Payment Coordination Patterns for MCP: A Design Taxonomy, Reference Wrapper, and Compatibility Snapshot

Eugene Lobachev   Prasanna Kumar   Arjun Subedi   Bishnu Bista

5 January 2026

Abstract

The Model Context Protocol (MCP) standardizes communication between large language model (LLM) clients and server-hosted tools, enabling flexible tool invocation. However, the introduction of payment tool execution raises a key coordination problem: the challenge of ensuring seamless payment validation between client and server without degrading user experience and affecting the reliability of LLM tool selection.

We categorize payment coordination patterns for MCP–TWO_STEP, RESUBMIT, X402, ELICITATION, PROGRESS, DYNAMIC_TOOLS, and an emerging URL_ELICITATION variant – identifying their assumptions, failure modes, and security/UX trade-offs. Originally, MCP tools followed a single-shot execution model; payment gating was later introduced by partitioning execution into request and confirm phases. New MCP features, including elicitation, progress updates, and dynamic tool lists, extend this design space, but are not yet uniformly implemented across clients.

Our contributions are threefold: (1) a capability-aligned taxonomy of payment coordination patterns, (2) a reference wrapper that transparently enforces payment gating for existing developer tools, and (3) a compatibility snapshot based on declared client capabilities and observed initialize payloads, with practical notes on how these features tend to surface in common MCP clients.

This work focuses on engineering design for AI systems rather than payment economics. Our aim is to help practitioners integrate paywalls into MCP safely, predictably, and with a better user experience.

1 Introduction

The Model Context Protocol (MCP) standardizes how LLM applications (clients) discover and invoke server-hosted tools, exchanging context over a JSON-RPC-based session with explicit capability negotiation. As MCP expands across IDEs and chat interfaces, developers increasingly expose paid tools whose execution must be gated upon successful payment. The central challenge is orchestrating this gate so that the MCP client, server, and human user remain in sync—without destabilizing the LLM's tool selection or degrading usability.

We use the term *payment coordination pattern* to denote how an MCP server and client cooperate to request, verify, and acknowledge payment during a tool's lifecycle. We adopt a neutral protocol-level framework and analyze the patterns independently of any single SDK.

Naming note: pattern labels (TWO_STEP, RESUBMIT, X402, ELICITATION, PROGRESS, DYNAMIC_TOOLS, URL_ELICITATION) are author-proposed taxonomy terms rather than official MCP specification names; each maps either to standard MCP features (for example, elicitation, progress, and `tools/listChanged`) or to an external protocol name (for example, X402).

Historically, MCP tools were executed as single-run functions: they could not pause mid-execution to ask for additional input or a payment. A common retrofit is TWO_STEP, which splits a paid operation into (i) a "request" tool that returns a payment link and (ii) a "confirm" tool that checks status before performing the actual work. Although simple and widely compatible, TWO_STEP can stress the selection of LLM tools as users connect more servers and tools; models can pick the wrong tool or hallucinate intermediate steps or incorrect flows.

A related variant, RESUBMIT, returns a structured error (for example, indicating that payment is required) with a payment link and id on the first call; after payment, the caller resubmits with `payment_id` and the server executes. An on-chain variation, X402, follows the same high-level retry shape but returns an X402 payment request instead of a generic link and requires the caller (or an attached wallet) to resubmit with a cryptographic payment signature rather than a bare `payment_id`. This keeps a single-tool surface, but risks argument drift and depends on error payloads being visible or otherwise exposed to the payment-handling layer.

Recent MCP features enable seamless in-flow coordination. Elicitation lets servers pause a tool to request user input (for example, payment confirmation) and then resume, turning formerly single-shot tools into interactive workflows. Progress notifications provide long running status updates and may carry a human-readable message that can include a payment link. The DYNAMIC_TOOLS pattern uses `tools/listChanged` to allow servers to expose or hide follow-up tools (for example, `confirm_payment_{short_id}`) at specific points, steering the client and LLM toward the next required step. These capabilities expand the design space but are unevenly supported across clients: elicitation and dynamic tools are not yet universal, and many clients suppress the progress message text, limiting its usefulness for payment links.

