populated:** invoice_id, amount, token, network, payee, nonce, expires_at, created_at
- **Database fields NULL:** payer (don't know yet), validated_at, tx_hash, settled_at

VALIDATED:

- **When:** Consumer's signed payment intent successfully verified
- **Duration:** Milliseconds to seconds (immediately enqueued for settlement)
- **Can transition to:** SETTLING (worker picks up job)
- **New database fields:** payer (from payment intent), validated_at (timestamp)
- **System behavior:** Settlement job created in tx_jobs table

SETTLING:

- **When:** Settlement worker submits blockchain transaction
- **Duration:** 2-30 seconds (waiting for blockchain confirmation)
- **Can transition to:** SETTLED (success) or FAILED (after retries exhausted)
- **New database fields:** tx_hash (blockchain transaction identifier)
- **System behavior:** Worker polling for transaction receipt

SETTLED:

- **When:** Blockchain transaction confirmed with required confirmations
- **Duration:** Permanent (terminal state)
- **Can transition to:** Never (terminal)
- **New database fields:** settled_at (timestamp), block_number (blockchain block)
- **System behavior:** Receipt with attestation generated and stored

FAILED:

- **When:** Settlement permanently failed after all retry attempts
- **Duration:** Permanent (terminal state)
- **Can transition to:** Never (terminal—requires manual intervention)
- **System behavior:** Refund process initiated, operators alerted

7.3 Preventing Invalid State Transitions

The database schema enforces state machine rules:

Enforcement Mechanism 1: Database Constraints

sql

```
-- Status must be one of the valid enum values
ALTER TABLE invoices
ADD CONSTRAINT invoice_status_check
CHECK (status IN ('PENDING', 'VALIDATED', 'SETTLING', 'SETTLED', 'FAILED'));
```

This prevents typos like setting status to "SETTLE" (missing D) or invalid values.

Enforcement Mechanism 2: Application-Level Guards

```
Function TransitionInvoice(invoice_id, new_status):
  current_invoice = SELECT * FROM invoices WHERE invoice_id = invoice_id

  // Define valid transitions
  valid_transitions = {
    'PENDING': ['VALIDATED'],
    'VALIDATED': ['SETTLING'],
    'SETTLING': ['SETTLED', 'FAILED'],
    'SETTLED': [],  // Terminal state
    'FAILED': []    // Terminal state
  }

  if new_status not in valid_transitions[current_invoice.status]:
    RAISE_ERROR("Invalid transition: {current_invoice.status} → {new_status}")

  // Transition is valid, proceed with update
  UPDATE invoices SET status = new_status WHERE invoice_id = invoice_id
```

