T402 Protocol

HTTP-Native Stablecoin Payment Protocol

Technical Whitepaper v2.0

T402 Team
`https://t402.io`

February 2026

Document Version History

Version	Date	Changes
2.0.0	January 2026	Initial whitepaper release
2.1.0	January 2026	Added NEAR, Aptos, Tezos, Polkadot, Stacks
2.2.0	January 2026	Expanded use cases and appendices
2.3.0	February 2026	Added Cosmos, up-to scheme, V2 headers, audit fixes
2.4.0	February 2026	Updated statistics to v2.6.0, added Cosmos payment scheme, corrected network c

Acknowledgments

The T402 protocol represents the collaborative efforts of many individuals and organizations.

Core Contributors: The T402 development team for protocol design, SDK implementations, and infrastructure deployment.

Technical Reviewers: Community members who provided feedback on the protocol specification, security model, and implementation patterns.

Standards Bodies: The work builds upon standards from the IETF (HTTP), CAIP (Chain Agnostic Improvement Proposals), Ethereum (EIPs), and blockchain-specific enhancement proposals (NEPs, SIPs, TZIPs).

Open Source Community: T402 depends on excellent open-source projects including viem, ethers.js, @solana/web3.js, ton-core, tronweb, near-api-js, and many others.

Tether: For developing and maintaining the USDT and USDT0 stablecoin infrastructure that enables global value transfer.

Layer 2 Networks: Base, Arbitrum, Optimism, and other L2 teams for building scalable, low-cost infrastructure that makes micropayments viable.

This whitepaper is dedicated to the vision of an open, programmable internet where value flows as freely as information.

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 ($0.30+ per transaction), settlement delays (2-5 business days), and geographic restrictions, are fundamentally incompatible with the pay-per-use economics of the digital age. Micropayments below $0.50 are economically unviable with credit card processing, leaving a vast design space unexplored.

T402 addresses these challenges through a novel payment protocol that bridges web standards with blockchain settlement:

- **Utilizes HTTP 402**: First practical implementation of the long-dormant HTTP 402 "Payment Required" status code, originally reserved in HTTP/1.1 (1997) for future digital payment systems
- **Supports 10 Blockchain Families**: Unified interface across EVM networks (19+ chains including Ethereum, Base, Arbitrum), Solana, TON, TRON, NEAR, Aptos, Tezos, Polkadot Asset Hub, Stacks, and Cosmos
- **Enables Gasless Transactions**: Leverages EIP-3009 (Transfer with Authorization) and similar mechanisms on non-EVM chains, allowing users to pay without holding native tokens for gas fees
- **Integrates with AI**: Native support for MCP (Model Context Protocol) and A2A (Agent-to-Agent) communication, enabling autonomous AI agents to discover, evaluate, and pay for resources programmatically
- **Minimizes Trust**: Cryptographic binding ensures facilitators cannot redirect funds—recipients are specified in signed authorizations, and facilitators can only submit transactions that honor these specifications
- **Sub-Cent Micropayments**: Transaction costs under 1% enable previously impossible business models—API calls at $0.001, sensor data at $0.0001, per-article content access

The protocol separates concerns into three layers: *Types* (transport-agnostic data structures using CAIP-2 network identifiers), *Logic* (chain-specific payment schemes like "exact" and "up-to"), and *Representation* (transport-layer encoding for HTTP headers, MCP tools, and A2A messages). This architecture enables extensibility across new blockchains and transport protocols while maintaining backward compatibility.

Ecosystem:

- **SDKs**: TypeScript (36 packages), Go, Python, Java
- **HTTP Integrations**: Express, Hono, Fastify, Next.js, Fetch, Axios
- **UI Components**: React, Vue, React Native, Universal Paywall
- **Infrastructure**: Public Facilitator API, Grafana monitoring

Production deployments demonstrate sub-cent transaction costs on Layer 2 networks, settlement finality in seconds (vs. days for traditional payments), and seamless integration with existing HTTP infrastructure requiring only 5-10 lines of middleware code.

The protocol is fully open-source under the Apache 2.0 license, with specifications, reference implementations, and this whitepaper available at `https://github.com/t402-io/t402`.

Keywords: Payment Protocol, HTTP 402, Stablecoin, USDT, USDT0, Cryptocurrency, API Monetization, Micropayments, AI Agents, MCP, EIP-3009, Gasless Transactions, Multi-chain

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:

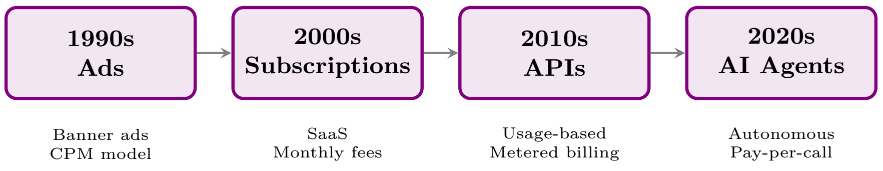


Figure 1.1: Evolution of web monetization models

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 [22] and USDT0 via LayerZero [15]) 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

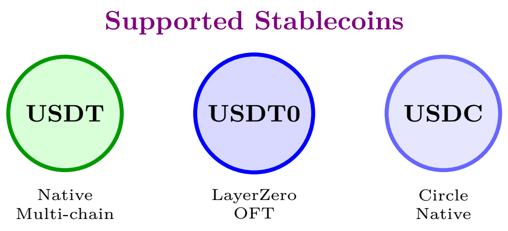


Figure 1.2: T402 supported stablecoins

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 [5]

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 [22], and standardized signing schemes (EIP-712 [2], EIP-3009 [13]) 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:

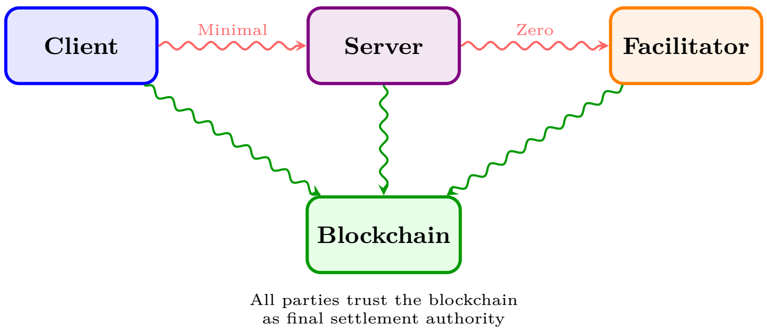

Client
Minimal
Server
Zero
Facilitator
Blockchain
All parties trust the blockchain
as final settlement authority

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 Family	**Mechanism**	**Signature**
EVM Chains (19+)	EIP-3009	ECDSA secp256k1
Solana	SPL Token	Ed25519
TON	Jetton	Ed25519
TRON	TRC-20	ECDSA secp256k1
NEAR	NEP-141	Ed25519
Aptos	Fungible Asset	Ed25519
Tezos	FA2	Ed25519
Polkadot	Asset Hub	Sr25519
Stacks	SIP-010	ECDSA secp256k1
Cosmos/Noble	Native Token	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 { createT402Fetch } from "@t402/fetch";

const t402Fetch = createT402Fetch({ signer });
const response = await t402Fetch(
  "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 all supported chains and the "up-to" scheme for metered payments.

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:

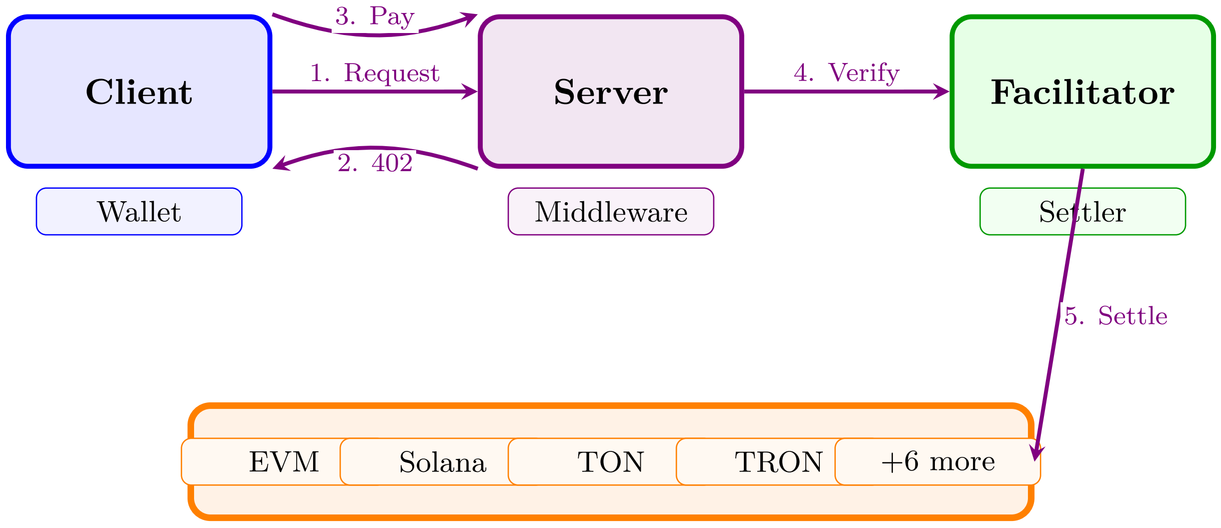


Figure 2.1: T402 System Architecture

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 [13], ERC-4337 [3])

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.

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:

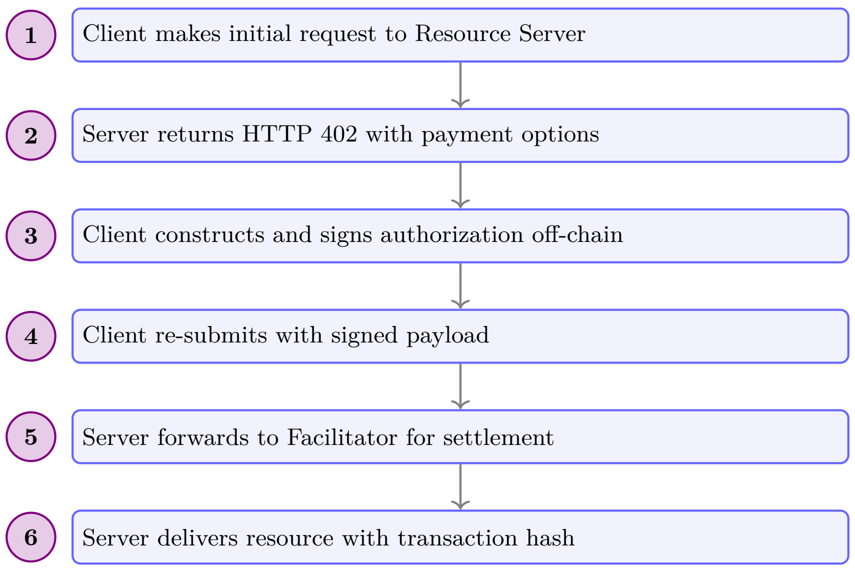

1
Client makes initial request to Resource Server
2
Server returns HTTP 402 with payment options
3
Client constructs and signs authorization off-chain
4
Client re-submits with signed payload
5
Server forwards to Facilitator for settlement
6
Server delivers resource with transaction hash

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

```
HTTP/1.1 402 Payment Required
Content-Type: application/json
PAYMENT-REQUIRED: eyJONDAyVmVyc2lvbiI6Mi4uLn0=

{
  "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: eyJONDAyVmVyc2lvbiI6Mi4uLn0=
```

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.

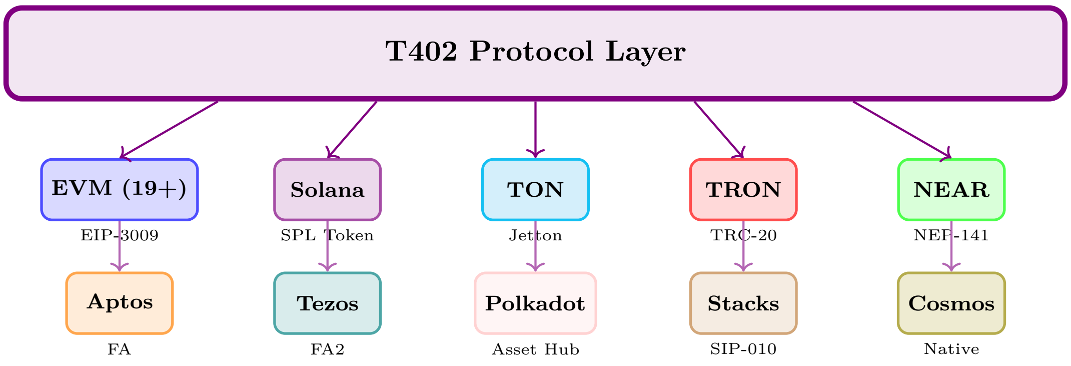


Figure 2.3: T402 multi-chain architecture: 10 blockchain families, 44+ networks

Each blockchain network is identified using CAIP-2 format [4] (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:

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.

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

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:

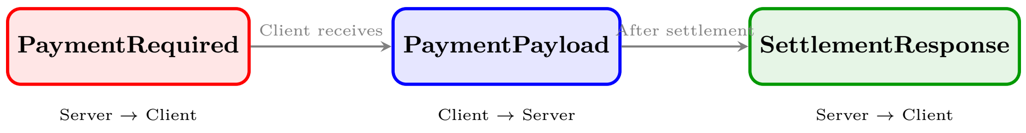


Figure 3.1: T402 message flow

3.2 Protocol Version

3.2.1 Version Field

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

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

Listing 3.1: Version field

For a complete version history and migration guide, see Table 2.3 in section 2.6.