An emerging URL_ELICITATION variant improves reliability by conveying links in a dedicated URL field that clients are expected to handle (for example, prompt the user and open a browser). However, the server still must learn completion state out-of-band (webhook or polling), making robust confirmation logic essential. The DYNAMIC_TOOLS approach also introduces UX risks (tools may disappear or reappear unexpectedly) and caching concerns that can confuse both users and models.

Our contributions are threefold: (1) a capability-aligned taxonomy of MCP payment coordination patterns (TWO_STEP, RESUBMIT, X402, ELICITATION, PROGRESS, DYNAMIC_TOOLS, URL_ELICITATION) with assumptions, failure modes, and UX/security trade-offs; (2) a lightweight reference wrapper that converts existing developer tools into payment-gated tools without duplicating business logic; and (3) a compatibility snapshot and practical notes on model behavior in common chat clients. Finally, Section 7 summarizes cross-client support, and Table 2 outlines client-declared capabilities relevant to portability. Later sections discuss implications for production deployments. We target engineering design for AI systems, with connections to AI tooling and human–computer interaction; we do not study pricing models or economic modeling.

2 Background

The Model Context Protocol (MCP) [1] standardizes how AI applications (clients) connect to server-hosted capabilities such as tools, resources, and prompts using JSON-RPC over standard transports (stdio or streamable HTTP). During initialization, clients and servers negotiate capabilities and then exchange bidirectional typed requests, responses, and notifications across an established session. This session model is the substrate on which payment coordination patterns operate. See the official architecture overview [2] for a concise primer.

Tools are the primary execution primitive: servers declare the tools capability [3] and may advertise support for dynamic listings (via `notifications/tools/list_changed`) [6], which allows a server to notify the client that its available tools have changed. Clients typically discover tools

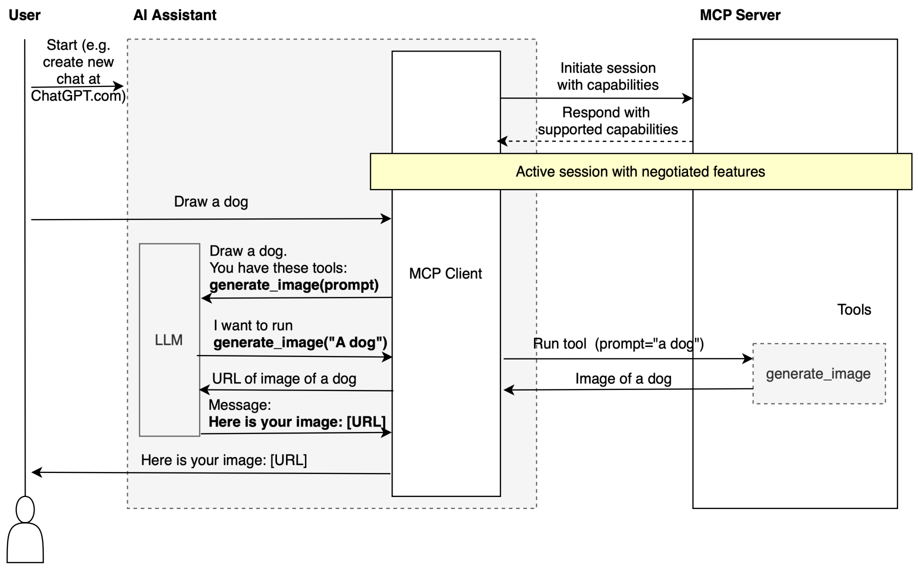
User
AI Assistant
MCP Server
Start (e.g. create new chat at ChatGPT.com)
Initiate session with capabilities
Respond with supported capabilities
Active session with negotiated features
Draw a dog
Draw a dog.
You have these tools:
generate_image(prompt)
MCP Client
Tools
LLM
I want to run
generate_image("A dog")
Run tool (prompt="a dog")
generate_image
URL of image of a dog
Image of a dog
Message:
Here is your image: [URL]
Here is your image: [URL]

Figure 1: Overview of an MCP client–server interaction for an example image tool. The server returns an image URL to the client; in practice some MCP clients may return data directly to the user (binary, JSON, or URL) without involving the LLM.

with `tools/list` [3] and then invoke them. Figure 1 illustrates a standard client–server interaction for a free tool.

