T402 Protocol

HTTP-Native Stablecoin Payment Protocol

Technical Whitepaper v2.0

T402 Team
https://t402.io

January 2026

Version History

Version	Date	Changes
2.0	January 2026	Initial whitepaper release

Contents

List of Figures

List of Tables

Abstract

T402 is an open payment protocol that enables HTTP-native stablecoin payments. By leveraging the HTTP 402 "Payment Required" status code, T402 provides a standardized mechanism for web services to require and process cryptocurrency payments without intermediaries.

The proliferation of web services, APIs, and AI agents has created an unprecedented demand for seamless, programmable payments. Traditional payment systems, with their high fees, settlement delays, and geographic restrictions, are fundamentally incompatible with the pay-per-use economics of the digital age.

T402 addresses these challenges through a novel payment protocol that:

- **Utilizes HTTP 402**: Implements the long-dormant HTTP 402 status code for payment signaling
- **Supports Multiple Chains**: Works across EVM networks, Solana, TON, and TRON
- **Enables Gasless Transactions**: Leverages EIP-3009 and ERC-4337 for zero-gas payments
- **Integrates with AI**: Native support for MCP (Model Context Protocol) enabling autonomous agent payments
- **Minimizes Trust**: The Facilitator cannot redirect funds—payments flow directly to merchants

The protocol separates concerns into three layers: *Types* (transport-agnostic data structures), *Logic* (chain-specific payment schemes), and *Representation* (transport-layer encoding). This architecture enables extensibility across new blockchains and transport protocols while maintaining backward compatibility.

Production deployments demonstrate sub-cent transaction costs on Layer 2 networks, immediate settlement, and seamless integration with existing HTTP infrastructure. The protocol is fully open-source under the Apache 2.0 license, with reference implementations available in TypeScript, Python, Go, and Java.

Keywords: Payment Protocol, HTTP 402, Stablecoin, USDT, Cryptocurrency, API Monetization, Micropayments, AI Agents

Chapter 1

Introduction

1.1 Background

The internet economy has evolved dramatically over the past three decades. What began as a platform for information sharing has transformed into a complex ecosystem of APIs, web services, SaaS platforms, and increasingly, autonomous AI agents. This evolution has created new monetization challenges that traditional payment systems are ill-equipped to address.

1.1.1 The Evolution of Web Monetization

Web monetization has progressed through several distinct phases:

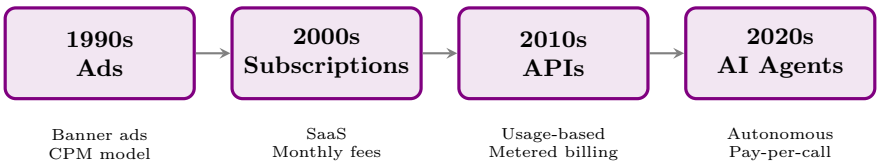

Figure 1.1: Evolution of web monetization models

Each transition has increased the granularity and automation of payments:

1. **Advertisement Era (1990s)**: Free content supported by display advertising. Payments flow from advertisers to publishers based on impressions.
2. **Subscription Era (2000s)**: Fixed monthly fees for access to services. Simple but inflexible—users pay for unused capacity while heavy users are subsidized.
3. **API Era (2010s)**: Usage-based pricing where customers pay for what they consume. Requires complex metering, billing cycles, and invoice processing.
4. **AI Agent Era (2020s)**: Autonomous agents making real-time purchasing decisions. Requires instant, machine-readable payment flows with no human intervention.

1.1.2 The Problem with Traditional Payments

Traditional payment systems were designed for human-initiated, high-value transactions. They exhibit several characteristics that make them unsuitable for the modern web:

Table 1.1: Traditional Payment System Limitations

Characteristic	Traditional	Impact
Minimum transaction	$0.50+	Micropayments impossible
Fee structure	2.9% + $0.30	Small payments uneconomical
Settlement time	1–3 days	Cash flow delays
Geographic limits	Regional	Global commerce barriers
API integration	Complex	Developer friction
Chargeback risk	Yes	Merchant liability
Machine access	Poor	AI agents cannot use

The Micropayment Problem

For a $0.01 API call, traditional payment processing would cost approximately $0.30—a 3,000% overhead. This economic reality has forced developers toward subscription models and credit systems that add complexity and friction.

1.1.3 The Cryptocurrency Promise

Cryptocurrencies theoretically solve many of these problems:

- **Low Fees**: Layer 2 solutions enable sub-cent transaction costs
- **Instant Settlement**: Transactions confirm in seconds
- **Global Access**: No geographic restrictions
- **Programmability**: Smart contracts enable complex payment logic
- **No Chargebacks**: Transactions are final

However, existing cryptocurrency payment approaches have their own limitations:

Table 1.2: Cryptocurrency Payment Challenges

Challenge	Description
Gas Fees	Variable costs can exceed payment value
User Experience	Wallet connections and signing are cumbersome
Volatility	Price fluctuations complicate pricing
Fragmentation	No standard protocol for payment requests
Integration	Complex blockchain-specific implementations

1.1.4 Why Stablecoins

T402 focuses exclusively on stablecoin payments (primarily USDT and USDT0) for several reasons:

1. **Price Stability**: 1:1 peg to USD eliminates volatility risk
2. **Familiar Denomination**: Prices expressed in dollars are intuitive
3. **Wide Adoption**: USDT is the most traded cryptocurrency by volume

4. **Multi-Chain**: Available on all major blockchain networks
5. **Regulatory Clarity**: Clearer compliance path than volatile tokens

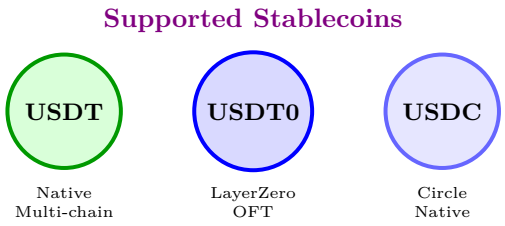

Figure 1.2: T402 supported stablecoins

1.1.5 The Rise of AI Agents

The emergence of AI agents capable of autonomous action introduces new requirements for payment systems. Unlike human users, AI agents:

1. **Operate Programmatically**: Cannot click buttons or fill forms
2. **Make Rapid Decisions**: May execute thousands of transactions per hour
3. **Require Machine-Readable APIs**: Need structured data, not web pages
4. **Have Budget Constraints**: Must evaluate costs against available funds
5. **Need Deterministic Outcomes**: Must handle failures gracefully

Agent Payment Gap

No existing payment protocol adequately addresses the needs of autonomous AI agents. Traditional systems require human interaction; existing crypto solutions lack standardization. T402 fills this gap.

1.2 HTTP 402: A Dormant Standard

In 1997, the HTTP/1.1 specification (RFC 2068) reserved status code 402 for "Payment Required." The specification noted it was "reserved for future use," anticipating the eventual need for native web payments.

> *"This code is reserved for future use. The initial motivation for this status code is that it might be useful for digital cash or micropayment schemes."* — RFC 7231 [2]

For over 25 years, this status code remained largely unused. The technical infrastructure for secure, programmable payments simply did not exist:

- No widely adopted digital currency
- No standardized signing schemes
- No programmable settlement layer
- No machine-readable payment protocols

The advent of blockchain technology, stablecoins, and standardized signing schemes (EIP-712, EIP-3009) has finally made the vision practical.

1.2.1 Why HTTP 402

Using HTTP 402 provides several advantages:

Table 1.3: Benefits of HTTP 402

Benefit	Description
Standardization	Part of HTTP specification since 1997
Semantic Clarity	Unambiguous meaning: payment required
Infrastructure	Works with existing proxies, CDNs, load balancers
Discoverability	Clients can detect paid resources automatically
Fallback	Graceful degradation for non-supporting clients

T402 activates HTTP 402 as the signaling mechanism for payment requirements, creating a standardized flow that works with existing HTTP infrastructure.

1.3 Design Goals

T402 was designed with the following core principles:

1.3.1 Minimize Trust

The protocol minimizes trust requirements between parties:

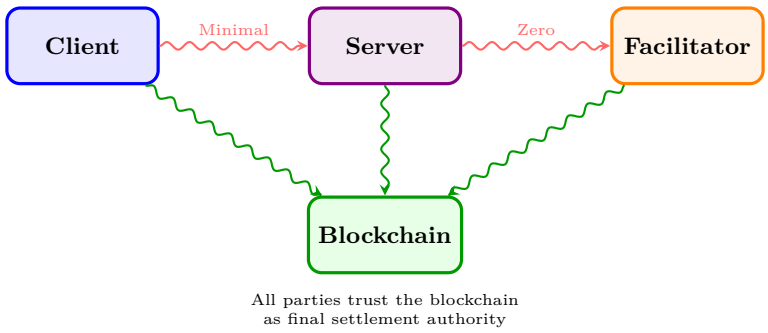

Figure 1.3: Trust relationships in T402

- **Client to Server**: Payment verified before resource delivery
- **Server to Facilitator**: Facilitator cannot redirect funds (cryptographically enforced)
- **All Parties**: Blockchain provides final settlement authority

1.3.2 Transport Agnosticism

Payment logic is separated from transport concerns, enabling the same payment flow across different communication protocols:

Table 1.4: Supported Transport Layers

Transport	Use Case	Description
HTTP	Web APIs	Traditional REST/GraphQL services
MCP	AI Tools	Model Context Protocol for LLMs
A2A	Agent Mesh	Agent-to-Agent communication

1.3.3 Chain Agnosticism

The protocol abstracts blockchain-specific details behind a unified interface:

Table 1.5: Supported Blockchain Networks

Network	Mechanism	Signature
EVM Chains	EIP-3009	ECDSA secp256k1
Solana	SPL Token	Ed25519
TON	Jetton	Ed25519
TRON	TRC-20	ECDSA secp256k1

1.3.4 Gasless Experience

Users pay only for the service—not for blockchain transaction fees:

Gasless Payments

The Facilitator sponsors all gas fees. Users sign authorizations off-chain (zero gas), and the Facilitator executes on-chain settlement. This provides a Web2-like experience while maintaining blockchain security.

1.3.5 Developer Experience

Integration should be simple. A complete server-side implementation requires minimal code:

```
import { paymentMiddleware } from "@t402/express";

app.use(paymentMiddleware({
  "GET /api/premium": {
    price: "$0.01",
    payTo: "0x..."
  }
}));
```

Listing 1.1: Express.js integration example

Client-side integration is equally straightforward:

```
import { T402Client } from "@t402/client";

const client = new T402Client({ signer });
```

```
const response = await client.fetch(
  "https://api.example.com/premium"
);
```

Listing 1.2: Client-side integration example

1.4 Comparison with Alternatives

Table 1.6: Protocol Comparison

Feature	T402	Stripe	Lightning	Request
Micropayments	✓	–	✓	–
No KYC	✓	–	✓	✓
Instant Settlement	✓	–	✓	✓
Stablecoin	✓	–	–	✓
Gasless	✓	N/A	✓	–
AI Agent Ready	✓	–	–	–
Multi-chain	✓	N/A	–	✓

1.5 Document Structure

This whitepaper is organized as follows:

Chapter 3: Architecture presents the protocol architecture, including system components, payment flows, and trust model.
Chapter 4: Core Specifications details data schemas, network identifiers, error codes, and protocol extensions.
Chapter 5: Payment Schemes describes payment schemes, focusing on the "exact" scheme across EVM, Solana, TON, and TRON.
Chapter 6: Transport Layers covers transport implementations for HTTP, MCP, and A2A protocols.
Chapter 7: Facilitator Service documents the Facilitator API, self-hosting options, and operational considerations.
Chapter 8: Security Analysis provides threat modeling, cryptographic analysis, and security recommendations.
Chapter 9: Implementation Guide offers practical guidance with SDK examples across TypeScript, Python, Go, and Java.
Chapter 10: Use Cases explores applications from API monetization to AI agent payments.
Chapter 11: Economics analyzes the cost model, fee structures, and economic comparisons.
Chapter 12: Future Work outlines planned extensions including Permit2, up-to scheme, and subscriptions.
Chapter 13: Conclusion summarizes the protocol and provides a call to action.

Chapter 2

Protocol Architecture

This chapter presents the high-level architecture of the T402 protocol, describing the roles of each component, the flow of payment information through the system, and the trust model that ensures secure operation.

2.1 Architectural Principles

The T402 protocol is built on three fundamental architectural principles:

1. **Separation of Concerns**: Payment logic, transport encoding, and blockchain operations are cleanly separated into independent layers.
2. **Trust Minimization**: No single party can unilaterally redirect funds or forge payments. The blockchain serves as the ultimate source of truth.
3. **Extensibility**: New payment schemes, transport layers, and blockchain networks can be added without modifying core protocol components.

2.2 System Components

The T402 protocol involves three primary actors, each with distinct responsibilities and trust boundaries.

2.2.1 Client

The *Client* is any application or agent that requests access to protected resources. Clients are responsible for:

- **Wallet Management**: Maintaining private keys securely and signing payment authorizations
- **Payment Construction**: Creating properly formatted `PaymentPayload` structures
- **Scheme Selection**: Choosing from available payment options in the `accepts` array
- **Error Handling**: Responding appropriately to payment failures and retrying when appropriate

Clients may take various forms:

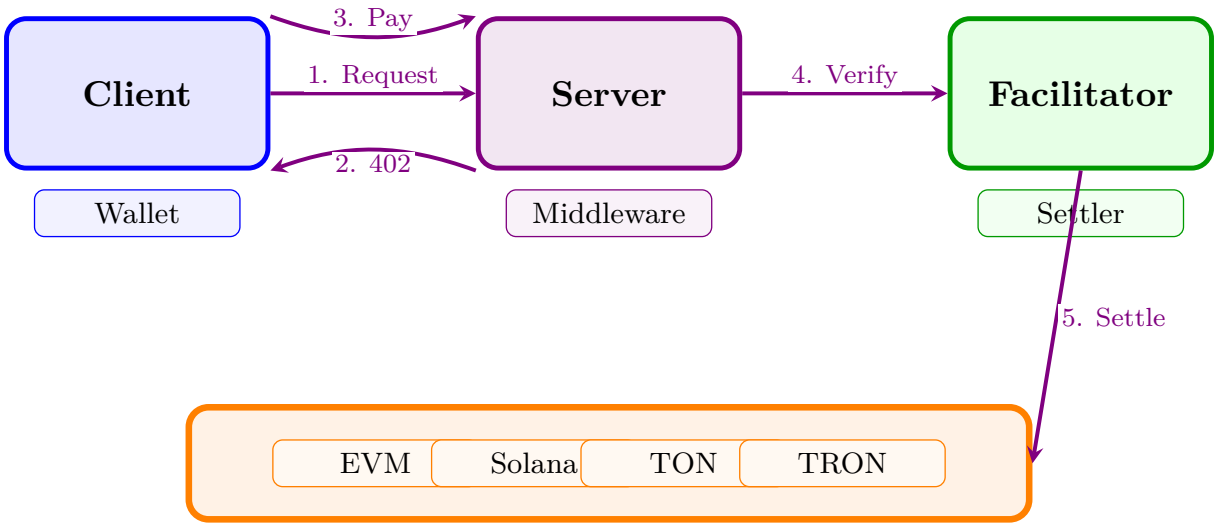

Figure 2.1: T402 System Architecture

Table 2.1: Client Types and Characteristics

Type	Signing	Use Case
Browser App	Wallet extension	Interactive web payments
Backend Service	Programmatic key	Automated API access
AI Agent	MCP integration	Autonomous resource acquisition
CLI Tool	Local keystore	Developer testing
Mobile App	Secure enclave	Consumer applications

2.2.2 Resource Server

The *Resource Server* provides access to protected resources—APIs, content, data, or computational services—and defines the payment requirements for access.

Responsibilities:

1. **Requirement Definition**: Specify acceptable payment methods including:
   - Payment scheme (e.g., "exact")
   - Supported networks (e.g., `eip155:8453`)
   - Required amount in atomic units
   - Token contract address
   - Recipient wallet address (`payTo`)
2. **Payment Signaling**: Return HTTP 402 responses with properly encoded `PaymentRequired` schemas
3. **Verification Delegation**: Forward received payment payloads to the Facilitator for verification
4. **Resource Delivery**: Serve the protected resource only after successful payment verification

Non-Custodial Design

Resource Servers never hold user funds. The `payTo` address in payment requirements is the merchant's own wallet. Payments flow directly from client to merchant via blockchain

settlement.

2.2.3 Facilitator

The *Facilitator* handles blockchain interactions on behalf of Resource Servers, providing a critical abstraction that allows servers to process payments without managing blockchain infrastructure.

Core Functions:

Verification Validate payment signatures and parameters before settlement:

- Cryptographic signature validity
- Sufficient token balance
- Correct recipient address
- Valid time window
- Unused nonce

Simulation Execute a dry-run of the transaction to ensure it would succeed on-chain
Settlement Broadcast the signed transaction to the blockchain and await confirmation
Gas Sponsorship Cover transaction fees for gasless payment flows (EIP-3009, ERC-4337)

Facilitator Security Property

The Facilitator **cannot** redirect funds to any address other than the `payTo` specified in the signed payment authorization. Any modification to payment parameters would invalidate the cryptographic signature, causing on-chain verification to fail.

2.3 Payment Flow

The standard T402 payment flow consists of six steps, illustrated in Figure 2.2.

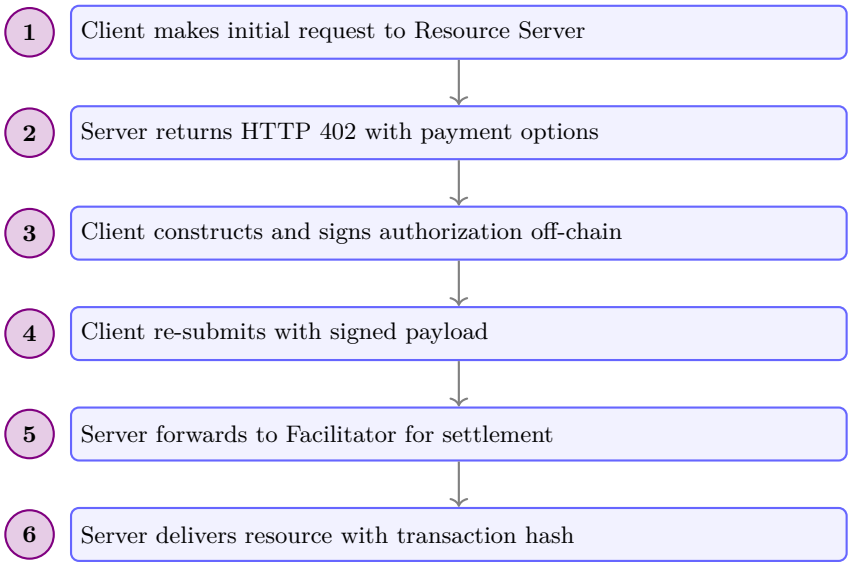

Figure 2.2: T402 Payment Flow: Six steps from request to delivery

2.3.1 Step 1: Initial Request

The client makes a standard HTTP request to the protected resource:

```
GET /api/premium-data HTTP/1.1
Host: api.example.com
Accept: application/json
```

Listing 2.1: Initial request without payment

2.3.2 Step 2: Payment Required Response

Without a valid payment, the server responds with HTTP 402 and includes payment requirements:

```
HTTP/1.1 402 Payment Required
Content-Type: application/json
PAYMENT-REQUIRED: eyJ0NDAyVmVyc2lvbiI6Mi4uLn0=

{
  "error": "Payment required to access this resource"
}
```

Listing 2.2: HTTP 402 response with payment requirements

The Base64-encoded `PAYMENT-REQUIRED` header contains the full `PaymentRequired` schema (see section 3.3).

2.3.3 Step 3: Payment Signing

The client:

1. Selects a payment option from the `accepts` array
2. Constructs the appropriate authorization (e.g., EIP-3009 for EVM)
3. Signs using the scheme-specific method (EIP-712 for EVM)

This step is **entirely off-chain** with zero gas cost to the user.

2.3.4 Step 4: Request with Payment

The client re-submits with the signed payment:

```
GET /api/premium-data HTTP/1.1
Host: api.example.com
Accept: application/json
PAYMENT-SIGNATURE: eyJ0NDAyVmVyc2lvbiI6Mi4uLn0=
```

Listing 2.3: Request with payment signature

2.3.5 Step 5: Verification and Settlement

The Resource Server:

1. Decodes the payment payload

2. Calls the Facilitator's `POST /verify` endpoint
3. Upon successful verification, calls `POST /settle`
4. Awaits blockchain confirmation

2.3.6 Step 6: Resource Delivery

Upon successful settlement:

```
HTTP/1.1 200 OK
Content-Type: application/json
PAYMENT-RESPONSE: eyJzdWNjZXNzIjp0cnVlLC4uLn0=

{
  "data": "premium content..."
}
```

Listing 2.4: Successful response with settlement proof

The `PAYMENT-RESPONSE` header contains the blockchain transaction hash, providing cryptographic proof of payment.

2.4 Trust Model

T402 minimizes trust requirements between all parties through cryptographic verification and blockchain settlement.

Table 2.2: Trust Relationships in T402

Relationship	Trust Level	Mechanism
Client → Server	Low	Payment verified before resource delivery
Server → Facilitator	Low	Facilitator cannot redirect funds
Server → Client	Zero	Blockchain provides settlement proof
Client → Facilitator	Low	Facilitator only executes signed authorizations
All → Blockchain	High	Consensus-based finality

2.4.1 Trust Assumptions

The protocol makes minimal trust assumptions:

1. **Blockchain Security**: The underlying blockchain networks provide consensus-based transaction finality
2. **Token Contract Integrity**: USDT/USDT0/USDC token contracts implement their specified interfaces correctly
3. **Transport Security**: TLS/HTTPS provides confidentiality and integrity for HTTP communications
4. **Client Key Security**: The client's private key has not been compromised

2.4.2 Facilitator Trust Properties

Theorem 2.1 (Facilitator Fund Safety)**.** *A correctly implemented Facilitator cannot transfer funds to any address other than the* `payTo` *address specified in the signed payment authorization.*

Proof. The EIP-712 typed signature covers all authorization parameters, including the recipient address (`to`). The ERC-20 contract's `transferWithAuthorization` function verifies the signature on-chain before executing the transfer. Any modification to the `to` address would produce a different message hash, causing `ecrecover` to return a different signer address, and the transaction would revert. □

2.5 Multi-Chain Architecture

T402 supports multiple blockchain networks through a unified interface.