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

```json
{
  "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

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.

Table 3.1: 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 3.2: 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

3.4.1 Structure

```json
{
  "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

Table 3.3: 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

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: 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 3.5: 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

```json
{
  "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 3.6: 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
1  if $P.t402Version \neq 2$ then
2      return $\{valid: false, error: \text{"invalid\_t402\_version"}\}$
   // Scheme match
3  if $P.accepted.scheme \neq R.scheme$ then
4      return $\{valid: false, error: \text{"invalid\_scheme"}\}$
   // Network match
5  if $P.accepted.network \neq R.network$ then
6      return $\{valid: false, error: \text{"invalid\_network"}\}$
   // Amount check
7  if $BigInt(P.accepted.amount) < BigInt(R.amount)$ then
8      return $\{valid: false, error: \text{"invalid\_amount"}\}$
   // Recipient check
9  if $P.accepted.payTo \neq R.payTo$ then
10     return $\{valid: false, error: \text{"invalid\_recipient"}\}$
11 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

3.6.3 Field Definitions

Table 3.7: 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

3.7 Network Identifiers

T402 v2 uses CAIP-2 (Chain Agnostic Improvement Proposal) format for network identification [4].

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 3.8: 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
`near`	Network name	NEAR Protocol networks
`aptos`	Chain ID	Aptos networks
`tezos`	Network hash	Tezos networks
`polkadot`	Genesis hash (partial)	Polkadot parachains
`stacks`	Chain ID	Stacks (Bitcoin L2) networks
`cosmos`	Chain ID string	Cosmos SDK chains

3.7.3 Supported Networks

Table 3.9: Supported Networks (Partial List)

Network	**CAIP-2 ID**	**Tokens**
EVM Networks (19+)		
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
Non-EVM Networks		
Solana	`solana:5eykt...Kvdp`	USDT
TON Mainnet	`ton:mainnet`	USDT
TRON Mainnet	`tron:mainnet`	USDT
NEAR Mainnet	`near:mainnet`	USDT
Aptos Mainnet	`aptos:1`	USDT
Tezos Mainnet	`tezos:NetXdQprcVkpaWU`	USDt
Polkadot Asset Hub	`polkadot:68d56f15...`	USDT
Stacks Mainnet	`stacks:1`	USDT
Cosmos/Noble	`cosmos:noble-1`	USDT

3.8 Error Codes

T402 defines standard error codes organized by category.

3.8.1 Payment Errors

Table 3.10: 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

3.8.2 Protocol Errors

Table 3.11: 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

3.8.3 Settlement Errors

Table 3.12: 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

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.

Table 3.13: HTTP Request Headers

Header	Description
`PAYMENT-SIGNATURE`	Base64-encoded PaymentPayload JSON
`X-PAYMENT`	V1 alias for PAYMENT-SIGNATURE

Table 3.14: HTTP Response Headers

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

3.10 Protocol Extensions

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

3.10.1 Extension Structure

```
{
  "extensions": {
    "siwx": {
      "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 [11] definitions are provided in Appendix A. Key validation rules:

- `t402Version` must equal 2
- `amount` must be a non-negative integer string

Table 3.15: 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

- `network` must match CAIP-2 format
- `accepts` must contain at least one element
- `scheme` must be a recognized scheme identifier

```json
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "type": "object",
  "required": ["t402Version", "resource", "accepts"],
  "properties": {
    "t402Version": {
      "type": "integer",
      "const": 2
    },
    "accepts": {
      "type": "array",
      "minItems": 1
    }
  }
}
```

Listing 3.8: Example validation with JSON Schema

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

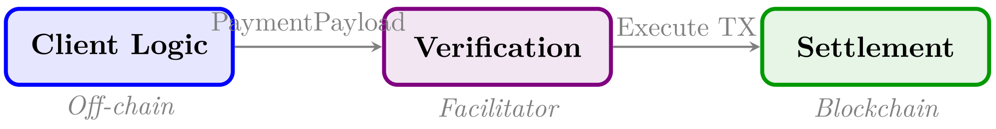


Figure 4.1: Payment scheme component flow

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 [13] (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 [2] structured signing for security and user experience:

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

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

```
  chainId: chainId,        // e.g., 8453 for Base
  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
1  $hash \leftarrow$ EIP712Hash($P.authorization$);
2  $signer \leftarrow$ ecrecover($hash, P.signature$);
3  if $signer \neq P.authorization.from$ then
4      return {isValid: false, invalidReason: "invalid_signature"};
// Step 2: Balance Verification
5  $balance \leftarrow$ balanceOf($R.asset, P.authorization.from$);
6  if $balance < R.amount$ then
7      return {isValid: false, invalidReason: "insufficient_funds"};
// Step 3: Amount Validation
8  if $P.authorization.value < R.amount$ then
9      return {isValid: false, invalidReason: "invalid_amount"};
// Step 4: Recipient Validation
10 if $P.authorization.to \neq R.payTo$ then
11     return {isValid: false, invalidReason: "invalid_recipient"};
// Step 5: Time Window Validation
12 $now \leftarrow$ currentTimestamp();
13 if $now < P.authorization.validAfter$ then
14     return {isValid: false, invalidReason: "authorization_not_yet_valid"};
15 if $now > P.authorization.validBefore$ then
16     return {isValid: false, invalidReason: "expired_authorization"};
// Step 6: Nonce Validation
17 $used \leftarrow$ authorizationState($P.authorization.from, P.authorization.nonce$);
18 if $used$ then
19     return {isValid: false, invalidReason: "nonce_already_used"};
// Step 7: Transaction Simulation
20 $result \leftarrow$ eth_call(transferWithAuthorization, $P$);
21 if $result.reverted$ then
22     return {isValid: false, invalidReason: "simulation_failed"};
23 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 [16], 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

```json
{
  "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

```json
{
  "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 [8], 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 [9], 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

```json
{
  "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 NEAR Implementation

On NEAR Protocol [18], the exact scheme uses NEP-141 [19] fungible token transfers with function call access keys for gasless execution.

4.7.1 NEP-141: Fungible Token Standard

NEP-141 defines NEAR's fungible token interface:

```json
{
  "receiver_id": "merchant.near",
  "amount": "1000000",
  "memo": "T402 payment"
}
```

Listing 4.9: NEP-141 transfer call

4.7.2 Payload Structure

```json
{
  "signedTransaction": "base64-encoded-signed-transaction",
  "publicKey": "ed25519:...",
  "accountId": "payer.near"
}
```

Listing 4.10: NEAR exact scheme payload

4.7.3 Gasless Architecture

NEAR enables gasless payments through:

- **Function Call Access Keys**: Limited keys that can only call specific contract methods
- **Relayer Pattern**: Facilitator submits transactions on behalf of users
- **Meta Transactions**: NEP-366 enables signed actions without gas

4.7.4 Verification Requirements

- Validate Ed25519 signature over serialized transaction
- Verify `receiver_id` matches `payTo` account
- Check token contract is official USDT (`usdt.tether-token.near`)
- Validate `amount` meets requirements
- Confirm account has sufficient balance

4.8 Aptos Implementation

On Aptos [14], the exact scheme uses the Fungible Asset standard (formerly Coin) for USDT transfers.

4.8.1 Fungible Asset Standard

Aptos migrated from the legacy Coin module to the Fungible Asset framework:

```json
{
  "function": "0x1::primary_fungible_store::transfer",
  "type_arguments": ["0x...(usdt_metadata)"],
  "arguments": ["recipient_address", "amount"]
}
```

Listing 4.11: Aptos transfer entry function

4.8.2 Payload Structure

```json
{
  "transaction": {
    "sender": "0x1234...5678",
    "sequence_number": "42",
    "max_gas_amount": "2000",
    "gas_unit_price": "100",
    "expiration_timestamp_secs": "1740672154",
    "payload": { ... }
  },
  "signature": {
    "type": "ed25519_signature",
    "public_key": "0x...",
    "signature": "0x..."
  }
}
```

Listing 4.12: Aptos exact scheme payload

4.8.3 Sponsored Transactions

Aptos supports fee sponsorship through:

- **Fee Payer**: Secondary signer pays gas fees

- **Multi-Agent**: Multiple signers with designated fee payer
- **Simulation**: Pre-flight checks before broadcast

4.8.4 Verification Requirements

- Validate Ed25519 signature
- Verify `sequence_number` is current (not replayed)
- Check `expiration_timestamp_secs` is in the future
- Validate recipient matches `payTo`
- Confirm USDT metadata address: `0xf73e...473cb`

4.9 Tezos Implementation

On Tezos [7], the exact scheme uses the FA2 [23] (TZIP-12) token standard for USDt transfers.

4.9.1 FA2: Multi-Asset Standard

FA2 is Tezos's unified token standard supporting fungible and non-fungible tokens:

```json
{
  "entrypoint": "transfer",
  "value": [{
    "from_": "tz1...",
    "txs": [{
      "to_": "tz1...",
      "token_id": 0,
      "amount": "1000000"
    }]
  }]
}
```

Listing 4.13: FA2 transfer entrypoint

4.9.2 Payload Structure

```json
{
  "operation": {
    "branch": "BL...",
    "contents": [{
      "kind": "transaction",
      "source": "tz1...",
      "destination": "KT1XnTn74bUtxHfDtBmm2bGZAQfhPbvKWR8o",
      "amount": "0",
      "parameters": { ... }
    }],
    "signature": "edsig..."
  }
}
```

Listing 4.14: Tezos exact scheme payload

4.9.3 Gasless via Permits

Tezos enables gasless transfers through TZIP-17 permits:

- **Off-chain Signatures**: Users sign permit data off-chain
- **Permit Entrypoint**: Relayer submits permit + transfer
- **Nonce Management**: Per-user counters prevent replay

4.9.4 Verification Requirements

- Validate Ed25519 signature (tz1) or secp256k1 (tz2)
- Verify USDt contract: `KT1XnTn74bUtxHfDtBmm2bGZAQfhPbvKWR8o`
- Check `token_id` is 0 (USDt)
- Validate counter for replay protection
- Confirm sufficient balance

4.10 Polkadot Implementation

On Polkadot [10], the exact scheme uses the Asset Hub [21] parachain's Assets pallet for USDT transfers.

4.10.1 Assets Pallet

The Assets pallet provides fungible token functionality on Asset Hub:

```
{
  "call": "Assets.transfer",
  "args": {
    "id": 1984,
    "target": "14...",
    "amount": "1000000"
  }
}
```

Listing 4.15: Assets.transfer call

4.10.2 Payload Structure

```
{
  "extrinsic": {
    "signature": {
      "signer": "14...",
      "signature": {
        "Sr25519": "0x..."
      },
      "era": { "mortal": [64, 0] },
      "nonce": 42,
      "tip": 0
    },
    "method": { ... }
  }
}
```

Listing 4.16: Polkadot exact scheme payload

4.10.3 Cross-Chain Considerations

- **Relay Chain**: Polkadot relay for security
- **Asset Hub**: Parachain 1000 hosts USDT (Asset ID 1984)
- **XCM**: Cross-chain messaging for multi-chain operations

4.10.4 Verification Requirements

- Validate Sr25519 signature
- Verify Asset ID is 1984 (USDT)
- Check era for transaction mortality
- Validate nonce for replay protection
- Confirm transfer target matches `payTo`

4.11 Stacks Implementation

On Stacks [6] (Bitcoin L2), the exact scheme uses SIP-010 [20] fungible token transfers secured by Bitcoin.

4.11.1 SIP-010: Fungible Token Standard

SIP-010 defines Stacks' standard trait for fungible tokens:

```json
{
  "contractAddress": "SP3Y2ZSH8P7D50B0VBTSX11S7XSG24M1VB9YFQA4K",
  "contractName": "token-susdc",
  "functionName": "transfer",
  "functionArgs": ["1000000", "sender", "recipient", "none"]
}
```

Listing 4.17: SIP-010 transfer call

4.11.2 Payload Structure

```json
{
  "transaction": {
    "version": 0,
    "chainId": 1,
    "auth": {
      "type": "standard",
      "spendingCondition": {
        "signer": "SP...",
        "nonce": 42,
        "fee": 1000,
        "signature": "..."
      }
    },
    "payload": {
      "type": "contract_call",
      ...
    }
  }
}
```

Listing 4.18: Stacks exact scheme payload

4.11.3 Bitcoin Anchoring

Stacks transactions are secured by Bitcoin:

- **Proof of Transfer**: Miners commit to Bitcoin blockchain
- **Bitcoin Finality**: Transactions finalize with Bitcoin blocks (∼10 min)
- **Clarity Language**: Smart contracts in decidable language

4.11.4 Verification Requirements

- Validate ECDSA secp256k1 signature
- Verify contract is official USDT: `SP3Y2ZSH8P7D50B0VBTSX11S7XSG24M1VB9YFQA4K.token-susdc`
- Check nonce for replay protection
- Validate recipient principal matches `payTo`
- Consider Bitcoin confirmation requirements for high-value payments

4.12 Cosmos Implementation

On Cosmos, the exact scheme uses native USDT on the Noble blockchain, a Cosmos appchain dedicated to native stablecoin issuance.

4.12.1 Native Token Standard

Noble hosts native USDT with the `uusdt` denomination (micro-USDT, 6 decimals). Transfers use the standard Cosmos `bank/MsgSend` message:

```
{
  "typeUrl": "/cosmos.bank.v1beta1.MsgSend",
  "value": {
    "fromAddress": "noble1...",
    "toAddress": "noble1...",
    "amount": [{
      "denom": "uusdt",
      "amount": "1000000"
    }]
  }
}
```

Listing 4.19: Cosmos bank send message

4.12.2 Payload Structure

```
{
  "signedTx": "base64-encoded-signed-transaction",
  "publicKey": "base64-encoded-public-key",
  "accountAddress": "noble1..."
}
```

Listing 4.20: Cosmos exact scheme payload

4.12.3 Verification Requirements

- Validate secp256k1 signature
- Verify sender matches `fromAddress`
- Check denomination is `uusdt`
- Validate recipient matches `payTo`
- Confirm sufficient balance
- Verify chain ID matches Noble mainnet (`noble-1`) or testnet (`grand-1`)

4.13 Scheme Comparison

Table 4.4: Exact Scheme Implementation Comparison (Primary Networks)

Aspect	EVM	Solana	TON	TRON	NEAR
Signature	ECDSA	Ed25519	Ed25519	ECDSA	Ed25519
Standard	EIP-3009	SPL Token	Jetton	TRC-20	NEP-141
Nonce	32 bytes	N/A (tx)	query_id	N/A	Nonce
Gasless	Native	Fee sponsor	Fee sponsor	Energy	Fee sponsor
Finality	∼2s (L2)	∼0.4s	∼5s	∼3s	∼1s

Table 4.5: Additional Network Implementations

Aspect	Aptos	Tezos	Polkadot	Stacks	Cosmos
Signature	Ed25519	Ed25519	Sr25519	ECDSA	secp256k1
Standard	Fungible Asset	FA2	Asset Hub	SIP-010	Native
Nonce	Seq number	Counter	Nonce	Nonce	Sequence
Gasless	Fee sponsor	Fee sponsor	Fee sponsor	Fee sponsor	Fee sponsor
Finality	∼1s	∼30s	∼12s	∼10min	∼6s

4.14 Future Schemes

Beyond the `exact` scheme detailed in this chapter, T402 supports and plans additional payment patterns:

- **Up-To Scheme (Implemented)**: Metered payments for usage-based billing, currently available on Solana, TON, TRON, and NEAR. See **??** for detailed specification.
- **Permit2 Integration (Planned)**: Uniswap's universal approval system [17] for improved gas efficiency
- **Subscription Scheme (Planned)**: Recurring payments with automatic renewal

For detailed specifications and implementation guides for these schemes, see **??**.

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

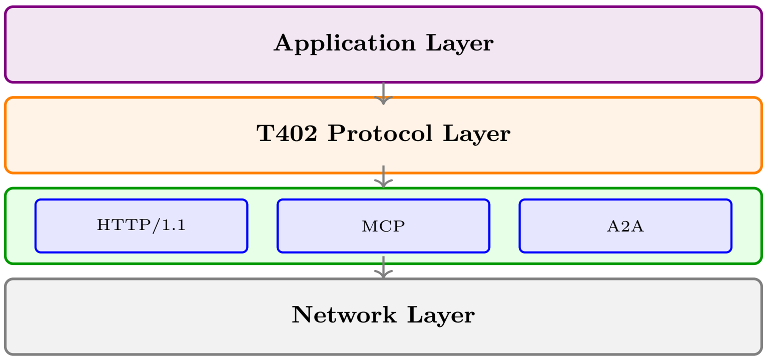


Figure 5.1: Transport layer architecture

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 [12] §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"
}
```

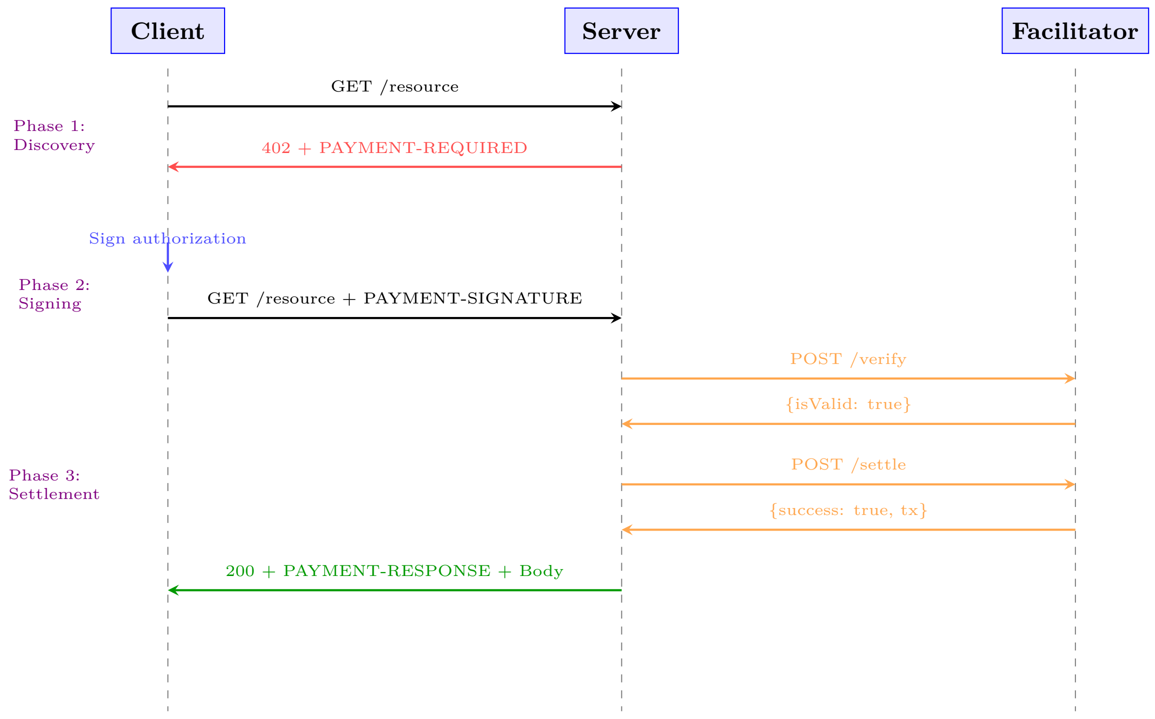


Figure 5.2: HTTP transport message flow

Listing 5.1: Phase 1: Initial request and 402 response

```
# Client sends request with signed payment
GET /api/premium/data HTTP/1.1
Host: api.example.com
Accept: application/json
PAYMENT-SIGNATURE: eyJONDAyVmVyc2lvbiI6Miwi...