MCP has introduced features that enable interactive flows inside tool execution. Elicitation [4] lets a server pause an in-flight tool to request additional user input (for example, confirmation or missing fields) and then resume, without forcing a separate tool call. The specification treats elicitation as a client capability and includes guidance on UX and safety.

For long-running operations, progress notifications [5] allow servers to report status increments and (optionally) attach a human-readable message; clients may choose how to display these updates (if the client surfaces message text).

To steer workflows, servers can use dynamic tools with `notifications/tools/list_changed` [6] so the client (and indirectly, the model's tool selection) sees only the next valid step—for example, briefly exposing a `confirm_payment_{short_id}` tool and then reverting to the original tool set after completion.

An emerging variant, URL_ELICITATION [9], conveys links in a dedicated `url` field for reliable, out-of-band actions (for example, opening a browser for OAuth or payment). Community proposals and SDK issues track this mode; servers still confirm completion via webhook or polling.

3 Related Work

Payment coordination for AI-powered tools intersects several research areas: protocol design for distributed systems, API monetization, and emerging AI agent architectures.

API monetization and metering. Traditional API monetization typically relies on authentication, metering, and billing at the HTTP/API layer. Platforms like Stripe and PayPal provide payment gateways that applications integrate directly into their request/response paths. However, these solutions assume direct client-server communication and do not account for an intermediary

LLM that selects and invokes tools on behalf of users. Our work addresses the unique challenge of coordinating payment when tool invocation is mediated by a language model.

MCP ecosystem and extensions. The Model Context Protocol [1] has rapidly evolved since its introduction, with community proposals for elicitation [4], progress notifications [5], and structured URL fields (SEP-1036) [9]. Security considerations have also emerged, including the SAFE-MCP framework [12] for threat modeling. Our taxonomy complements these efforts by focusing specifically on the payment coordination problem.

AI agent architectures. Recent work on autonomous AI agents explores how LLMs can plan and execute multi-step tasks [2]. As agents gain access to paid services, the question of payment authorization becomes critical. Unlike traditional software that executes deterministically, LLM-based agents may exhibit non-deterministic behavior, complicating payment flows. Our patterns address this by providing structured coordination mechanisms that account for model uncertainty.

Positioning of this work. To our knowledge, this is the first systematic taxonomy of payment coordination patterns for the MCP ecosystem. While prior work has addressed MCP security and capability negotiation, the specific problem of gating tool execution on payment verification—while preserving UX and model reliability—has not been formally analyzed.

Finally, the specification documents related building blocks—lifecycle (init/operate/shutdown), logging, cancellation, and evolving authorization guidance—which underpin robust client–server behavior and are relevant when discussing reliability and security considerations. See also the draft Security Best Practices [7] for additional guidance on notifications and cross-server interactions.

4 Taxonomy of Payment Coordination Patterns

We define a payment coordination pattern as a client–server protocol strategy that gates tool execution on successful payment while keeping the MCP client, the MCP server, and the human user in sync. Patterns are specified in terms of (i) intent, (ii) required client capabilities, (iii) mechanics (message or notification sequence), (iv) guarantees and failure modes, and (v) UX notes.

Notation. Let EXEC be the developer's original tool logic, PAID the authoritative payment state (as seen by the server), and CONFIRM the act of checking payment state (by polling or webhook-triggered lookup). Safety requires EXEC only after PAID. Liveness requires that under honest participants the system eventually reaches PAID $\rightarrow$ EXEC.

4.1 TWO_STEP (request $\rightarrow$ confirm)

Intent. Split the flow into two tools: (1) a request tool that returns {`payment_link`, `payment_id`}; and (2) a confirm tool that takes `payment_id`, verifies PAID, and then runs the original logic. Baseline pattern with broad compatibility.

Preconditions (client capabilities). None beyond standard tool invocation and result rendering.