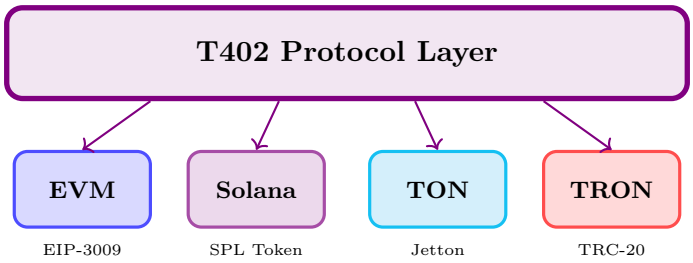

Figure 2.3: T402 multi-chain architecture with unified protocol layer

Each blockchain network is identified using CAIP-2 format (e.g., `eip155:8453` for Base), enabling:

- Unambiguous network identification
- Support for multiple networks per blockchain family
- Future extensibility to new networks

2.6 Protocol Versioning

T402 uses the `t402Version` field for protocol versioning:

Table 2.3: Protocol Version History

Version	Date	Changes
1	2025-08	Initial release with legacy network identifiers
2	2025-12	CAIP-2 networks, `resource` separation, extensions support

Version 1 Deprecation

Protocol version 1 is deprecated and no longer receives security updates. All implementations should migrate to version 2. See Appendix for migration guidance.

Chapter 3

Core Specifications

This chapter defines the core data structures, schemas, and encoding rules used throughout the T402 protocol. These specifications are transport-agnostic and chain-agnostic, forming the foundation upon which specific implementations are built.

3.1 Overview

All T402 messages use JSON encoding with UTF-8 character set. The protocol defines three primary message types:

Figure 3.1: T402 message flow

3.2 Protocol Version

3.2.1 Version Field

All T402 messages include a `t402Version` field indicating the protocol version:

```
{
  "t402Version": 2
}
```

Listing 3.1: Version field

Table 3.1: Protocol Version History

Version	Status	Key Features
1	Deprecated	Legacy network identifiers, limited extension support
2	Current	CAIP-2 networks, resource separation, extensions

3.2.2 Version Negotiation

When a client supports multiple versions, it should:

1. Attempt the highest supported version first
2. Fall back to lower versions if server rejects
3. Cache server version preference for future requests

Version 1 Deprecation

Protocol version 1 is deprecated and should not be used for new implementations. It lacks CAIP-2 network identifiers and has limited security features. All implementations should use version 2.

3.3 PaymentRequired Schema

When a resource server requires payment, it responds with the `PaymentRequired` message in the response body.

3.3.1 Structure

```
{
  "t402Version": 2,
  "error": "Payment required for resource access",
  "resource": {
    "url": "https://api.example.com/data",
    "description": "Premium market data API",
    "mimeType": "application/json"
  },
  "accepts": [
    {
      "scheme": "exact",
      "network": "eip155:8453",
      "amount": "10000",
      "asset": "0x833589fCD...02913",
      "payTo": "0x209693Bc6...287C",
      "maxTimeoutSeconds": 60,
      "extra": {
        "name": "USDC",
        "version": "2"
      }
    }
  ],
  "extensions": {}
}
```

Listing 3.2: PaymentRequired schema

Table 3.2: PaymentRequired Fields

Field	Type	Req	Description
`t402Version`	integer	Yes	Protocol version (must be 2)
`error`	string	No	Human-readable error message
`resource`	ResourceInfo	Yes	Information about the resource
`accepts`	array	Yes	Acceptable payment options
`extensions`	object	No	Protocol extensions

Table 3.3: ResourceInfo Fields

Field	Type	Req	Description
`url`	string	Yes	Canonical URL of the resource
`description`	string	No	Human-readable description
`mimeType`	string	No	Expected response MIME type

3.3.2 Field Definitions

3.3.3 ResourceInfo Object

3.4 PaymentRequirements Schema

Each element in the `accepts` array specifies one acceptable payment method.

3.4.1 Structure

```
{
  "scheme": "exact",
  "network": "eip155:8453",
  "amount": "10000",
  "asset": "0x833589fCD6eDb6E08f4c7C32D4f71b54bda02913",
  "payTo": "0x209693Bc6afc0C5328bA36FaF03C514EF312287C",
  "maxTimeoutSeconds": 60,
  "extra": {
    "name": "USDC",
    "version": "2"
  }
}
```

Listing 3.3: PaymentRequirements schema

3.4.2 Field Definitions

3.4.3 Amount Encoding

Amounts are encoded as decimal strings representing the smallest unit of the token:

Definition 3.1 (Atomic Units)**.** An atomic unit is the smallest indivisible unit of a token. For tokens with d decimals, the atomic amount equals the display amount multiplied by 10^d.

Table 3.4: PaymentRequirements Fields

Field	Type	Req	Description
`scheme`	string	Yes	Payment scheme (e.g., “exact”)
`network`	string	Yes	CAIP-2 network identifier
`amount`	string	Yes	Amount in atomic units
`asset`	string	Yes	Token contract address
`payTo`	string	Yes	Recipient wallet address
`maxTimeoutSeconds`	integer	Yes	Maximum payment timeout
`extra`	object	No	Scheme-specific data

Table 3.5: Amount Encoding Examples

Token	Decimals	Display	Atomic (string)
USDT/USDC	6	$1.00	“1000000”
USDT/USDC	6	$0.01	“10000”
USDT/USDC	6	$0.001	“1000”
ETH	18	1.0 ETH	“1000000000000000000”

String Encoding Required

Amounts MUST be encoded as strings, not numbers. JSON numbers have limited precision and can cause errors with large values. The string “1000000000000000000” is safe; the number 1000000000000000000 may lose precision.

3.4.4 Asset Address Formatting

Asset addresses follow chain-specific formatting rules:

Table 3.6: Asset Address Formats

Chain	Format	Example
EVM	0x + 40 hex chars	`0x833589fCD...`
Solana	Base58 (32-44 chars)	`EPjFWdd5Au...`
TON	EQ/UQ + Base64	`EQCxE6mUtQ...`
TRON	T + Base58Check	`TR7NHqjeKQ...`

3.5 PaymentPayload Schema

Clients construct a `PaymentPayload` to authorize payment for a resource.

3.5.1 Structure

```
{
  "t402Version": 2,
  "resource": {
    "url": "https://api.example.com/data"
  },
  "accepted": {
    "scheme": "exact",
```

```
    "network": "eip155:8453",
    "amount": "10000",
    "asset": "0x833589fCD...02913",
    "payTo": "0x209693Bc6...287C"
  },
  "payload": {
    "signature": "0x2d6a7588...",
    "authorization": {
      "from": "0x857b0651...",
      "to": "0x209693Bc...",
      "value": "10000",
      "validAfter": "0",
      "validBefore": "1740672154",
      "nonce": "0xf3746613..."
    }
  },
  "extensions": {}
}
```

Listing 3.4: PaymentPayload schema

3.5.2 Field Definitions

Table 3.7: PaymentPayload Fields

Field	Type	Req	Description
`t402Version`	integer	Yes	Protocol version (must be 2)
`resource`	ResourceInfo	No	Resource being paid for
`accepted`	Requirements	Yes	Selected payment option
`payload`	object	Yes	Scheme-specific payment data
`extensions`	object	No	Protocol extensions

3.5.3 Payload Validation Rules

The server MUST validate the following before accepting a payment:

Algorithm 1: PaymentPayload Validation

Input : PaymentPayload P, PaymentRequirements R
Output : ValidationResult

```
// Version check
if P.t402Version ≠ 2 then
    return {valid: false, error: "invalid_t402_version"}

// Scheme match
if P.accepted.scheme ≠ R.scheme then
    return {valid: false, error: "invalid_scheme"}

// Network match
if P.accepted.network ≠ R.network then
    return {valid: false, error: "invalid_network"}

// Amount check
if BigInt(P.accepted.amount) < BigInt(R.amount) then
    return {valid: false, error: "invalid_amount"}

// Recipient check
if P.accepted.payTo ≠ R.payTo then
    return {valid: false, error: "invalid_recipient"}
return {valid: true}
```

3.6 SettlementResponse Schema

After successful on-chain settlement, the server returns a `SettlementResponse`.

3.6.1 Success Response

```json
{
  "success": true,
  "transaction": "0x1234567890abcdef...",
  "network": "eip155:8453",
  "payer": "0x857b06519E91e3A54538791bDbb0E22373e36b66"
}
```

Listing 3.5: Successful settlement response

3.6.2 Error Response

```json
{
  "success": false,
  "error": "insufficient_funds",
  "message": "Payer balance is 5000, required 10000"
}
```

Listing 3.6: Failed settlement response

Table 3.8: SettlementResponse Fields

Field	Type	Req	Description
`success`	boolean	Yes	Whether settlement succeeded
`transaction`	string	If success	Transaction hash
`network`	string	If success	Network where settled
`payer`	string	If success	Payer's address
`error`	string	If failure	Error code
`message`	string	No	Human-readable error

3.6.3 Field Definitions

3.7 Network Identifiers

T402 v2 uses CAIP-2 (Chain Agnostic Improvement Proposal) format for network identification [1].

3.7.1 CAIP-2 Format

Definition 3.2 (CAIP-2 Identifier)**.** A CAIP-2 network identifier has the format:

`namespace:reference`

where:

- `namespace` identifies the blockchain family
- `reference` identifies the specific chain within that family

3.7.2 Namespace Registry

Table 3.9: CAIP-2 Namespaces

Namespace	Reference	Description
`eip155`	Chain ID	EVM-compatible chains
`solana`	Genesis hash (partial)	Solana networks
`ton`	Workchain ID	TON networks
`tron`	Genesis block hash	TRON networks

3.7.3 Supported Networks

3.8 Error Codes

T402 defines standard error codes organized by category.

Table 3.10: Supported Networks

Network	CAIP-2 ID	Tokens
Ethereum	`eip155:1`	USDT0
Base	`eip155:8453`	USDT0, USDC
Arbitrum One	`eip155:42161`	USDT0
Optimism	`eip155:10`	USDT0
Ink	`eip155:57073`	USDT0
Berachain	`eip155:80094`	USDT0
Solana	`solana:5eykt...Kvdp`	USDT
TON	`ton:-239`	USDT
TRON	`tron:0x2b6653dc`	USDT

Table 3.11: Payment Error Codes

Code	Description
`insufficient_funds`	Payer lacks sufficient token balance
`invalid_signature`	Payment signature verification failed
`invalid_amount`	Payment amount below required
`invalid_recipient`	Recipient address doesn't match payTo
`expired_authorization`	Time window has passed
`nonce_already_used`	Nonce used in previous transaction

3.8.1 Payment Errors

3.8.2 Protocol Errors

3.8.3 Settlement Errors

3.9 HTTP Headers

When using HTTP transport, T402 uses custom headers for payment data.

3.9.1 Request Headers

3.9.2 Response Headers

Header Encoding

All header values are Base64-encoded JSON to ensure safe transmission through HTTP infrastructure. The JSON is first serialized to UTF-8 bytes, then Base64-encoded.

3.10 Protocol Extensions

The `extensions` field enables optional functionality beyond core payment mechanics.

3.10.1 Extension Structure

```
{
  "extensions": {
    "siwx": {
```

Table 3.12: Protocol Error Codes

Code	Description
`invalid_t402_version`	Protocol version not supported
`invalid_network`	Network not supported
`invalid_scheme`	Payment scheme not supported
`invalid_payload`	Malformed payment payload
`invalid_payment_requirements`	Invalid requirements format

Table 3.13: Settlement Error Codes

Code	Description
`simulation_failed`	Transaction simulation failed
`settlement_failed`	On-chain settlement failed
`unexpected_verify_error`	Unexpected verification error
`unexpected_settle_error`	Unexpected settlement error

```
      "info": {
        "address": "0x...",
        "chainId": 8453,
        "signature": "0x..."
      },
      "schema": "https://t402.io/schemas/siwx.json"
    }
  }
}
```

Listing 3.7: Extension structure

3.10.2 Extension Processing Rules

1. Extensions MUST be ignored if not understood
2. Extensions MUST NOT affect core payment processing
3. Extension names SHOULD be namespaced (e.g., "t402:siwx")
4. Extensions MAY include validation schemas

3.10.3 Standard Extensions

3.11 JSON Schema Definitions

Complete JSON Schema definitions are provided in Appendix 12.3. Key validation rules:

- `t402Version` must equal 2
- `amount` must be a non-negative integer string
- `network` must match CAIP-2 format
- `accepts` must contain at least one element
- `scheme` must be a recognized scheme identifier

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "type": "object",
  "required": ["t402Version", "resource", "accepts"],
  "properties": {
```

Table 3.14: HTTP Request Headers

Header	Description
`X-PAYMENT-SIGNATURE`	Base64-encoded PaymentPayload JSON
`X-PAYMENT`	Alias for X-PAYMENT-SIGNATURE

Table 3.15: HTTP Response Headers

Header	Description
`X-PAYMENT-REQUIRED`	Base64-encoded PaymentRequired JSON
`X-PAYMENT-RESPONSE`	Base64-encoded SettlementResponse JSON

```
    "t402Version": {
      "type": "integer",
      "const": 2
    },
    "accepts": {
      "type": "array",
      "minItems": 1
    }
  }
}
```

Listing 3.8: Example validation with JSON Schema

Table 3.16: Standard Extensions

Extension	Status	Description
`siwx`	Planned	Sign-In-With-X identity (CAIP-122)
`subscription`	Planned	Recurring payment authorization
`escrow`	Planned	Conditional payment release
`receipt`	Planned	Cryptographic payment receipts

Chapter 4

Payment Schemes

Payment schemes define how payments are constructed, validated, and settled on specific blockchain networks. This chapter details the "exact" scheme implementation across all supported chains.

4.1 Scheme Architecture

Each payment scheme consists of three components:

1. **Client Logic**: How to construct the `payload` field within `PaymentPayload`
2. **Verification Logic**: How to validate payment parameters and signatures
3. **Settlement Logic**: How to execute the on-chain transaction

Figure 4.1: Payment scheme component flow

4.2 Exact Scheme Overview

The `exact` scheme enables payments of a precise amount from payer to recipient.

Definition 4.1 (Exact Scheme)**.** A payment scheme where the authorized transfer amount exactly equals the required payment amount, with no partial payments, refunds, or variability.

4.2.1 Properties

4.2.2 Scheme Identifier

The exact scheme is identified by `"scheme": "exact"` in payment requirements and payloads.

Table 4.1: Exact Scheme Properties

Property	Description
Deterministic	Payment amount known at request time
Atomic	Payment succeeds or fails completely
Non-custodial	Funds flow directly to recipient
Gasless	User pays no transaction fees (Facilitator sponsors)
Single-use	Each authorization can only be used once
Time-bounded	Authorizations expire after `validBefore` timestamp

4.3 EVM Implementation

On EVM-compatible chains, the exact scheme uses EIP-3009 (Transfer with Authorization) for gasless, signature-based transfers.

4.3.1 EIP-3009: Transfer with Authorization

EIP-3009 defines a standard interface for token transfers authorized by cryptographic signatures rather than on-chain approval transactions:

```
interface IEIP3009 {
    function transferWithAuthorization(
        address from,
        address to,
        uint256 value,
        uint256 validAfter,
        uint256 validBefore,
        bytes32 nonce,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) external;

    function authorizationState(
        address authorizer,
        bytes32 nonce
    ) external view returns (bool);
}
```

Listing 4.1: EIP-3009 interface

4.3.2 Authorization Parameters

4.3.3 EIP-712 Typed Data Signing

The authorization uses EIP-712 structured signing for security and user experience:

```
const domain = {
  name: tokenName,         // e.g., "USD Coin"
  version: tokenVersion,   // e.g., "2"
  chainId: chainId,        // e.g., 8453 for Base
```

Table 4.2: EIP-3009 Authorization Parameters

Parameter	Type	Description
`from`	address	Payer's wallet address
`to`	address	Recipient's wallet address (must match `payTo`)
`value`	uint256	Transfer amount in atomic units
`validAfter`	uint256	Unix timestamp when authorization becomes valid
`validBefore`	uint256	Unix timestamp when authorization expires
`nonce`	bytes32	Random 32-byte value for replay protection
`v, r, s`	uint8, bytes32, bytes32	ECDSA signature components

```
  verifyingContract: tokenAddress
};

const types = {
  TransferWithAuthorization: [
    { name: "from", type: "address" },
    { name: "to", type: "address" },
    { name: "value", type: "uint256" },
    { name: "validAfter", type: "uint256" },
    { name: "validBefore", type: "uint256" },
    { name: "nonce", type: "bytes32" }
  ]
};
```

Listing 4.2: EIP-712 domain and types

4.3.4 Payload Structure

The `payload` field for EVM exact scheme:

```
{
  "signature": "0x2d6a7588...af148b571c",
  "authorization": {
    "from": "0x857b...36b66",
    "to": "0x2096...287C",
    "value": "10000",
    "validAfter": "1740672089",
    "validBefore": "1740672154",
    "nonce": "0xf374...3480"
  }
}
```

Listing 4.3: EVM exact scheme payload

4.3.5 Verification Algorithm

Algorithm 2: EVM Exact Scheme Verification

Input : PaymentPayload P, PaymentRequirements R
Output : VerifyResponse

`// Step 1: Signature Recovery`
$hash \leftarrow \text{EIP712Hash}(P.authorization)$;
$signer \leftarrow \text{ecrecover}(hash, P.signature)$;
if $signer \neq P.authorization.from$ **then**
  return *{isValid: false, invalidReason: "invalid_signature"}*;

`// Step 2: Balance Verification`
$balance \leftarrow \text{balanceOf}(R.asset, P.authorization.from)$;
if $balance < R.amount$ **then**
  return *{isValid: false, invalidReason: "insufficient_funds"}*;

`// Step 3: Amount Validation`
if $P.authorization.value < R.amount$ **then**
  return *{isValid: false, invalidReason: "invalid_amount"}*;

`// Step 4: Recipient Validation`
if $P.authorization.to \neq R.payTo$ **then**
  return *{isValid: false, invalidReason: "invalid_recipient"}*;

`// Step 5: Time Window Validation`
$now \leftarrow \text{currentTimestamp}()$;
if $now < P.authorization.validAfter$ **then**
  return *{isValid: false, invalidReason: "authorization_not_yet_valid"}*;
if $now > P.authorization.validBefore$ **then**
  return *{isValid: false, invalidReason: "expired_authorization"}*;

`// Step 6: Nonce Validation`
$used \leftarrow \text{authorizationState}(P.authorization.from, P.authorization.nonce)$;
if *used* **then**
  return *{isValid: false, invalidReason: "nonce_already_used"}*;

`// Step 7: Transaction Simulation`
$result \leftarrow \text{eth\_call}(\text{transferWithAuthorization}, P)$;
if *result.reverted* **then**
  return *{isValid: false, invalidReason: "simulation_failed"}*;
return *{isValid: true, payer: P.authorization.from}*;

4.3.6 Settlement Process

Upon successful verification, the Facilitator executes settlement:

1. Construct the `transferWithAuthorization` transaction
2. Sign with the Facilitator's hot wallet (for gas payment)
3. Broadcast to the network
4. Wait for confirmation (typically 1-2 blocks on L2)
5. Return transaction hash in `SettlementResponse`

4.4 Solana (SVM) Implementation

On Solana, the exact scheme uses SPL Token `TransferChecked` instructions with Facilitator-sponsored fee payment.

4.4.1 Transaction Structure

Solana payments require a specific instruction layout:

Table 4.3: Required Solana Transaction Instructions

Index	Program	Instruction
0	ComputeBudget	SetComputeUnitLimit
1	ComputeBudget	SetComputeUnitPrice
2	Token/Token2022	TransferChecked

4.4.2 Payload Structure

```
{
  "transaction": "AQAAAAAAAAAAAAAA...base64-encoded..."
}
```

Listing 4.4: Solana exact scheme payload

The `transaction` field contains a Base64-encoded, serialized, **partially-signed** versioned Solana transaction. The client signs as the token authority; the Facilitator adds its signature as the fee payer.

4.4.3 PaymentRequirements Extra Fields

```
{
  "extra": {
    "feePayer": "EwWqGE4ZFKLofuestmU4LDdK7XM1N4ALgdZccwYugwGd"
  }
}
```

Listing 4.5: Solana-specific extra fields

4.4.4 Facilitator Security Rules

Critical Security Checks for Solana

The Facilitator **MUST** enforce all of the following:

1. **Instruction Layout**: Exactly 3 instructions in the specified order
2. **Fee Payer Safety**: Fee payer address must NOT appear in any instruction accounts
3. **Authority Safety**: Fee payer must NOT be the transfer authority or source
4. **Compute Price Bound**: Compute unit price $\leq$ 5 lamports per CU
5. **Destination Validation**: Destination must equal the ATA PDA for (payTo, asset)
6. **Amount Exactness**: Transfer amount must exactly equal requirements amount

4.5 TON Implementation

On TON, payments use Jetton (TON's fungible token standard) transfers.

4.5.1 Jetton Transfer Message

TON Jettons use an internal message-based transfer mechanism:

```
transfer#0f8a7ea5
  query_id:uint64
  amount:(VarUInteger 16)
  destination:MsgAddress
  response_destination:MsgAddress
  custom_payload:(Maybe ^Cell)
  forward_ton_amount:(VarUInteger 16)
  forward_payload:(Either Cell ^Cell)
= InternalMsgBody;
```

Listing 4.6: TON Jetton transfer cell structure

4.5.2 Payload Structure