Enforcement Mechanism 3: Idempotency Checks

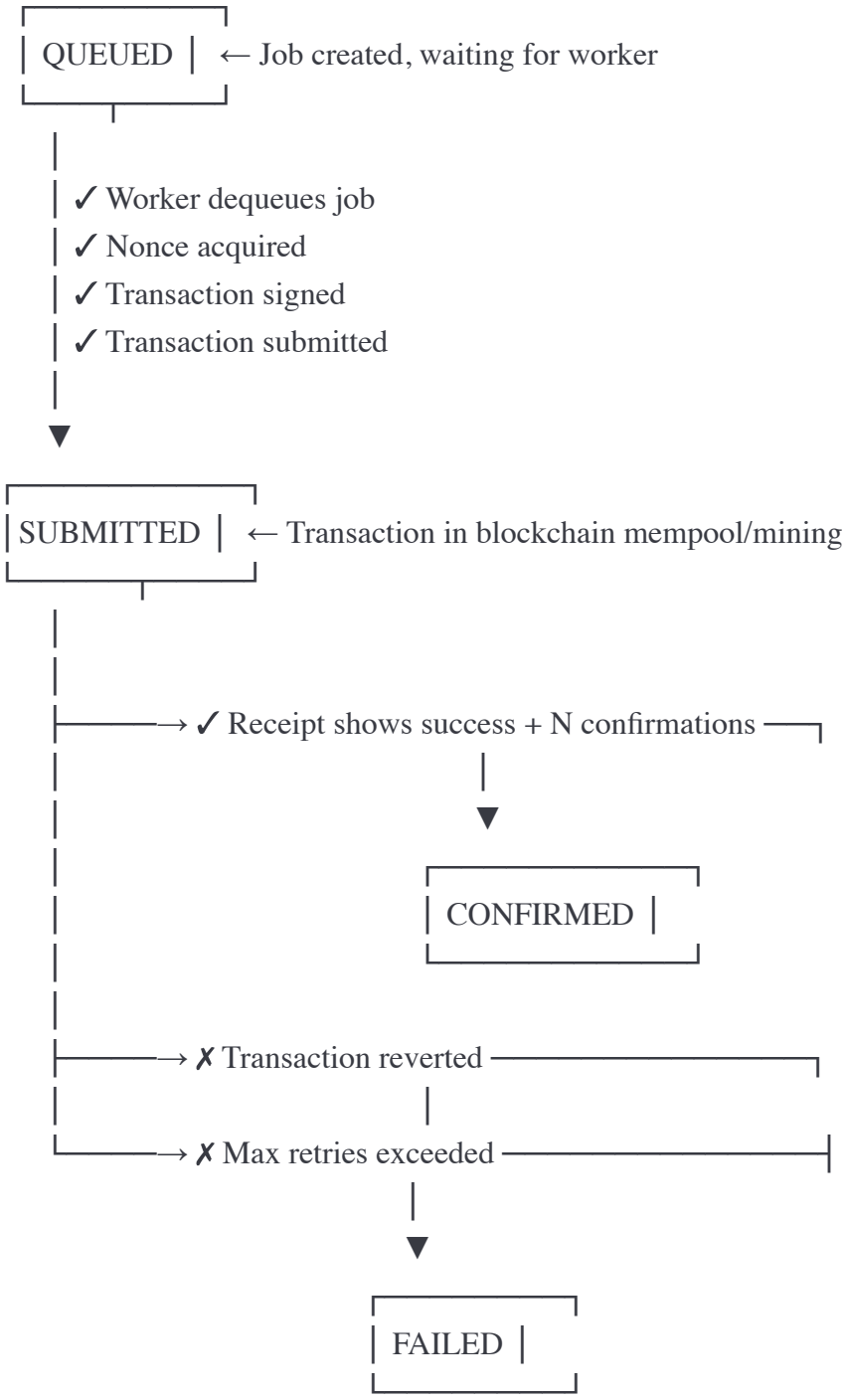
```
When processing payment validation:
  BEGIN TRANSACTION;
    SELECT * FROM invoices WHERE invoice_id = X FOR UPDATE;

    IF status != 'PENDING':
      ROLLBACK;
      RAISE_ERROR("Invoice already processed (status: {status})")

    // Only proceed if status is PENDING
    UPDATE invoices SET status = 'VALIDATED'...
  COMMIT;
```

This prevents:

- Two concurrent requests both validating the same invoice
- Accidentally re-validating an already-settled invoice
- Race conditions in distributed environment

7.4 Settlement Job State Machine

Parallel to invoice states, settlement jobs have their own state machine:



Correlation Between Invoice and Job States:

[DIAGRAM_019_INVOICE_JOB_CORRELATION] *Matrix showing how invoice states correlate with settlement job states throughout lifecycle, with valid combinations and normal progression path highlighted.*

Invoice State	Job State	Meaning
PENDING	(no job yet)	Waiting for payment
VALIDATED	QUEUED	Payment approved, job created
SETTLING	QUEUED	Job waiting for worker
SETTLING	SUBMITTED	Transaction submitted to blockchain
SETTLED	CONFIRMED	Transaction confirmed
FAILED	FAILED	Settlement failed permanently

Invariants (rules that must always be true):

1. **One-to-one relationship:** Each invoice has at most one settlement job
2. **Job existence:** If invoice.status >= VALIDATED, then job exists
3. **Job reference:** If invoice has settle_job_id, that job exists in tx_jobs
4. **Status correlation:** If job.status = CONFIRMED, then invoice.status = SETTLED
5. **Terminal states:** If invoice.status = SETTLED or FAILED, then job.status = CONFIRMED or FAILED

These invariants are checked by:

- Database foreign key constraints
- Application-level assertions
- Reconciliation jobs (detect violations)

7.5 Consistency in Distributed Systems

Our system is distributed—multiple gateway instances, multiple workers, shared database. How do we maintain consistency?

Consistency Challenge:

[DIAGRAM_020_DISTRIBUTED_CONSISTENCY] *Timeline showing how database locking prevents race conditions when multiple gateway instances attempt to process same invoice simultaneously.*

Time	Gateway-1	Gateway-2	Database
T1	Receives request (same invoice_id)	Receives request (same invoice_id)	invoice status=PENDING
T2	Validates payment	Validates payment	???
T3	Updates to VALIDATED	Updates to VALIDATED	PROBLEM!

Both gateways try to validate the same invoice. Without coordination, both might succeed, creating duplicate settlement jobs.