Mechanics. `request_tool(args)` $\rightarrow$ {`payment_link`, `payment_id`}; persist {`args_snapshot`, user or session identifier, price, `created_at`} keyed by `payment_id` in durable storage; then `confirm_tool(payment_id)` $\rightarrow$ (PAID? $\rightarrow$ EXEC(`args_snapshot`) : pending). Implementations often wrap the original tool so developers do not duplicate logic.

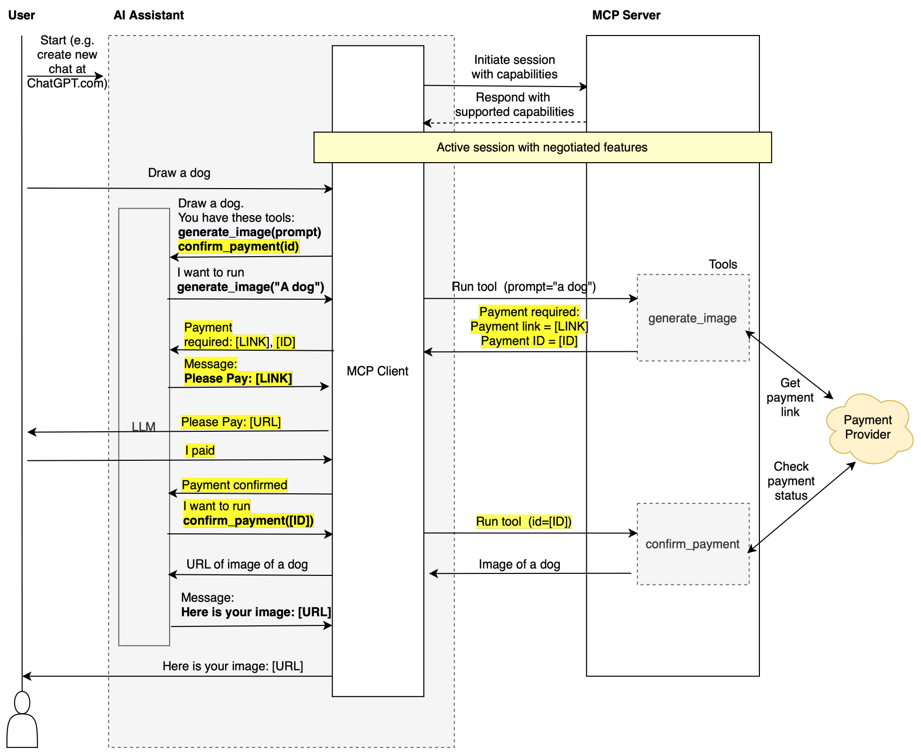

Figure 2: TWO_STEP coordination pattern: the server surfaces a payment link and requires an explicit confirmation call before executing the original logic.

Figure 2 illustrates this request-and-confirm interaction.

Guarantees and failure modes.

- Safety: strong, since EXEC is gated in `confirm_tool` and runs against the stored `args_snapshot`.
- Liveness: depends on the client surfacing the payment link and the user following it.
- Routing avoidance: models may try cheaper or free alternatives (web search, other tools, local code) unless explicitly instructed to use `request_tool` for paid operations.
- Premature confirm: models sometimes call `confirm_tool` immediately after `request_tool`; implementations typically return a structured pending status that clearly indicates user action is required (and re-issue the canonical link) rather than failing.
- Post-payment argument drift: if the user changes the ask after paying, EXEC typically runs with the captured `args_snapshot`; changes after payment introduce a mismatch between what was paid for and what is requested.
- Concurrency mix-ups: with multiple pending payments, the wrong job may run if calls are not uniquely associated with a `payment_id`.
- Paid-then-fail: EXEC can error after PAID; compensation or re-run policies may be required by the deployment.

UX notes. Extra cognitive steps; two tools appear in the catalog; works everywhere but scales poorly in multi-server setups. Expectation mismatch is common: payment authorizes execution of the captured `args_snapshot`, not later edits (for example, "elephant" paid but user switches to "giraffe"); communicate this explicitly. When `confirm_tool` returns "pending," surface the canonical link again and provide an explicit "I've paid" button (or equivalent UI action).