```
{
  "boc": "te6ccgEBAgEAhgAB...base64-encoded-BOC..."
}
```

Listing 4.7: TON exact scheme payload

4.5.3 Verification Requirements

- Validate Ed25519 signature
- Verify sender wallet address matches authorization
- Check Jetton master contract address
- Validate workchain ID (0 for basechain)
- Verify sufficient balance

4.6 TRON Implementation

On TRON, payments use TRC-20 token transfers.

4.6.1 Transaction Structure

TRON uses Protobuf-encoded transactions with ECDSA secp256k1 signatures.

4.6.2 Payload Structure

```
{
  "transaction": "0a02...hex-encoded-protobuf...",
  "signature": "0x..."
}
```

Listing 4.8: TRON exact scheme payload

4.6.3 Special Considerations

- **Energy/Bandwidth**: TRON uses energy and bandwidth instead of gas
- **Address Format**: Base58Check encoding (starts with 'T')
- **Contract Address**: Official USDT: `TR7NHqjeKQxGTCi8q8ZY4pL8otSzgjLj6t`

4.7 Scheme Comparison

Table 4.4: Exact Scheme Implementation Comparison

Aspect	EVM	Solana	TON	TRON
Signature	ECDSA	Ed25519	Ed25519	ECDSA
Standard	EIP-3009	SPL Token	Jetton	TRC-20
Nonce	32 bytes	N/A (tx sig)	query_id	N/A
Gasless	Native	Fee sponsor	Fee sponsor	Energy
Finality	∼2s (L2)	∼0.4s	∼5s	∼3s

4.8 Future Schemes

4.8.1 Up-To Scheme

The planned "up-to" scheme will support metered payments:

- **Mechanism**: EIP-2612 (Permit) for EVM
- **Use Case**: Pay for actual usage up to a maximum
- **Settlement**: Partial settlement of authorized amount

```
{
  "scheme": "up-to",
  "network": "eip155:8453",
  "maxAmount": "1000000",
  "asset": "0x...",
  "payTo": "0x..."
}
```

Listing 4.9: Up-to scheme requirements (proposed)

4.8.2 Permit2 Integration

Uniswap's Permit2 offers improved efficiency:

- Single approval for multiple protocols
- Batch transfers
- Expiring approvals for better security

Chapter 5

Transport Layers

Transport layers define how T402 messages are encoded and transmitted across different communication protocols. The protocol is designed to be transport-agnostic, allowing the same payment logic to work across HTTP, MCP, A2A, and custom transports.

5.1 Transport Architecture

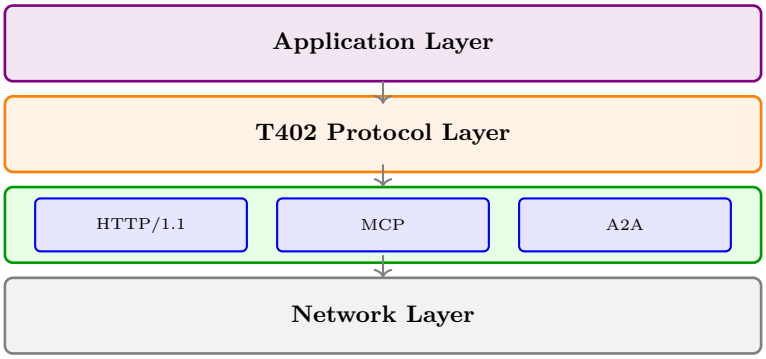

Figure 5.1: Transport layer architecture

Each transport must implement three core operations:

Table 5.1: Transport Operations

Operation	Direction	Description
Signal Payment	Server → Client	Transmit PaymentRequirements
Receive Payment	Client → Server	Accept PaymentPayload
Return Result	Server → Client	Send SettlementResponse

5.2 HTTP Transport

The HTTP transport is the canonical implementation, using standard HTTP mechanisms for maximum compatibility with existing infrastructure.

5.2.1 Header Specification

T402 defines three custom HTTP headers:

Table 5.2: HTTP Headers for T402

Header	Direction	Content
`PAYMENT-REQUIRED`	S → C	Base64url(PaymentRequirements)
`PAYMENT-SIGNATURE`	C → S	Base64url(PaymentPayload)
`PAYMENT-RESPONSE`	S → C	Base64url(SettlementResponse)

Base64url Encoding

All header values use Base64url encoding (RFC 4648 §5) without padding. This ensures compatibility with HTTP header character restrictions and avoids issues with `+`, `/`, and `=` characters.

5.2.2 Status Code Mapping

Table 5.3: HTTP Status Code Mapping

T402 State	HTTP	Header	Body
Payment Required	402	PAYMENT-REQUIRED	Error JSON
Invalid Payload	400	–	Error details
Verification Failed	402	PAYMENT-RESPONSE	Error details
Settlement Failed	402	PAYMENT-RESPONSE	Error details
Success	200	PAYMENT-RESPONSE	Resource
Server Error	500	–	Error details

5.2.3 Request Flow

5.2.4 Complete Example

```
# Client sends initial request
GET /api/premium/data HTTP/1.1
Host: api.example.com
Accept: application/json
User-Agent: T402-Client/2.0

# Server responds with payment requirements
HTTP/1.1 402 Payment Required
Content-Type: application/json
PAYMENT-REQUIRED: eyJ0NDAyVmVyc2lvbiI6Miwic...
X-T402-Version: 2

{
  "error": "Payment required",
  "code": "payment_required",
  "resource": "/api/premium/data"
}
```

Listing 5.1: Phase 1: Initial request and 402 response

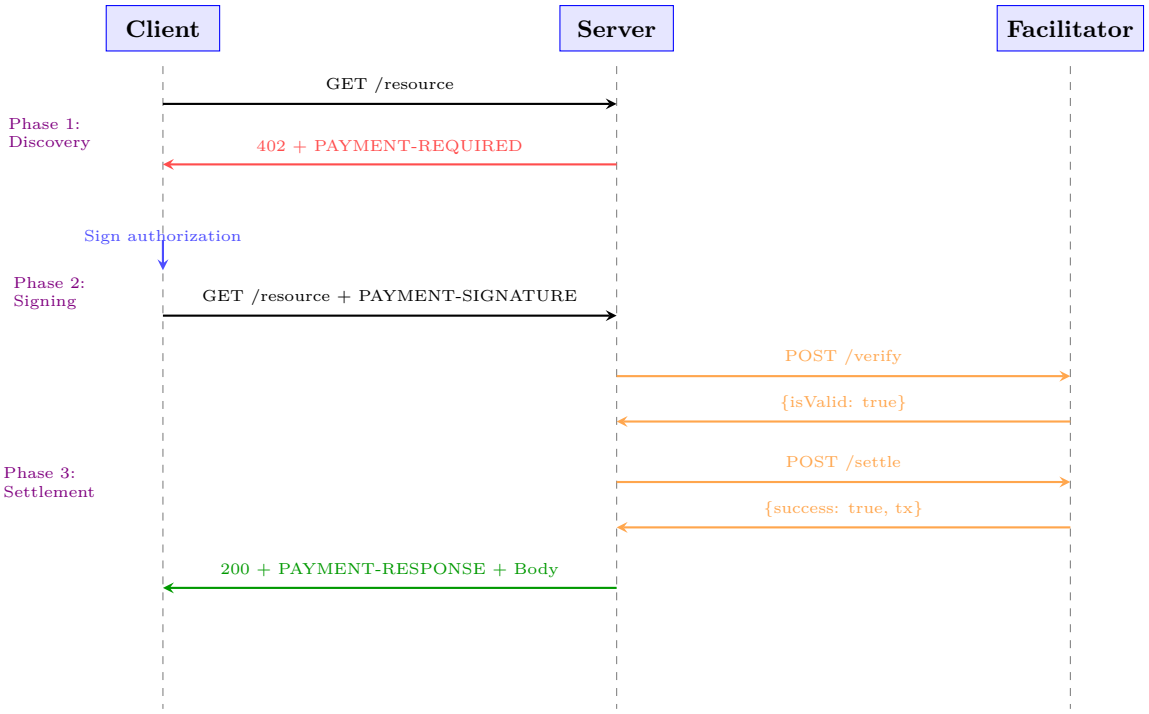

Figure 5.2: HTTP transport message flow

```
# Client sends request with signed payment
GET /api/premium/data HTTP/1.1
Host: api.example.com
Accept: application/json
PAYMENT-SIGNATURE: eyJ0NDAyVmVyc2lvbiI6Miwi...

# Server returns resource with settlement confirmation
HTTP/1.1 200 OK
Content-Type: application/json
PAYMENT-RESPONSE: eyJzdWNjZXNzIjp0cnVlLC...
X-T402-Transaction: 0x7a8b9c...

{
  "data": { ... }
}
```

Listing 5.2: Phase 2: Request with payment

5.2.5 Error Responses

```
HTTP/1.1 402 Payment Required
Content-Type: application/json
PAYMENT-RESPONSE: eyJzdWNjZXNzIjpmYWxzZS...

{
  "error": "Payment verification failed",
  "code": "invalid_signature",
  "details": "Signature does not match payer address"
}
```

Listing 5.3: Payment verification failed

```
HTTP/1.1 402 Payment Required
Content-Type: application/json
PAYMENT-RESPONSE: eyJzdWNjZXNzIjpmYWxzZS...

{
  "error": "Payment failed",
  "code": "insufficient_funds",
  "details": "Payer balance: 0.50 USDT, required: 1.00 USDT"
}
```

Listing 5.4: Insufficient funds

5.2.6 Content Negotiation

Servers MAY support content negotiation for payment requirements:

```
# Client indicates preferred networks
GET /api/data HTTP/1.1
Accept-Payment: network=eip155:8453, network=eip155:42161

# Server responds with filtered options
HTTP/1.1 402 Payment Required
PAYMENT-REQUIRED: eyJ0NDAyVmVyc2lvbiI6Mi...
Vary: Accept-Payment
```

Listing 5.5: Content negotiation headers

5.2.7 Caching Considerations

Payment-protected resources require careful cache handling:

Table 5.4: Cache-Control Directives

Response	Recommended Headers
402 Response	`Cache-Control: no-store`
200 with Payment	`Cache-Control: private, no-cache`
Idempotent Resource	`Cache-Control: private, max-age=N`

5.3 MCP Transport

The Model Context Protocol (MCP) transport enables AI agents to make payments when invoking tools. MCP uses JSON-RPC 2.0 over stdio or HTTP.

5.3.1 Architecture Overview

Figure 5.3: MCP transport architecture

5.3.2 Tool Discovery

Paid tools advertise payment requirements in their schema:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "tools": [{
      "name": "financial_analysis",
      "description": "Analyze stock performance",
      "inputSchema": {
        "type": "object",
        "properties": {
          "ticker": { "type": "string" }
        }
      },
      "t402": {
        "price": "0.05",
        "asset": "USDT",
        "description": "Per-analysis fee"
      }
    }]
  }
}
```

Listing 5.6: MCP tool with payment requirements

5.3.3 Payment Required Response

When a tool requires payment, the server returns a JSON-RPC error:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "error": {
    "code": -32402,
    "message": "Payment required",
    "data": {
      "t402Version": 2,
      "resource": {
        "url": "mcp://server/tools/financial_analysis",
        "method": "tools/call"
      },
      "description": "Financial analysis tool",
      "mimeType": "application/json",
      "accepts": [{
        "scheme": "exact",
        "network": "eip155:8453",
        "asset": "0x833589fCD6...USDC",
        "amount": "50000",
        "payTo": "0xMerchant...",
        "maxTimeoutSeconds": 300
      }]
    }
  }
}
```

Listing 5.7: MCP payment required error

5.3.4 Payment Transmission

Payments are included in the request's `_meta` field:

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "method": "tools/call",
  "params": {
    "name": "financial_analysis",
    "arguments": {
      "ticker": "AAPL",
      "period": "1Y"
    },
    "_meta": {
      "t402/payment": {
        "t402Version": 2,
        "accepted": {
          "scheme": "exact",
          "network": "eip155:8453",
          "asset": "0x833589fCD6...USDC",
          "amount": "50000",
          "payTo": "0xMerchant..."
        },
        "payload": {
          "from": "0xPayer...",
          "validAfter": 1704067200,
          "validBefore": 1704070800,
          "nonce": "0x7f8a9b...",
          "signature": "0x2d6a75..."
        }
      }
    }
  }
}
```

Listing 5.8: MCP tool call with payment

5.3.5 Success Response

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "result": {
    "content": [{
      "type": "text",
      "text": "Analysis for AAPL: ..."
    }],
    "_meta": {
      "t402/settlement": {
        "success": true,
        "network": "eip155:8453",
```

```
        "transaction": "0x7a8b9c...",
        "payer": "0xPayer...",
        "amount": "50000"
      }
    }
  }
}
```

Listing 5.9: MCP successful response with settlement

5.3.6 Error Codes

Table 5.5: MCP T402 Error Codes

Code	Meaning	Description
-32402	Payment Required	Tool requires payment
-32403	Payment Failed	Verification or settlement failed
-32404	Invalid Payment	Malformed payment payload
-32405	Expired Payment	Authorization has expired

5.4 A2A Transport

The Agent-to-Agent (A2A) transport enables direct payments between autonomous AI agents using Google's A2A protocol.

5.4.1 A2A Protocol Overview

A2A defines agent communication primitives:

- **Agent Card**: Discovery and capability advertisement
- **Tasks**: Work requests with structured inputs/outputs
- **Messages**: Communication within task context
- **Artifacts**: Deliverables produced by agents

5.4.2 Payment in Agent Cards

Agents advertise payment requirements in their Agent Card:

```
{
  "name": "DataAnalysisAgent",
  "description": "Performs statistical analysis",
  "url": "https://agent.example.com",
  "capabilities": {
    "streaming": false,
    "pushNotifications": false
  },
  "skills": [{
    "id": "regression_analysis",
    "name": "Regression Analysis",
```

```
    "inputModes": ["application/json"],
    "outputModes": ["application/json"]
  }],
  "extensions": {
    "t402": {
      "version": 2,
      "paymentEndpoint": "/t402/payment",
      "pricing": {
        "regression_analysis": {
          "amount": "100000",
          "asset": "USDT",
          "network": "eip155:8453"
        }
      }
    }
  }
}
```

Listing 5.10: A2A Agent Card with T402 payment

5.4.3 Task Payment Flow

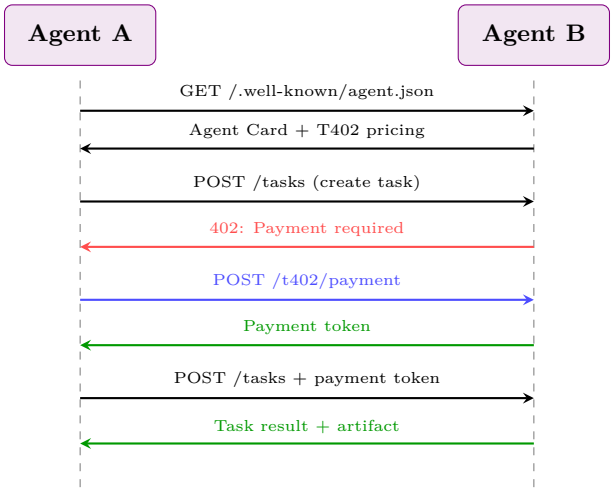

Figure 5.4: A2A payment flow

5.4.4 Payment Request

```
POST /tasks HTTP/1.1
Content-Type: application/json

{
  "skill": "regression_analysis",
  "input": {
    "data": [...],
    "variables": ["x", "y"]
  }
}

# Response
HTTP/1.1 402 Payment Required
```

```
Content-Type: application/json

{
  "error": "payment_required",
  "t402": {
    "t402Version": 2,
    "resource": {
      "url": "a2a://agent.example.com/tasks",
      "skill": "regression_analysis"
    },
    "accepts": [{
      "scheme": "exact",
      "network": "eip155:8453",
      "amount": "100000",
      "payTo": "0xAgent..."
    }]
  }
}
```

Listing 5.11: A2A task creation requiring payment

5.4.5 Task with Payment

```
POST /tasks HTTP/1.1
Content-Type: application/json
X-T402-Payment-Token: eyJhbGciOiJFUzI1NiI...

{
  "skill": "regression_analysis",
  "input": {
    "data": [...],
    "variables": ["x", "y"]
  }
}

# Response
HTTP/1.1 200 OK

{
  "taskId": "task_123",
  "status": "completed",
  "output": {
    "coefficients": [1.5, 2.3],
    "r_squared": 0.95
  },
  "t402Settlement": {
    "transaction": "0x7a8b9c...",
    "amount": "100000"
  }
}
```

Listing 5.12: A2A task with payment token

5.5 WebSocket Transport

For real-time applications, T402 supports WebSocket connections with payment state management.

5.5.1 Connection Establishment

```
# Client initiates WebSocket connection
GET /ws HTTP/1.1
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Protocol: t402.v2

# Server accepts
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Sec-WebSocket-Protocol: t402.v2
```

Listing 5.13: WebSocket connection with T402 support

5.5.2 Message Types

Table 5.6: WebSocket Message Types

Type	Direction	Purpose
`payment_required`	S → C	Signal payment needed
`payment`	C → S	Submit payment
`payment_ack`	S → C	Confirm settlement
`payment_error`	S → C	Report failure

```
{
  "type": "payment",
  "id": "msg_123",
  "t402Version": 2,
  "accepted": { ... },
  "payload": { ... }
}
```

Listing 5.14: WebSocket payment message

5.6 Custom Transport Implementation

Organizations can implement custom transports by following the transport interface specification.

5.6.1 Transport Interface

Algorithm 3: Transport Interface

```
interface T402Transport
    signalPaymentRequired(req: PaymentRequirements) → void;
    receivePayment() → PaymentPayload | null;
    sendSettlement(resp: SettlementResponse) → void;
    sendError(error: T402Error) → void;
```

5.6.2 Implementation Checklist

1. **Encoding**: Choose appropriate encoding (Base64, JSON, Protocol Buffers)
2. **Framing**: Define message boundaries
3. **Error Handling**: Map T402 errors to transport errors
4. **Timeout**: Implement payment timeout handling
5. **Idempotency**: Handle duplicate payments gracefully
6. **Security**: Ensure transport security (TLS)

5.6.3 Example: gRPC Transport

```
syntax = "proto3";

service T402Service {
  rpc GetResource(ResourceRequest)
    returns (ResourceResponse);
}

message ResourceRequest {
  string path = 1;
  bytes payment_payload = 2;  // Optional
}

message ResourceResponse {
  oneof result {
    PaymentRequired payment_required = 1;
    ResourceData data = 2;
  }
  bytes settlement_response = 3;
}

message PaymentRequired {
  bytes requirements = 1;  // JSON PaymentRequirements
}
```

Listing 5.15: gRPC service definition

Table 5.7: Transport Layer Comparison

Feature	HTTP	MCP	A2A	WebSocket
Stateless	✓	✓	✓	–
Streaming	–	✓	–	✓
Bidirectional	–	–	–	✓
Browser Support	✓	–	–	✓
AI Native	–	✓	✓	–
Caching	✓	–	–	–

5.7 Transport Comparison

Choosing a Transport

- **HTTP**: Best for REST APIs and web services
- **MCP**: Best for LLM tool integrations
- **A2A**: Best for agent-to-agent communication
- **WebSocket**: Best for real-time streaming

Chapter 6

Facilitator Service

The Facilitator is a critical infrastructure component that handles blockchain interactions on behalf of Resource Servers. It provides verification, settlement, and discovery services while abstracting the complexity of multi-chain operations.

6.1 Architecture Overview

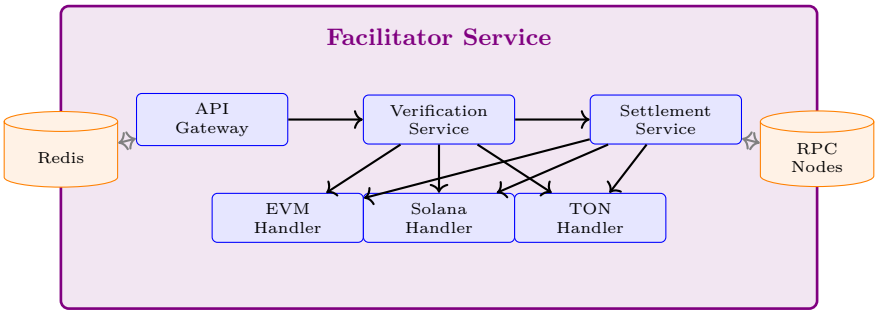

Figure 6.1: Facilitator service architecture

6.1.1 Core Responsibilities

Table 6.1: Facilitator Responsibilities

Function	Description
Verification	Validate signatures, balances, and nonces
Settlement	Execute on-chain token transfers
Gas Sponsorship	Pay transaction fees on behalf of users
Multi-chain	Abstract blockchain-specific details
Rate Limiting	Protect against abuse
Discovery	Enable resource discovery (Bazaar)

6.2 API Reference

The Facilitator exposes a RESTful API at https://facilitator.t402.io.

6.2.1 POST /verify

Validates a payment authorization without executing on-chain settlement.

Table 6.2: /verify Endpoint

Property	Value
Method	POST
Content-Type	application/json
Rate Limit	100 requests/minute
Timeout	30 seconds

6.2.1.1 Request Schema

```
{
  "paymentPayload": {
    "t402Version": 2,
    "accepted": {
      "scheme": "exact",
      "network": "eip155:8453",
      "asset": "0x833589fCD6eDb6...USDC",
      "amount": "100000",
      "payTo": "0xMerchant..."
    },
    "payload": {
      "from": "0xPayer...",
      "validAfter": 1704067200,
      "validBefore": 1704070800,
      "nonce": "0x7f8a9b0c1d2e3f4a...",
      "signature": "0x2d6a7588ce58f84b..."
    }
  },
  "paymentRequirements": {
    "t402Version": 2,
    "resource": { "url": "https://api.example.com/data" },
    "accepts": [{ ... }]
  }
}
```

Listing 6.1: /verify request body

6.2.1.2 Success Response

```
{
  "isValid": true,
  "payer": "0x857b06519E91e3A54538791bDbb0E22373e36b66",
  "network": "eip155:8453",
  "balance": "1500000",
  "validUntil": 1704070800
}
```

Listing 6.2: /verify success response

6.2.1.3 Error Response

```
{
  "isValid": false,
  "error": {
    "code": "insufficient_funds",
    "message": "Payer has insufficient balance",
    "details": {
      "required": "100000",
      "available": "50000",
      "asset": "USDC"
    }
  }
}
```

Listing 6.3: /verify error response

6.2.2 POST /settle

Executes the verified payment on the blockchain.

Idempotency

Settlement requests are idempotent. If the same nonce is submitted multiple times, only the first will be processed. Subsequent requests return the original transaction hash.

6.2.2.1 Request Schema

```
{
  "paymentPayload": {
    "t402Version": 2,
    "accepted": { ... },
    "payload": { ... }
  },
  "paymentRequirements": { ... }
}
```

Listing 6.4: /settle request body

6.2.2.2 Success Response

```
{
  "success": true,
  "network": "eip155:8453",
  "transaction": "0x7a8b9c0d1e2f3a4b5c6d7e8f9a0b1c2d...",
  "payer": "0x857b06519E91e3A54538791bDbb0E22373e36b66",
  "amount": "100000",
  "asset": "0x833589fCD6eDb6...USDC",
  "blockNumber": 12345678,
  "gasUsed": "65000",
  "effectiveGasPrice": "1000000000"
}
```

Listing 6.5: /settle success response

6.2.2.3 Error Response

```
{
  "success": false,
  "error": {
    "code": "settlement_failed",
    "message": "Transaction reverted",
    "details": {
      "reason": "ERC20: transfer amount exceeds balance",
      "transaction": "0x7a8b9c..."
    }
  }
}
```

Listing 6.6: /settle error response

6.2.3 GET /supported

Returns supported networks, schemes, and signer addresses.