Solution: Database Transactions with Locking

```
Gateway-1:
  BEGIN TRANSACTION;
  SELECT * FROM invoices WHERE invoice_id=X FOR UPDATE;  ← Acquires lock
  // Gateway-2 now blocks here, waiting for lock

  if status == 'PENDING':
    UPDATE invoices SET status='VALIDATED'
    INSERT INTO tx_jobs (...)
  COMMIT;  ← Releases lock

Gateway-2:
  BEGIN TRANSACTION;
  SELECT * FROM invoices WHERE invoice_id=X FOR UPDATE;  ← Now acquires lock
  // Reads updated status = 'VALIDATED'

  if status == 'PENDING':
    // This check fails! Status is now 'VALIDATED'
    ROLLBACK
    return ERROR "Invoice already processed"
```

FOR UPDATE is the key:

- Acquires exclusive lock on the row
- Other transactions attempting to read that row (with FOR UPDATE) must wait
- This serializes access—only one transaction at a time
- Prevents race conditions

Consistency Model: Linearizability

Our system provides **linearizable consistency** for invoice operations:

- Once an invoice transitions to a new state, all subsequent reads see the new state
- No stale reads (you never see PENDING after VALIDATED)
- Operations appear to occur in a single, atomic order

This is achieved through:

- Database transactions (ACID properties)
- Row-level locking (FOR UPDATE)
- Monotonic state transitions (never backwards)

Eventual Consistency for Settlements:

While invoice state is strictly consistent, blockchain settlement is **eventually consistent**:

- We mark invoice as SETTLING immediately
- Blockchain transaction may take seconds/minutes to confirm
- During this window, invoice.status = SETTLING but onchain transfer hasn't confirmed yet
- Eventually, settlement completes and invoice.status = SETTLED matches onchain state

This is acceptable because:

- Consumer already received resource (optimistic delivery)
- Risk is on marketplace, not consumer
- Reconciliation ensures eventual consistency

7.6 Failure Recovery and System Restarts

What happens if a worker crashes mid-settlement?

Scenario: Worker Crashes After Submitting Transaction

T1: Worker dequeues job_123
T2: Worker builds transaction
T3: Worker signs transaction
T4: Worker submits transaction → tx_hash = 0xABCD...
T5: Worker updates DB: tx_jobs.status = SUBMITTED, tx_hash = 0xABCD
T6: CRASH! ← Worker process dies

Transaction is now in blockchain mempool, but worker isn't monitoring it anymore.

Recovery Process:

[DIAGRAM_021_CRASH_RECOVERY_FLOW] *Flowchart showing system recovery after worker crash: detect crash, query stuck jobs, check blockchain status, update database, finalize settlement.*

T7: System detects worker crash (health check failure)
T8: New worker instance starts up
T9: Reconciliation query runs:

```
SELECT * FROM tx_jobs
WHERE status = 'SUBMITTED'
AND submitted_at < NOW() - INTERVAL '5 minutes';
```

→ Finds job_123 (submitted but not confirmed)

T10: Worker queries blockchain:
eth_getTransactionReceipt(0xABCD...)
→ Receipt exists! Transaction mined successfully in block 12847392

T11: Worker updates database:
UPDATE tx_jobs SET status='CONFIRMED', confirmed_at=NOW() WHERE job_id='job_123'
UPDATE invoices SET status='SETTLED', block_number=12847392 WHERE invoice_id=...

T12: Generate attestation and receipt (finalize settlement)

Key Insight: Because we persisted tx_hash before the crash, we can recover by querying the blockchain for the transaction status.

Scenario: Worker Crashes Before Submitting Transaction

T1: Worker dequeues job_456
T2: Worker builds transaction
T3: Worker signs transaction
T4: CRASH! ← Before submitting

Transaction never submitted to blockchain.

Recovery:

T5: New worker starts
T6: Scan for stuck jobs:

```
SELECT * FROM tx_jobs
WHERE status = 'QUEUED'
AND created_at < NOW() - INTERVAL '10 minutes';
```

→ Finds job_456 (queued but never processed)

T7: Worker re-enqueues job:
LPUSH settlement_queue "job_456"

T8: Worker eventually picks up job again
T9: Worker processes normally this time

Idempotency: Because job_456 was never submitted, processing it again is safe. We'll use the next available nonce and submit a new transaction.

Reconciliation Schedule:

- **Every 1 minute:** Check for jobs in SUBMITTED status older than 5 minutes (stuck transactions)
- **Every 5 minutes:** Check for jobs in QUEUED status older than 10 minutes (stuck queue)
- **Every hour:** Full consistency check (invoice states vs blockchain state)
- **Every 24 hours:** Comprehensive audit (all SETTLED invoices have valid blockchain transactions)

---

8. Security Architecture

Security is paramount in a payment system. Let's explore every aspect of our security design.