# Server returns resource with settlement confirmation
HTTP/1.1 200 OK
Content-Type: application/json
PAYMENT-RESPONSE: eyJzdWNjZXNzIjpOcnVlLC...
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) [1] transport enables AI agents to make payments when invoking tools. MCP uses JSON-RPC 2.0 over stdio or HTTP.

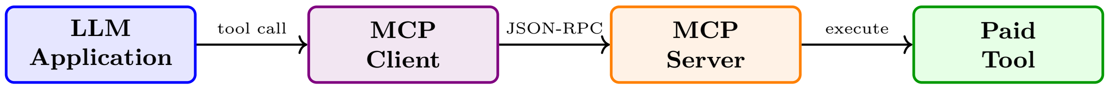


Figure 5.3: MCP transport architecture

5.3.1 Architecture Overview

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:

```json
{
  "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

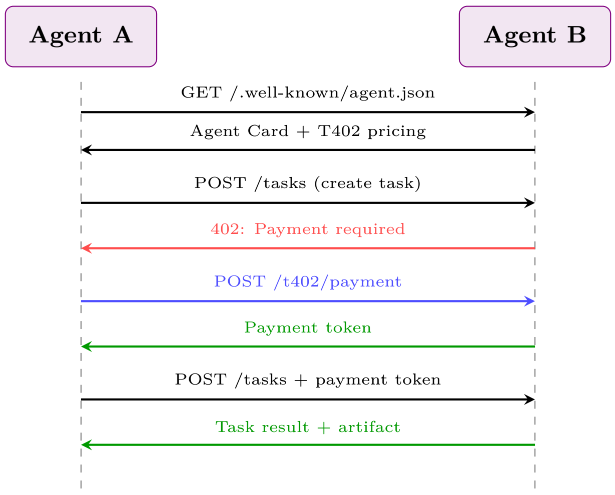

Agent A
Agent B
GET /.well-known/agent.json
Agent Card + T402 pricing
POST /tasks (create task)
402: Payment required
POST /t402/payment
Payment token
POST /tasks + payment token
Task result + artifact

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

```
1 interface T402Transport
2     signalPaymentRequired(req: PaymentRequirements) → void;
3     receivePayment() → PaymentPayload | null;
4     sendSettlement(resp: SettlementResponse) → void;
5     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

5.7 Transport Comparison

Table 5.7: Transport Layer Comparison

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

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

Security Analysis

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

6.1 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.

6.2 Formal Threat Model

6.2.1 Adversary Capabilities

We consider adversaries with the following capabilities:

Table 6.1: 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

6.2.2 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

6.2.3 Threat Categories

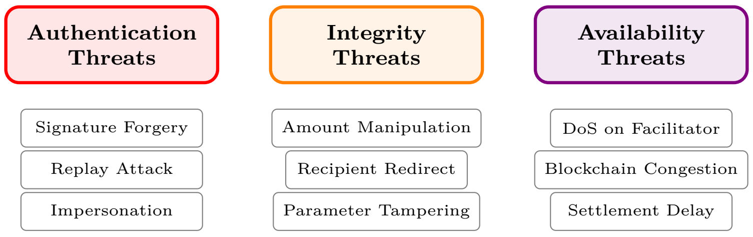


Figure 6.1: Threat category taxonomy

6.3 Attack Analysis

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

6.3.1 Replay Attacks

Replay Attack Definition

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

6.3.1.1 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

6.3.1.2 Mitigations

T402 employs three-layer replay protection:

Table 6.2: 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
1  if $t > A.validBefore$ then
2      return true // Expired, cannot be replayed
   // Nonce check (on-chain)
3  $used \leftarrow$ authorizationState($A.from, A.nonce$);
4  if $used$ then
5      return true // Already used, cannot be replayed
   // Domain check (signature verification)
6  $domain \leftarrow \{chainId, verifyingContract\}$;
7  if $domain \neq A.signedDomain$ then
8      return true // Wrong chain/contract, invalid signature
9  return false // Valid, could potentially be replayed
```

6.3.2 Signature Forgery

Signature Forgery Definition

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

6.3.2.1 Attack Resistance

Theorem 6.1 (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 [13] 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)

□

6.3.3 Man-in-the-Middle (MitM)

6.3.3.1 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

6.3.3.2 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.

6.3.4 Double Spending

6.3.4.1 Attack Vector

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

6.3.4.2 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

6.3.5 Insufficient Payment Attack

6.3.5.1 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

6.3.5.2 Mitigations

1. **Amount Verification**: Facilitator checks $authorization.value \geq requirements.amount$

Table 6.3: 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
TRON	∼3 seconds	Low
NEAR	∼1.2 seconds	Low
Aptos	∼1 second	Very Low
Tezos	∼30 seconds	Low
Polkadot	∼12 seconds	Low
Stacks	∼10 minutes	Moderate
Cosmos/Noble	∼6 seconds	Low
Ethereum L1	∼13 minutes	Moderate

2. **Exact Match**: For `exact` scheme, values must match precisely
3. **Server Validation**: Server re-verifies before serving content

6.3.6 Wrong Recipient Attack

6.3.6.1 Attack Vector

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

6.3.6.2 Mitigations

Theorem 6.2 (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 [2] 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
```

□

6.4 Cryptographic Security

6.4.1 Signature Algorithms

Table 6.4: 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
NEAR	Ed25519	256 bits	128 bits
Aptos	Ed25519	256 bits	128 bits
Tezos	Ed25519 / secp256k1	256 bits	128 bits
Polkadot	Sr25519 (Schnorrkel)	256 bits	128 bits
Stacks	ECDSA secp256k1	256 bits	128 bits
Cosmos	ECDSA secp256k1	256 bits	128 bits

6.4.2 Hash Functions

Table 6.5: 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

6.4.3 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 6.1: Secure nonce generation

6.5 Facilitator Security

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

6.5.1 Security Architecture

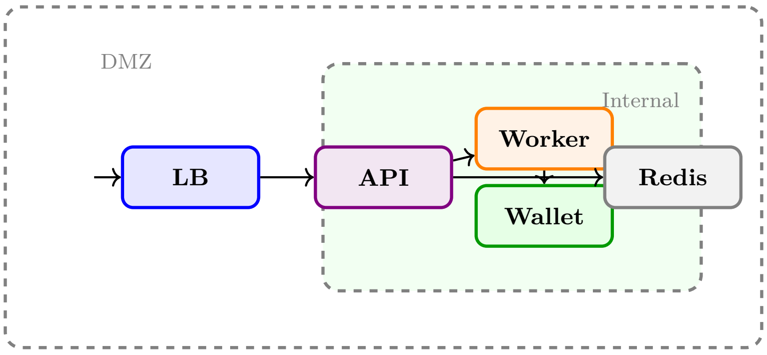


Figure 6.2: Facilitator security architecture

6.5.2 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.

6.5.2.1 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 6.2: Secure key loading example

6.5.3 API Security

Table 6.6: 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

6.5.4 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

6.6 Chain-Specific Security

6.6.1 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

6.6.2 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

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

    // 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 6.3: Solana security validation

6.6.3 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

6.6.4 TRON Security Considerations

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

6.7 Deployment Security

6.7.1 Container Security

```yaml
version: '3.8'
services:
  facilitator:
    image: ghcr.io/t402-io/facilitator:latest
    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:
      - internal
```

Listing 6.4: Docker security configuration

6.7.2 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

6.7.3 Secret Management

Table 6.7: 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)

6.8 Incident Response

6.8.1 Severity Classification

Table 6.8: 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

6.8.2 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

6.9 Security Audit Status

6.9.1 Completed Audits

Table 6.9: Security Audit History

Component	Auditor	Date	Status
Protocol Design	Internal	2024 Q4	Complete
SDK Code Review	Internal	2025 Q1	Complete
Security Hardening	Internal	2026 Q1	Complete
Smart Contract Audit	External	TBD	Pending

6.9.2 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

6.10 Responsible Disclosure

6.10.1 Reporting Channels

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

6.11 Summary

Table 6.10: 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 7

Implementation Guide

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

7.1 SDK Overview

T402 provides official SDKs for four major languages:

Table 7.1: Official SDK Comparison

SDK	Version	Registry	Frameworks	License
TypeScript	2.6.0	npm @t402/*	Express, Hono, Fastify, Next.js	Apache 2.0
Python	1.11.0	PyPI	FastAPI, Flask, Django, Starlette	Apache 2.0
Go	1.11.0	Go Modules	Gin, Echo, Chi, Fiber	Apache 2.0
Java	1.11.0	Maven Central	Servlet, Spring, WebFlux, Micronaut, Quarkus	Apache 2.0

7.1.1 TypeScript SDK Architecture

The TypeScript SDK is organized as a monorepo with 36 modular packages:

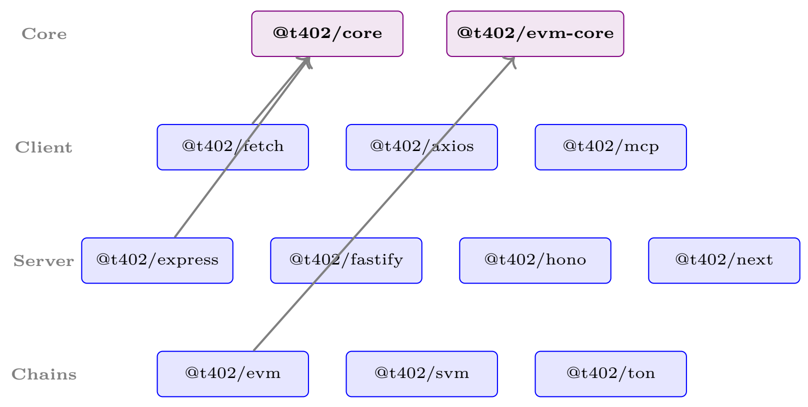


Figure 7.1: TypeScript SDK package architecture

Table 7.2: TypeScript Package Categories

Category	**Packages**
Core	core, evm-core, extensions
Client	fetch, axios
Server	express, fastify, hono, next
AI	mcp, a2a
Chains	evm, svm, ton, tron, near, aptos, tezos, polkadot, stacks, cosmos, btc
UI	paywall, react, vue, react-native
CLI	cli
WDK	wdk, wdk-gasless, wdk-bridge, wdk-multisig, wdk-protocol, wdk-defi, wdk-ton, wdk-ton-gasless, wdk-tron-gasfree

7.2 Server-Side Integration

7.2.1 Express.js (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 7.1: Complete Express.js server setup

7.2.2 FastAPI (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
```

```python
    content_length = int(request.headers.get("content-length", 0))
    return f"{content_length / 100000:.4f}"  # $0.01 per 100KB
```

Listing 7.2: Complete FastAPI server setup

7.2.3 Gin (Go)

```go
package main

import (
    "github.com/gin-gonic/gin"
    t402 "github.com/t402-io/t402/sdks/go"
    t402http "github.com/t402-io/t402/sdks/go/http"
    t402gin "github.com/t402-io/t402/sdks/go/http/gin"
)

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 7.3: Complete Gin server setup

7.2.4 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 7.4: Complete Spring Boot setup

7.3 Client-Side Integration

7.3.1 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 7.5: TypeScript client with wallet integration

7.3.2 Python Client

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

# 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 7.6: Python client with async support

7.3.3 Go Client

```go
package main
```

```go

import (
    "context"
    "fmt"
    "log"

    t402http "github.com/t402-io/t402/sdks/go/http"
    "github.com/t402-io/t402/sdks/go/mechanisms/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 7.7: Go client implementation

7.4 MCP Server Integration

7.4.1 Creating a Paid MCP Tool

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

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 7.8: MCP server with paid tools (conceptual)

7.5 Testing Guide

7.5.1 Unit Testing

```
import { describe, it, expect, vi } from "vitest";
import { createMockPayment, MockFacilitator } from "@t402/core/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 7.9: Unit testing with mocks (conceptual)

7.5.2 Integration Testing

```
import { T402TestClient } from "@t402/core/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 7.10: Integration testing with testnet (conceptual)

7.6 Error Handling

7.6.1 Client-Side Error Handling

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

```typescript

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 7.11: Comprehensive error handling

7.6.2 Server-Side Error Handling

```typescript
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 7.12: Server error handling

7.7 Best Practices

7.7.1 Security Checklist

Table 7.3: 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

7.7.2 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 7.13: Caching and optimization

7.7.3 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 7.14: Metrics and monitoring

7.8 Migration Guide

7.8.1 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 });
});