```
{
  "kinds": [
    {
      "t402Version": 2,
      "scheme": "exact",
      "network": "eip155:8453"
    },
    {
      "t402Version": 2,
      "scheme": "exact",
      "network": "eip155:42161"
    },
    {
      "t402Version": 2,
      "scheme": "exact",
      "network": "solana:5eykt4UsFv8P8NJdTREpY1vzqKqZKvdp"
    }
  ],
  "extensions": [
    "subscription",
    "refund"
  ],
  "signers": {
    "eip155:*": ["0xC88f67e776f16DcFBf42e6bDda1B82604448899B"],
    "solana:*": ["8GGtWHRQ1wz5gDKE2KXZLktqzcfV1CBqSbeUZjA7hoWL"],
    "tron:*": ["TT1MqNNj2k5qdGA6nrrCodW6oyHbbAreQ5"]
  },
  "version": "2.0.0"
}
```

Listing 6.7: /supported response

6.2.4 GET /health

Health check endpoint for monitoring.

```json
{
  "status": "healthy",
  "version": "2.0.0",
  "uptime": 864000,
  "chains": {
    "eip155:8453": {
      "status": "healthy",
      "latency": 45,
      "blockHeight": 12345678
    },
    "eip155:42161": {
      "status": "healthy",
      "latency": 32,
      "blockHeight": 98765432
    },
    "solana:5eykt4UsFv8P8NJdTREpY1vzqKqZKvdp": {
      "status": "degraded",
      "latency": 250,
      "slot": 234567890
    }
  },
  "redis": {
    "status": "healthy",
    "connections": 10
  }
}
```

Listing 6.8: /health response

6.2.5 GET /metrics

Prometheus-compatible metrics endpoint.

```
# HELP t402_settlements_total Total settlements
# TYPE t402_settlements_total counter
t402_settlements_total{network="eip155:8453"} 12345
t402_settlements_total{network="eip155:42161"} 6789

# HELP t402_settlement_amount_total Total settled amount
# TYPE t402_settlement_amount_total counter
t402_settlement_amount_total{asset="USDC"} 1234567890000

# HELP t402_verification_duration_seconds Verification time
# TYPE t402_verification_duration_seconds histogram
t402_verification_duration_seconds_bucket{le="0.1"} 9500
t402_verification_duration_seconds_bucket{le="0.5"} 9900
t402_verification_duration_seconds_bucket{le="1.0"} 9990
```

Listing 6.9: /metrics response (Prometheus format)

6.3 Discovery API

The Discovery API enables the “Bazaar” concept—a decentralized marketplace for T402-enabled resources.

6.3.1 GET /discovery/resources

Lists publicly discoverable T402-enabled resources.

```
{
  "resources": [
    {
      "url": "https://api.example.com/premium",
      "description": "Premium market data API",
      "category": "finance",
      "pricing": {
        "scheme": "exact",
        "amount": "10000",
        "asset": "USDC",
        "networks": ["eip155:8453", "eip155:42161"]
      },
      "provider": {
        "name": "Example Corp",
        "verified": true,
        "rating": 4.8
      },
      "stats": {
        "totalCalls": 1234567,
        "avgLatency": 45
      }
    }
  ],
  "pagination": {
    "page": 1,
    "pageSize": 20,
    "total": 150
  }
}
```

Listing 6.10: /discovery/resources response

6.3.2 POST /discovery/register

Register a resource for discovery.

```
{
  "url": "https://api.myservice.com/data",
  "description": "Real-time weather data",
  "category": "weather",
  "pricing": {
    "scheme": "exact",
    "amount": "1000",
    "asset": "USDC"
  },
  "signature": "0x..."  // Signed by payTo address
}
```

Listing 6.11: /discovery/register request

6.4 Error Codes

Table 6.3: Facilitator Error Codes

Code	HTTP	Description
`invalid_payload`	400	Malformed request body
`invalid_signature`	400	Signature verification failed
`invalid_network`	400	Network not supported
`invalid_scheme`	400	Scheme not supported
`insufficient_funds`	402	Payer lacks balance
`nonce_already_used`	409	Nonce used in prior tx
`expired_authorization`	410	Authorization expired
`simulation_failed`	422	Tx simulation failed
`settlement_failed`	500	On-chain tx failed
`rate_limited`	429	Too many requests
`service_unavailable`	503	Temporary outage

6.5 Rate Limiting

The Facilitator implements rate limiting to prevent abuse.

Table 6.4: Rate Limits

Endpoint	Limit	Window
/verify	100 requests	1 minute
/settle	50 requests	1 minute
/supported	1000 requests	1 minute
/health	Unlimited	–
/discovery/*	60 requests	1 minute

Rate limit headers are included in responses:

```
X-RateLimit-Limit: 100
X-RateLimit-Remaining: 95
X-RateLimit-Reset: 1704067260
Retry-After: 45
```

Listing 6.12: Rate limit headers

6.6 Self-Hosting

Organizations can deploy their own Facilitator instance for enhanced control and privacy.

Table 6.5: Self-Hosting Requirements

Component	Minimum	Notes
CPU	2 cores	4+ for production
Memory	2 GB	4+ GB recommended
Storage	10 GB	SSD recommended
Network	100 Mbps	Low latency to RPCs

6.6.1 Requirements

6.6.2 Hot Wallet Setup

Security Critical

The Facilitator requires a hot wallet with gas funds. Implement proper key management:

- Use hardware security modules (HSM) in production
- Limit wallet balance to operational needs
- Monitor for unauthorized transactions
- Implement automated alerts

6.6.3 Environment Configuration

```
# Server Configuration
PORT=8080
LOG_LEVEL=info
ENVIRONMENT=production

# Network RPC URLs (required)
EVM_RPC_BASE=https://mainnet.base.org
EVM_RPC_ARBITRUM=https://arb1.arbitrum.io/rpc
EVM_RPC_ETHEREUM=https://eth.llamarpc.com
SOLANA_RPC_URL=https://api.mainnet-beta.solana.com
TON_RPC_URL=https://toncenter.com/api/v2
TRON_RPC_URL=https://api.trongrid.io

# Hot Wallet (use HSM in production)
EVM_PRIVATE_KEY=0x...
SOLANA_PRIVATE_KEY=base58...
TRON_PRIVATE_KEY=hex...

# Redis (required for rate limiting)
REDIS_URL=redis://localhost:6379
REDIS_PASSWORD=

# Rate Limiting
RATE_LIMIT_VERIFY=100
RATE_LIMIT_SETTLE=50
RATE_LIMIT_WINDOW=60

# Monitoring
METRICS_ENABLED=true
METRICS_PORT=9090
TRACING_ENABLED=true
```

```
JAEGER_ENDPOINT=http://jaeger:14268/api/traces
```

Listing 6.13: Facilitator environment variables

6.6.4 Docker Deployment

```yaml
version: "3.8"

services:
  facilitator:
    image: ghcr.io/t402-io/facilitator:latest
    ports:
      - "8080:8080"
      - "9090:9090"
    environment:
      - PORT=8080
      - REDIS_URL=redis://redis:6379
      - EVM_RPC_BASE=${EVM_RPC_BASE}
      - EVM_PRIVATE_KEY=${EVM_PRIVATE_KEY}
    depends_on:
      - redis
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8080/health"]
      interval: 30s
      timeout: 10s
      retries: 3

  redis:
    image: redis:7-alpine
    volumes:
      - redis-data:/data
    command: redis-server --appendonly yes

  prometheus:
    image: prom/prometheus:latest
    volumes:
      - ./prometheus.yml:/etc/prometheus/prometheus.yml
    ports:
      - "9091:9090"

volumes:
  redis-data:
```

Listing 6.14: docker-compose.yml

6.6.5 Kubernetes Deployment

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: facilitator
  labels:
    app: facilitator
spec:
  replicas: 3
  selector:
```

```
    matchLabels:
      app: facilitator
  template:
    metadata:
      labels:
        app: facilitator
    spec:
      containers:
      - name: facilitator
        image: ghcr.io/t402-io/facilitator:latest
        ports:
        - containerPort: 8080
        - containerPort: 9090
        envFrom:
        - secretRef:
            name: facilitator-secrets
        - configMapRef:
            name: facilitator-config
        resources:
          requests:
            memory: "512Mi"
            cpu: "500m"
          limits:
            memory: "2Gi"
            cpu: "2000m"
        livenessProbe:
          httpGet:
            path: /health
            port: 8080
          initialDelaySeconds: 10
          periodSeconds: 30
        readinessProbe:
          httpGet:
            path: /health
            port: 8080
          initialDelaySeconds: 5
          periodSeconds: 10
```

Listing 6.15: Kubernetes Deployment manifest

6.7 Monitoring and Observability

6.7.1 Key Metrics

Table 6.6: Critical Monitoring Metrics

Metric	Type	Alert Threshold
Settlement success rate	Gauge	$< 99\%$
Verification latency p99	Histogram	> 500ms
Settlement latency p99	Histogram	> 30s
Hot wallet balance	Gauge	$<$ \$100 equivalent
RPC error rate	Counter	$> 1\%$
Rate limit hits	Counter	> 100/min

6.7.2 Grafana Dashboard

A pre-built Grafana dashboard is available at:

https://grafana.facilitator.t402.io

Figure 6.2: Grafana dashboard overview

6.7.3 Alerting Rules

```yaml
groups:
- name: facilitator
  rules:
  - alert: HighSettlementFailureRate
    expr: |
      rate(t402_settlements_total{status="failed"}[5m])
      / rate(t402_settlements_total[5m]) > 0.01
    for: 5m
    labels:
      severity: critical
    annotations:
      summary: "Settlement failure rate > 1%"

  - alert: LowWalletBalance
    expr: t402_wallet_balance_usd < 100
    for: 1m
    labels:
      severity: warning
    annotations:
      summary: "Hot wallet balance low"

  - alert: HighVerificationLatency
    expr: |
      histogram_quantile(0.99,
        rate(t402_verification_seconds_bucket[5m])
      ) > 0.5
    for: 10m
    labels:
      severity: warning
    annotations:
      summary: "p99 verification latency > 500ms"
```

Listing 6.16: Prometheus alerting rules

6.8 Security Considerations

6.8.1 Authentication

For self-hosted deployments, implement API authentication:

```
# Request with API key
curl -X POST https://facilitator.example.com/verify \
  -H "Authorization: Bearer sk_live_abc123..." \
  -H "Content-Type: application/json" \
  -d '{"paymentPayload": {...}}'
```

Listing 6.17: API key authentication

6.8.2 Network Security

- Deploy behind a reverse proxy (nginx, Cloudflare)
- Enable TLS 1.3 only
- Implement IP allowlisting for /settle endpoint
- Use private networking for Redis connections
- Rotate API keys regularly

6.8.3 Audit Logging

All settlement operations are logged with:

```
{
  "timestamp": "2026-01-15T10:30:00Z",
  "event": "settlement",
  "network": "eip155:8453",
  "transaction": "0x7a8b9c...",
  "payer": "0x857b06...",
  "payTo": "0xMerchant...",
  "amount": "100000",
  "asset": "USDC",
  "gasUsed": "65000",
  "clientIP": "192.168.1.100",
  "userAgent": "T402-Server/2.0"
}
```

Listing 6.18: Audit log entry

6.9 Operational Runbook

6.9.1 Common Issues

6.9.2 Emergency Procedures

1. **Service Degradation**: Enable circuit breaker, switch to backup RPCs
2. **Wallet Compromise**: Immediately rotate keys, pause settlements
3. **Chain Outage**: Mark chain as unhealthy, inform users
4. **Data Breach**: Rotate all API keys, audit access logs

Table 6.7: Troubleshooting Guide

Symptom	**Cause**	**Resolution**
High verification latency	RPC congestion	Switch to backup RPC
Settlement failures	Low gas balance	Refill hot wallet
Rate limit errors	Traffic spike	Scale horizontally
Nonce errors	Concurrent txs	Implement nonce manager

Chapter 7

Security Analysis

This chapter provides a comprehensive security analysis of T402, including formal threat modeling, cryptographic primitive analysis, attack scenarios, and deployment recommendations.

7.1 Security Philosophy

T402 follows a **defense-in-depth** approach with multiple layers of protection:

1. **Cryptographic Layer**: Strong signature schemes prevent forgery
2. **Protocol Layer**: Nonces and deadlines prevent replay
3. **Blockchain Layer**: On-chain verification ensures finality
4. **Transport Layer**: HTTPS protects data in transit

Core Security Principle

T402 is designed so that a malicious Facilitator cannot steal user funds. The Facilitator can only settle payments to the `payTo` address specified in the cryptographically signed authorization. This is verified on-chain by the token contract.

7.2 Formal Threat Model

7.2.1 Adversary Capabilities

We consider adversaries with the following capabilities:

Table 7.1: Adversary Capability Matrix

Capability	Description
Network Access	Can observe and modify network traffic (MitM)
API Access	Can send arbitrary requests to public APIs
Prior Payments	Has access to previously completed payment data
Blockchain Access	Can submit transactions to public blockchains

7.2.2 Adversary Limitations

The security model assumes adversaries **cannot**:

- Compute discrete logarithms (break ECDSA/Ed25519)
- Find hash collisions (break Keccak-256/SHA-256)
- Compromise the user's private key
- Perform 51% attacks on supported blockchains

7.2.3 Threat Categories

Figure 7.1: Threat category taxonomy

7.3 Attack Analysis

This section provides detailed analysis of specific attack vectors and their mitigations.

7.3.1 Replay Attacks

Replay Attack Definition

An adversary captures a valid payment authorization and attempts to reuse it for additional payments.

7.3.1.1 Attack Vector

1. Adversary observes valid `PaymentPayload` from Client to Server
2. Adversary saves the payload including the signature
3. Adversary submits the same payload to a different Server or the same Server later

7.3.1.2 Mitigations

T402 employs three-layer replay protection:

Table 7.2: Replay Attack Mitigations

Layer	Mechanism	Protection
Temporal	`validBefore`	Authorization expires after deadline
Uniqueness	`nonce`	32-byte random, checked on-chain
Domain	EIP-712 Domain	Chain and contract-specific binding

Algorithm 4: Replay Protection Verification

Input : Authorization A, Current time t
Output : Boolean (protected)

```
// Temporal check
if t > A.validBefore then
    return true // Expired, cannot be replayed

// Nonce check (on-chain)
used ← authorizationState(A.from, A.nonce);
if used then
    return true // Already used, cannot be replayed

// Domain check (signature verification)
domain ← {chainId, verifyingContract};
if domain ≠ A.signedDomain then
    return true // Wrong chain/contract, invalid signature
return false // Valid, could potentially be replayed
```

7.3.2 Signature Forgery

Signature Forgery Definition

An adversary attempts to create a valid authorization signature without possessing the private key.

7.3.2.1 Attack Resistance

Theorem 7.1 (Signature Unforgeability)**.** *Under the Elliptic Curve Discrete Logarithm Problem (ECDLP) assumption, an adversary cannot forge a valid EIP-3009 authorization signature in probabilistic polynomial time.*

Proof Sketch. EIP-3009 uses ECDSA signatures over the secp256k1 curve. The security reduces to:

1. **ECDLP Hardness**: Given G and kG, computing k is computationally infeasible
2. **EIP-712 Binding**: The typed data hash binds all authorization parameters
3. **On-chain Verification**: `ecrecover` extracts signer from signature

Any signature forgery would imply either:

- Solving ECDLP (contradicts assumption)
- Finding hash collision (Keccak-256 is collision-resistant)
- Exploiting `ecrecover` (extensively audited)

□

7.3.3 Man-in-the-Middle (MitM)

7.3.3.1 Attack Vector

1. Adversary intercepts 402 response from Server
2. Modifies `payTo` to adversary's address
3. Client signs payment to adversary instead of Server

7.3.3.2 Mitigations

1. **HTTPS Requirement**: All T402 traffic MUST use TLS 1.2+
2. **Certificate Verification**: Clients MUST verify server certificates
3. **Signed Parameters**: All payment parameters are included in signature

Client Implementation Requirement

Clients MUST verify the `payTo` address belongs to the expected recipient before signing. This is typically done by matching against the request domain or a trusted list.

7.3.4 Double Spending

7.3.4.1 Attack Vector

1. Client creates authorization for payment
2. Server verifies and serves content
3. Client attempts to spend same funds elsewhere before settlement

7.3.4.2 Mitigations

1. **Balance Check**: Facilitator verifies sufficient balance during `/verify`
2. **Prompt Settlement**: Settlement typically occurs within seconds
3. **Blockchain Finality**: Once settled, transaction is irreversible

Table 7.3: Double Spend Window by Chain

Chain	Finality	Risk Window
Base/Optimism	∼2 seconds	Low
Arbitrum	∼1 second	Low
Solana	∼0.4 seconds	Very Low
TON	∼5 seconds	Low
Ethereum L1	∼13 minutes	Moderate

7.3.5 Insufficient Payment Attack

7.3.5.1 Attack Vector

1. Server requires payment of X tokens
2. Client signs authorization for $X - \epsilon$ tokens
3. Client attempts to claim full value of service

7.3.5.2 Mitigations

1. **Amount Verification**: Facilitator checks $authorization.value \geq requirements.amount$
2. **Exact Match**: For `exact` scheme, values must match precisely
3. **Server Validation**: Server re-verifies before serving content

7.3.6 Wrong Recipient Attack

7.3.6.1 Attack Vector

1. Malicious Facilitator intercepts payment request
2. Modifies `payTo` in settlement call
3. Redirects funds to attacker's wallet

7.3.6.2 Mitigations

Theorem 7.2 (Recipient Binding)**.** *A correctly signed EIP-3009 authorization cannot be settled to any address other than the* `to` *address included in the signed data.*

Proof. The EIP-712 typed data includes the `to` parameter in the hash:

```
const types = {
  TransferWithAuthorization: [
    { name: "from", type: "address" },
    { name: "to", type: "address" },     // Bound in signature
    { name: "value", type: "uint256" },
    // ...
  ]
};
```

The token contract verifies:

```
require(
    ecrecover(hash, v, r, s) == from,
    "Invalid signature"
);
// If signature valid, 'to' is cryptographically bound
```

□

Table 7.4: Cryptographic Primitive Security Levels

Chain	Algorithm	Key Size	Security Level
EVM	ECDSA secp256k1	256 bits	128 bits
Solana	Ed25519	256 bits	128 bits
TON	Ed25519	256 bits	128 bits
TRON	ECDSA secp256k1	256 bits	128 bits

Table 7.5: Hash Function Properties

Function	Output	Collision Resistance	Usage
Keccak-256	256 bits	128 bits	EIP-712 domain hash
SHA-256	256 bits	128 bits	Solana, TON, TRON

7.4 Cryptographic Security

7.4.1 Signature Algorithms

7.4.2 Hash Functions

7.4.3 Nonce Generation

Nonce Requirements

Nonces MUST be generated using a cryptographically secure random number generator (CSPRNG). Using predictable or sequential nonces enables attack vectors.

```
import { randomBytes } from 'crypto';

// Good: Cryptographically secure random bytes
const nonce = '0x' + randomBytes(32).toString('hex');

// Bad: Predictable
const nonce = Date.now().toString(16).padStart(64, '0');

// Bad: Sequential
const nonce = (lastNonce + 1n).toString(16).padStart(64, '0');
```

Listing 7.1: Secure nonce generation

7.5 Facilitator Security

The Facilitator is a critical component that handles payment settlement. This section details its security requirements.

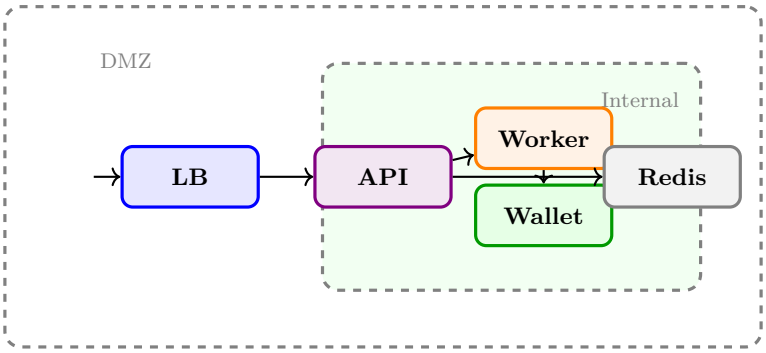

Figure 7.2: Facilitator security architecture

7.5.1 Security Architecture

7.5.2 Hot Wallet Security

Hot Wallet Risk

The Facilitator hot wallet contains funds for gas payment. While it cannot access user funds, compromising this wallet results in loss of operational capacity.