8.1 Threat Landscape

Who Might Attack?

[DIAGRAM_027_ATTACK_DEFENSE_MATRIX] *Matrix showing attack types (Payment Forgery, Replay, Man-in-Middle, etc.) mapped to defense mechanisms (Signature Verification, Nonce, Encryption, etc.).*

Malicious Consumers:

- Goal: Get services without paying
- Methods: Forge signatures, replay old payments, claim never received service

Malicious Providers:

- Goal: Collect payment without delivering service
- Methods: Return fake results, claim payment never received

External Attackers:

- Goal: Steal funds, disrupt operations
- Methods: DDOS, exploit vulnerabilities, social engineering

Compromised Insiders:

- Goal: Steal funds, manipulate system
- Methods: Access to databases, keys, infrastructure

8.2 Cryptographic Security

Signature Security:

Every payment involves two signatures:

1. **Facilitator signs payment offer** (proves offer is authentic)
2. **Consumer signs payment intent** (proves authorization to pay)

Signature Algorithm: ECDSA (Elliptic Curve Digital Signature Algorithm)

Why ECDSA?

- Ethereum-native (used for all blockchain transactions)
- Smaller keys than RSA (256-bit key = 128-bit security)
- Fast verification
- Deterministic address recovery (ecrecover)

How ECDSA Signatures Work:

[DIAGRAM_022_ECDSA_SIGNATURE_PROCESS] *ECDSA signing and verification process showing key derivation (Private Key → Public Key → Address), signature generation (r, s, v), and address recovery (ecrecover).*

Alice has:
Private Key (secret): 32 random bytes, e.g., 0x1234abcd...
Public Key (derived): Point on elliptic curve
Address (derived): keccak256(PublicKey)[12:] → 20 bytes (0xABCD...)

Alice wants to sign message M:
1. Hash message: h = keccak256(M)
2. Generate random k (per-signature randomness)
3. Compute signature point: (r, s)
   - r = x-coordinate of k*G (G is generator point)
   - s = k^-1 * (h + r * PrivateKey) mod n
4. Signature is (r, s, v) where v is recovery ID

Bob verifies signature:
1. Given message M and signature (r, s, v)
2. Compute h = keccak256(M)
3. Recover public key: PubKey = ecrecover(h, r, s, v)
4. Derive address: Addr = keccak256(PubKey)[12:]
5. Check if Addr matches claimed signer

No need for Bob to have Alice's public key upfront!

Why This Prevents Forgery:

Without Alice's private key, attacker cannot compute valid (r, s) pair because:

- `s = k^-1 * (h + r * PrivateKey)` requires knowledge of PrivateKey
- Even if attacker intercepts a valid signature, they can't create new signatures for different messages
- Each signature requires a unique random k (NEVER reuse k or private key can be recovered!)

Replay Attack Prevention:

Each payment offer includes a **nonce**—a unique value never reused.

Attack Without Nonce:

1. Alice signs: PaymentIntent{amount:0.05, token:USDC}
2. Alice makes API call, gets service
3. Attacker intercepts the signed PaymentIntent
4. Attacker replays it: makes request with same signed PaymentIntent
5. System validates signature ✓ (signature is valid!)
6. Attacker gets free service!

Defense With Nonce:

1. Alice receives offer with nonce="1699564832000-a3f2d9c8"
2. Alice signs: PaymentIntent{nonce:"1699564832000-a3f2d9c8", ...}
3. Alice makes API call, invoice moves to VALIDATED
4. Attacker intercepts signed PaymentIntent
5. Attacker replays with same PaymentIntent
6. System checks: invoice with this nonce is already VALIDATED
7. System rejects: "Invoice already processed"
8. Attacker fails!

Nonce is checked in database as part of invoice_id. Once used, can never be used again.

EIP-712 Structured Data Signing:

Instead of signing arbitrary strings, we sign structured data with a schema.

Without EIP-712 (Dangerous):

Message to sign: "Pay 0.05 USDC to 0x1234... for sentiment analysis"

Problem: User's wallet shows gibberish:
"Sign message: 0x1234abcd5678..."

User can't tell what they're signing. Could be anything!
Phishing risk: Attacker tricks user into signing malicious message.