```

```javascript
// 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 7.15: 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 8

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, examining both the business rationale and technical implementation for each use case.

8.1 AI Agent Payments

The emergence of autonomous AI agents creates unprecedented demand for machine-to-machine payments. Unlike human users who can manually enter credit card details or authenticate through OAuth flows, AI agents operate programmatically and require payment mechanisms that integrate seamlessly into their execution loops.

8.1.1 The Agent Payment Problem

Traditional payment systems assume human involvement at critical decision points: entering payment details, reviewing charges, and authorizing transactions. This assumption breaks down completely when the "user" is an AI agent executing hundreds of API calls per minute to complete a complex task.

Consider a research agent tasked with analyzing market trends. To complete its work, the agent might need to:

- Query financial data APIs for stock prices and trading volumes
- Access news APIs for sentiment analysis on recent articles
- Use specialized ML services for natural language processing
- Retrieve historical datasets for trend comparison

Each of these services may be operated by different providers, each with their own payment requirements. Without a programmatic payment protocol, the agent would be blocked at the first paywall, unable to proceed without human intervention.

T402 solves this by providing a fully programmatic payment flow. When an agent encounters a 402 response, it can automatically parse the payment requirements, sign an authorization with its configured wallet, and retry the request—all without human involvement. The entire process completes in milliseconds, allowing agents to navigate paid services as easily as free ones.

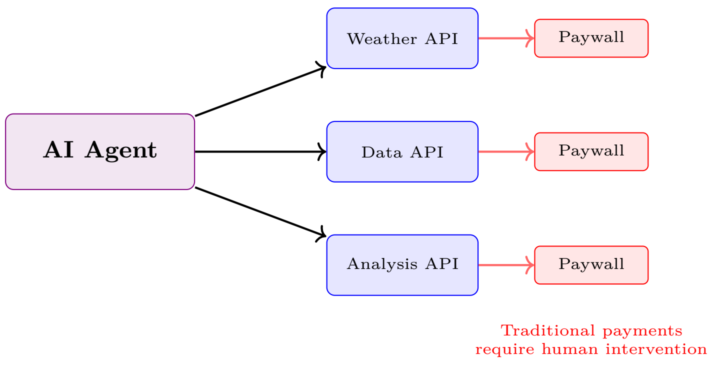


Figure 8.1: AI agents blocked by traditional paywalls requiring human interaction

Table 8.1: AI Agent Payment Requirements

Requirement	**Description**
Programmatic Access	Pure API integration without UI interaction or browser automation
Budget Control	Configurable spending limits per call, per hour, and per day to prevent runaway costs
Cost Visibility	Ability to inspect prices before execution for cost-benefit decisions
Autonomous Execution	No human approval required for individual transactions within budget
Failure Handling	Graceful degradation when payments fail or budgets are exceeded

8.1.2 MCP Tool Marketplace

The Model Context Protocol (MCP), developed by Anthropic, provides a standardized way for AI models to discover and interact with external tools. T402 extends MCP with payment capabilities, enabling a marketplace where tool providers can monetize their services while agents maintain full autonomy.

In this model, tool providers register their services with pricing information. When an agent discovers available tools, it receives not just the tool's capabilities but also its cost. The agent can then make intelligent decisions about which tools to use based on both capability and cost-effectiveness.

Budget management becomes crucial in this context. An agent operator might configure daily spending limits of $10, with per-call maximums of $0.10. The agent respects these limits automatically, choosing cheaper alternatives when available or gracefully declining tasks that would exceed its budget.

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

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 8.1: AI agent with budget management for MCP tools

The code above demonstrates several key patterns. First, the agent is configured with a hierarchical budget structure—per-call, hourly, and daily limits—providing multiple layers of cost control. Second, the agent can filter tools by price, excluding expensive options that don't fit its budget. Third, when budgets are exceeded, a callback notifies the operator, who can then decide whether to increase limits or accept the limitation.

8.1.3 Agent-to-Agent Economy

As AI agents become more specialized, an ecosystem emerges where agents pay other agents for services. A research agent might pay a data collection agent for raw information, which then pays an analysis agent for insights, creating a value chain of specialized services.

This agent economy enables unprecedented specialization. Rather than building monolithic agents that handle every aspect of a task, developers can create focused agents that excel at specific functions. A data scraping agent might be optimized for web extraction, while a separate summarization agent excels at condensing information. By paying each other for services, these agents can collaborate to solve problems neither could handle alone.

The economic implications are significant. Agent operators can monetize their agents' capabilities, creating a marketplace where the best-performing agents command premium prices. This creates incentives for continuous improvement and specialization, driving the overall quality of available agent services upward.

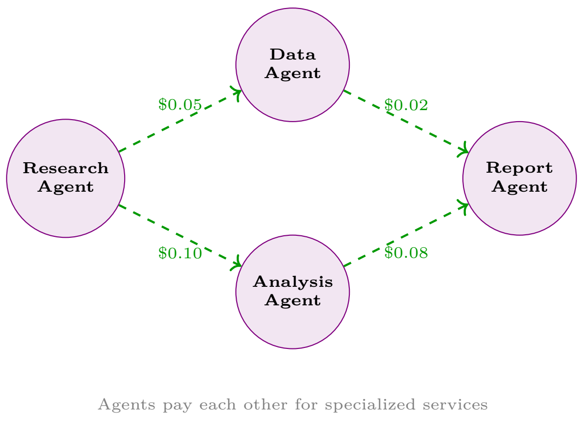


Figure 8.2: Agent-to-agent payment flows in a specialized service ecosystem

8.2 API Monetization

API monetization traditionally requires significant infrastructure: user registration systems, subscription management, billing cycles, and payment processing integration. T402 dramatically simplifies this by enabling per-request billing with a single middleware addition.

8.2.1 Traditional vs T402 Monetization

Traditional API monetization follows a subscription model. Users sign up, provide payment details, choose a plan, and then access the API within their plan's limits. This model works well for predictable, high-volume usage but creates friction for occasional users and barriers for experimentation.

Consider a developer evaluating multiple weather APIs. Under the traditional model, they might need to create accounts and provide credit cards to four or five services just to compare response quality. With T402, they can make paid test calls to any API instantly, paying only for what they use.

Table 8.2: API Monetization Comparison

Aspect	**Traditional (Stripe)**	**T402**
Setup	Create plans, pricing tiers, billing integration	Add middleware (5 lines)
User Flow	Sign up → Subscribe → Use	Use → Pay → Access
Billing	Monthly invoices with reconciliation	Per-request instant settlement
Minimum Viable	$5-10/month subscription floor	$0.001 per call possible
Global Access	Credit card availability varies by region	Worldwide with crypto wallet
Settlement	2-3 business days to bank	Instant to wallet
Chargeback Risk	Yes, 1-2% typical	No, cryptographic finality

The "Use first, pay per use" model fundamentally changes the relationship between API providers and consumers. Providers no longer need to convince users to commit to subscriptions; they simply need to provide value, and payment follows naturally. This lowers the barrier to adoption while ensuring providers are compensated for every request served.

8.2.2 Weather Data API Example

Weather data demonstrates tiered pricing well. Current conditions require minimal computation, while extended forecasts demand sophisticated modeling. Historical data involves database queries proportional to the time range. T402 allows pricing each endpoint according to its true cost and value.

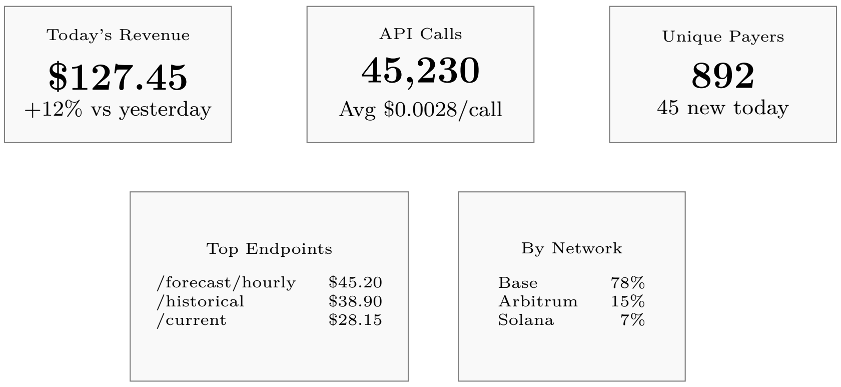
```
import express from "express";
import { paymentMiddleware } from "@t402/express";

const app = express();

// Tiered pricing based on data resolution and computation
const pricingTiers = {
  "GET /api/weather/current": {
    price: "$0.001",
    description: "Current conditions - minimal computation"
  },
  "GET /api/weather/forecast/hourly": {
    price: "$0.005",
    description: "48-hour hourly forecast - moderate modeling"
  },
  "GET /api/weather/forecast/daily": {
    price: "$0.01",
    description: "14-day daily forecast - complex modeling"
  },
  "GET /api/weather/historical": {
    price: async (req) => {
      // Price scales with data volume requested
      const days = calculateDays(req.query.start, req.query.end);
      return `$${(days * 0.001).toFixed(4)}`;
    },
    description: "Historical weather data - database intensive"
  }
};

app.use(paymentMiddleware({
  routes: pricingTiers,
  payTo: "0xWeatherServiceWallet..."
}));

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

Listing 8.2: Weather API with tiered pricing reflecting computational cost

The dynamic pricing for historical data demonstrates an important pattern. Rather than charging a flat rate regardless of the query scope, the price scales with the actual resources consumed. A query for one day of history costs $0.001, while a year of data costs $0.365. This fairness encourages usage while ensuring the provider covers their costs.

8.2.3 Revenue Analytics

T402 payments provide rich analytics data. Every payment includes the payer's address, the exact endpoint accessed, and the settlement transaction. Providers can analyze this data to understand usage patterns, identify high-value customers, and optimize pricing.



Today's Revenue
$127.45
+12% vs yesterday

API Calls
45,230
Avg $0.0028/call

Unique Payers
892
45 new today

Top Endpoints

/forecast/hourly	$45.20
/historical	$38.90
/current	$28.15

By Network

Base	78%
Arbitrum	15%
Solana	7%

Figure 8.3: API monetization dashboard showing real-time revenue metrics

The network distribution data reveals which blockchain ecosystems your users prefer, informing decisions about which networks to prioritize for gas optimization. The endpoint breakdown helps identify which features drive revenue, guiding product development priorities.

8.3 Content Access

Digital content—articles, videos, music, research papers—has traditionally been monetized through advertising or subscriptions. Both models have significant drawbacks: advertising compromises user experience and privacy, while subscriptions create commitment barriers and "subscription fatigue."

8.3.1 The Subscription Fatigue Problem

Modern consumers face an overwhelming array of subscription services. News outlets, streaming platforms, productivity tools, and specialized content providers each demand monthly commitments. The result is subscription fatigue: consumers either over-subscribe (paying for services they rarely use) or under-subscribe (missing valuable content behind paywalls they won't cross).

Subscription Fatigue

The average consumer subscribes to 6+ content services at $10-15/month each, totaling $60-90 monthly. Studies show most subscribed content goes unconsumed—users pay for access they don't use. T402 enables pay-per-use: read one article for $0.25 instead of subscribing for $10/month, paying only for actual consumption.

Pay-per-access fundamentally changes this dynamic. A reader interested in a single investigative journalism piece shouldn't need to subscribe to an entire publication. A viewer wanting to watch one documentary shouldn't commit to a streaming platform. T402 enables granular access where users pay precisely for what they consume.

8.3.2 News Article Paywall

Implementing a pay-per-article news paywall demonstrates how content pricing can reflect value. Breaking news might command premium prices due to timeliness, while evergreen content could be priced lower. In-depth analysis and original research justify higher prices than commodity news.

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

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

    // Full article delivered after payment
    res.json({
      title: article.title,
      content: article.content,
      author: article.author,
      publishedAt: article.publishedAt
    });
  },
  {
    // Dynamic pricing reflects article value
    price: async (req) => {
      const article = await getArticleMeta(req.query.slug);
      const prices = {
        "breaking": "0.10",    // Time-sensitive, high value
        "analysis": "0.25",    // Expert insight
        "research": "0.50",    // Original investigation
        "standard": "0.05"     // General news
      };
      return prices[article.type] || "0.10";
    },
    network: "eip155:8453"  // Base network for low fees
  }
);
```

Listing 8.3: News paywall with value-based dynamic pricing

This approach benefits both publishers and readers. Publishers can price content according to production cost and value, rather than averaging everything into a single subscription price. Readers pay fair prices for exactly what they read, without subsidizing content they'd never access.

8.3.3 Video Streaming

Video content naturally lends itself to pay-per-view models. A two-hour film requires more infrastructure and delivers more value than a two-minute clip. Pricing by duration aligns cost with consumption.

```
// 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
      const pricePerMinute = 0.01;
      const minutes = video.duration / 60;
      return `$${(minutes * pricePerMinute).toFixed(4)}`;
    }
  }),
  async (req, res) => {
    const { payer, transaction } = req.t402Payment;

    // Generate time-limited access token
    // Prevents sharing and ensures payment per view
    const accessToken = await generateAccessToken({
      videoId: req.params.id,
      payer,
      transaction,
      expiresIn: "24h"  // 24-hour viewing window
    });

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

Listing 8.4: Pay-per-view video with duration-based pricing

The access token mechanism ensures that payment grants temporary access rather than permanent ownership. This mirrors the traditional rental model while providing the convenience of instant, global access without account creation.

8.4 Micropayments

Perhaps no use case demonstrates T402's advantages more clearly than micropayments. Traditional payment processors impose minimum fees that make sub-dollar transactions economically impossible. A $0.30 minimum fee on a $0.01 transaction represents a 3,000% overhead, rendering the payment absurd.

8.4.1 Economic Viability

T402 enables true micropayments by reducing transaction costs to under 1% even for tiny amounts. This unlocks entirely new business models that were previously impossible.

Table 8.3: Micropayment Economics: Traditional vs T402

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%

The economic impact is transformative. Services that couldn't exist with traditional payments—per-sensor IoT data, per-millisecond compute, per-byte storage access—become viable. The long tail of small transactions, previously abandoned, becomes a significant revenue stream.

8.4.2 IoT Data Marketplace

Internet of Things devices generate continuous streams of valuable data: temperature readings, air quality measurements, traffic counts, energy consumption. This data has value, but each individual reading is worth fractions of a cent. T402 enables monetizing this data at its natural granularity.

```
// Sensor data pricing: fraction of a cent per reading
const sensorPricing = {
  temperature: "0.0001",      // $0.0001 per reading
  humidity: "0.0001",
  air_quality: "0.0005",      // More valuable sensor type
  traffic: "0.001",           // High-value infrastructure data
  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 with volume discount
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]);
      // 20% discount for bulk purchases
      return `$${(count * basePrice * 0.8).toFixed(6)}`;
    }
  }),
  async (req, res) => {
    const readings = await getBulkReadings(
      req.params.type,
      req.query.count
    );
    res.json(readings);
  }
);
```

Listing 8.5: IoT sensor data marketplace with granular pricing

This model enables entirely new IoT business models. A smart city might deploy thousands of environmental sensors, each generating revenue through data sales. The sensors pay for themselves through micro-transactions, creating sustainable infrastructure funding.

8.4.3 Compute-on-Demand

Cloud computing typically bills by the hour or minute, with minimum commitments. T402 enables billing at the actual resource consumption level: per CPU-second and per megabyte of memory.

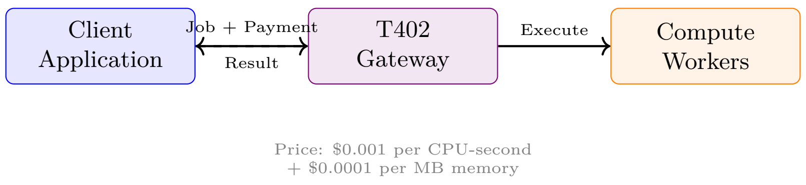


Figure 8.4: Pay-per-compute architecture with resource-based pricing

This granular billing enables new use cases. A developer running a quick script doesn't need to provision a server or commit to hourly billing—they pay for the 500 milliseconds of compute they actually used. Batch processing jobs pay exactly for their resource consumption without rounding up to billing intervals.

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

      // Calculate price from resource limits
      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,
        // Could refund difference if actual < estimated
        actualCost: calculateActualCost(result)
      }
    });
  }
);
```

Listing 8.6: Serverless compute with resource-based pricing

8.5 Cross-Border Payments

International payments remain one of the most friction-laden areas of traditional finance. Currency conversion, correspondent banking, regulatory compliance, and processing delays create barriers that especially impact small transactions and underserved regions.

8.5.1 Traditional Cross-Border Challenges

A freelancer in Southeast Asia completing a $50 project for a client in Europe faces significant hurdles. Wire transfers cost $25-50 in fees alone, making small payments impractical. PayPal may not operate in their country, or charges 4-5% plus currency conversion fees. Even when payment succeeds, funds might take 3-5 business days to arrive.

Table 8.4: Cross-Border Payment Comparison

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

T402 eliminates these barriers. USDT provides a stable value reference regardless of local currency volatility. Blockchain settlement is global by design—a transaction from Japan to Brazil settles as quickly as one within the same city. Fees remain minimal regardless of transaction geography or amount.

8.5.2 Freelancer Platform Example

Consider a platform connecting global freelancers with clients. Traditional payment processing requires integrating multiple payment methods for different regions, handling currency conversion, and managing payment disputes. With T402, payment flows directly from client to freelancer upon work acceptance.