7.5.2.1 Security Measures

1. **Minimum Balance**: Keep only funds necessary for gas
2. **Monitoring**: Real-time alerts for unusual activity
3. **Key Storage**: Use HSM or secure enclave where possible
4. **Rotation**: Periodic key rotation
5. **Separation**: Different wallets per environment

```go
func loadPrivateKey() (*ecdsa.PrivateKey, error) {
    // Option 1: Environment variable (basic)
    keyHex := os.Getenv("PRIVATE_KEY")

    // Option 2: Secret manager (recommended)
    keyHex, err := vault.GetSecret("facilitator/evm-key")
    if err != nil {
        return nil, err
    }

    // Option 3: HSM (production)
    key, err := hsm.GetSigningKey("facilitator-key-id")
    if err != nil {
        return nil, err
    }

    return key, nil
}
```

Listing 7.2: Secure key loading example

Table 7.6: Facilitator API Security Controls

Control	Description
Rate Limiting	1000 req/60s per IP (configurable)
API Key Auth	Required for settlement endpoints
HTTPS Only	TLS 1.2+ mandatory
Input Validation	Strict schema validation on all inputs
CORS Policy	Configurable origin restrictions
Request Logging	Full audit trail of all operations

7.5.3 API Security

7.5.4 Denial of Service Protection

1. **Rate Limiting**: Per-IP and per-API-key limits
2. **Request Size Limits**: Maximum payload size enforced
3. **Timeout Configuration**: Aggressive timeouts on all operations
4. **Circuit Breaker**: Automatic degradation under load
5. **Geographic Distribution**: CDN and edge deployment

7.6 Chain-Specific Security

7.6.1 EVM Security Considerations

EVM-Specific Checks

- Verify `chainId` in EIP-712 domain matches target network
- Check `verifyingContract` is the expected token address
- Validate `validBefore` is reasonable (not too far in future)
- Use cryptographically random 32-byte nonces

7.6.2 Solana Security Considerations

Solana-Specific Checks

The Facilitator MUST enforce:

1. Exactly 3 instructions (ComputeBudget x2, TransferChecked)
2. Fee payer NOT in any instruction accounts
3. Compute unit price $\leq$ 5 lamports/CU
4. Destination is correct ATA PDA
5. Amount exactly matches requirements

```
1 func validateSolanaTransaction(tx *solana.Transaction, req *Requirements
    ) error {
2     // Check instruction count
3     if len(tx.Message.Instructions) != 3 {
4         return ErrInvalidInstructionCount
5     }
6
```

```
    // Validate fee payer safety
    feePayer := tx.Message.AccountKeys[0]
    for _, inst := range tx.Message.Instructions {
        for _, acc := range inst.Accounts {
            if tx.Message.AccountKeys[acc] == feePayer {
                return ErrFeePayerInInstruction
            }
        }
    }

    // Validate compute price
    computePrice := extractComputePrice(tx)
    if computePrice > maxComputePrice {
        return ErrComputePriceTooHigh
    }

    return nil
}
```

Listing 7.3: Solana security validation

7.6.3 TON Security Considerations

- Validate Ed25519 signature over the BOC (Bag of Cells)
- Verify workchain ID is 0 (basechain)
- Check Jetton master contract matches expected USDT
- Validate sender wallet address computation

7.6.4 TRON Security Considerations

- Validate Base58Check address format
- Verify contract is official USDT (`TR7NHqjeKQxGTCi8q8ZY4pL8otSzgjLj6t`)
- Check sufficient energy for transaction
- Validate protobuf transaction structure

7.7 Deployment Security

7.7.1 Container Security

```
version: '3.8'
services:
  facilitator:
    image: t402/facilitator:2.0.0
    security_opt:
      - no-new-privileges:true
    read_only: true
    tmpfs:
      - /tmp:noexec,nosuid,size=64M
    cap_drop:
      - ALL
    cap_add:
      - NET_BIND_SERVICE
    user: "1000:1000"
    networks:
```

```
16        - internal
```

Listing 7.4: Docker security configuration

7.7.2 Network Security

1. **Internal Network**: Service-to-service communication isolated
2. **Reverse Proxy**: Only Caddy/Nginx exposed externally
3. **Firewall Rules**: Whitelist only required ports
4. **TLS Termination**: At edge, internal traffic optional

7.7.3 Secret Management

Table 7.7: Secret Management Requirements

Secret	Storage	Rotation
Private Keys	HSM / Vault	90 days
API Keys	Vault / Env	30 days
Database Passwords	Vault	90 days
TLS Certificates	ACME Auto	90 days (Let's Encrypt)

7.8 Incident Response

7.8.1 Severity Classification

Table 7.8: Security Incident Severity Levels

Severity	Initial Response	Resolution	Example
Critical	24 hours	7 days	Private key leak
High	48 hours	30 days	Auth bypass
Medium	72 hours	Next release	XSS vulnerability
Low	7 days	Best effort	Minor info leak

7.8.2 Response Procedure

1. **Detection**: Automated monitoring or external report
2. **Containment**: Isolate affected systems
3. **Investigation**: Determine scope and impact
4. **Eradication**: Remove threat
5. **Recovery**: Restore services
6. **Post-mortem**: Document and improve

Table 7.9: Security Audit History

Component	Auditor	Date	Status
Protocol Design	Internal	2024 Q4	Complete
SDK Code Review	Internal	2025 Q1	Complete
External Audit	TBD	Planned	Pending

7.9 Security Audit Status

7.9.1 Completed Audits

7.9.2 Continuous Security Measures

- **Dependency Scanning**: Dependabot, govulncheck, npm audit
- **Container Scanning**: Trivy in CI/CD
- **Secret Scanning**: GitHub secret scanning enabled
- **SBOM Generation**: Per-release software bill of materials

7.10 Responsible Disclosure

7.10.1 Reporting Channels

- **GitHub Security Advisories**: Preferred method
- **Email**: `security@t402.io`

7.10.2 Bug Bounty Program

T402 maintains an active bug bounty program:

Table 7.10: Bug Bounty Rewards

Severity	Reward (USDT)
Critical (fund loss possible)	$5,000 – $10,000
High (security bypass)	$1,000 – $5,000
Medium (significant impact)	$250 – $1,000
Low (minor issues)	$50 – $250

7.11 Summary

Table 7.11: Security Property Summary

Property	Guarantee
Non-custodial	Facilitator cannot access user funds
Recipient Binding	Payments can only go to signed `payTo`
Replay Protection	Nonces + deadlines + domain separation
Signature Security	128-bit security level (ECDSA/Ed25519)
Settlement Finality	Blockchain consensus guarantees

Chapter 8

Implementation Guide

This chapter provides practical guidance for integrating T402 into applications, with examples across all supported languages and frameworks.

8.1 SDK Overview

T402 provides official SDKs for four major languages:

Table 8.1: Official SDK Comparison

SDK	Version	Registry	Frameworks	License
TypeScript	2.0.0	npm @t402/*	Express, Fastify, Next.js	Apache 2.0
Python	1.7.1	PyPI	FastAPI, Flask, Django	Apache 2.0
Go	1.5.0	Go Modules	Gin, Echo, Chi	Apache 2.0
Java	1.1.0	Maven Central	Spring Boot, WebFlux	Apache 2.0

8.1.1 TypeScript SDK Architecture

The TypeScript SDK is organized as a monorepo with 21 modular packages:

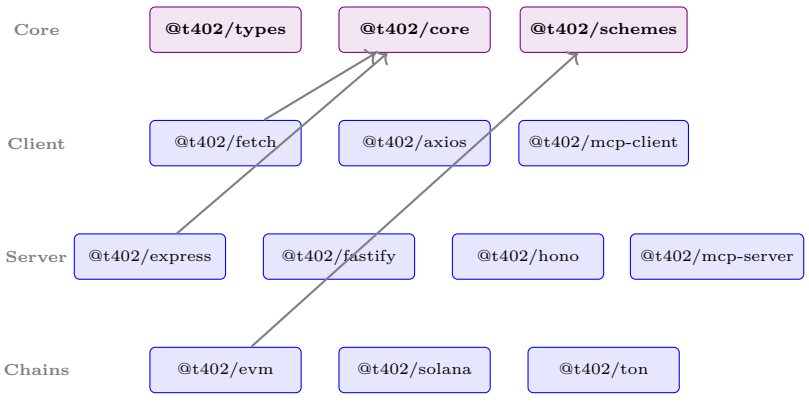

Figure 8.1: TypeScript SDK package architecture

Table 8.2: TypeScript Package Categories

Category	Packages	Purpose
Core	types, core, schemes	Type definitions, core logic
Client	fetch, axios, mcp-client	HTTP clients with auto-payment
Server	express, fastify, hono, next	Server middleware
MCP	mcp-server, mcp-client	Model Context Protocol
Chains	evm, solana, ton, tron	Chain-specific handlers

8.2 Server-Side Integration

8.2.1 Express.js (TypeScript)

```typescript
import express from "express";
import { paymentMiddleware, T402Config } from "@t402/express";

const app = express();
app.use(express.json());

// Configuration
const paymentConfig: T402Config = {
  facilitator: "https://facilitator.t402.io",
  defaultPayTo: "0xYourWalletAddress...",
  routes: {
    "GET /api/premium": {
      accepts: [{
        scheme: "exact",
        network: "eip155:8453",  // Base
        asset: "0x833589fCD6...USDC",
        amount: "10000",  // $0.01
      }],
      description: "Premium API access"
    },
    "POST /api/analyze": {
      accepts: async (req) => {
        // Dynamic pricing based on request
        const dataSize = JSON.stringify(req.body).length;
        const price = Math.ceil(dataSize / 1000) * 1000;
        return [{
          scheme: "exact",
          network: "eip155:8453",
          asset: "0x833589fCD6...USDC",
          amount: String(price),
        }];
      }
    }
  }
};

// Apply middleware
app.use(paymentMiddleware(paymentConfig));

// Protected endpoints
app.get("/api/premium", (req, res) => {
  // Payment already verified by middleware
  const { payer, amount } = req.t402Payment!;
  res.json({
```

```
    data: "Premium content",
    paidBy: payer,
    amount: amount
  });
});

app.listen(3000);
```

Listing 8.1: Complete Express.js server setup

8.2.2 FastAPI (Python)

```python
from fastapi import FastAPI, Request, Depends
from t402.fastapi import T402Middleware, PaymentConfig
from t402.fastapi import require_payment, get_payment_info

app = FastAPI()

# Global configuration
config = PaymentConfig(
    facilitator_url="https://facilitator.t402.io",
    default_pay_to="0xYourWalletAddress..."
)

# Apply middleware
app.add_middleware(T402Middleware, config=config)

# Route-specific payment requirement
@app.get("/api/premium")
@require_payment(
    price="0.01",
    network="eip155:8453",
    asset="USDC"
)
async def premium_endpoint(request: Request):
    payment = get_payment_info(request)
    return {
        "data": "Premium content",
        "paid_by": payment.payer,
        "transaction": payment.transaction
    }

# Dynamic pricing
@app.post("/api/analyze")
@require_payment(price_fn=lambda req: calculate_price(req))
async def analyze_endpoint(request: Request):
    body = await request.json()
    result = perform_analysis(body)
    return {"result": result}

def calculate_price(request: Request) -> str:
    # Price based on request complexity
    content_length = int(request.headers.get("content-length", 0))
    return f"{content_length / 100000:.4f}"  # $0.01 per 100KB
```

Listing 8.2: Complete FastAPI server setup

8.2.3 Gin (Go)

```go
package main

import (
    "github.com/gin-gonic/gin"
    "github.com/t402-io/t402/go/middleware"
    "github.com/t402-io/t402/go/types"
)

func main() {
    r := gin.Default()

    // Configure T402
    config := middleware.Config{
        FacilitatorURL: "https://facilitator.t402.io",
        DefaultPayTo:   "0xYourWalletAddress...",
    }

    // Apply to specific routes
    premium := r.Group("/api")
    premium.Use(middleware.RequirePayment(config, types.PaymentOption{
        Scheme:  "exact",
        Network: "eip155:8453",
        Asset:   "0x833589fCD6...USDC",
        Amount:  "10000",
    }))

    premium.GET("/premium", func(c *gin.Context) {
        payment := middleware.GetPayment(c)
        c.JSON(200, gin.H{
            "data":   "Premium content",
            "payer":  payment.Payer,
            "amount": payment.Amount,
        })
    })

    // Dynamic pricing
    premium.POST("/analyze", middleware.DynamicPayment(config,
        func(c *gin.Context) types.PaymentOption {
            var body map[string]interface{}
            c.ShouldBindJSON(&body)
            price := calculatePrice(body)
            return types.PaymentOption{
                Scheme:  "exact",
                Network: "eip155:8453",
                Amount:  price,
            }
        }),
        handleAnalyze,
    )

    r.Run(":8080")
}
```

Listing 8.3: Complete Gin server setup

8.2.4 Spring Boot (Java)

```java
package com.example.api;

import io.t402.spring.*;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.*;

@SpringBootApplication
@EnableT402Payments
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}

@RestController
@RequestMapping("/api")
public class PremiumController {

    @GetMapping("/premium")
    @RequirePayment(
        price = "0.01",
        network = "eip155:8453",
        asset = "USDC"
    )
    public ResponseEntity<?> getPremium(
            @PaymentInfo T402Payment payment) {
        return ResponseEntity.ok(Map.of(
            "data", "Premium content",
            "payer", payment.getPayer(),
            "transaction", payment.getTransaction()
        ));
    }

    @PostMapping("/analyze")
    @RequirePayment(priceProvider = DynamicPriceProvider.class)
    public ResponseEntity<?> analyze(
            @RequestBody AnalysisRequest request,
            @PaymentInfo T402Payment payment) {
        AnalysisResult result = analysisService.analyze(request);
        return ResponseEntity.ok(result);
    }
}

@Component
public class DynamicPriceProvider implements PriceProvider {
    @Override
    public String getPrice(HttpServletRequest request) {
        int contentLength = request.getContentLength();
        double price = contentLength / 100000.0 * 0.01;
        return String.format("%.6f", price);
    }
}
```

Listing 8.4: Complete Spring Boot setup

8.3 Client-Side Integration

8.3.1 TypeScript Fetch Client

```typescript
import { T402Client, registerExactEvmScheme } from "@t402/fetch";
import { createWalletClient, http } from "viem";
import { privateKeyToAccount } from "viem/accounts";
import { base } from "viem/chains";

// Setup wallet
const account = privateKeyToAccount("0x...");
const walletClient = createWalletClient({
  account,
  chain: base,
  transport: http()
});

// Create T402 client
const client = new T402Client({
  maxAutoPayment: "1000000",  // Max $1 auto-pay
  onPaymentRequired: async (requirements) => {
    console.log("Payment required:", requirements);
    return true;  // Approve payment
  },
  onPaymentComplete: (settlement) => {
    console.log("Paid:", settlement.transaction);
  }
});

// Register EVM scheme handler
registerExactEvmScheme(client, {
  signer: walletClient,
  preferredNetworks: ["eip155:8453", "eip155:42161"]
});

// Make requests - payments handled automatically
async function fetchPremiumData() {
  const response = await client.fetch(
    "https://api.example.com/premium"
  );

  if (!response.ok) {
    throw new Error(`Request failed: ${response.status}`);
  }

  return response.json();
}
```

Listing 8.5: TypeScript client with wallet integration

8.3.2 Python Client

```python
import asyncio
from t402.client import T402Client
from t402.schemes.evm import EvmSigner
from eth_account import Account
```

```python

# Setup wallet
private_key = "0x..."
account = Account.from_key(private_key)

# Create signer
signer = EvmSigner(
    account=account,
    rpc_url="https://mainnet.base.org"
)

# Create client
client = T402Client(
    signer=signer,
    max_auto_payment=1_000_000,  # $1 in micro-units
    preferred_networks=["eip155:8453"]
)

async def fetch_premium_data():
    async with client:
        response = await client.get(
            "https://api.example.com/premium"
        )
        return response.json()

# Callback for payment events
@client.on_payment_required
async def handle_payment(requirements):
    print(f"Payment required: {requirements.description}")
    print(f"Price: ${requirements.accepts[0].amount / 1e6}")
    return True  # Approve

@client.on_payment_complete
async def handle_complete(settlement):
    print(f"Transaction: {settlement.transaction}")

# Run
data = asyncio.run(fetch_premium_data())
```

Listing 8.6: Python client with async support

8.3.3 Go Client

```go
package main

import (
    "context"
    "fmt"
    "log"

    "github.com/t402-io/t402/go/client"
    "github.com/t402-io/t402/go/schemes/evm"
)

func main() {
    // Create EVM signer
    signer, err := evm.NewSigner(
```

```
        "0x...",  // Private key
        "https://mainnet.base.org",
    )
    if err != nil {
        log.Fatal(err)
    }

    // Create T402 client
    c := client.New(client.Config{
        Signer:            signer,
        MaxAutoPayment:    1_000_000,
        PreferredNetworks: []string{"eip155:8453"},
        OnPaymentRequired: func(req *types.PaymentRequirements) bool {
            fmt.Printf("Payment: %s\n", req.Description)
            return true
        },
    })

    // Make request
    ctx := context.Background()
    resp, err := c.Get(ctx, "https://api.example.com/premium")
    if err != nil {
        log.Fatal(err)
    }
    defer resp.Body.Close()

    // Process response
    var data map[string]interface{}
    json.NewDecoder(resp.Body).Decode(&data)
    fmt.Println(data)
}
```

Listing 8.7: Go client implementation

8.4 MCP Server Integration

8.4.1 Creating a Paid MCP Tool

```
import { McpServer } from "@modelcontextprotocol/sdk/server";
import { T402McpPlugin } from "@t402/mcp-server";

const server = new McpServer({
  name: "financial-tools",
  version: "1.0.0"
});

// Add T402 payment plugin
const t402 = new T402McpPlugin({
  facilitatorUrl: "https://facilitator.t402.io",
  payTo: "0xYourWallet..."
});

server.use(t402);

// Define paid tool
server.tool("stock_analysis", {
```

```
  description: "Analyze stock performance",
  inputSchema: {
    type: "object",
    properties: {
      ticker: { type: "string" },
      period: { type: "string", enum: ["1M", "3M", "1Y", "5Y"] }
    },
    required: ["ticker"]
  },

  // Payment configuration
  t402: {
    price: "0.10",       // $0.10 per call
    network: "eip155:8453",
    description: "Stock analysis fee"
  },

  // Tool handler
  handler: async ({ ticker, period = "1Y" }) => {
    const analysis = await performAnalysis(ticker, period);
    return {
      content: [{
        type: "text",
        text: JSON.stringify(analysis, null, 2)
      }]
    };
  }
});

// Tool with dynamic pricing
server.tool("custom_report", {
  description: "Generate custom financial report",
  inputSchema: {
    type: "object",
    properties: {
      tickers: { type: "array", items: { type: "string" } },
      metrics: { type: "array", items: { type: "string" } }
    }
  },

  t402: {
    priceFunction: (args) => {
      // $0.05 per ticker + $0.02 per metric
      const tickerCost = args.tickers.length * 0.05;
      const metricCost = args.metrics.length * 0.02;
      return String(tickerCost + metricCost);
    },
    network: "eip155:8453"
  },

  handler: async (args) => {
    return generateReport(args);
  }
});

server.listen();
```

Listing 8.8: MCP server with paid tools

8.5 Testing Guide

8.5.1 Unit Testing

```typescript
import { describe, it, expect, vi } from "vitest";
import { createMockPayment, MockFacilitator } from "@t402/testing";

describe("Payment Middleware", () => {
  it("should require payment for protected routes", async () => {
    const mockFacilitator = new MockFacilitator();

    const response = await request(app)
      .get("/api/premium")
      .expect(402);

    expect(response.headers["payment-required"]).toBeDefined();
    const requirements = decodePaymentRequired(
      response.headers["payment-required"]
    );
    expect(requirements.accepts[0].amount).toBe("10000");
  });

  it("should accept valid payment", async () => {
    const payment = createMockPayment({
      scheme: "exact",
      network: "eip155:8453",
      amount: "10000",
      payer: "0xTestPayer..."
    });

    const response = await request(app)
      .get("/api/premium")
      .set("PAYMENT-SIGNATURE", encodePayment(payment))
      .expect(200);

    expect(response.body.data).toBeDefined();
  });

  it("should reject insufficient payment", async () => {
    const payment = createMockPayment({
      amount: "5000"  // Less than required
    });

    await request(app)
      .get("/api/premium")
      .set("PAYMENT-SIGNATURE", encodePayment(payment))
      .expect(402);
  });
});
```

Listing 8.9: Unit testing with mocks

8.5.2 Integration Testing

```typescript
import { T402TestClient } from "@t402/testing";

```

```
describe("Payment Flow Integration", () => {
  let client: T402TestClient;

  beforeAll(async () => {
    // Use Base Sepolia testnet
    client = await T402TestClient.create({
      network: "eip155:84532",  // Base Sepolia
      faucetUrl: "https://faucet.t402.io"
    });

    // Fund test wallet
    await client.fundWallet("1000000");  // 1 USDC
  });

  it("should complete end-to-end payment", async () => {
    const response = await client.fetch(
      "https://testnet-api.example.com/premium"
    );

    expect(response.ok).toBe(true);
    expect(client.lastPayment).toBeDefined();
    expect(client.lastPayment.transaction).toMatch(/^0x/);
  });
});
```

Listing 8.10: Integration testing with testnet

8.6 Error Handling

8.6.1 Client-Side Error Handling

```
import {
  T402Error,
  InsufficientFundsError,
  PaymentExpiredError,
  NetworkError
} from "@t402/core";

async function fetchWithRetry(url: string, maxRetries = 3) {
  for (let attempt = 0; attempt < maxRetries; attempt++) {
    try {
      return await client.fetch(url);
    } catch (error) {
      if (error instanceof InsufficientFundsError) {
        // Notify user to add funds
        await notifyUser("Insufficient balance", {
          required: error.required,
          available: error.available
        });
        throw error;  // Don't retry
      }

      if (error instanceof PaymentExpiredError) {
        // Retry with fresh authorization
        console.log("Payment expired, retrying...");
        continue;
```

```
      }

      if (error instanceof NetworkError) {
        // Try different network
        if (attempt < maxRetries - 1) {
          client.setPreferredNetwork(getNextNetwork());
          continue;
        }
      }

      throw error;
    }
  }
}
```