With EIP-712 (Safe):

Wallet displays structured data:

```
┌─────────────────────────────────────────────────┐
│ Sign Structured Data                            │
├─────────────────────────────────────────────────┤
│ Domain: marketplace.io                          │
│ Chain: Base (8453)                              │
│ Type: PaymentIntent                             │
├─────────────────────────────────────────────────┤
│ invoice_id: inv_abc123                          │
│ payer: 0x1234... (you)                          │
│ payee: 0x5678... (Sentiment Pro API)            │
│ amount: 50000 (0.05 USDC)                       │
│ token: USDC on Base                             │
│ network: base                                   │
├─────────────────────────────────────────────────┤
│ [Sign]  [Reject]                                │
└─────────────────────────────────────────────────┘
```

User sees exactly what they're authorizing. Much safer!

EIP-712 Structure:

[DIAGRAM_023_EIP712_STRUCTURE] *EIP-712 structured data signing architecture showing Domain Separator, Type Hash, and Final Message Hash construction with domain binding.*

```
Domain Separator = hash({
  name: "x402 Marketplace",
  version: "1",
  chainId: 8453,  // Base
  verifyingContract: FACILITATOR_ADDRESS
})

PaymentIntent Type = hash({
  PaymentIntent: [
    {name: "invoice_id", type: "string"},
    {name: "payer", type: "address"},
    {name: "amount", type: "uint256"},
    {name: "token", type: "address"},
    {name: "network", type: "string"},
    {name: "nonce", type: "string"},
    {name: "timestamp", type: "uint256"}
  ]
})

Final message hash = keccak256(
  "\x19\x01" +  // EIP-712 prefix
  DomainSeparator +
  hash(PaymentIntent data)
)

Signature = ECDSA_sign(FinalMessageHash, PrivateKey)
```

Domain Binding: Signature includes domain name (marketplace.io) and chain ID. Prevents:

- Using signature on different domain (phishing site)
- Replaying signature on different blockchain network
- Cross-application attacks

8.3 Infrastructure Security

Key Management with KMS/HSM:

The most critical security decision: where do we store private keys?

Bad: In Code

```
// NEVER DO THIS!
const RELAYER_PRIVATE_KEY = "0x1234abcd5678...";
```

Why bad:

- Key exposed in source code repository
- Anyone with code access has the key
- Can't rotate without code deployment
- If committed to Git, key is in history forever

Bad: In Environment Variables

```
// Still bad!
const RELAYER_PRIVATE_KEY = process.env.RELAYER_KEY;
```

Why bad:

- Key exposed in system environment
- Visible in process listings
- Logs may leak it
- Can't audit who accessed it

Good: In KMS/HSM

```
// Key never leaves KMS
const signature = await kms.sign({
  KeyId: "relayer-base-v1",
  Message: transaction_hash
});
```

Why good:

- Key stored in hardware security module (tamper-proof)
- Every signing operation logged
- Access controlled by IAM policies
- Keys can be rotated without code changes
- Even with full AWS access, can't extract key

KMS Security Features:

Tamper Resistance:

- Keys stored in FIPS 140-2 Level 3 certified hardware (CloudHSM)
- Physical tampering destroys keys
- Can't be extracted, even by AWS employees

Auditing:

- Every KMS API call logged to CloudWatch
- Can query: "Who signed what, when?"
- Detect unauthorized access attempts

Access Control:

- IAM policies restrict which roles can use keys
- Example policy:

```
{
  "Effect": "Allow",
  "Action": "kms:Sign",
  "Resource": "arn:aws:kms:*:*:key/relayer-base-v1",
  "Principal": {
    "AWS": "arn:aws:iam::*:role/settlement-worker"
  }
}
```

- Only settlement-worker role can sign
- Gateway role CANNOT sign (even if compromised)

Key Rotation:

Current key: relayer-base-v1 (created Jan 2025)

Step 1: Create new key
- Generate relayer-base-v2 in KMS
- Derive Ethereum address from public key: 0xNEW...

Step 2: Fund new address
- Transfer USDC from old (0xOLD...) to new (0xNEW...) address
- Transfer ETH for gas

Step 3: Update configuration
- Change KeyId in worker config: relayer-base-v1 → relayer-base-v2
- Deploy updated workers

Step 4: Monitor transition
- New settlements use new key/address
- Old key/address no longer active

Step 5: Archive old key
- After 90 days, disable signing on old key
- Keep for verification of historical signatures
- Eventually delete (after all old receipts expire)