```
// 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;  // Price agreed upfront
    },
    payTo: async (req) => {
      // Pay directly to freelancer's wallet - no intermediary
      const deliverable = await getDeliverable(req.params.id);
      return deliverable.freelancerWallet;
    }
  }),
  async (req, res) => {
    const { transaction } = req.t402Payment;

    // Mark as paid with on-chain proof
    await updateDeliverable(req.params.id, {
      status: "paid",
      paymentTx: transaction,
      paidAt: new Date()
    });

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

Listing 8.7: Global freelancer payments with direct settlement

This model provides significant advantages. The freelancer receives payment instantly upon work acceptance—no waiting days for bank transfers. The platform doesn't need to hold funds or manage escrow; payment flows directly between parties with on-chain proof. Disputes can reference the immutable transaction record.

8.6 Research Data Access

Academic and commercial research increasingly depends on data. Researchers need access to datasets for analysis, model training, and validation. Data providers need sustainable revenue models. Traditional licensing involves lengthy negotiations, minimum commitments, and one-size-fits-all pricing.

T402 enables tiered data access where researchers pay for exactly the access level they need. A student exploring a dataset can access a free preview. A researcher validating a methodology can purchase a sample. A production system can access the full dataset. Each tier is priced appropriately.

```
// Dataset pricing tiers reflecting access levels
const datasetPricing = {
  preview: { rows: 100, price: "0" },           // Free exploration
  sample: { rows: 1000, price: "1.00" },        // Validation
  standard: { rows: 10000, price: "5.00" },     // Research use
  full: { rows: -1, price: "25.00" }            // Production access
};

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 requires no payment
    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,
      // Help users understand what they're getting
      accessedRows: data.length,
      fullAccessPrice: datasetPricing.full.price
    });
  }
);
```

Listing 8.8: Research dataset marketplace with tiered access

This model democratizes data access. Researchers in underfunded institutions can access data affordably at lower tiers. Data providers earn revenue proportional to the value delivered. The free preview tier encourages exploration and discovery while paid tiers support sustainable data curation.

8.7 Gaming and Virtual Goods

Video games have long experimented with digital economies: in-game purchases, virtual currencies, and item trading. T402 brings several advantages: direct player-to-player payments, no platform fees on trades, and cross-game item portability potential.

In-game purchases through T402 eliminate traditional app store fees (typically 30%). Players pay directly for items, with the game developer receiving the full amount minus minimal blockchain fees. This either increases developer revenue or allows lower prices for players—or both.

```
// In-game item purchase - no app store fees
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 with provenance
    await grantItem(playerId, itemId, {
      purchasedBy: payer,
      transaction,
      purchasedAt: new Date()
    });

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

```

```
// Player-to-player trading - direct P2P payment
app.post("/api/game/trade/execute",
  paymentMiddleware.route({
    price: (req) => req.body.price,
    payTo: async (req) => {
      // Pay seller directly - platform takes no cut
      const seller = await getPlayer(req.body.sellerId);
      return seller.walletAddress;
    }
  }),
  async (req, res) => {
    const { transaction } = req.t402Payment;

    // Transfer item between players
    await transferItem(
      req.body.sellerId,
      req.body.buyerId,
      req.body.itemId,
      { transaction }  // On-chain proof of sale
    );

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

Listing 8.9: Game item marketplace with direct payments

Player-to-player trading demonstrates T402's peer-to-peer capabilities. The `payTo` field dynamically routes payment to the seller's wallet, enabling true player economies without platform intermediation. The on-chain transaction provides proof of sale, enabling dispute resolution if needed.

8.8 Decision Framework

Not every payment scenario is ideal for T402. Understanding when to use it—and when traditional methods might be more appropriate—helps developers make informed architectural decisions.

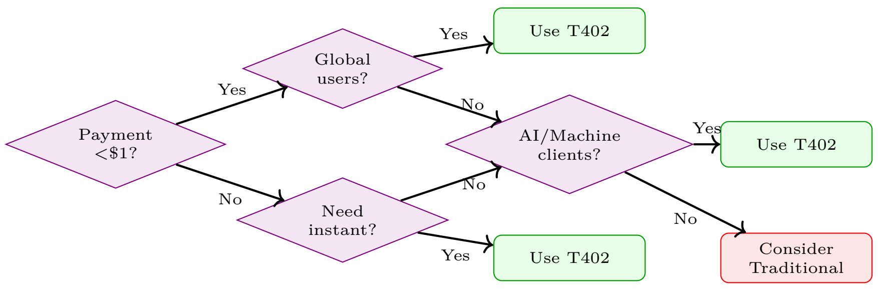


Figure 8.5: Decision framework for evaluating T402 fit

8.8.1 Ideal Use Cases

Certain characteristics make a use case particularly well-suited for T402:

Table 8.5: T402 Fit Assessment by Use Case

Use Case	T402 Fit	Key Benefit
AI Agent Tools	Excellent	Fully autonomous payments without human intervention
API Micropayments	Excellent	Sub-cent transactions economically viable
Global Content	Excellent	No geographic restrictions or currency barriers
IoT Data	Excellent	High-volume micro-transactions at scale
Freelance Platforms	Good	Instant settlement, no intermediary holding funds
E-commerce	Moderate	Users may prefer familiar payment methods
Recurring Billing	Limited	Use subscription extensions or traditional methods

8.8.2 When NOT to Use T402

Equally important is understanding scenarios where traditional payments may be more appropriate:

When NOT to Use T402

- **Users unfamiliar with crypto wallets**: Consumer applications targeting mainstream users may face adoption friction
- **Strict regulatory compliance**: Some jurisdictions have unclear cryptocurrency regulations
- **Recurring subscription billing**: Monthly subscriptions are better served by traditional recurring billing (though T402 extensions can help)
- **Refund-heavy business models**: Blockchain transactions are irreversible; businesses requiring frequent refunds need additional infrastructure
- **Fiat-only requirements**: Some business contexts legally require fiat currency settlement

The best implementations often combine T402 with traditional payments, offering users choice. A content platform might accept both credit cards and T402 payments, capturing users comfortable with either method while providing the benefits of each.

Chapter 9

Conclusion

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 that works seamlessly with existing web infrastructure.

9.1 Summary of Contributions

T402 introduces several significant advances in the intersection of web protocols and blockchain payments.

9.1.1 Technical Contributions

1. **HTTP 402 Activation**: T402 provides the first practical implementation of the long-dormant HTTP 402 "Payment Required" status code. Originally reserved in HTTP/1.1 (1997) for future use, this status code finally fulfills its intended purpose—signaling that a resource requires payment before access is granted.
2. **Transport Agnosticism**: The protocol cleanly separates payment logic from transport mechanisms. Whether payments flow over HTTP headers, MCP tool calls, or A2A agent messages, the underlying verification and settlement logic remains consistent. This separation enables future transport protocols without requiring changes to the core payment infrastructure.
3. **Chain Agnosticism**: T402 provides a unified interface across ten distinct blockchain ecosystems: EVM (19+ networks), Solana, TON, TRON, NEAR, Aptos, Tezos, Polkadot, Stacks, and Cosmos. Each chain's unique signing and token mechanisms are abstracted behind consistent interfaces, allowing developers to support multiple chains without chain-specific code paths.
4. **Gasless User Experience**: By leveraging EIP-3009 (Transfer with Authorization) for EVM chains and similar mechanisms on other networks, T402 eliminates the need for end users to hold native tokens for gas. The facilitator sponsors all transaction fees, creating a seamless payment experience comparable to traditional credit cards.
5. **Trust Minimization**: The facilitator architecture ensures that payment recipients are cryptographically bound in the signed authorization. Facilitators cannot redirect funds to unauthorized addresses—they can only submit transactions that transfer funds to the merchant's specified wallet.
6. **AI-Native Design**: T402 provides first-class support for autonomous agent payments via MCP integration. AI agents can discover, evaluate, and pay for tools programmatically,

enabling machine-to-machine commerce without human intervention.

9.1.2 Ecosystem Achievements

The T402 ecosystem has achieved significant milestones:

Table 9.1: T402 Ecosystem Statistics

Metric	**Value**
Supported Blockchain Networks	10 families (44+ networks)
SDK Languages	4 (TypeScript, Go, Python, Java)
TypeScript Packages	36 packages
HTTP Framework Integrations	18 (across 4 SDKs)
Frontend Component Libraries	3 (React, Vue, Paywall)
Protocol Version	v2

9.2 Comparison with Alternatives

T402 occupies a unique position in the payment landscape, combining the simplicity of traditional web payments with the global reach of cryptocurrency.

Table 9.2: Payment Protocol Comparison

Feature	**T402**	**Stripe**	**Lightning**	**Web3 dApps**
Micropayments (<$0.10)	✓		✓	
Global Access	✓		✓	✓
No User Signup	✓		✓	✓
Instant Settlement	✓		✓	✓
Price Stability	✓	✓		
AI Agent Support	✓			
HTTP Native	✓	✓		
Multi-chain	✓			✓
Gasless UX	✓	✓		

vs. Traditional Payments (Stripe, PayPal): T402 enables micropayments below the $0.50 minimum viable for credit cards, provides instant settlement instead of 2-3 day delays, and works globally without credit card infrastructure dependencies. However, traditional payments offer greater consumer familiarity and regulatory clarity.

vs. Lightning Network: Both enable micropayments, but T402 uses stablecoins (eliminating price volatility) and doesn't require channel management or liquidity provisioning. Lightning offers stronger decentralization.

vs. General Web3: T402 provides gasless UX and HTTP-native integration, avoiding the complexity of wallet connections and transaction signing UIs. Traditional Web3 dApps offer greater decentralization and on-chain composability.

9.3 Limitations and Future Directions

9.3.1 Current Limitations

We acknowledge several limitations in the current T402 design:

- **Facilitator Centralization**: While facilitators cannot steal funds, they represent a centralization point for transaction submission. Users must trust facilitators for liveness and timely settlement.
- **Stablecoin Dependency**: T402 relies on USDT/USDT0 availability and stability. Regulatory actions against Tether or stablecoin depegging events would impact the protocol.
- **User Onboarding**: Despite gasless UX, users still need cryptocurrency wallets and initial USDT balances, creating friction compared to credit card payments.
- **Refund Complexity**: Blockchain transactions are irreversible. Implementing refunds requires additional infrastructure and trust assumptions.
- **Regulatory Uncertainty**: Cryptocurrency payment regulations vary by jurisdiction and continue to evolve, creating compliance challenges for some use cases.

9.3.2 Future Development

The T402 roadmap addresses these limitations and extends capabilities:

Decentralized Facilitator Network Replace single facilitators with decentralized networks using threshold signatures and economic incentives, eliminating centralization concerns.

Up-To Payment Scheme Enable streaming payments and variable-amount authorizations for metered services, subscriptions, and pay-as-you-go models.

Multi-Asset Support Extend beyond USDT to support USDC, DAI, and other stablecoins, providing users with asset choice and reducing single-issuer dependency.

Privacy Enhancements Integrate zero-knowledge proofs for payment verification without revealing transaction amounts or participant identities.

Fiat On-Ramps Partner with fiat on-ramp providers to enable direct credit card to T402 payment flows, eliminating the need for pre-existing crypto balances.

9.4 Ecosystem Impact

T402 enables business models that were previously impractical:

API Economy Developers can monetize APIs with per-request pricing, eliminating subscription barriers and enabling true pay-per-use models. A weather API can charge $0.001 per request rather than requiring $10/month subscriptions.

Content Creators Writers, artists, and video creators can sell individual pieces without platform intermediaries taking 30%+ fees. A journalist can charge $0.25 per article, receiving nearly the full amount.

AI Agents Autonomous agents can acquire resources, pay for tools, and transact with other agents without human intervention. This enables new categories of AI applications that require economic agency.

IoT Networks Sensor networks can monetize data at natural granularities—$0.0001 per reading—creating sustainable funding models for infrastructure that generates continuous small-value data streams.

Global Freelancing Workers in underbanked regions can receive instant payments without 4-5% PayPal fees or 3-5 day wire transfer delays. A $50 project pays out $49.50 instantly, not $45 in a week.

9.5 Call to Action

We invite the developer community, enterprises, and researchers to participate in the T402 ecosystem:

Build Integrate T402 into your applications using our SDKs. Start with a single endpoint and expand as you validate the model with your users.

Contribute Submit improvements, bug fixes, and new features via GitHub. The protocol is open source under the Apache 2.0 license, and we welcome contributions across all SDKs.

Extend Develop new payment schemes, transport bindings, or chain mechanisms. The modular architecture supports extension without core protocol changes.

Research Investigate advanced topics: privacy-preserving payments, decentralized facilitator networks, cross-chain atomic settlements, and formal verification of payment protocols.

Deploy Run your own Facilitator instance for specialized use cases or enhanced privacy. The facilitator software is open source and designed for self-hosting.

9.6 Resources

Table 9.3: T402 Resources

Resource	**URL**
Website	`https://t402.io`
Documentation	`https://docs.t402.io`
GitHub Repository	`https://github.com/t402-io/t402`
Facilitator API	`https://facilitator.t402.io`
npm Packages	`https://www.npmjs.com/org/t402`
PyPI Package	`https://pypi.org/project/t402`
Go Module	`https://pkg.go.dev/github.com/t402-io/t402/sdks/go`

9.7 Closing Remarks

The web was designed with payments in mind—HTTP 402 exists as proof of this intention. For three decades, that vision remained unrealized due to the complexity of integrating traditional payment systems with web protocols. Cryptocurrency and stablecoins provide the missing piece: programmable money that flows as easily as data.

T402 bridges this gap, bringing the simplicity of REST APIs to the world of payments. A developer can add payment requirements to an endpoint with five lines of code. A user can pay without creating accounts or sharing sensitive financial data. An AI agent can autonomously acquire resources to complete its tasks.

We believe this represents a fundamental shift in how value flows on the internet. The protocol is ready. The tools are available. The community is growing.

The future of payments is HTTP-native.

Build it with us.

Appendix A

Complete JSON Schemas

This appendix provides complete JSON Schema definitions for all T402 data structures, including core protocol types and network-specific payload formats.

A.1 Core Protocol Schemas

A.1.1 PaymentRequired Schema

The `PaymentRequired` structure is returned by servers in HTTP 402 responses, describing acceptable payment methods.

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://t402.io/schemas/payment-required.json",
  "title": "PaymentRequired",
  "description": "Server response indicating payment requirements",
  "type": "object",
  "required": ["t402Version", "resource", "accepts"],
  "properties": {
    "t402Version": {
      "type": "integer",
      "const": 2,
      "description": "Protocol version (must be 2)"
    },
    "error": {
      "type": "string",
      "description": "Human-readable error message"
    },
    "resource": {
      "$ref": "#/$defs/ResourceInfo"
    },
    "accepts": {
      "type": "array",
      "items": { "$ref": "#/$defs/PaymentRequirements" },
      "minItems": 1,
      "description": "Acceptable payment options"
    },
    "extensions": {
      "type": "object",
      "description": "Protocol extensions"
    }
  },
  "$defs": {
    "ResourceInfo": {
      "type": "object",
      "required": ["url"],
      "properties": {
        "url": {
          "type": "string",
          "format": "uri",
          "description": "Canonical URL of the resource"
        },
        "description": {
          "type": "string",
          "description": "Human-readable resource description"
        },
        "mimeType": {
          "type": "string",
          "description": "Expected response MIME type"
        }
      }
    },
    "PaymentRequirements": {
      "type": "object",
      "required": ["scheme", "network", "amount", "asset", "payTo",
    ↪ "maxTimeoutSeconds"],
      "properties": {
        "scheme": {
          "type": "string",
          "enum": ["exact", "upto"],
          "description": "Payment scheme identifier"
        },
        "network": {
          "type": "string",
          "pattern": "^[a-z0-9]+:[a-zA-Z0-9]+$",
          "description": "CAIP-2 network identifier"
        },
        "amount": {
          "type": "string",
          "pattern": "^[0-9]+$",
          "description": "Amount in smallest units (e.g., wei)"
        },
        "asset": {
          "type": "string",
          "description": "Token contract address"
        },
        "payTo": {
          "type": "string",
          "description": "Recipient address"
        },
        "maxTimeoutSeconds": {
          "type": "integer",
          "minimum": 60,
          "maximum": 3600,
          "description": "Maximum authorization validity"
        },
        "extra": {
          "type": "object",
          "description": "Network-specific parameters"
        }
      }
    }
  }
}
```

Listing A.1: PaymentRequired JSON Schema

A.1.2 PaymentPayload Schema

The `PaymentPayload` structure is sent by clients containing the signed payment authorization.