Listing 8.11: Comprehensive error handling

8.6.2 Server-Side Error Handling

```
import { T402ErrorHandler } from "@t402/express";

// Custom error handler
app.use(T402ErrorHandler({
  onVerificationFailed: (error, req, res) => {
    logger.error("Payment verification failed", {
      error: error.code,
      payer: error.payer,
      ip: req.ip
    });

    res.status(402).json({
      error: "payment_failed",
      code: error.code,
      message: error.message,
      retry: error.retryable
    });
  },

  onSettlementFailed: async (error, req, res) => {
    // Log for manual review
    await alertOps("Settlement failed", error);

    res.status(500).json({
      error: "settlement_error",
      message: "Payment processing failed",
      support: "support@example.com"
    });
  }
}));
```

Listing 8.12: Server error handling

Table 8.3: Security Best Practices

Practice	Description
Verify on Server	Always verify payments server-side
Use HTTPS	Never transmit payments over HTTP
Validate Amounts	Check payment matches requirements
Check Expiry	Reject expired authorizations
Log Transactions	Maintain audit trail
Handle Errors	Never expose internal errors

8.7 Best Practices

8.7.1 Security Checklist

8.7.2 Performance Optimization

```
import { createPaymentCache } from "@t402/express";

// Cache verified payments (idempotent by nonce)
const paymentCache = createPaymentCache({
  ttl: 300,  // 5 minutes
  maxSize: 10000
});

app.use(paymentMiddleware({
  cache: paymentCache,

  // Batch verification for high-throughput
  batchVerification: {
    enabled: true,
    maxBatchSize: 10,
    maxWaitMs: 50
  },

  // Circuit breaker for facilitator
  circuitBreaker: {
    enabled: true,
    failureThreshold: 5,
    resetTimeout: 30000
  }
}));
```

Listing 8.13: Caching and optimization

8.7.3 Monitoring Integration

```
import { T402Metrics } from "@t402/express";
import { Counter, Histogram } from "prom-client";

const metrics = new T402Metrics({
  prefix: "myapp_t402_",

  // Custom metrics
  customMetrics: {
```

```
    revenueByEndpoint: new Counter({
      name: "myapp_revenue_total",
      help: "Total revenue by endpoint",
      labelNames: ["endpoint", "network"]
    })
  },

  onPaymentComplete: (payment, req) => {
    metrics.customMetrics.revenueByEndpoint.inc({
      endpoint: req.path,
      network: payment.network
    }, Number(payment.amount) / 1e6);
  }
});

app.use(metrics.middleware());
app.get("/metrics", metrics.handler());
```

Listing 8.14: Metrics and monitoring

8.8 Migration Guide

8.8.1 From Stripe to T402

```
// Before: Stripe
app.post("/api/premium", async (req, res) => {
  const session = await stripe.checkout.sessions.create({
    payment_method_types: ["card"],
    line_items: [{ price: "price_xxx", quantity: 1 }],
    mode: "payment",
    success_url: "https://example.com/success",
    cancel_url: "https://example.com/cancel",
  });
  res.json({ url: session.url });
});

// After: T402 - No redirect, instant access
app.get("/api/premium",
  paymentMiddleware.route({
    price: "$0.01",
    network: "eip155:8453"
  }),
  (req, res) => {
    res.json({ data: "Premium content" });
  }
);
```

Listing 8.15: Migration from Stripe

Migration Benefits

- No checkout redirect flow
- Instant settlement (vs 2-3 day)
- No subscription management
- Global access without KYC

- 90%+ lower fees for micropayments

Chapter 9

Use Cases

T402 enables diverse payment scenarios across web services, AI agents, digital content, and IoT devices. This chapter explores practical applications with detailed implementation patterns.

9.1 AI Agent Payments

The emergence of autonomous AI agents creates unprecedented demand for machine-to-machine payments. Unlike humans, AI agents cannot use traditional payment methods that require manual intervention.

9.1.1 The Agent Payment Problem

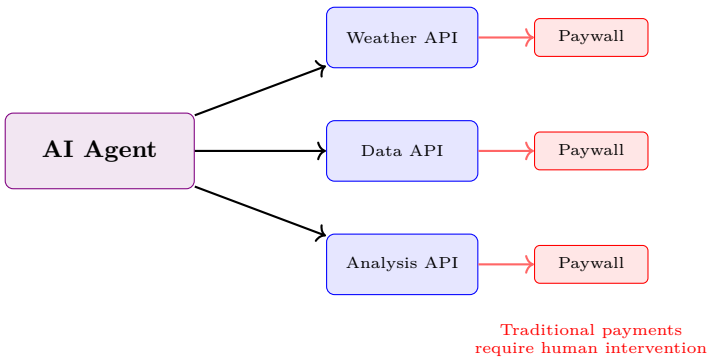

Figure 9.1: AI agents blocked by traditional paywalls

AI agents require:

Table 9.1: AI Agent Payment Requirements

Requirement	Description
Programmatic Access	No UI interaction, pure API
Budget Control	Configurable spending limits
Cost Visibility	Know price before execution
Autonomous Execution	No human approval per transaction
Failure Handling	Graceful degradation on payment failure

9.1.2 MCP Tool Marketplace

The Model Context Protocol enables AI agents to discover and use paid tools:

```
import { Agent } from "@langchain/core";
import { T402McpClient } from "@t402/mcp-client";

const mcpClient = new T402McpClient({
  signer: agentWallet,
  budget: {
    maxPerCall: "100000",     // $0.10 max per call
    maxPerHour: "1000000",    // $1.00 max per hour
    maxPerDay: "10000000"     // $10.00 max per day
  },
  onBudgetExceeded: async (cost) => {
    // Notify operator
    await alertOperator(`Budget exceeded: ${cost}`);
    return false;  // Reject payment
  }
});

// Agent discovers available tools
const tools = await mcpClient.listTools();
// Returns: [
//   { name: "stock_analysis", price: "$0.10" },
//   { name: "sentiment_analysis", price: "$0.05" },
//   { name: "market_report", price: "$0.50" }
// ]

// Agent evaluates cost vs value
const agent = new Agent({
  tools: tools.filter(t =>
    parseFloat(t.price.slice(1)) <= 0.10
  ),
  mcpClient
});

// Execute task with automatic payments
const result = await agent.run(
  "Analyze AAPL stock sentiment from recent news"
);
```

Listing 9.1: AI agent with budget management

9.1.3 Agent-to-Agent Economy

9.2 API Monetization

Transform any REST API into a paid service with per-request billing.

9.2.1 Traditional vs T402 Monetization

9.2.2 Weather Data API Example

```
import express from "express";
```

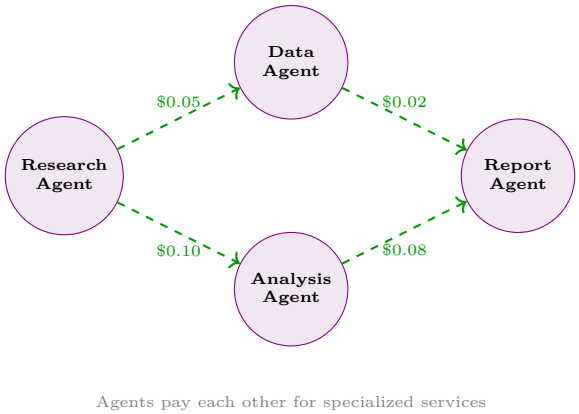

Figure 9.2: Agent-to-agent payment flows

Table 9.2: API Monetization Comparison

Aspect	Traditional (Stripe)	T402
Setup	Create plans, pricing tiers	Add middleware
User Flow	Sign up → Subscribe → Use	Use → Pay → Access
Billing	Monthly invoices	Per-request instant
Minimum	$5-10/month subscription	$0.001 per call
Global	Credit card regions	Worldwide (crypto)
Settlement	2-3 business days	Instant

```
import { paymentMiddleware } from "@t402/express";

const app = express();

// Tiered pricing based on data resolution
const pricingTiers = {
  "GET /api/weather/current": {
    price: "$0.001",
    description: "Current weather"
  },
  "GET /api/weather/forecast/hourly": {
    price: "$0.005",
    description: "48-hour hourly forecast"
  },
  "GET /api/weather/forecast/daily": {
    price: "$0.01",
    description: "14-day daily forecast"
  },
  "GET /api/weather/historical": {
    price: async (req) => {
      // Price based on date range
      const days = calculateDays(req.query.start, req.query.end);
      return `$${(days * 0.001).toFixed(4)}`;
    },
    description: "Historical weather data"
  }
};

app.use(paymentMiddleware({
  routes: pricingTiers,
  payTo: "0xWeatherServiceWallet..."
```

```
}));

// Endpoints
app.get("/api/weather/current", async (req, res) => {
  const { lat, lon } = req.query;
  const weather = await getWeather(lat, lon);
  res.json(weather);
});
```

Listing 9.2: Weather API with tiered pricing

9.2.3 Revenue Dashboard

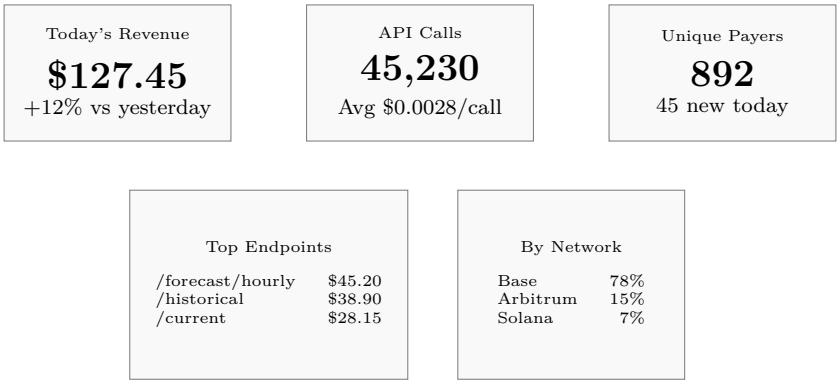

Figure 9.3: API monetization dashboard

9.3 Content Access

Enable pay-per-article, pay-per-video, and premium content access without subscriptions.

9.3.1 The Subscription Fatigue Problem

Subscription Fatigue

The average consumer subscribes to 6+ content services at $10-15/month each. Most content goes unconsumed. T402 enables pay-per-use: read one article for $0.25 instead of subscribing for $10/month.

9.3.2 News Article Paywall

```
// Next.js API route with T402
import { withPayment } from "@t402/next";

export default withPayment(
  async function handler(req, res) {
    const { slug } = req.query;
    const article = await getArticle(slug);

    res.json({
```

```
      title: article.title,
      content: article.content,
      author: article.author
    });
  },
  {
    // Dynamic pricing by article type
    price: async (req) => {
      const article = await getArticleMeta(req.query.slug);
      const prices = {
        "breaking": "0.10",
        "analysis": "0.25",
        "research": "0.50",
        "standard": "0.05"
      };
      return prices[article.type] || "0.10";
    },
    network: "eip155:8453"
  }
);
```

Listing 9.3: News paywall implementation

9.3.3 Video Streaming

```
// Video access with time-based pricing
app.get("/api/video/:id/access",
  paymentMiddleware.route({
    price: async (req) => {
      const video = await getVideoMeta(req.params.id);
      // $0.01 per minute of content
      return `$${(video.duration / 60 * 0.01).toFixed(4)}`;
    }
  }),
  async (req, res) => {
    const { payer, transaction } = req.t402Payment;

    // Generate time-limited access token
    const accessToken = await generateAccessToken({
      videoId: req.params.id,
      payer,
      transaction,
      expiresIn: "24h"
    });

    res.json({
      accessToken,
      streamUrl: `/stream/${req.params.id}?token=${accessToken}`
    });
  }
);
```

Listing 9.4: Pay-per-view video streaming

9.4 Micropayments

Enable payments below traditional minimums ($0.50) for IoT, compute, and data services.

9.4.1 Economic Viability

Table 9.3: Micropayment Economics

Payment	Stripe Fee	T402 Fee	Savings
$0.01	$0.30 (3000%)	$0.0001 (1%)	99.97%
$0.10	$0.33 (330%)	$0.001 (1%)	99.70%
$1.00	$0.33 (33%)	$0.01 (1%)	97.0%
$10.00	$0.59 (5.9%)	$0.10 (1%)	83.1%

9.4.2 IoT Data Marketplace

```
// Sensor data pricing: $0.0001 per reading
const sensorPricing = {
  temperature: "0.0001",
  humidity: "0.0001",
  air_quality: "0.0005",
  traffic: "0.001",
  energy: "0.0002"
};

app.get("/api/sensors/:type/latest",
  paymentMiddleware.route({
    price: (req) => sensorPricing[req.params.type] || "0.001"
  }),
  async (req, res) => {
    const reading = await getSensorReading(req.params.type);
    res.json(reading);
  }
);

// Bulk data: $0.05 per 1000 readings
app.get("/api/sensors/:type/bulk",
  paymentMiddleware.route({
    price: async (req) => {
      const count = Math.min(req.query.count || 100, 10000);
      const basePrice = parseFloat(sensorPricing[req.params.type]);
      return `$${(count * basePrice * 0.8).toFixed(6)}`;  // 20% bulk discount
    }
  }),
  async (req, res) => {
    const readings = await getBulkReadings(
      req.params.type,
      req.query.count
    );
    res.json(readings);
  }
);
```

Listing 9.5: IoT sensor data marketplace

9.4.3 Compute-on-Demand

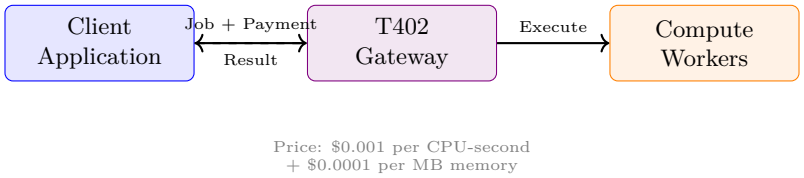

Figure 9.4: Pay-per-compute architecture

```
// Price based on estimated compute resources
app.post("/api/compute/run",
  paymentMiddleware.route({
    price: async (req) => {
      const { code, timeout, memory } = req.body;

      // Estimate cost
      const cpuSeconds = timeout / 1000;
      const memoryMB = memory / (1024 * 1024);

      const cpuCost = cpuSeconds * 0.001;    // $0.001/CPU-sec
      const memoryCost = memoryMB * 0.0001;  // $0.0001/MB

      return `$${(cpuCost + memoryCost).toFixed(6)}`;
    }
  }),
  async (req, res) => {
    const result = await executeInSandbox(req.body);
    res.json({
      result: result.output,
      metrics: {
        cpuTime: result.cpuTime,
        memoryUsed: result.memoryUsed
      }
    });
  }
);
```

Listing 9.6: Serverless compute pricing

9.5 Cross-Border Payments

Enable instant global payments without currency conversion or banking restrictions.

9.5.1 Traditional Cross-Border Challenges

9.5.2 Freelancer Platform

```
// Freelancer delivers work, client pays instantly
app.post("/api/deliverables/:id/accept",
  paymentMiddleware.route({
    price: async (req) => {
      const deliverable = await getDeliverable(req.params.id);
      return deliverable.agreedPrice;
```

Table 9.4: Cross-Border Payment Comparison

Method	Time	Fee	Access
Wire Transfer	3-5 days	$25-50	Bank account
PayPal	1-3 days	4-5%	Limited countries
Stripe	2-7 days	3.9% + $0.30	Credit card
T402	Instant	<1%	Global (crypto wallet)

```
    },
    payTo: async (req) => {
      // Pay directly to freelancer's wallet
      const deliverable = await getDeliverable(req.params.id);
      return deliverable.freelancerWallet;
    }
  }),
  async (req, res) => {
    const { transaction } = req.t402Payment;

    // Mark as paid
    await updateDeliverable(req.params.id, {
      status: "paid",
      paymentTx: transaction
    });

    // Release to client
    const deliverable = await getDeliverable(req.params.id);
    res.json({
      files: deliverable.files,
      paymentConfirmation: transaction
    });
  }
);
```

Listing 9.7: Global freelancer payments

9.6 Research Data Access

Academic and commercial research data marketplaces.

```
// Dataset pricing tiers
const datasetPricing = {
  preview: { rows: 100, price: "0" },
  sample: { rows: 1000, price: "1.00" },
  standard: { rows: 10000, price: "5.00" },
  full: { rows: -1, price: "25.00" }  // Unlimited
};

app.get("/api/datasets/:id/download",
  paymentMiddleware.route({
    price: (req) => {
      const tier = req.query.tier || "sample";
      return datasetPricing[tier]?.price || "5.00";
    },
    // Free preview tier
    skipPayment: (req) => req.query.tier === "preview"
  }),
```

```
  async (req, res) => {
    const tier = req.query.tier || "sample";
    const dataset = await getDataset(req.params.id);
    const limit = datasetPricing[tier].rows;

    const data = limit === -1
      ? dataset.data
      : dataset.data.slice(0, limit);

    res.json({
      metadata: dataset.metadata,
      data,
      tier,
      totalRows: dataset.data.length
    });
  }
);
```

Listing 9.8: Research dataset marketplace

9.7 Gaming and Virtual Goods

In-game purchases and virtual item trading.

```
// In-game item purchase
app.post("/api/game/items/purchase",
  paymentMiddleware.route({
    price: async (req) => {
      const item = await getGameItem(req.body.itemId);
      return item.price;
    }
  }),
  async (req, res) => {
    const { payer, transaction } = req.t402Payment;
    const { itemId, playerId } = req.body;

    // Grant item to player
    await grantItem(playerId, itemId, {
      purchasedBy: payer,
      transaction
    });

    res.json({
      success: true,
      item: await getPlayerInventory(playerId, itemId)
    });
  }
);

// Player-to-player trading
app.post("/api/game/trade/execute",
  paymentMiddleware.route({
    price: (req) => req.body.price,
    payTo: async (req) => {
      // Pay seller directly
      const seller = await getPlayer(req.body.sellerId);
      return seller.walletAddress;
```

```
    }
  }),
  async (req, res) => {
    // Transfer item between players
    await transferItem(
      req.body.sellerId,
      req.body.buyerId,
      req.body.itemId
    );

    res.json({ success: true });
  }
);
```

Listing 9.9: Game item marketplace

9.8 Decision Framework

Use this framework to determine if T402 is appropriate for your use case.

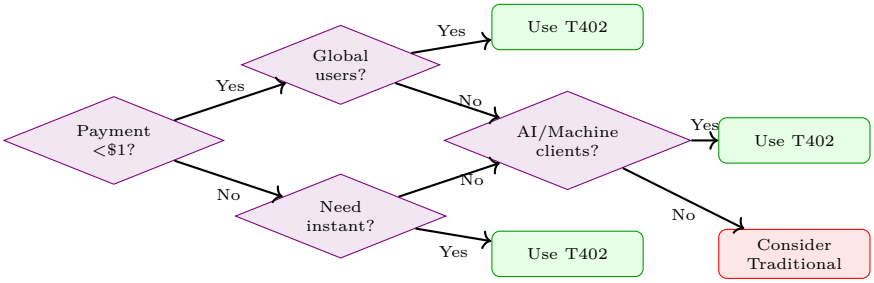

Figure 9.5: T402 decision framework

9.8.1 Ideal Use Cases

Table 9.5: T402 Fit Assessment

Use Case	T402 Fit	Key Benefit
AI Agent Tools	Excellent	Autonomous payments
API Micropayments	Excellent	Sub-cent transactions
Global Content	Excellent	No geographic limits
IoT Data	Excellent	High-volume micro-tx
Freelance Platforms	Good	Instant settlement
E-commerce	Moderate	User familiarity
Recurring Billing	Limited	Use subscriptions

When NOT to Use T402

- Users unfamiliar with crypto wallets
- Strict regulatory compliance requirements
- Recurring subscription billing (use extensions)
- Refund-heavy business models

Chapter 10

Economic Model

This chapter provides a comprehensive analysis of the economic properties, cost structures, and sustainability model of T402.