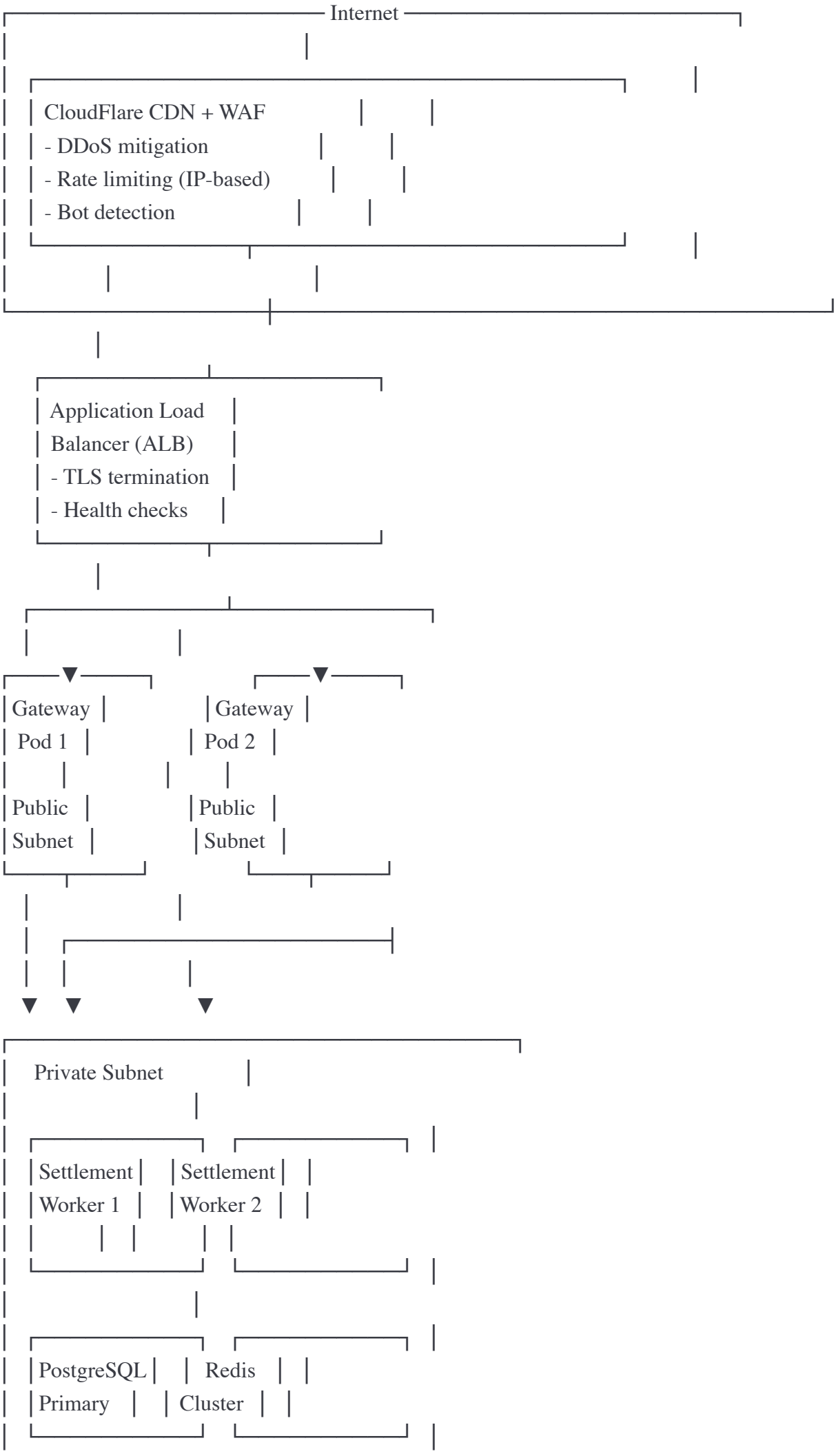
Rotation schedule:

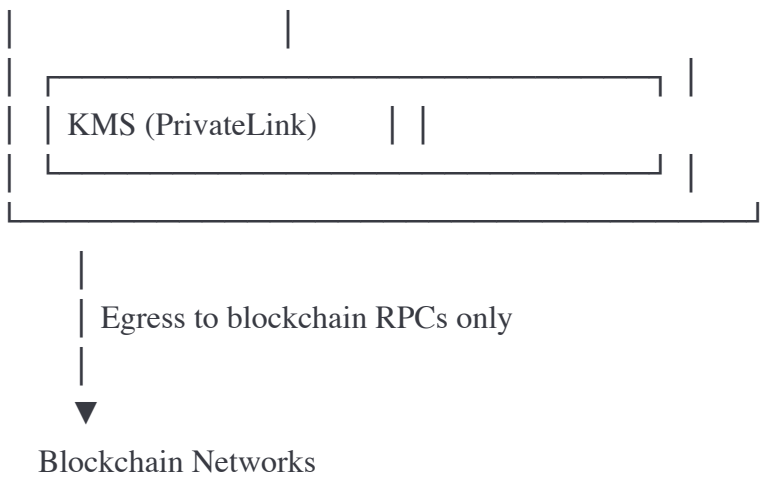
- **Attestation keys:** Every 90 days
- **Relayer keys:** Every 180 days (more complex due to onchain setup)
- **Emergency rotation:** Immediately if compromise suspected