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://t402.io/schemas/payment-payload.json",
  "title": "PaymentPayload",
  "description": "Client payment authorization",
  "type": "object",
  "required": ["t402Version", "accepted", "payload"],
  "properties": {
    "t402Version": {
      "type": "integer",
      "const": 2
    },
    "resource": {
      "$ref": "payment-required.json#/$defs/ResourceInfo"
    },
    "accepted": {
      "$ref": "payment-required.json#/$defs/PaymentRequirements",
      "description": "The accepted payment option"
    },
    "payload": {
      "type": "object",
      "description": "Network-specific signed payload"
    },
    "extensions": {
      "type": "object"
    }
  }
}
```

Listing A.2: PaymentPayload JSON Schema

A.1.3 Facilitator Response Schemas

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://t402.io/schemas/verify-response.json",
  "title": "VerifyResponse",
  "description": "Facilitator verification result",
  "type": "object",
  "required": ["isValid"],
  "properties": {
    "isValid": {
```

```json
      "type": "boolean",
      "description": "Whether the payment authorization is valid"
    },
    "invalidReason": {
      "type": "string",
      "description": "Reason for invalidity (if isValid=false)"
    },
    "payer": {
      "type": "string",
      "description": "Recovered payer address"
    }
  }
}
```

Listing A.3: VerifyResponse JSON Schema

```json
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://t402.io/schemas/settle-response.json",
  "title": "SettleResponse",
  "description": "Facilitator settlement result",
  "type": "object",
  "required": ["success"],
  "properties": {
    "success": {
      "type": "boolean",
      "description": "Whether settlement succeeded"
    },
    "errorReason": {
      "type": "string",
      "description": "Error reason (if success=false)"
    },
    "payer": {
      "type": "string",
      "description": "Payer address"
    },
    "transaction": {
      "type": "string",
      "description": "Transaction hash/signature"
    },
    "network": {
      "type": "string",
      "description": "Network where settlement occurred"
    }
  }
}
```

Listing A.4: SettleResponse JSON Schema

A.2 Network-Specific Payload Schemas

Each blockchain network has a unique payload format reflecting its signing mechanisms.

A.2.1 EVM Payload (EIP-3009)

```json
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://t402.io/schemas/evm-payload.json",
  "title": "EVMPayload",
  "description": "EIP-3009 TransferWithAuthorization payload",
  "type": "object",
  "required": ["from", "to", "value", "validAfter", "validBefore", "nonce",
    ↪ "signature"],
  "properties": {
    "from": {
      "type": "string",
      "pattern": "^0x[a-fA-F0-9]{40}$",
      "description": "Payer address"
    },
    "to": {
      "type": "string",
      "pattern": "^0x[a-fA-F0-9]{40}$",
      "description": "Recipient address"
    },
    "value": {
      "type": "string",
      "pattern": "^[0-9]+$",
      "description": "Amount in token smallest units"
    },
    "validAfter": {
      "type": "integer",
      "description": "Unix timestamp after which authorization is valid"
    },
    "validBefore": {
      "type": "integer",
      "description": "Unix timestamp before which authorization is valid"
    },
    "nonce": {
      "type": "string",
      "pattern": "^0x[a-fA-F0-9]{64}$",
      "description": "32-byte random nonce"
    },
    "signature": {
      "type": "string",
      "pattern": "^0x[a-fA-F0-9]{130}$",
      "description": "EIP-712 signature (65 bytes)"
    }
  }
}
```

Listing A.5: EVM TransferWithAuthorization Payload

A.2.2 Solana Payload

```json
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://t402.io/schemas/solana-payload.json",
  "title": "SolanaPayload",
  "description": "Solana SPL token transfer authorization",
  "type": "object",
  "required": ["payer", "recipient", "amount", "mint", "signature",
    ↪ "recentBlockhash"],
  "properties": {
    "payer": {
      "type": "string",
      "description": "Base58-encoded payer public key"
    },
    "recipient": {
      "type": "string",
      "description": "Base58-encoded recipient public key"
    },
    "amount": {
      "type": "string",
      "pattern": "^[0-9]+$",
      "description": "Amount in lamports"
    },
    "mint": {
      "type": "string",
      "description": "SPL token mint address"
    },
    "signature": {
      "type": "string",
      "description": "Base58-encoded Ed25519 signature"
    },
    "recentBlockhash": {
      "type": "string",
      "description": "Recent blockhash for transaction freshness"
    }
  }
}
```

Listing A.6: Solana SPL Token Transfer Payload

A.2.3 TON Payload

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://t402.io/schemas/ton-payload.json",
  "title": "TONPayload",
  "description": "TON Jetton transfer authorization",
  "type": "object",
  "required": ["sender", "recipient", "amount", "jettonMaster", "queryId",
    ↪ "signature"],
  "properties": {
    "sender": {
      "type": "string",
      "description": "Sender address (bounceable or non-bounceable)"
    },
    "recipient": {
      "type": "string",
      "description": "Recipient address"
    },
    "amount": {
      "type": "string",
      "pattern": "^[0-9]+$",
      "description": "Amount in nano-units"
    },
    "jettonMaster": {
      "type": "string",
      "description": "Jetton master contract address"
    },
    "queryId": {
      "type": "string",
      "description": "64-bit query ID for message uniqueness"
    },
    "forwardPayload": {
      "type": "string",
      "description": "Optional forward payload (base64)"
    },
    "signature": {
      "type": "string",
      "description": "Ed25519 signature (base64)"
    },
    "validUntil": {
      "type": "integer",
      "description": "Message expiration timestamp"
    }
  }
}
```

Listing A.7: TON Jetton Transfer Payload

A.2.4 TRON Payload

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://t402.io/schemas/tron-payload.json",
  "title": "TRONPayload",
  "description": "TRON TRC-20 transfer authorization",
  "type": "object",
  "required": ["from", "to", "amount", "contract", "signature"],
  "properties": {
    "from": {
      "type": "string",
      "pattern": "^T[a-zA-Z0-9]{33}$",
      "description": "Sender address (Base58Check)"
    },
    "to": {
      "type": "string",
      "pattern": "^T[a-zA-Z0-9]{33}$",
      "description": "Recipient address"
    },
    "amount": {
      "type": "string",
      "pattern": "^[0-9]+$",
      "description": "Amount in smallest units"
    },
    "contract": {
      "type": "string",
      "description": "TRC-20 contract address"
    },
    "expiration": {
      "type": "integer",
      "description": "Transaction expiration timestamp"
    },
    "signature": {
      "type": "string",
      "description": "ECDSA secp256k1 signature (hex)"
    }
  }
}
```

Listing A.8: TRON TRC-20 Transfer Payload

A.2.5 NEAR Payload

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://t402.io/schemas/near-payload.json",
  "title": "NEARPayload",
  "description": "NEAR NEP-141 fungible token transfer",
  "type": "object",
  "required": ["signerId", "receiverId", "amount", "tokenContract",
   ↪ "signature"],
  "properties": {
    "signerId": {
      "type": "string",
      "description": "Sender account ID (e.g., alice.near)"
    },
    "receiverId": {
      "type": "string",
      "description": "Recipient account ID"
    },
    "amount": {
      "type": "string",
      "pattern": "^[0-9]+$",
      "description": "Amount in yoctoNEAR (10^-24)"
    },
    "tokenContract": {
      "type": "string",
      "description": "NEP-141 token contract ID"
    },
    "nonce": {
      "type": "integer",
      "description": "Access key nonce"
    },
    "blockHash": {
      "type": "string",
      "description": "Recent block hash (base58)"
    },
    "signature": {
      "type": "string",
      "description": "Ed25519 signature (base58)"
    }
  }
}
```

Listing A.9: NEAR NEP-141 Transfer Payload

A.2.6 Aptos Payload

```json
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://t402.io/schemas/aptos-payload.json",
  "title": "AptosPayload",
  "description": "Aptos Fungible Asset transfer",
  "type": "object",
  "required": ["sender", "recipient", "amount", "metadata", "signature"],
  "properties": {
    "sender": {
      "type": "string",
      "pattern": "^0x[a-fA-F0-9]{64}$",
      "description": "Sender address (32-byte hex)"
    },
    "recipient": {
      "type": "string",
      "pattern": "^0x[a-fA-F0-9]{64}$",
      "description": "Recipient address"
    },
    "amount": {
      "type": "string",
      "pattern": "^[0-9]+$",
      "description": "Amount in smallest units"
    },
    "metadata": {
      "type": "string",
      "description": "Fungible Asset metadata address"
    },
    "sequenceNumber": {
      "type": "integer",
      "description": "Account sequence number"
    },
    "expirationTimestamp": {
      "type": "integer",
      "description": "Transaction expiration (Unix seconds)"
    },
    "signature": {
      "type": "string",
      "description": "Ed25519 signature (hex)"
    }
  }
}
```

Listing A.10: Aptos Fungible Asset Transfer Payload

A.2.7 Tezos Payload

```json
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://t402.io/schemas/tezos-payload.json",
  "title": "TezosPayload",
  "description": "Tezos FA2 token transfer",
  "type": "object",
  "required": ["source", "destination", "amount", "contract", "tokenId",
    ↪ "signature"],
  "properties": {
    "source": {
      "type": "string",
      "pattern": "^tz[123][a-zA-Z0-9]{33}$",
      "description": "Source address (tz1/tz2/tz3)"
    },
    "destination": {
      "type": "string",
      "description": "Destination address"
    },
    "amount": {
      "type": "string",
      "pattern": "^[0-9]+$",
      "description": "Token amount"
    },
    "contract": {
      "type": "string",
      "pattern": "^KT1[a-zA-Z0-9]{33}$",
      "description": "FA2 contract address"
    },
    "tokenId": {
      "type": "integer",
      "description": "Token ID within FA2 contract"
    },
    "counter": {
      "type": "integer",
      "description": "Account operation counter"
    },
    "signature": {
      "type": "string",
      "description": "Ed25519/secp256k1 signature"
    }
  }
}
```

Listing A.11: Tezos FA2 Transfer Payload

A.2.8 Polkadot Payload

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://t402.io/schemas/polkadot-payload.json",
  "title": "PolkadotPayload",
  "description": "Polkadot Asset Hub transfer",
  "type": "object",
  "required": ["sender", "recipient", "amount", "assetId", "signature"],
  "properties": {
    "sender": {
      "type": "string",
      "description": "SS58-encoded sender address"
    },
    "recipient": {
      "type": "string",
      "description": "SS58-encoded recipient address"
    },
    "amount": {
      "type": "string",
      "pattern": "^[0-9]+$",
      "description": "Amount in smallest units"
    },
    "assetId": {
      "type": "integer",
      "description": "Asset ID on Asset Hub (1984 for USDT)"
    },
    "nonce": {
      "type": "integer",
      "description": "Account nonce"
    },
    "era": {
      "type": "string",
      "description": "Transaction mortality (immortal or mortal)"
    },
    "signature": {
      "type": "string",
      "description": "Sr25519/Ed25519 signature (hex)"
    }
  }
}
```

Listing A.12: Polkadot Asset Hub Transfer Payload

A.2.9 Stacks Payload

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://t402.io/schemas/stacks-payload.json",
  "title": "StacksPayload",
  "description": "Stacks SIP-010 fungible token transfer",
  "type": "object",
  "required": ["sender", "recipient", "amount", "contract", "signature"],
  "properties": {
    "sender": {
      "type": "string",
      "pattern": "^S[PM][A-Z0-9]{38,39}$",
      "description": "Stacks address (SP for mainnet, ST for testnet)"
    },
    "recipient": {
      "type": "string",
      "description": "Recipient Stacks address"
    },
    "amount": {
      "type": "string",
      "pattern": "^[0-9]+$",
      "description": "Amount in micro-units"
    },
    "contract": {
      "type": "string",
      "description": "SIP-010 contract identifier"
    },
    "nonce": {
      "type": "integer",
      "description": "Account nonce"
    },
    "fee": {
      "type": "string",
      "description": "Transaction fee in microSTX"
    },
    "signature": {
      "type": "string",
      "description": "secp256k1 signature (hex)"
    }
  }
}
```

Listing A.13: Stacks SIP-010 Transfer Payload

A.3 Extension Schemas

A.3.1 Receipt Extension

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://t402.io/schemas/extensions/receipt.json",
  "title": "ReceiptExtension",
  "description": "Payment receipt for record-keeping",
  "type": "object",
  "properties": {
    "receipt": {
      "type": "object",
      "properties": {
        "id": {
          "type": "string",
          "format": "uuid",
          "description": "Unique receipt identifier"
        },
        "timestamp": {
          "type": "string",
          "format": "date-time",
          "description": "Payment timestamp (ISO 8601)"
        },
        "merchant": {
          "type": "object",
          "properties": {
            "name": { "type": "string" },
            "address": { "type": "string" },
            "taxId": { "type": "string" }
          }
        },
        "items": {
          "type": "array",
          "items": {
            "type": "object",
            "properties": {
              "description": { "type": "string" },
              "quantity": { "type": "number" },
              "unitPrice": { "type": "string" },
              "total": { "type": "string" }
            }
          }
        }
      }
    }
  }
}
```