Mitigations (recommended).

- Tool descriptions: include price and an explicit instruction to use `request_tool` for paid operations; add a short example to reduce routing errors ("If payment is required, call `request_tool`, then `confirm_tool`").
- Bind and persist: store `args_snapshot` keyed by `payment_id`; verify at confirm; enforce single use; cap outstanding intents per user to avoid confusion in parallel requests.
- Premature confirm handling: return `PENDING` with the canonical link and a brief cooldown to reduce thrashing; log such attempts.
- Failure compensation: provide a cancel or refund path (or credit) when EXEC fails post-PAID; emit a user-facing status and a receipt.

4.2 RESUBMIT (error-based retry with payment ID)

Intent. Add an optional `payment_id` parameter to the tool. On the first call the caller omits `payment_id`; the server returns a structured MCP error that includes {`payment_link`, `payment_id`}. After the user pays, the caller re-invokes the same tool with `payment_id`; the server verifies PAID and then runs EXEC(`args`).

Preconditions. The client can surface structured error details (including `payment_link` and `payment_id`) and retry the same tool call with updated arguments. Ideally, error handling is implemented in the MCP client itself: display the payment link, await completion, and transparently resubmit with `payment_id` without invoking the LLM. When this is not available, it is sufficient that the client forwards the full structured error body to the LLM so the model can reliably surface the link and guide a correct retry.

Mechanics. `tool(args)` $\rightarrow$ structured error with {`payment_link`, `payment_id`}; user pays; `tool(args, payment_id)` $\rightarrow$ (PAID? $\rightarrow$ EXEC(`args`) : pending). Figure 3 illustrates this retry-based coordination.

Guarantees and failure modes.

- Safety: EXEC only runs after PAID is verified for the provided `payment_id`; the server remains the source of truth.
- Liveness: depends on callers (client or LLM) handling error correctly: surfacing the link, awaiting payment, and resubmitting with `payment_id`.
- Model-in-the-loop confusion: when the structured error is forwarded to the LLM, the model may (a) ask for a `payment_id` before any exists, (b) retry with a stale or hallucinated `payment_id`, or (c) call the tool with `payment_id` without ever showing the link to the user, leaving them stuck in a "payment pending" state.

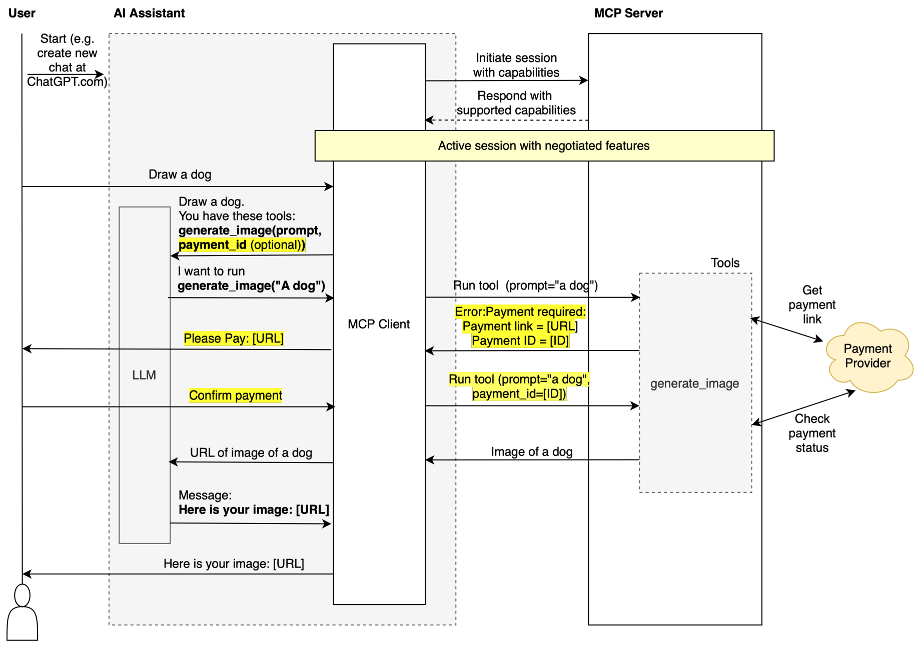
User
AI Assistant
MCP Server
Start (e.g. create new chat at ChatGPT.com)
Initiate session with capabilities
Respond with supported capabilities
Active session with negotiated features
Draw a dog
Draw a dog. You have these tools: generate_image(prompt, payment_id (optional))
I want to run generate_image("A dog")
Run tool (prompt="a dog")
Tools
Get payment link
MCP Client
Error:Payment required: Payment link = [URL] Payment ID = [ID]
Payment Provider
Please Pay: [URL]
LLM
Run tool (prompt="a dog", payment_id=[ID])
generate_image
Confirm payment
Check payment status
URL of image of a dog
Image of a dog
Message: Here is your image: [URL]
Here is your image: [URL]