8.4 Network Security

Defense in Depth:

[DIAGRAM_024_NETWORK_SECURITY_LAYERS] *Multi-layer network architecture showing security zones: CloudFlare WAF, Load Balancer, Gateway (public subnet), Workers/Database (private subnet), KMS (PrivateLink).*


Internet
CloudFlare CDN + WAF
- DDoS mitigation
- Rate limiting (IP-based)
- Bot detection
Application Load
Balancer (ALB)
- TLS termination
- Health checks
Gateway
Pod 1
Gateway
Pod 2
Public
Subnet
Public
Subnet
Private Subnet
Settlement
Worker 1
Settlement
Worker 2
PostgreSQL
Primary
Redis
Cluster



Layer-by-Layer Defense:

Layer 1: CloudFlare

- Protects against DDoS (distributed denial of service)
- Rate limits: Max 100 requests/second per IP
- WAF (Web Application Firewall) blocks:
  - SQL injection attempts
  - XSS (cross-site scripting)
  - Known malicious IPs
- Bot detection: CAPTCHA for suspicious traffic

Layer 2: Load Balancer

- TLS 1.3 encryption (all traffic encrypted)
- Certificate management (auto-renewal)
- Health checks: Remove unhealthy instances automatically
- Connection limits: Max 10,000 concurrent connections per instance

Layer 3: Gateway (Application)

- Authentication (optional): API key or JWT token
- Authorization: Check if consumer allowed to access provider
- Rate limiting (wallet-based): Max 10 invoices/minute per wallet
- Input validation: Reject malformed requests
- SQL injection prevention: Parameterized queries only

Layer 4: Network Segmentation

- Gateway in **public subnet** (can receive internet traffic)
- Workers in **private subnet** (no direct internet access)
- Database in **private subnet** (only internal VPC access)
- KMS via **PrivateLink** (never traverses internet)

Why Segmentation?

- If gateway compromised, attacker can't directly access database
- If worker compromised, attacker can't receive incoming connections (no shells)
- Limits blast radius of any breach

Layer 5: Secrets Management

- No secrets in code or environment variables
- All secrets in AWS Secrets Manager
- Secrets automatically rotated (passwords every 90 days)

- Access logged and audited

8.5 Application-Level Security

Rate Limiting Strategy:

Why Rate Limit?

- Prevent abuse (one consumer monopolizing system)
- Prevent DDoS (many requests overwhelming system)
- Protect providers (limit requests to their APIs)
- Economic fairness (prevent gaming system)

Multi-Level Rate Limiting:

[DIAGRAM_025_RATE_LIMITING_LEVELS] *Three-tier rate limiting architecture: Level 1 (IP-based at CDN), Level 2 (Wallet-based at application), Level 3 (Provider-based protection).*

Level 1: IP-Based (CloudFlare)

100 requests/second per IP address

If exceeded:
- Return 429 Too Many Requests
- Include Retry-After header (e.g., 60 seconds)
- Log IP for abuse monitoring

Protects against: Simple DDoS from single IP

Level 2: Wallet-Based (Application)

10 invoices/minute per wallet address
1000 invoices/day per wallet address

If exceeded:
- Return 429 Too Many Requests
- Include reason: "Wallet rate limit exceeded"
- Suggest: "Wait 60 seconds or use different wallet"

Protects against: Malicious agent spamming requests

Level 3: Provider-Based (Application)

1000 requests/minute per provider API

If exceeded:
- Return 503 Service Unavailable
- Include reason: "Provider capacity limit"
- Suggest alternative providers

Protects against: Overloading provider APIs

Implementation (Redis-based):

```
Function CheckRateLimit(wallet_address):
  key = "ratelimit:wallet:" + wallet_address
  current_count = REDIS.GET(key)

  if current_count >= 10:
    RAISE_ERROR("Rate limit exceeded")

  REDIS.INCR(key)
  REDIS.EXPIRE(key, 60)  // Key expires after 60 seconds

  return OK
```

Sliding Window Algorithm:

Problem with simple counter: burst at boundary

```
Minute 1: 10 requests at 00:59 (allowed)
Minute 2: 10 requests at 01:00 (allowed)
Result: 20 requests in 1 second! (boundary burst)
```