10.1 Cost Structure Overview

T402 introduces a fundamentally different cost model compared to traditional payment systems. The total cost of a payment consists of three components:

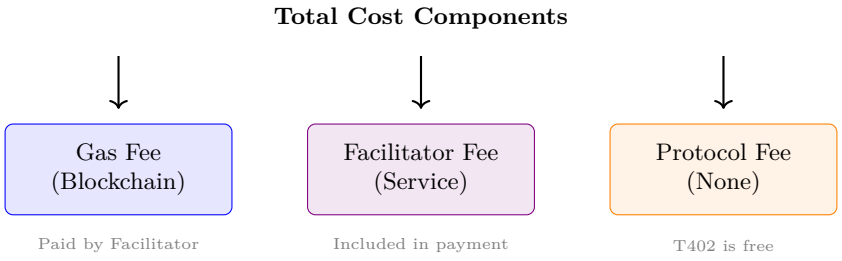

Figure 10.1: T402 cost structure

Table 10.1: Cost Component Breakdown

Component	Typical Cost	Who Pays
Gas Fee	\$0.001–0.10	Facilitator (sponsored)
Facilitator Fee	0–1%	Deducted from payment
Protocol Fee	\$0.00	N/A (open source)

Table 10.2: Payment Provider Fee Comparison

Provider	Fixed Fee	% Fee	Intl Fee	Settlement
Stripe	$0.30	2.9%	+1.5%	2 days
PayPal	$0.49	3.49%	+1.5%	1-3 days
Square	$0.30	2.6%	+1%	1-2 days
Adyen	$0.12	2.9%	+1%	3 days
T402	$0.00	0-1%	0%	Instant

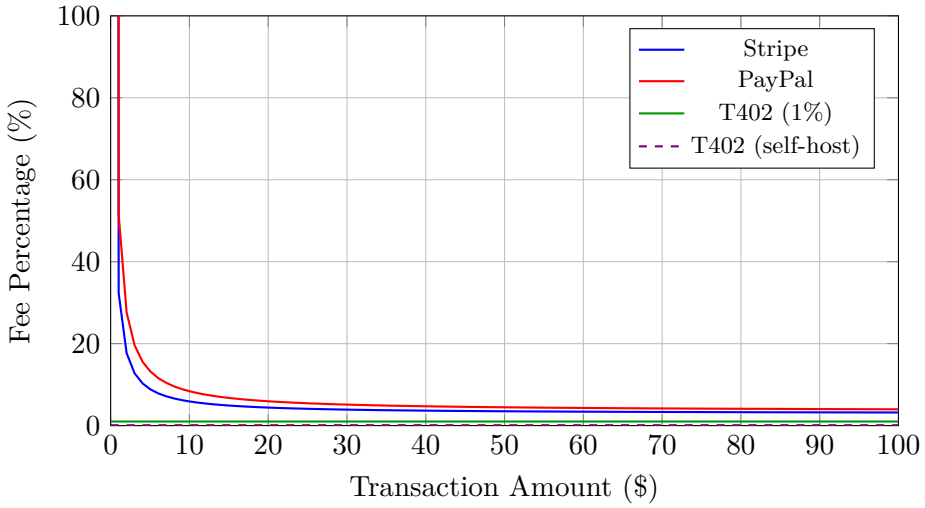

Figure 10.2: Fee percentage by transaction size

10.2 Traditional Payment Comparison

10.2.1 Fee Structure Comparison

10.2.2 Cost by Transaction Size

10.2.3 Break-Even Analysis

Micropayment Sweet Spot

T402 provides the greatest advantage for payments under $10, where traditional fixed fees dominate. For a $0.10 payment, T402 is 300x more cost-effective than Stripe.

10.3 Network Economics

Different blockchain networks offer varying cost profiles:

Table 10.3: Break-Even: T402 vs Stripe

Amount	Stripe Fee	T402 Fee	Savings
$0.01	$0.30 (3000%)	$0.0001 (1%)	99.97%
$0.10	$0.30 (303%)	$0.001 (1%)	99.67%
$1.00	$0.33 (33%)	$0.01 (1%)	96.97%
$5.00	$0.45 (8.9%)	$0.05 (1%)	88.89%
$10.00	$0.59 (5.9%)	$0.10 (1%)	83.05%
$50.00	$1.75 (3.5%)	$0.50 (1%)	71.43%
$100.00	$3.20 (3.2%)	$1.00 (1%)	68.75%

Table 10.4: Network Gas Fee Comparison (EIP-3009 Transfer)

Network	Gas Units	Gas Price	USD Cost
Ethereum L1	65,000	30 gwei	$4.50–15.00
Arbitrum One	65,000	0.1 gwei	$0.01–0.05
Base	65,000	0.001 gwei	$0.001–0.01
Optimism	65,000	0.001 gwei	$0.001–0.01
Solana	5,000 CU	–	$0.0001
TON	–	–	$0.01–0.05
TRON	30,000	–	$0.10–0.50

10.3.1 Gas Fee Comparison

10.3.2 Network Selection Matrix

10.4 Facilitator Economics

The Facilitator service operates as the economic bridge between users and blockchain networks.

10.4.1 Revenue Model

Table 10.5: Facilitator Revenue Sources

Source	Rate	Description
Settlement Fee	0–1%	Per-transaction fee
Priority Processing	Fixed	Guaranteed fast settlement
Enterprise SLA	Monthly	Dedicated infrastructure
White-label	License	Self-hosted deployment

Network	Cost	Speed	Security	Best For
Ethereum	High	12s	Highest	>$100 tx
Base	Very Low	2s	High	Micropay
Arbitrum	Low	<1s	High	DeFi
Solana	Lowest	400ms	Medium	High-freq

Figure 10.3: Network selection matrix

Table 10.6: Facilitator Operating Costs

Category	Monthly	Notes
Infrastructure	$500–2,000	Servers, databases
RPC Services	$200–1,000	Alchemy, QuickNode
Gas Reserves	Variable	Based on volume
Monitoring	$100–500	Grafana, alerts
Security	$200–1,000	Audits, HSM

10.4.2 Cost Structure

10.4.3 Break-Even Volume

10.5 Token Economics

10.5.1 USDT vs USDC Comparison

10.5.2 USDT0 LayerZero Economics

USDT0 provides native cross-chain transfers via LayerZero:

- **No Bridge Fees**: Native OFT transfer vs bridge fees
- **Unified Liquidity**: Same token across all chains
- **Fast Settlement**: Minutes vs hours for bridges
- **Lower Risk**: No bridge exploits possible

10.6 Volume Economics

10.6.1 Economies of Scale

10.6.2 Batch Settlement Optimization

High-volume operators can reduce costs through batching:

```
// Configure batch settlement
const config = {
  batchSize: 100,           // Settle 100 tx per batch
```

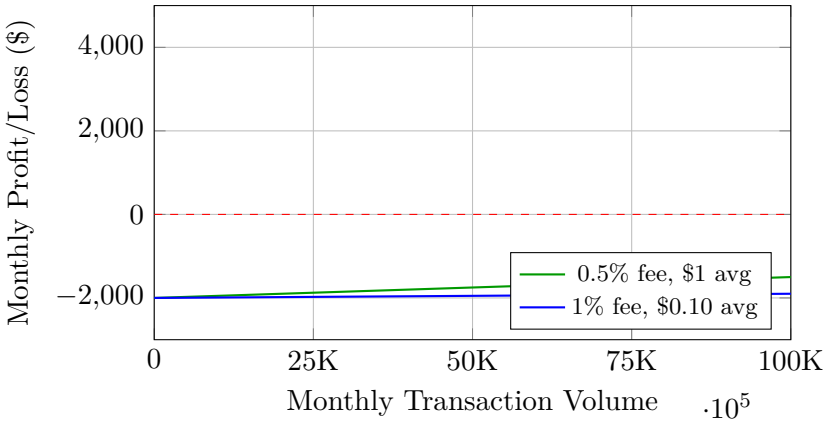

Figure 10.4: Facilitator break-even analysis

Table 10.7: Stablecoin Comparison

Property	USDT	USDC	USDT0
Market Cap	$120B+	$30B+	–
Daily Volume	$50B+	$5B+	–
Chains Supported	15+	10+	20+ (OFT)
EIP-3009 Support	Yes	Yes	Yes
Cross-chain Native	No	No	Yes

```
  maxWaitTime: 60000,        // Max 60 seconds
  minBatchValue: "100000",   // Min $0.10 per batch
};

// Cost savings:
// - Single tx: $0.01 gas
// - Batch of 100: $0.02 gas total
// - Savings: 98% on gas costs
```

Listing 10.1: Batch settlement for cost optimization

10.7 ROI Analysis

10.7.1 API Provider Example

Traditional Limitation

With Stripe's $0.30 minimum fee, the weather API would need to charge at least $0.50 per call to be viable—500x higher than market rates.

Table 10.8: Cost Optimization by Volume

Monthly Volume	Avg Gas/Tx	Effective Rate	Savings vs Stripe
1,000 tx	$0.01	2.0%	50%
10,000 tx	$0.005	1.5%	60%
100,000 tx	$0.002	1.2%	65%
1,000,000 tx	$0.001	1.0%	70%

Table 10.9: ROI: Weather API Provider

Metric	Stripe	T402
Monthly API Calls	1,000,000	1,000,000
Price per Call	$0.001	$0.001
Gross Revenue	$1,000	$1,000
Payment Fees	$300+	$10
Net Revenue	$700	$990
Effective Margin	**70%**	**99%**

10.7.2 Content Platform Example

10.8 Market Opportunity

10.8.1 Total Addressable Market

10.8.2 Growth Segments

10.9 Economic Sustainability

10.9.1 Long-term Viability

T402's economic sustainability is ensured by:

1. **Open Protocol**: No protocol fees, community-driven
2. **Facilitator Competition**: Multiple providers ensure competitive fees
3. **Chain Agnosticism**: Not dependent on single network
4. **Stablecoin Diversity**: Support for multiple stablecoins

10.9.2 Risk Mitigation

Table 10.10: ROI: News Platform Pay-Per-Article

Metric	Subscription	T402 PPV
Monthly Users	10,000	50,000
Conversion Rate	2%	15%
Paying Users	200	7,500
Avg Revenue/User	$10.00	$0.75
Gross Revenue	$2,000	$5,625
Payment Fees	$118	$56
Net Revenue	**$1,882**	**$5,569**

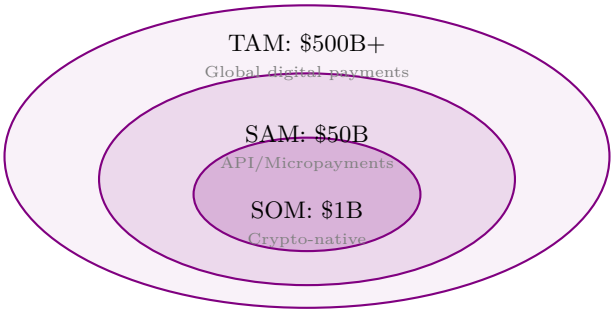

Figure 10.5: Market opportunity sizing

Table 10.11: High-Growth Market Segments

Segment	2024 Size	CAGR
AI Agent Services	$2B	85%
API Economy	$15B	25%
Creator Economy	$100B	20%
IoT Payments	$5B	40%
Cross-border Freelance	$50B	15%

Table 10.12: Economic Risk Factors

Risk	Impact	Mitigation
L2 Fee Spikes	Increased settlement cost	Multi-chain support
Stablecoin Depeg	Payment value mismatch	Multi-stablecoin support
Facilitator Failure	Service disruption	Self-hosting option
Regulatory Change	Operational restrictions	Geographic diversity

Chapter 11

Future Work

This chapter outlines planned enhancements, research directions, and long-term vision for T402. The protocol's modular architecture enables incremental improvements while maintaining backward compatibility.

11.1 Development Roadmap

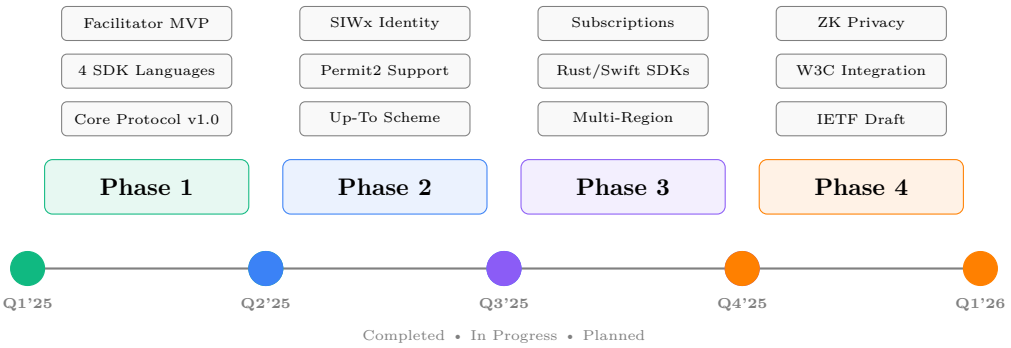

Figure 11.1: T402 Protocol Development Roadmap

Figure 11.1 illustrates the four-phase development plan for T402:

Phase 1 – Foundation (Completed) Core protocol specification, SDK implementations in TypeScript, Python, Go, and Java, plus the Facilitator service MVP.

Phase 2 – Enhancement (Current) Protocol extensions including Up-To metered billing, Permit2 integration for improved gas efficiency, and Sign-In-With-X for identity.

Phase 3 – Scale Infrastructure scaling with multi-region deployment, new SDK platforms (Rust, Swift), and subscription billing support.

Phase 4 – Standardize Industry standardization through IETF, W3C integration, and advanced privacy features using zero-knowledge proofs.

11.2 Protocol Enhancements

11.2.1 Up-To Scheme

The "up-to" scheme enables metered, usage-based billing where clients authorize a maximum amount upfront, and providers settle only the actual usage. This pattern is essential for:

- **Variable-cost APIs**: AI inference, cloud compute, data processing
- **Streaming services**: Real-time data feeds, video streaming
- **Metered resources**: Storage, bandwidth, API calls

11.2.1.1 Authorization Structure

```json
{
  "scheme": "up-to",
  "maxAmount": "10.00",
  "asset": "usdt",
  "network": "eip155:1",
  "validUntil": "2025-03-01T00:00:00Z",
  "nonce": "abc123",
  "authorization": {
    "type": "eip3009",
    "signature": "0x...",
    "validAfter": 1704067200,
    "validBefore": 1706745600
  },
  "meteringEndpoint": "https://api.example.com/usage"
}
```

Listing 11.1: Up-To Authorization Object

11.2.1.2 Settlement Flow

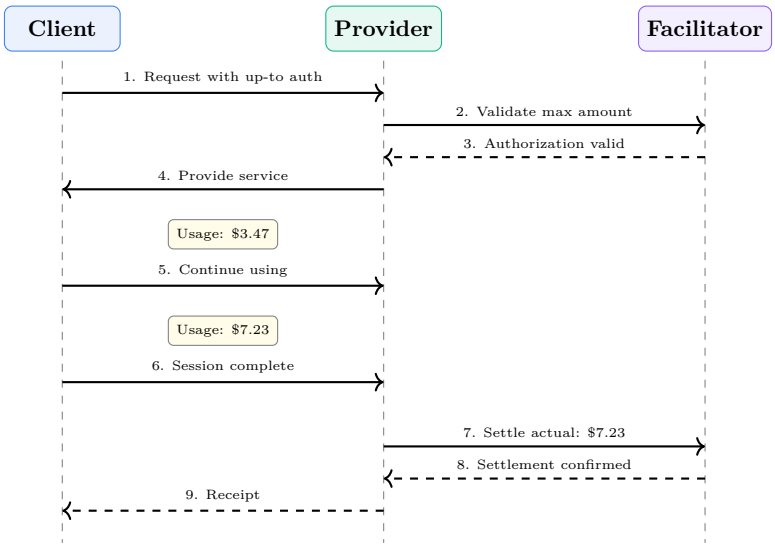

Figure 11.2: Up-To Scheme Settlement Flow

11.2.1.3 Implementation via EIP-2612 Permit

For EVM networks, the up-to scheme leverages EIP-2612 permits with batch settlement:

```solidity
contract UpToSettlement {
    struct Session {
        address client;
        uint256 maxAmount;
```

```
        uint256 usedAmount;
        uint256 deadline;
        bool settled;
    }

    mapping(bytes32 => Session) public sessions;
    IERC20Permit public usdt;

    function initSession(
        address client,
        uint256 maxAmount,
        uint256 deadline,
        uint8 v, bytes32 r, bytes32 s
    ) external {
        // Permit for max amount approval
        usdt.permit(client, address(this), maxAmount, deadline, v, r, s)
            ;

        bytes32 sessionId = keccak256(abi.encodePacked(
            client, maxAmount, deadline, block.timestamp
        ));

        sessions[sessionId] = Session({
            client: client,
            maxAmount: maxAmount,
            usedAmount: 0,
            deadline: deadline,
            settled: false
        });

        emit SessionCreated(sessionId, client, maxAmount);
    }

    function settle(
        bytes32 sessionId,
        uint256 actualAmount
    ) external onlyProvider {
        Session storage session = sessions[sessionId];
        require(!session.settled, "Already settled");
        require(actualAmount <= session.maxAmount, "Exceeds max");
        require(block.timestamp <= session.deadline, "Expired");

        session.usedAmount = actualAmount;
        session.settled = true;

        // Transfer only actual amount used
        usdt.transferFrom(session.client, msg.sender, actualAmount);

        emit SessionSettled(sessionId, actualAmount);
    }
}
```

Listing 11.2: Up-To Settlement Contract

11.2.2 Permit2 Integration

Uniswap's Permit2 contract provides significant improvements over traditional token approvals:

Table 11.1: Permit2 Advantages

Feature	Traditional	Permit2
Approval scope	Per-contract	Universal
Gas for approval	~46,000	One-time
Expiration support	No	Yes
Nonce management	Per-token	Unified
Batch transfers	No	Yes

11.2.2.1 Permit2 Payment Flow

```typescript
import { Permit2, SignatureTransfer } from '@uniswap/permit2-sdk';

async function createPermit2Payment(
  amount: bigint,
  recipient: string,
  deadline: number
): Promise<T402PaymentHeader> {
  const permit: SignatureTransfer.PermitTransferFrom = {
    permitted: {
      token: USDT_ADDRESS,
      amount: amount,
    },
    spender: FACILITATOR_ADDRESS,
    nonce: await getNextNonce(),
    deadline: deadline,
  };

  const signature = await wallet._signTypedData(
    SignatureTransfer.DOMAIN(chainId),
    SignatureTransfer.PERMIT_TRANSFER_FROM_TYPES,
    permit
  );

  return {
    scheme: 'permit2',
    payload: encodeBase64({
      permit,
      signature,
      transferDetails: {
        to: recipient,
        requestedAmount: amount,
      }
    })
  };
}
```

Listing 11.3: Permit2 T402 Integration

11.2.3 Sign-In-With-X (SIWx)

CAIP-122 defines a chain-agnostic authentication standard enabling wallet-based identity:

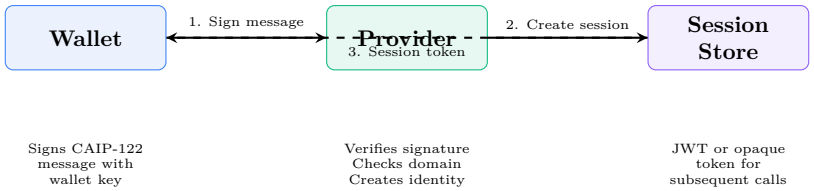

Figure 11.3: SIWx Authentication Flow

11.2.3.1 SIWx Message Format

```
{
  "domain": "api.example.com",
  "address": "0x1234...5678",
  "statement": "Sign in to Example API",
  "uri": "https://api.example.com/auth",
  "version": "1",
  "chainId": "eip155:1",
  "nonce": "32891756",
  "issuedAt": "2025-01-15T12:00:00Z",
  "expirationTime": "2025-01-15T13:00:00Z",
  "resources": [
    "https://api.example.com/weather/*",
    "https://api.example.com/news/*"
  ]
}
```

Listing 11.4: CAIP-122 Sign-In Message

11.2.3.2 Integration with T402

SIWx complements T402 by enabling:

- **Returning customer recognition**: Skip payment for pre-funded accounts
- **Subscription verification**: Prove active subscription status
- **Credit balance**: Maintain per-user credit balances
- **Rate limiting**: Apply per-wallet rate limits

```
async function authMiddleware(req: Request): Promise<AuthResult> {
  // Check for session token
  const sessionToken = req.headers.get('Authorization');
  if (sessionToken) {
    const session = await verifySession(sessionToken);
    if (session?.creditBalance > 0) {
      return { type: 'credit', wallet: session.wallet };
    }
  }

  // Check for T402 payment
  const paymentHeader = req.headers.get('X-Payment');
  if (paymentHeader) {
    return { type: 'payment', payment: parsePayment(paymentHeader) };
  }

  // Check for SIWx authentication
```

```
  const siwxHeader = req.headers.get('X-SIWx');
  if (siwxHeader) {
    const identity = await verifySIWx(siwxHeader);
    return { type: 'identity', wallet: identity.address };
  }

  // Require payment
  throw new PaymentRequiredError();
}
```

Listing 11.5: SIWx + T402 Middleware

11.2.4 Subscription Billing

Recurring payments present unique challenges in the blockchain context. The T402 subscription extension enables:

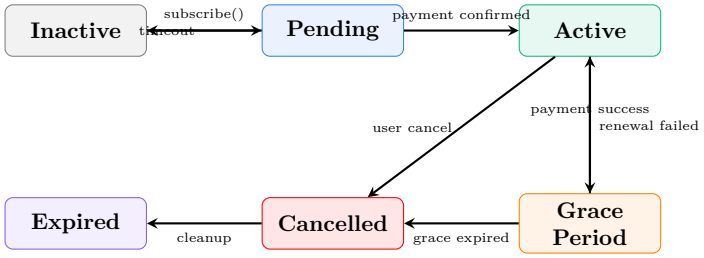

Figure 11.4: Subscription State Machine

11.2.4.1 Subscription Models

Table 11.2: Supported Subscription Types

Type	Billing	Use Case
Fixed	Monthly/Annual	SaaS subscriptions, premium tiers
Usage-capped	Per-period limit	API plans with quota
Hybrid	Base + overage	Enterprise APIs with burst capacity
Prepaid	Credit-based	Gaming, streaming credits

11.2.4.2 Implementation Approaches

1. **Pull-based (Permit2)**: Provider initiates monthly charges using stored permit
2. **Push-based (Streaming)**: Superfluid-style continuous payment streams
3. **Escrow-based**: Prepaid escrow with periodic release

11.2.5 Refund Mechanism

Unlike traditional payment systems, blockchain payments are irreversible. The refund mechanism addresses legitimate refund scenarios:

```
{
  "originalPayment": {
    "transactionHash": "0xabc...",
    "amount": "5.00",
    "timestamp": "2025-01-10T15:30:00Z"
  },
  "refundRequest": {
    "reason": "service_unavailable",
    "requestedAmount": "5.00",
    "evidence": {
      "type": "service_log",
      "data": "Error 503 at 15:30:05"
    }
  },
  "refundDestination": {
    "network": "eip155:1",
    "address": "0x1234...5678"
  }
}
```

Listing 11.6: Refund Request Schema

11.2.5.1 Refund Policies

1. **Automatic**: Service failures trigger immediate refund
2. **Dispute-based**: Human review for contested charges
3. **Partial**: Pro-rated refunds for partial service
4. **Credit**: Refund to account credit instead of on-chain

11.3 New Platforms

11.3.1 Rust SDK

The Rust SDK targets high-performance applications and WebAssembly deployments:

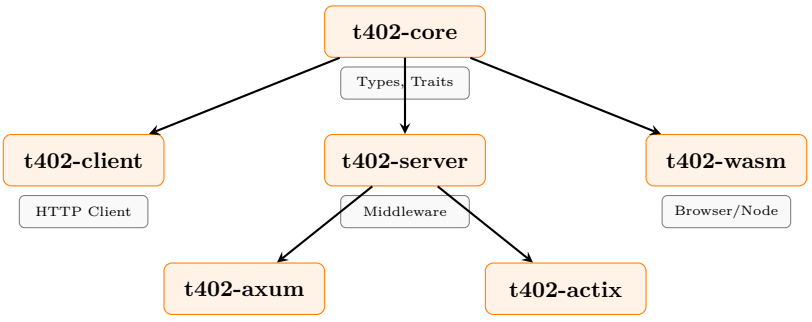

Figure 11.5: Rust SDK Architecture

11.3.1.1 Key Features

```
// Server middleware (Axum)
use t402_axum::T402Layer;

```

```rust
let app = Router::new()
    .route("/api/data", get(handler))
    .layer(T402Layer::new(config));

// Client
use t402_client::T402Client;

let client = T402Client::new(wallet);
let response = client
    .get("https://api.example.com/weather")
    .with_payment(usdt("0.01"))
    .send()
    .await?;
```

Listing 11.7: Rust SDK Usage Example (Rust syntax)

11.3.1.2 WebAssembly Support

The Rust SDK compiles to WebAssembly for:

- Browser-based T402 clients
- Edge computing (Cloudflare Workers, Deno Deploy)
- Node.js native addon performance

11.3.2 Swift SDK

Native iOS and macOS support with modern Swift features:

```swift
// Swift client usage
import T402

let client = T402Client(wallet: wallet)

let weather = try await client
    .request("https://api.example.com/weather")
    .pay(.usdt(0.01, network: .ethereum))
    .decode(WeatherResponse.self)

// SwiftUI integration
struct PaidContentView: View {
    @StateObject var payment = T402Payment()

    var body: some View {
        PaidContent(payment: payment) {
            ArticleView(article: article)
        } paywall: {
            PaywallView(amount: 0.10, asset: .usdt)
        }
    }
}
```

Listing 11.8: Swift SDK Example (Swift syntax)

11.3.2.1 Platform Integration

- **WalletConnect**: Native mobile wallet integration

- **Apple Pay Bridge**: Fiat-to-crypto via Apple Pay
- **Keychain**: Secure key storage
- **App Clips**: Lightweight payment flows

11.3.3 Kotlin SDK

Android-native SDK with Kotlin coroutines:

```kotlin
// Kotlin Android usage
val client = T402Client.create(wallet)

val response = client.get("https://api.example.com/data")
    .pay(USDT.amount("0.01").on(Network.POLYGON))
    .await()

// Jetpack Compose integration
@Composable
fun PaidScreen(url: String) {
    val payment = rememberT402Payment()

    T402Gate(
        payment = payment,
        amount = 0.10.usdt(),
        content = { PremiumContent() },
        paywall = { PaywallDialog(onPay = payment::initiate) }
    )
}
```

Listing 11.9: Kotlin SDK Example

11.3.4 C++ SDK

Embedded systems and IoT devices require a lightweight C++ implementation:

- **Minimal dependencies**: Only libcurl and OpenSSL
- **No heap allocation**: Static memory for constrained devices
- **FreeRTOS support**: IoT device integration
- **Arduino library**: Maker-friendly API

11.4 Infrastructure Scaling

11.4.1 Multi-Region Architecture

11.4.1.1 Regional Deployment Goals

11.4.2 Horizontal Scaling

- **Stateless API**: All state in distributed cache (Redis Cluster)
- **Auto-scaling**: Kubernetes HPA based on request rate
- **Connection pooling**: Shared RPC connections to blockchain nodes
- **Read replicas**: Database read scaling for verification queries

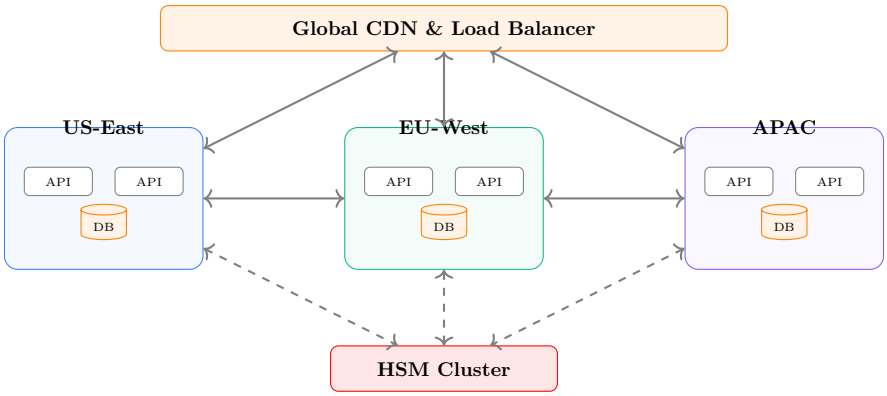

Figure 11.6: Multi-Region Facilitator Architecture

Table 11.3: Regional Deployment Targets

Region	Location	Latency Target	Status
US-East	Virginia	<50ms	Active
US-West	Oregon	<50ms	Planned
EU-West	Frankfurt	<50ms	Planned
APAC-East	Tokyo	<100ms	Planned
APAC-South	Singapore	<100ms	Planned

11.4.2.1 Capacity Planning

Table 11.4: Scaling Milestones

Tx/Day	API Pods	DB Size	Cache RAM
10,000	3	10 GB	2 GB
100,000	10	100 GB	8 GB
1,000,000	50	1 TB	32 GB
10,000,000	200	10 TB	128 GB

11.4.3 Security Enhancements

11.4.3.1 Hardware Security Modules (HSM)

Production hot wallets will migrate to HSM-backed key management:

- **AWS CloudHSM**: FIPS 140-2 Level 3 compliance
- **Key rotation**: Automated monthly rotation
- **Multi-signature**: 2-of-3 threshold for high-value settlements
- **Audit logging**: All key operations logged to immutable store

11.4.3.2 Rate Limiting Evolution

```
rateLimiting:
  tiers:
    anonymous:
```

```
      rps: 10
      burst: 20
      dailyLimit: 1000
    authenticated:
      rps: 100
      burst: 200
      dailyLimit: 50000
    enterprise:
      rps: 1000
      burst: 5000
      dailyLimit: unlimited

  adaptive:
    enabled: true
    cpuThreshold: 80%
    memoryThreshold: 85%
    reductionFactor: 0.5

  ddosProtection:
    enabled: true
    provider: cloudflare
    challengeThreshold: 1000
```

Listing 11.10: Advanced Rate Limiting Configuration

11.5 Standardization

11.5.1 IETF Standardization

The T402 specification will be proposed as an Internet-Draft:

11.5.1.1 Proposed RFC Structure

1. **RFC 9XXX: HTTP Payment Required (402) Protocol**
   - Updates RFC 7231 with 402 semantics
   - Defines `X-Payment` and `X-Payment-Response` headers
   - Specifies scheme registry mechanism
2. **RFC 9XXY: Cryptocurrency Payment Schemes for HTTP 402**
   - Defines `exact`, `up-to`, `permit2` schemes
   - Network identifier format (CAIP-2)
   - Authorization encoding standards

11.5.1.2 Timeline

1. **Q3 2025**: Draft submission to IETF HTTP Working Group
2. **Q4 2025**: Working group adoption
3. **Q2 2026**: Last Call
4. **Q4 2026**: RFC publication

11.5.2 W3C Web Payments Integration

Integration with W3C Payment Request API:

```javascript
// Browser payment flow
const paymentRequest = new PaymentRequest(
  [
    {
      supportedMethods: 'https://t402.io/payment',
      data: {
        supportedNetworks: ['eip155:1', 'eip155:137'],
        supportedAssets: ['usdt', 'usdc'],
      }
    }
  ],
  {
    total: {
      label: 'API Access',
      amount: { currency: 'USDT', value: '0.10' }
    }
  }
);

const response = await paymentRequest.show();
const paymentHeader = response.details.paymentHeader;
// Use paymentHeader in X-Payment header
```

Listing 11.11: W3C Payment Request Integration

11.5.3 CAIP Alignment

Full alignment with Chain Agnostic Improvement Proposals:

Table 11.5: CAIP Alignment Status

CAIP	Description	Status
CAIP-2	Blockchain ID Specification	Implemented
CAIP-10	Account ID Specification	Implemented
CAIP-19	Asset Type and ID	Planned
CAIP-122	Sign-In-With-X	Planned
CAIP-25	JSON-RPC Provider Request	Planned

11.6 Research Directions

11.6.1 Privacy-Preserving Payments

11.6.1.1 Zero-Knowledge Proofs

Future protocol versions may incorporate ZK proofs for:

- **Payment amount privacy**: Prove payment $\geq$ threshold without revealing exact amount
- **Identity privacy**: Prove wallet membership in allowed set (ZK set membership)
- **Compliance proofs**: Prove KYC completion without revealing identity

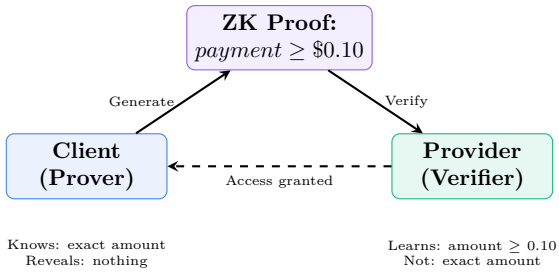

Figure 11.7: Zero-Knowledge Payment Verification

11.6.2 Cross-Chain Atomic Payments

Research into atomic cross-chain payments for multi-chain scenarios:

- **Hash Time-Locked Contracts (HTLC)**: Trustless cross-chain swaps
- **Layered settlements**: L2-to-L2 transfers via bridges
- **Intent-based payments**: Express payment intent, let solvers optimize execution

11.6.3 AI Agent Integration Research

11.6.3.1 Autonomous Agent Economics

As AI agents become economic actors, research focuses on:

1. **Agent identity**: How do agents prove identity across services?
2. **Budget management**: Safe delegation of spending authority
3. **Reputation systems**: Trust scoring for agent-to-agent commerce
4. **Dispute resolution**: Automated arbitration for agent disputes

11.6.3.2 Multi-Agent Payment Protocols

```
{
  "taskId": "research-task-123",
  "coordinator": "agent:researcher-001",
  "budget": {
    "total": "50.00",
    "asset": "usdt",
    "network": "eip155:137"
  },
  "subtasks": [
    {
      "agent": "agent:data-collector",
      "allocation": "20.00",
      "services": ["https://api.weather.com/*"]
    },
    {
      "agent": "agent:analyzer",
      "allocation": "25.00",
      "services": ["https://api.openai.com/*"]
    }
  ],
  "settlement": "atomic",
```

```
  "deadline": "2025-01-20T00:00:00Z"
}
```

Listing 11.12: Multi-Agent Payment Coordination

11.6.4 Streaming Payments

Research into continuous payment streams for real-time services:

- **Superfluid integration**: Per-second payment streams
- **Dynamic rate adjustment**: Auto-adjust stream rate based on usage
- **Stream aggregation**: Combine multiple micro-streams for efficiency

11.7 Community and Ecosystem

11.7.1 Developer Ecosystem

- **SDK contributions**: Open-source community SDKs (Elixir, Scala, PHP)
- **Integration templates**: Ready-to-deploy templates for popular frameworks
- **Developer grants**: Funding for innovative T402 applications
- **Hackathons**: Quarterly hackathons with prizes

11.7.2 Governance

Long-term protocol governance considerations:

1. **Technical Steering Committee**: Protocol evolution decisions
2. **Specification process**: RFC-style proposal and review
3. **Reference implementation**: Canonical SDK implementations
4. **Certification program**: Verified compliant implementations

11.7.3 Interoperability Initiatives

- **Payment gateway bridges**: Connect T402 to traditional payment rails
- **Cross-protocol payments**: Interoperability with Lightning, Bolt12
- **Identity federation**: Accept multiple identity standards (SIWx, Verifiable Credentials)

11.8 Conclusion

The T402 protocol has established a foundation for HTTP-native cryptocurrency payments. The roadmap outlined in this chapter charts a path toward:

- **Enhanced flexibility**: Up-To metering, subscriptions, and refunds
- **Broader reach**: New SDKs for Rust, Swift, Kotlin, and C++
- **Global scale**: Multi-region infrastructure with enterprise-grade security
- **Industry adoption**: IETF and W3C standardization
- **Advanced capabilities**: Privacy, cross-chain, and AI agent integration

The protocol's modular architecture ensures these enhancements can be adopted incrementally, maintaining backward compatibility while enabling continuous innovation in the intersection of web services and blockchain payments.

Contributing to T402

The T402 protocol is open source. Contributions are welcome:

- **GitHub**: https://github.com/t402-io/t402
- **Documentation**: https://docs.t402.io
- **Discussions**: https://github.com/t402-io/t402/discussions

Chapter 12

Conclusion

12.1 Summary

This whitepaper has presented T402, an HTTP-native payment protocol for stablecoin transactions. The protocol addresses fundamental limitations of traditional payment systems while providing a developer-friendly integration path.

12.1.1 Key Contributions

1. **HTTP 402 Activation**: First practical implementation of the long-dormant HTTP 402 status code for payment signaling
2. **Transport Agnosticism**: Clean separation between payment logic and transport mechanisms (HTTP, MCP, A2A)
3. **Chain Agnosticism**: Unified interface across EVM, Solana, TON, and TRON blockchains
4. **Gasless Payments**: Leveraging EIP-3009 and ERC-4337 for zero-gas user experience
5. **Trust Minimization**: Facilitators cannot redirect funds—payments flow directly to merchants
6. **AI-Native Design**: First-class support for autonomous agent payments via MCP

12.1.2 Ecosystem Impact

T402 enables new business models:

- **API Economy**: Per-request monetization without subscription friction
- **Content Creators**: Article-level payments vs subscription fatigue
- **AI Agents**: Autonomous resource acquisition
- **IoT**: Sub-cent data transactions

12.2 Call to Action

We invite the community to:

- **Build**: Integrate T402 into your applications
- **Contribute**: Submit improvements via GitHub
- **Extend**: Develop new schemes and transports
- **Deploy**: Run your own Facilitator instance

12.3 Resources

Website https://t402.io
Documentation https://docs.t402.io
GitHub https://github.com/t402-io/t402
Facilitator API https://facilitator.t402.io

The future of payments is HTTP-native.

Complete JSON Schemas

This appendix provides complete JSON Schema definitions for all T402 data structures.

.1 PaymentRequired Schema

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "type": "object",
  "required": ["t402Version", "resource", "accepts"],
  "properties": {
    "t402Version": {
      "type": "integer",
      "const": 2
    },
    "error": {
      "type": "string"
    },
    "resource": {
      "$ref": "#/$defs/ResourceInfo"
    },
    "accepts": {
      "type": "array",
      "items": {
        "$ref": "#/$defs/PaymentRequirements"
      },
      "minItems": 1
    },
    "extensions": {
      "type": "object"
    }
  },
  "$defs": {
    "ResourceInfo": {
      "type": "object",
      "required": ["url"],
      "properties": {
        "url": { "type": "string", "format": "uri" },
        "description": { "type": "string" },
        "mimeType": { "type": "string" }
      }
    },
    "PaymentRequirements": {
      "type": "object",
      "required": ["scheme", "network", "amount", "asset", "payTo", "
         maxTimeoutSeconds"],
      "properties": {
```

```
        "scheme": { "type": "string" },
        "network": { "type": "string" },
        "amount": { "type": "string" },
        "asset": { "type": "string" },
        "payTo": { "type": "string" },
        "maxTimeoutSeconds": { "type": "integer" },
        "extra": { "type": "object" }
      }
    }
  }
}
```

Listing 1: PaymentRequired JSON Schema

.2 PaymentPayload Schema

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "type": "object",
  "required": ["t402Version", "accepted", "payload"],
  "properties": {
    "t402Version": {
      "type": "integer",
      "const": 2
    },
    "resource": {
      "$ref": "#/$defs/ResourceInfo"
    },
    "accepted": {
      "$ref": "#/$defs/PaymentRequirements"
    },
    "payload": {
      "type": "object"
    },
    "extensions": {
      "type": "object"
    }
  }
}
```

Listing 2: PaymentPayload JSON Schema

.3 SettlementResponse Schema

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "type": "object",
  "required": ["success", "transaction", "network"],
  "properties": {
    "success": { "type": "boolean" },
    "errorReason": { "type": "string" },
    "payer": { "type": "string" },
    "transaction": { "type": "string" },
    "network": { "type": "string" }
```

```
  }
}
```

Listing 3: SettlementResponse JSON Schema

.4 VerifyResponse Schema

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "type": "object",
  "required": ["isValid"],
  "properties": {
    "isValid": { "type": "boolean" },
    "invalidReason": { "type": "string" },
    "payer": { "type": "string" }
  }
}
```

Listing 4: VerifyResponse JSON Schema

Supported Networks

This appendix lists all supported networks and token addresses.

.5 EVM Networks

Table 1: Supported EVM Networks

Network	CAIP-2 ID	USDT0 Address
Ethereum	eip155:1	0x6C96dE32CEa08842dcc...Fa41dee
Arbitrum One	eip155:42161	0xFd086bC7CD5C481DCC...FCbb9
Base	eip155:8453	0x0200C29006150606B6...470c1
Optimism	eip155:10	-
Ink	eip155:57073	0x0200C29006150606B6...470c1
Berachain	eip155:80094	0x779Ded0c9e1022225f...13736
Unichain	eip155:130	0x588ce4F028D8e7B53B...75518

.6 Non-EVM Networks

Table 2: Non-EVM Networks

Network	CAIP-2 ID	Token Address
Solana	solana:5eykt...Kvdp	EPjFWdd5Aufq...Dt1v
TON	ton:-239	EQCxE6mUtQJK...sDs
TRON	tron:0x2b6653dc	TR7NHqjeKQxG...Lj6t

.7 Facilitator Wallet Addresses

Table 3: Official Facilitator Addresses

Chain Type	Address
EVM (all)	0xC88f67e776f16DcFBf42e6bDda1B82604448899B
Solana	8GGtWHRQ1wz5gDKE2KXZLktqzcfV1CBqSbeUZjA7hoWL
TRON	TT1MqNNj2k5qdGA6nrrCodW6oyHbbAreQ5

Error Code Reference

This appendix provides a complete reference of T402 error codes.

.8 Payment Errors

Table 4: Payment Error Codes

Code	Description	Resolution
`insufficient_funds`	Payer lacks sufficient balance	Fund wallet
`invalid_signature`	Signature verification failed	Re-sign with correct key
`invalid_amount`	Amount below required	Increase amount
`invalid_recipient`	Recipient address mismatch	Use correct payTo
`expired_authorization`	Time window passed	Create new authorization
`nonce_already_used`	Nonce used in prior tx	Generate new nonce

.9 Protocol Errors

Table 5: Protocol Error Codes

Code	Description	Resolution
`invalid_network`	Network not supported	Use supported network
`invalid_scheme`	Scheme not supported	Use supported scheme
`invalid_t402_version`	Version not supported	Upgrade to v2
`invalid_payload`	Malformed payload	Validate format
`invalid_payment_req`	Invalid requirements	Validate requirements

.10 Settlement Errors

Table 6: Settlement Error Codes

Code	Description	Resolution
`simulation_failed`	Transaction simulation failed	Check balance/params
`settlement_failed`	On-chain settlement failed	Retry or contact support

Table 6 – continued

Code	Description	Resolution
`unexpected_verify_error`	Unexpected verification error	Check logs, retry
`unexpected_settle_error`	Unexpected settlement error	Check logs, retry

Glossary

A2A (Agent-to-Agent)
: Protocol for direct communication between AI agents, enabling autonomous agent payments.

ATA (Associated Token Account)
: Solana account derived from a wallet address and token mint, used for holding SPL tokens.

CAIP-2
: Chain Agnostic Improvement Proposal 2, a standard for blockchain network identifiers (e.g., `eip155:8453`).

Client
: Any application or agent that requests access to protected resources and submits payments.

EIP-3009
: Ethereum Improvement Proposal enabling gasless token transfers via cryptographic signatures.

EIP-712
: Ethereum standard for typed structured data signing, providing human-readable signing prompts.

ERC-4337
: Account abstraction standard enabling smart contract wallets and gasless transactions.

Facilitator
: Service that handles payment verification and blockchain settlement on behalf of Resource Servers.

HTTP 402
: HTTP status code "Payment Required," used by T402 to signal payment requirements.

Jetton
: TON blockchain's fungible token standard, analogous to ERC-20 on Ethereum.

LayerZero
: Cross-chain messaging protocol used by USDT0 for omnichain transfers.

MCP (Model Context Protocol)
: Protocol enabling AI agents to interact with tools and resources, with T402 support for paid tools.

Nonce
: Unique value used once to prevent replay attacks; in T402, a 32-byte random value.

OFT (Omnichain Fungible Token)
: LayerZero token standard enabling native cross-chain transfers; USDT0 is an OFT.

PaymentPayload
: JSON structure containing signed payment authorization sent by clients.

PaymentRequired
: JSON structure returned by servers describing acceptable payment methods.

Resource Server
: Service providing protected resources (APIs, content) that require payment for access.

Settlement
: The process of executing a payment transaction on the blockchain.

SPL Token
: Solana Program Library token standard, analogous to ERC-20 on Ethereum.

Stablecoin
: Cryptocurrency designed to maintain stable value, typically pegged to fiat currency (e.g., USDT).

TRC-20
: TRON blockchain's token standard, analogous to ERC-20 on Ethereum.

USDT
: Tether USD, the most widely used stablecoin, available on multiple blockchains.

USDT0
: Tether's omnichain USDT implementation using LayerZero OFT standard.

Verification
: The process of validating a payment signature and parameters before settlement.

Bibliography

[1] CASA. CAIP-2: Blockchain ID Specification. Chain Agnostic Improvement Proposal, 2019.

[2] R. Fielding and J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. RFC 7231, June 2014.