Figure 3: RESUBMIT coordination pattern: the first call (without `payment_id`) returns a payment-required error with a canonical link and identifier; after payment, retrying the same tool with `payment_id` executes with the supplied arguments.

- Argument drift: arguments are not bound to `payment_id` by default in this pattern; the second call executes with whatever arguments are supplied, which may differ from what the user intended to pay for.
- Premature retry: callers may re-submit with `payment_id` before payment has settled, causing noisy pending states.
- Hidden or truncated error payloads: some clients strip or downplay structured error bodies; the LLM may see "payment required" without the `payment_link` or `payment_id`, resulting in vague or unhelpful guidance.
- Stale or mismatched identifiers: calls may include unknown, already-consumed, or cross-session `payment_id` values if the server does not enforce strict binding.

UX notes. RESUBMIT keeps a single-tool surface and fits naturally with clients that treat errors as an invitation to retry. The experience is strongly dependent on how error responses are propagated: if the client or LLM hides the link, improvises around the error, or retries without explanation, users may never see a clear "pay and retry" path.

Mitigations (recommended).

- Canonical schema: always return a structured error, machine-readable `payment_link` and `payment_id`, and a concise instruction ("Open this link to pay, then retry this tool including `payment_id`.").
- Client-first handling: when possible, implement error handling in the MCP client itself (display link, wait for completion, transparently resubmit) instead of delegating all coordination to the LLM.

- Single-use semantics: mark each `payment_id` as consumed once PAID and EXEC have succeeded; subsequent uses should be rejected or return an idempotent result to prevent double execution.
- Binding and validation: associate `payment_id` with `args_snapshot`, user or session, and price; reject mismatched or cross-context identifiers, and log anomalies.
- Retry control: apply minimal backoff or clear `PENDING` responses to discourage rapid-fire retries while payment is in flight.

4.3 X402 (on-chain payments)

Intent. Provide an on-chain variation of RESUBMIT using the X402 protocol [10], so that payment is cryptographically bound to a concrete payment request rather than mediated via a generic URL and opaque `payment_id`. Instead of returning a browser URL, the server returns a structured X402 payment request that encodes order details (amount, asset, description) and the payee's blockchain address. The pattern is designed primarily for automatic or agent-mediated payments (for example, an MCP client with an attached agent-controlled wallet), but it also supports user-approved flows where a human signs in an existing crypto wallet. The client or wallet signs the payment request and then retries the same tool call with a payment signature; the server verifies the signature and settles on-chain via a facilitator before running EXEC.

Preconditions. All RESUBMIT preconditions apply, plus the availability of an X402-capable payment provider and a client that understands X402 hand-offs.

We use *facilitator* to mean a payment service that turns a signed X402 request into an on-chain transaction and reports settlement under a configured confirmation policy. We use *provider* to denote the integration layer that creates/verifies X402 requests and exposes settlement status to the MCP server. A *settlement policy* is the confirmation rule (for example, mempool accepted vs. N confirmations) under which the facilitator/provider reports a request as paid.

In practice, the server must be able to emit an X402 payment request either (i) alongside an HTTP 402 status using a response header (for example, `Payment-Request` or `X-402-Payment`) or (ii) in the response body, while the MCP client (or a transport adapter) extracts this payload and exposes it to a wallet. (By *Payment-Request header* we mean the transport-layer header that carries the serialized X402 payment request.) After signing, the client retries the tool call with a payment signature that is conveyed either via HTTP headers (for example, `Payment-Signature` or `X-Payment`) or in MCP metadata (for example, `_meta["x402/payment"]`).