Solution: Sliding window

Track timestamps of each request
Limit = requests in last 60 seconds (rolling window)

00:59: request (count in last 60s = 1)
01:00: request (count in last 60s = 2)
01:01: request (count in last 60s = 2, old request fell out of window)

SQL Injection Prevention:

Vulnerable Code (NEVER DO THIS):

```
invoice_id = req.params.invoice_id;
query = "SELECT * FROM invoices WHERE invoice_id = '" + invoice_id + "'";
```

Attack:

```
invoice_id = "'; DROP TABLE invoices; --"
Resulting query: SELECT * FROM invoices WHERE invoice_id = '; DROP TABLE invoices; --'
Result: Entire invoices table deleted!
```

Safe Code (ALWAYS DO THIS):

```
invoice_id = req.params.invoice_id;
query = "SELECT * FROM invoices WHERE invoice_id = $1";
result = db.query(query, [invoice_id]);  // Parameterized query
```

Database driver handles escaping. Even if invoice_id contains SQL, it's treated as literal string.

XSS Prevention:

If we ever return user input in responses, must sanitize:

```
// Dangerous
user_provided_text = "<script>alert('XSS')</script>";
response = "<div>" + user_provided_text + "</div>";
// Browser executes script!

// Safe
user_provided_text = "<script>alert('XSS')</script>";
escaped_text = escape_html(user_provided_text);
// Result: "&lt;script&gt;alert('XSS')&lt;/script&gt;"
response = "<div>" + escaped_text + "</div>";
// Browser displays text, doesn't execute
```

8.6 Operational Security

Principle of Least Privilege:

Every system component has minimum necessary permissions.

Gateway IAM Role:

```
Permissions:
  ✓ Read from invoices table
  ✓ Write to invoices table
  ✓ Write to tx_jobs table (create settlement jobs)
  ✗ Delete from invoices table (never needed)
  ✗ Access to KMS signing keys (doesn't sign transactions)
  ✗ SSH access to database (application access only)
```

Worker IAM Role:

```
Permissions:
  ✓ Read from tx_jobs, invoices tables
  ✓ Update tx_jobs, invoices tables
  ✓ KMS sign operations (for relayer key)
  ✓ Write to receipts table
  ✗ Create new invoices (only gateway does this)
  ✗ Delete any tables
```

Admin Role:

Permissions:
✓ Read all tables
✓ Manual invoice updates (emergency only, with audit log)
✓ KMS key management (rotation, not signing)
✓ Infrastructure changes (deploy new code)
✗ Direct KMS signing (can't sign arbitrary transactions)

Multi-Factor Authentication (MFA):

All human access requires MFA:

- Admin console login: Username + password + TOTP code
- AWS console access: IAM user + MFA device
- Database admin access: Certificate + password + MFA
- SSH to servers: Certificate + MFA

Audit Logging:

Every security-relevant action is logged:

- **KMS API calls:** Who signed what, when?
- **Database modifications:** What changed, by whom?
- **IAM policy changes:** Who modified permissions?
- **Failed authentication attempts:** Potential breach attempts?
- **Rate limit violations:** Abusive behavior?

Logs are:

- Immutable (write-once, can't be altered)
- Stored for 1 year (compliance requirement)
- Monitored for anomalies (automated alerts)
- Backed up offsite (disaster recovery)

Security Incident Response Plan:

[DIAGRAM_028_SECURITY_INCIDENT_RESPONSE] *Decision tree for security incident response showing escalation path based on severity: Suspected Breach, Key Compromise, Database Breach with steps for each.*

Level 1: Suspected Breach

1. Isolate: Disconnect affected components from network
2. Preserve evidence: Take snapshots, copy logs
3. Analyze: Determine scope of breach
4. Contain: Patch vulnerability, reset credentials
5. Notify: Inform affected parties (if data exposed)
6. Review: Post-mortem, improve security

Level 2: Key Compromise

1. Immediate: Disable compromised key in KMS
2. Rotate: Generate new key, update configuration
3. Audit: Review all signatures from compromised key
4. Investigate: How was key compromised?
5. Notify: If consumer/provider funds at risk
6. Compensate: From reserve fund if needed

Level 3: Database Breach

1. Immediate: Revoke all database credentials
2. Assess: What data was exposed? (No private keys, but invoice data)
3. Compliance: GDPR requires notification within 72 hours
4. Forensics: External security firm investigates
5. Remediate: Patch vulnerability, enhance monitoring
6. Communication: Transparent post-mortem to users