Listing A.14: Receipt Extension Schema

A.3.2 Subscription Extension

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://t402.io/schemas/extensions/subscription.json",
  "title": "SubscriptionExtension",
  "description": "Recurring payment parameters",
  "type": "object",
  "properties": {
    "subscription": {
      "type": "object",
      "properties": {
        "id": {
          "type": "string",
          "description": "Subscription identifier"
        },
        "interval": {
          "type": "string",
          "enum": ["daily", "weekly", "monthly", "yearly"],
          "description": "Billing interval"
        },
        "startDate": {
          "type": "string",
          "format": "date",
          "description": "Subscription start date"
        },
        "endDate": {
          "type": "string",
          "format": "date",
          "description": "Subscription end date (optional)"
        },
        "maxPayments": {
          "type": "integer",
          "description": "Maximum number of payments"
        }
      }
    }
  }
}
```

Listing A.15: Subscription Extension Schema

Appendix B

Supported Networks

This appendix provides comprehensive network configuration details for all supported chains.

B.1 EVM Networks

T402 supports 25+ EVM-compatible networks through the unified EIP-3009 interface. The tables below list networks by token standard.

Table B.1: EVM Networks with USDT0 (LayerZero OFT)

Network	CAIP-2 ID	USDT0 Contract
Ethereum	eip155:1	`0x6C96...1dee`
Arbitrum One	eip155:42161	`0xFd08...Cbb9`
Optimism	eip155:10	`0x01bF...1071`
Ink	eip155:57073	`0x0200...70c1`
Berachain	eip155:80094	`0x779D...3736`
Unichain	eip155:130	`0x9151...EcC5`
Polygon	eip155:137	`0xc213...8e8F`
Mantle	eip155:5000	`0x779D...3736`
Plasma	eip155:9745	`0xB8CE...5ebb`
Sei	eip155:1329	`0x9151...EcC5`
Conflux eSpace	eip155:1030	`0xaf37...47ff`
Monad	eip155:143	`0xe7cd...C82D`
Rootstock	eip155:30	`0x779d...3736`
XLayer	eip155:196	`0x779D...3736`
Flare	eip155:14	`0xe7cd...C82D`
Corn	eip155:21000000	`0xB8CE...5ebb`
HyperEVM	eip155:999	`0xB8CE...5ebb`
MegaETH	eip155:4326	`0xb8ce...5ebb`
Stable	eip155:988	`0x779D...3736`

Full Contract Addresses

For complete contract addresses, query the Facilitator API at `https://facilitator.t402.io/supported`.

Table B.2: EVM Networks with Legacy USDT

Network	CAIP-2 ID	USDT Contract
BNB Chain	eip155:56	`0x55d3...7955`
Avalanche	eip155:43114	`0x9702...A8c7`
Fantom	eip155:250	`0x049d...3C7A`
Celo	eip155:42220	`0x4806...D5e`
Kaia	eip155:8217	`0xcee8...d167`

Table B.3: EVM Networks with USDC

Network	CAIP-2 ID	USDC Contract
Base	eip155:8453	`0x8335...2913`

B.2 Solana

Table B.4: Solana Network Configuration

Property	Value
CAIP-2 ID	`solana:5eykt4UsFv8P8NJdTREpY1vzqKqZKvdp`
USDT Mint	`Es9vMFrzaCERmJfrF4H2FYD4KCoNkY11McCe8BenwNYB`
Token Program	`TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA`
Decimals	6

B.3 TON

B.4 TRON

B.5 NEAR

B.6 Aptos

B.7 Tezos

B.8 Polkadot Asset Hub

B.9 Stacks

B.10 Cosmos

B.11 Facilitator Wallet Addresses

The official T402 Facilitator service uses the following addresses for gas sponsorship:

Table B.5: TON Network Configuration

Property	**Value**
CAIP-2 ID (Mainnet)	`ton:mainnet`
CAIP-2 ID (Testnet)	`ton:testnet`
USDT Jetton Master (Mainnet)	`EQCxE6mUtQJKFnGfaROTKOt1lZbDiiX1kCixRv7Nw2Id_sDs`
USDT Jetton Master (Testnet)	`kQD0GKBM8ZbryVk2aESmzfU6b9b_8era_IkvBSELujFZPsyy`
Workchain	0 (basechain)
Decimals	6

Table B.6: TRON Network Configuration

Property	**Value**
CAIP-2 ID (Mainnet)	`tron:mainnet`
CAIP-2 ID (Testnet)	`tron:nile`
USDT Contract (Mainnet)	`TR7NHqjeKQxGTCi8q8ZY4pL8otSzgjLj6t`
USDT Contract (Nile)	`TXYZopYRdj2D9XRtbG411XZZ3kM5VkAeBf`
Decimals	6

Table B.7: NEAR Network Configuration

Property	**Value**
CAIP-2 ID (Mainnet)	`near:mainnet`
CAIP-2 ID (Testnet)	`near:testnet`
USDT Contract (Mainnet)	`usdt.tether-token.near`
USDT Contract (Testnet)	`usdt.fakes.testnet`
Standard	NEP-141
Decimals	6

Table B.8: Aptos Network Configuration

Property	**Value**
CAIP-2 ID (Mainnet)	`aptos:1`
CAIP-2 ID (Testnet)	`aptos:2`
USDT Metadata (Mainnet)	`0xf73e887a8754f540ee6e1a93bdc6dde2af69fc7ca5de32013e89dd4`
Standard	Fungible Asset
Decimals	6

Table B.9: Tezos Network Configuration

Property	**Value**
CAIP-2 ID (Mainnet)	`tezos:NetXdQprcVkpaWU`
CAIP-2 ID (Ghostnet)	`tezos:NetXnHfVqm9iesp`
USDt Contract (Mainnet)	`KT1XnTn74bUtxHfDtBmm2bGZAQfhPbvKWR8o`
Token ID	0
Standard	FA2 (TZIP-12)
Decimals	6

Table B.10: Polkadot Asset Hub Configuration

Property	**Value**
CAIP-2 ID (Asset Hub)	`polkadot:68d56f15f85d3136970ec16946040bc1`
CAIP-2 ID (Westend)	`polkadot:e143f23803ac50e8f6f8e62695d1ce9e`
USDT Asset ID	1984
Parachain ID	1000
Decimals	6

Table B.11: Stacks Network Configuration

Property	**Value**
CAIP-2 ID (Mainnet)	`stacks:1`
CAIP-2 ID (Testnet)	`stacks:2147483648`
USDT Contract	`SP3Y2ZSH8P7D50B0VBTSX11S7XSG24M1VB9YFQA4K.token-susdc`
Standard	SIP-010
Decimals	6
Bitcoin Finality	∼10 minutes

Network Updates

For the most current network support and addresses, refer to the Facilitator's `/supported` endpoint:

`https://facilitator.t402.io/supported`

Table B.12: Cosmos/Noble Network Configuration

Property	**Value**
CAIP-2 ID (Mainnet)	`cosmos:noble-1`
CAIP-2 ID (Testnet)	`cosmos:grand-1`
USDT Asset	`uusdt`
Standard	Native token
Decimals	6

Table B.13: Official Facilitator Addresses

Chain	**Address**
EVM (all networks)	`0xC88f67e776f16DcFBf42e6bDda1B82604448899B`
Solana	`8GGtWHRQ1wz5gDKE2KXZLktqzcfV1CBqSbeUZjA7hoWL`
TON (mainnet)	`EQDjv9CUEJ__D_3-3J4trQtqVklMBiNoGVSf3Fu6AaDGkEUe`
TON (testnet)	`kQDjv9CUEJ__D_3-3J4trQtqVklMBiNoGVSf3Fu6AaDGkP6U`
TRON	`TT1MqNNj2k5qdGA6nrrCodW6oyHbbAreQ5`
Stacks	`SP36B1B191JTQAZTRKKWRN7J0YHHM41W9P9P7EPR5`
Cosmos/Noble	`noble1ejc2c2gvk46h7kyulx9fus85vdpq0zdjwkfav0`

Appendix C

Error Code Reference

This appendix provides a complete reference of T402 error codes, organized by category. Each error includes a machine-readable code, human-readable description, and recommended resolution.

C.1 Payment Errors

Payment errors occur during the payment authorization phase, typically due to insufficient funds, invalid signatures, or parameter mismatches.

Table C.1: Payment Error Codes

Code	Description	Resolution
`insufficient_funds`	Payer's token balance is less than the required payment amount	Fund wallet with sufficient USDT/USDT0
`insufficient_allowance`	Token allowance to facilitator is insufficient (EVM only)	Approve token spending or use EIP-3009
`invalid_signature`	Cryptographic signature verification failed	Re-sign with correct private key
`signature_mismatch`	Recovered signer does not match claimed payer	Ensure signing key matches payer address
`invalid_amount`	Payment amount is below the required minimum	Increase amount to meet requirements
`amount_overflow`	Payment amount exceeds maximum safe value	Use smaller payment amount
`invalid_recipient`	Recipient address does not match payTo in requirements	Use exact payTo address from requirements
`recipient_mismatch`	Payment directed to wrong recipient	Verify payTo address before signing
`expired_authorization`	Authorization deadline has passed	Create new authorization with future deadline
`deadline_too_short`	Deadline is too close to current time	Set deadline at least 60 seconds in future

Table C.1 – continued

Code	Description	Resolution
`deadline_too_long`	Deadline exceeds maximum allowed window	Set deadline within 1 hour of current time
`nonce_already_used`	Nonce has been used in a previous transaction	Generate new random 32-byte nonce
`nonce_invalid_format`	Nonce does not match expected format	Use valid hex-encoded 32-byte value
`invalid_payer`	Payer address is malformed or invalid	Use valid address for the target network
`payer_blacklisted`	Payer address is on USDT blacklist	Use different wallet address
`invalid_asset`	Asset address does not match expected token	Use correct USDT/USDT0 contract address

C.2 Protocol Errors

Protocol errors indicate issues with the T402 request format, unsupported features, or version incompatibilities.

Table C.2: Protocol Error Codes

Code	Description	Resolution
`invalid_network`	Specified network is not supported by facilitator	Query `/supported` for available networks
`network_mismatch`	Payment network does not match requirements	Use network specified in PaymentRequired
`invalid_scheme`	Payment scheme is not supported	Use `exact` or `upto` scheme
`scheme_mismatch`	Payment scheme does not match requirements	Use scheme specified in PaymentRequired
`invalid_t402_version`	Protocol version is not supported	Upgrade client to support v2
`version_mismatch`	Payload version incompatible with server	Match version in requirements
`invalid_payload`	Payment payload is malformed or incomplete	Validate JSON structure against schema
`payload_too_large`	Payload exceeds maximum allowed size	Reduce payload size (max 64KB)
`invalid_payment_req`	Payment requirements are invalid	Server configuration error; contact provider
`missing_required_field`	Required field is missing from payload	Include all required fields per spec
`invalid_field_format`	Field value does not match expected format	Validate field formats against schema
`unsupported_extension`	Requested extension is not available	Remove unsupported extension from request

Table C.2 – continued

Code	Description	Resolution
`invalid_resource_info`	ResourceInfo structure is malformed	Validate ResourceInfo JSON structure

C.3 Network-Specific Errors

Each supported blockchain network may return specific errors related to its unique characteristics.

C.3.1 EVM Errors

Table C.3: EVM-Specific Error Codes

Code	Description	Resolution
`evm_invalid_chain_id`	Chain ID does not match network	Use correct chain ID for target network
`evm_contract_not_found`	Token contract not deployed on network	Verify network supports USDT0/USDT
`evm_eip3009_not_supported`	Token does not support EIP-3009	Use network with EIP-3009 compatible token
`evm_domain_separator_mismatch`	EIP-712 domain separator invalid	Verify contract address and chain ID
`evm_gas_estimation_failed`	Unable to estimate gas for transaction	Check contract state and parameters
`evm_nonce_too_low`	On-chain nonce higher than provided	Use fresh nonce from contract
`evm_revert`	Contract execution reverted	Check revert reason in transaction receipt

C.3.2 Solana Errors

Table C.4: Solana-Specific Error Codes

Code	Description	Resolution
`svm_invalid_pubkey`	Public key is not valid Ed25519	Use valid Solana public key
`svm_ata_not_found`	Associated Token Account does not exist	Create ATA before payment
`svm_ata_creation_failed`	Failed to create Associated Token Account	Ensure payer has SOL for rent
`svm_blockhash_expired`	Recent blockhash has expired	Fetch fresh blockhash and re-sign
`svm_signature_invalid`	Ed25519 signature verification failed	Re-sign with correct keypair

Table C.4 – continued

Code	Description	Resolution
`svm_program_error`	SPL Token program returned error	Check program error code

C.3.3 TON Errors

Table C.5: TON-Specific Error Codes

Code	Description	Resolution
`ton_invalid_address`	Address is not valid TON format	Use valid bounceable/non-bounceable address
`ton_jetton_wallet_not_found`	Jetton wallet not deployed	Deploy Jetton wallet first
`ton_query_id_used`	query_id already used in prior message	Generate new unique query_id
`ton_insufficient_ton`	Insufficient TON for gas fees	Fund wallet with TON for fees
`ton_message_expired`	Message validity period expired	Create new message with fresh timestamp
`ton_workchain_mismatch`	Address workchain does not match	Use basechain (workchain 0) addresses

C.3.4 TRON Errors

Table C.6: TRON-Specific Error Codes

Code	Description	Resolution
`tron_invalid_address`	Address is not valid Base58Check format	Use valid T-prefixed address
`tron_bandwidth_exceeded`	Account bandwidth limit exceeded	Wait or freeze TRX for bandwidth
`tron_energy_insufficient`	Insufficient energy for contract call	Freeze TRX for energy or pay fees
`tron_signature_invalid`	ECDSA signature verification failed	Re-sign with correct private key
`tron_contract_error`	TRC-20 contract returned error	Check contract error message

C.3.5 NEAR Errors

Table C.7: NEAR-Specific Error Codes

Code	Description	Resolution
near_account_not_found	Account ID does not exist	Create account before payment
near_invalid_account_id	Account ID format is invalid	Use valid NEAR account ID format
near_not_registered	Account not registered with token	Call storage_deposit first
near_storage_insufficient	Insufficient storage deposit	Add storage deposit to account
near_access_key_invalid	Access key not valid for account	Use correct access key
near_gas_exceeded	Transaction exceeded gas limit	Reduce transaction complexity

C.3.6 Aptos Errors

Table C.8: Aptos-Specific Error Codes

Code	Description	Resolution
aptos_account_not_found	Account does not exist on-chain	Create account with initial funding
aptos_invalid_address	Address is not valid 32-byte hex	Use valid 0x-prefixed address
aptos_sequence_number_invalid	Sequence number already used	Fetch current sequence number
aptos_fa_store_not_found	Fungible Asset store not found	Register for asset first
aptos_gas_insufficient	Insufficient APT for gas fees	Fund account with APT
aptos_module_not_found	Move module not deployed	Verify contract deployment

C.3.7 Tezos Errors

Table C.9: Tezos-Specific Error Codes

Code	Description	Resolution
tezos_invalid_address	Address is not valid tz/KT format	Use valid Tezos address format
tezos_counter_invalid	Operation counter already used	Fetch current counter from chain
tezos_balance_too_low	Insufficient XTZ for fees	Fund account with XTZ
tezos_fa2_not_operator	Facilitator not approved as operator	Call update_operators first
tezos_token_undefined	Token ID not defined in contract	Use correct token ID (0 for USDt)

Table C.9 – continued

Code	Description	Resolution
`tezos_gas_exhausted`	Operation exceeded gas limit	Reduce operation complexity

C.3.8 Polkadot Errors

Table C.10: Polkadot Asset Hub Error Codes

Code	Description	Resolution
`dot_invalid_address`	SS58 address is invalid	Use valid SS58 address format
`dot_asset_not_found`	Asset ID does not exist	Use correct asset ID (1984 for USDT)
`dot_account_not_found`	Account has no existential deposit	Fund account with minimum DOT
`dot_frozen_balance`	Asset balance is frozen	Unfreeze balance or use different account
`dot_nonce_invalid`	Transaction nonce already used	Fetch current nonce from chain
`dot_xcm_failed`	Cross-chain message failed	Check XCM routing configuration

C.3.9 Stacks Errors

Table C.11: Stacks-Specific Error Codes

Code	Description	Resolution
`stx_invalid_address`	Address is not valid Stacks format	Use valid SP/ST-prefixed address
`stx_nonce_invalid`	Transaction nonce incorrect	Fetch current nonce from API
`stx_fee_insufficient`	Transaction fee too low	Increase fee for faster confirmation
`stx_contract_not_found`	SIP-010 contract not deployed	Verify contract address
`stx_clarity_error`	Clarity contract returned error	Check contract error code
`stx_anchor_block_pending`	Waiting for Bitcoin anchor block	Wait for Bitcoin finality

C.4 Settlement Errors

Settlement errors occur when attempting to execute the on-chain transaction after verification succeeds.

Table C.12: Settlement Error Codes

Code	Description	Resolution
`simulation_failed`	Transaction simulation failed	Check balance, allowance, parameters
`settlement_failed`	On-chain transaction failed	Check transaction hash for details
`settlement_timeout`	Settlement did not confirm in time	Check transaction status manually
`settlement_reverted`	Transaction confirmed but reverted	Check revert reason in receipt
`gas_price_too_low`	Gas price below network minimum	Increase gas price and retry
`gas_limit_exceeded`	Transaction exceeded gas limit	Optimize transaction or split
`facilitator_balance_low`	Facilitator lacks gas funds	Contact T402 support
`network_congestion`	Network too congested for settlement	Retry during lower congestion
`rpc_error`	RPC node returned error	Retry or use alternate RPC
`rpc_timeout`	RPC request timed out	Retry with longer timeout
`unexpected_verify_error`	Unexpected error during verification	Check logs, contact support
`unexpected_settle_error`	Unexpected error during settlement	Check logs, contact support

C.5 HTTP Transport Errors

HTTP-specific errors related to the transport layer when using the HTTP/402 protocol.

Table C.13: HTTP Transport Error Codes

Code	Description	Resolution
`missing_payment_header`	PAYMENT-SIGNATURE header not present	Include signed payload in header
`invalid_payment_header`	PAYMENT-SIGNATURE header malformed	Base64url encode JSON payload
`missing_accept_header`	Client does not accept payment types	Include Accept header with payment types
`header_too_large`	Payment header exceeds size limit	Reduce payload size
`invalid_content_type`	Request Content-Type invalid	Use application/json

C.6 Facilitator Errors

Errors specific to the T402 Facilitator service.

Table C.14: Facilitator Error Codes

Code	Description	Resolution
`rate_limit_exceeded`	Too many requests from client	Implement backoff and retry
`api_key_invalid`	API key is not valid	Use valid API key
`api_key_expired`	API key has expired	Renew API key
`api_key_required`	Endpoint requires API key	Include X-API-Key header
`facilitator_unavailable`	Facilitator service is down	Retry later or use fallback
`maintenance_mode`	Facilitator in maintenance	Wait for maintenance to complete
`network_disabled`	Network temporarily disabled	Use alternate network
`internal_error`	Internal server error	Retry or contact support

C.7 Error Response Format

All T402 errors are returned in a consistent JSON format:

```json
{
  "error": {
    "code": "insufficient_funds",
    "message": "Payer balance (1.50 USDT) is less than required (5.00 USDT)",
    "details": {
      "required": "5000000",
      "available": "1500000",
      "asset": "0x6C96dE32CEa08842dcc4058c14d3aaAD7Fa41dee",
      "network": "eip155:1"
    }
  }
}
```

Listing C.1: Error Response Structure

- **code**: Machine-readable error code from tables above
- **message**: Human-readable description
- **details**: Optional object with error-specific context

Appendix D

Glossary

This glossary provides definitions for technical terms used throughout the T402 whitepaper, organized alphabetically.

A

A2A (Agent-to-Agent)
Protocol for direct communication between AI agents, enabling autonomous agent-to-agent payments without human intervention. T402 supports A2A as a transport layer.

Account Abstraction
Blockchain design pattern that separates transaction signing from account ownership, enabling features like gasless transactions, social recovery, and multi-signature wallets. ERC-4337 is the primary standard for EVM chains.

APT
Native token of the Aptos blockchain, used for gas fees and staking.

Aptos
Layer-1 blockchain using Move language for smart contracts, developed by former Meta engineers. Features parallel transaction execution and the Fungible Asset standard for tokens.

Asset Hub
Polkadot system parachain dedicated to asset management, where USDT is deployed as asset ID 1984.

ATA (Associated Token Account)
Solana account derived deterministically from a wallet address and token mint address. Required for holding SPL tokens; created automatically by the facilitator if needed.

B

Base58Check
Encoding format used by TRON for addresses, featuring a T-prefix and built-in checksum for error detection.

Blockhash
Solana's mechanism for transaction freshness, requiring a recent blockhash (within 2 minutes) to prevent replay attacks.

Bounceable Address
TON address format that returns funds if the destination contract cannot process the message. Used for smart contracts.

C

CAIP-2
: Chain Agnostic Improvement Proposal 2, a standard format for blockchain network identifiers. Format: `namespace:reference` (e.g., `eip155:8453` for Base, `solana:5eykt...Kvdp` for Solana mainnet).

CAIP-10
: Chain Agnostic Improvement Proposal 10, extending CAIP-2 to include account addresses. Format: `namespace:reference:address`.

Clarity
: Smart contract language for the Stacks blockchain, designed for predictability and security with decidable execution.

Cosmos/Noble
: Cosmos ecosystem blockchain focused on native asset issuance. Noble hosts native USDT with the `uusdt` denomination. Uses CAIP-2 identifiers `cosmos:noble-1` (mainnet) and `cosmos:grand-1` (testnet).

Client
: Any application, service, or AI agent that requests access to protected resources and submits payment authorizations. Clients sign payments off-chain without needing native tokens for gas.

D

Deadline
: Unix timestamp specifying when a payment authorization expires. Prevents stale authorizations from being settled after the client's intent has lapsed.

DOT
: Native token of the Polkadot network, used for governance, staking, and parachain bonding.

E

EIP-712
: Ethereum Improvement Proposal for typed structured data signing. Provides human-readable signing prompts in wallets, showing exactly what is being authorized rather than opaque hex data.

EIP-3009
: Ethereum Improvement Proposal enabling gasless token transfers via cryptographic signatures. The payer signs an authorization; a third party (facilitator) submits and pays for the transaction.

ERC-20
: Ethereum token standard defining a common interface for fungible tokens. USDT on Ethereum follows this standard.

ERC-4337
: Ethereum account abstraction standard enabling smart contract wallets with features like gasless transactions, batched operations, and session keys.

Exact Scheme
: The primary T402 payment scheme where the payment amount exactly matches the required amount. The most common and simplest scheme.

Existential Deposit
Minimum balance required to keep an account alive on Substrate-based chains like Polkadot. Below this threshold, the account is reaped.

F

FA2 (TZIP-12)
Tezos token standard supporting fungible, non-fungible, and multi-asset tokens. USDt on Tezos is an FA2 token with token ID 0.

Facilitator
Central service in T402 that handles payment verification and on-chain settlement. Maintains hot wallets on all supported networks, sponsors gas fees, and provides unified API for clients and servers.

Finality
The point at which a blockchain transaction becomes irreversible. Varies by network: Solana (400ms), TON (5s), EVM (12s-15min), Stacks (10min via Bitcoin).

Fungible Asset
Aptos's modern token standard replacing legacy Coin module. USDT on Aptos uses the Fungible Asset standard.

G

Gas Computational unit measuring execution cost on blockchain networks. T402 eliminates gas concerns for payers by having the facilitator sponsor all gas fees.

Gasless Transaction
Transaction where the end user does not pay gas fees. In T402, the facilitator pays gas using EIP-3009 (EVM), meta-transactions, or similar mechanisms.

Ghostnet
Tezos permanent testnet, identified by CAIP-2 as `tezos:NetXnHfVqm9iesp`.

H

Hot Wallet
Cryptocurrency wallet connected to the internet for immediate transaction execution. The facilitator maintains hot wallets for gas sponsorship.

HTTP 402
HTTP status code "Payment Required," originally reserved in HTTP/1.1. T402 implements the intended semantics: the server requires payment before granting access.

I–J

Jetton
TON blockchain's fungible token standard, analogous to ERC-20. USDT on TON is implemented as a Jetton with separate wallet contracts per holder.

Jetton Wallet
Per-user contract on TON that holds a specific Jetton balance. Created automatically when receiving Jettons for the first time.

K–L

LayerZero
Cross-chain messaging protocol enabling tokens to exist natively on multiple chains. USDT0 uses LayerZero's Omnichain Fungible Token (OFT) standard.

M

MCP (Model Context Protocol)
Anthropic's protocol enabling AI models to interact with external tools and resources. T402 provides an MCP server allowing AI agents to make payments.

Mechanism
In T402, the combination of a payment scheme and network implementation. Example: "exact scheme on EVM" is a mechanism.

Meta-transaction
Transaction signed by one party but submitted and paid for by another. Core enabler of gasless user experiences.

Move
Smart contract language used by Aptos (and Sui), featuring resource-oriented programming and formal verification support.

N

NEAR
Layer-1 blockchain using sharding for scalability. Features human-readable account names and the NEP-141 token standard.

NEP-141
NEAR Enhancement Proposal 141, the fungible token standard for NEAR Protocol. USDT on NEAR follows this standard.

Nile Testnet
TRON's test network for development, identified by CAIP-2 as `tron:nile`.

Non-bounceable Address
TON address format that does not return funds if delivery fails. Used for externally owned accounts (wallets).

Nonce
Number used once to prevent replay attacks. In T402, a cryptographically random 32-byte value included in payment authorizations.

O

OFT (Omnichain Fungible Token)
LayerZero token standard enabling native cross-chain transfers without bridges. USDT0 is Tether's OFT implementation.

Off-chain Signing
Creating cryptographic signatures without submitting transactions to the blockchain. T402 clients sign payment authorizations off-chain.

On-chain Settlement
Executing the actual token transfer on the blockchain. In T402, the facilitator performs settlement after verifying signatures.

Operator
In Tezos FA2, an address authorized to transfer tokens on behalf of the owner. The facilitator must be added as an operator.

P

Parachain
Independent blockchain connected to the Polkadot relay chain, benefiting from shared security. Asset Hub is Polkadot's system parachain for tokens.

PaymentPayload
JSON structure containing a signed payment authorization sent by clients in the `PAYMENT-SIGNATURE` header. Includes payer, signature, and authorization details.

PaymentRequired
JSON structure returned by servers in 402 responses, describing acceptable payment methods including network, scheme, amount, and recipient address.

Polkadot
Heterogeneous multi-chain protocol enabling cross-chain communication. USDT is deployed on the Asset Hub parachain.

Q

query_id
TON's mechanism for transaction identification, a 64-bit value ensuring message uniqueness and enabling response correlation.

R

Replay Attack
Reusing a valid transaction or authorization to execute it multiple times. T402 prevents replays via nonces and deadlines.

Resource Server
Service providing protected resources (APIs, content, compute) that require payment for access. Integrates T402 middleware.

ResourceInfo
Optional metadata in T402 v2 describing the protected resource, enabling client UIs to display meaningful payment context.

S

Scheme
Payment logic defining how amounts are calculated and validated. T402 supports "exact" (fixed amount) and "up-to" (metered/variable, available on Solana, TON, TRON, NEAR).

Settlement
Process of executing a payment transaction on the blockchain, transferring tokens from payer to recipient.

SIP-010
Stacks Improvement Proposal 010, the fungible token standard for the Stacks blockchain, analogous to ERC-20.

SPL Token
Solana Program Library token standard for fungible and non-fungible tokens on Solana. USDT on Solana is an SPL token.

SS58
Substrate-based address format used by Polkadot, featuring a network prefix and checksum.

Stablecoin
Cryptocurrency designed to maintain a stable value relative to a reference asset, typically USD. USDT is the largest stablecoin by market capitalization.

Stacks
Bitcoin Layer-2 network enabling smart contracts secured by Bitcoin's proof-of-work through its Proof of Transfer consensus.

Storage Deposit
NEAR's mechanism for paying storage costs, required before an account can hold tokens. Part of NEP-141 registration.

T

Tezos
Self-amending blockchain with formal verification support. Uses the FA2 (TZIP-12) standard for tokens including USDt.

TL-B
Type Language - Binary, TON's binary serialization format for message encoding.

TON (The Open Network)
Layer-1 blockchain originally developed by Telegram, featuring high throughput and the Jetton token standard.

TRC-20
TRON blockchain's fungible token standard, modeled after ERC-20. USDT on TRON follows this standard.

Transport
In T402, the communication layer for exchanging payment data. Supported transports: HTTP (402 status), MCP (AI tools), A2A (agent communication).

TRON
High-throughput blockchain with delegated proof-of-stake consensus. Hosts the largest USDT circulation.

TRX
Native token of the TRON network, used for bandwidth, energy, and governance.

U

Up-To Scheme
T402 payment scheme for streaming or variable amounts, where the client authorizes up to a maximum and the server settles the actual consumed amount. Implemented for Solana, TON, TRON, and NEAR.

USDT
Tether USD, the most widely used stablecoin, maintaining approximate 1:1 parity with the US dollar. Available on 12+ blockchains.

USDT0
Tether's omnichain USDT implementation using LayerZero OFT standard, enabling native transfers across 19+ EVM chains.

USDt
Tether USD on Tezos, using the FA2 standard. Note the lowercase "t" distinguishing it from USDT on other chains.

V

Verification
Process of validating a payment signature and parameters before settlement. Checks signature validity, balance sufficiency, nonce freshness, and deadline.

V1 (Protocol Version 1)
Legacy T402 format using simple chain identifiers. Still supported for backwards compatibility.

V2 (Protocol Version 2)
Current T402 format using CAIP-2 network identifiers, ResourceInfo, and extension support.

W

Wallet
Software or hardware managing private keys and enabling blockchain interactions. In T402, client wallets sign payment authorizations.

WDK (Wallet Development Kit)
Tether's toolkit for building wallets with USDT/USDT0 support. T402 provides WDK integration packages.

Westend
Polkadot's canary network for testing, with Asset Hub at CAIP-2 `polkadot:e143f238...1ce9e`.

Workchain
TON's parallel blockchain shards. Workchain 0 (basechain) handles standard transactions and smart contracts.

X–Z

PAYMENT-SIGNATURE
HTTP header (V2) containing the base64url-encoded payment payload in T402 requests. V1 alias: `X-PAYMENT`.

PAYMENT-REQUIRED
HTTP header (V2) containing base64url-encoded payment requirements in 402 responses. V1 alias: `X-PAYMENT-REQUIRED`.

PAYMENT-RESPONSE
HTTP header (V2) containing settlement confirmation details after successful payment. V1 alias: `X-PAYMENT-RESPONSE`.

XCM (Cross-Consensus Messaging)
Polkadot's protocol for cross-chain communication between parachains and with external networks.

XTZ
Native token of the Tezos blockchain, used for gas fees, staking, and governance.

Bibliography

[1] Anthropic. Model Context Protocol Specification. Protocol Specification, 2024.

[2] Remco Bloemen, Leonid Logvinov, and Jacob Evans. EIP-712: Ethereum typed structured data hashing and signing. Ethereum Improvement Proposal, 2017.

[3] Vitalik Buterin et al. EIP-4337: Account Abstraction Using Alt Mempool. Ethereum Improvement Proposal, 2021.

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

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

[6] Stacks Foundation. Stacks Documentation. Technical Documentation, 2021.

[7] Tezos Foundation. Tezos Developer Documentation. Technical Documentation, 2018.

[8] TON Foundation. TON Blockchain Documentation. Technical Documentation, 2023.

[9] TRON Foundation. TRON Protocol Documentation. Technical Documentation, 2017.

[10] Web3 Foundation. Polkadot Wiki. Technical Documentation, 2020.

[11] IETF. JSON Schema: A Media Type for Describing JSON Documents. Internet-Draft, 2020.

[12] S. Josefsson. RFC 4648: The Base16, Base32, and Base64 Data Encodings. RFC 4648, 2006.

[13] Peter Jihoon Kim, Kevin Britz, and David Knott. EIP-3009: Transfer With Authorization. Ethereum Improvement Proposal, 2020.

[14] Aptos Labs. Aptos Developer Documentation. Technical Documentation, 2022.

[15] LayerZero Labs. LayerZero V2 Documentation. Technical Documentation, 2024.

[16] Solana Labs. Solana Documentation. Technical Documentation, 2020.

[17] Uniswap Labs. Permit2. Smart Contract, 2022.

[18] NEAR Protocol. NEAR Protocol Documentation. Technical Documentation, 2020.

[19] NEAR Protocol. NEP-141: Fungible Token Standard. NEAR Enhancement Proposal, 2020.

[20] Stacks. SIP-010: Standard Trait Definition for Fungible Tokens. Stacks Improvement Proposal, 2021.

[21] Parity Technologies. Asset Hub Parachain. Technical Documentation, 2022.

[22] Tether. Tether Whitepaper. Technical Whitepaper, 2014.

[23] Tezos. FA2: Multi-Asset Interface. Token Standard, 2020.