MCP is transport-agnostic and does not mandate HTTP status-code semantics or arbitrary header propagation at the protocol layer; whether 402/headers are visible depends on the transport adapter and client. X402 therefore assumes a specialized client or gateway that bridges between MCP and the underlying X402 transport. In practice, there are currently no general-purpose MCP chat or IDE clients that ship first-class X402 handling; existing deployments rely on custom clients or thin proxies that understand both MCP and X402. As a result, this pattern is best viewed as a specialized option for controlled environments rather than a default choice for broad public use.

In this pattern, a "payment signature" denotes a cryptographic signature over the X402 payment request that the facilitator can use to construct and submit an on-chain transaction on behalf of the payer; depending on the chain and wallet, this may be represented as a signed transaction payload.

Mechanics. The client first calls the paid tool without a payment signature: `tool(args)`. The server constructs an X402 payment request from the captured `args_snapshot` and returns a

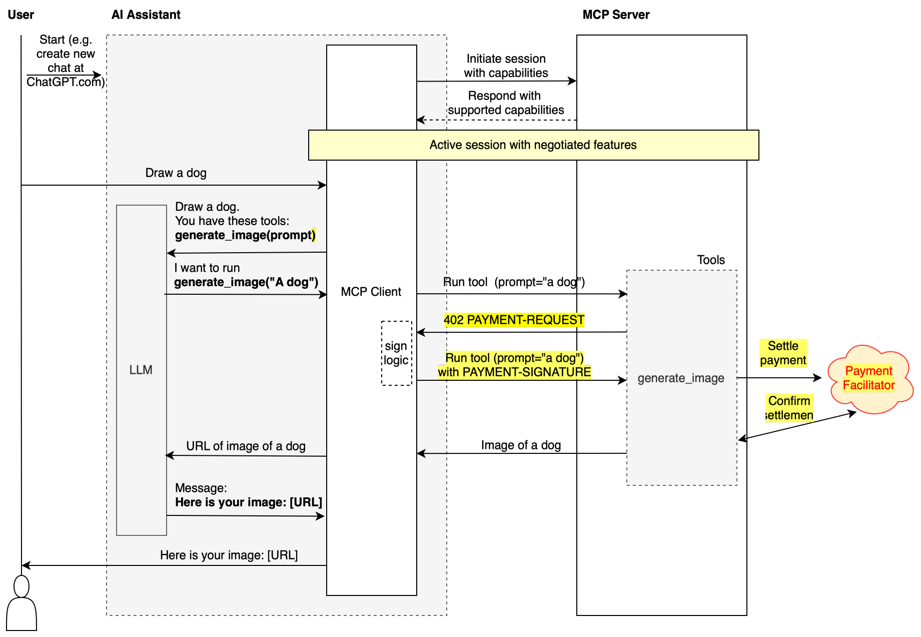
User
AI Assistant
MCP Server
Start (e.g. create new chat at ChatGPT.com)
Initiate session with capabilities
Respond with supported capabilities
Active session with negotiated features
Draw a dog
Draw a dog. You have these tools: generate_image(prompt)
I want to run generate_image("A dog")
MCP Client
LLM
Tools
Run tool (prompt="a dog")
402 PAYMENT-REQUEST
sign logic
Run tool (prompt="a dog") with PAYMENT-SIGNATURE
generate_image
Settle payment
Payment Facilitator
Confirm settlemen
URL of image of a dog
Image of a dog
Message: Here is your image: [URL]
Here is your image: [URL]

Figure 4: X402 coordination pattern: the first call returns a 402 Payment Required response carrying an X402 payment request; the client or wallet signs a transaction and then retries the same tool with `payment_signature`, after which the server verifies and executes with the original arguments.

structured failure that carries the payment request (often using HTTP status 402 on the underlying transport). The client obtains a payer signature on this request from a wallet (either automatically via an agent-controlled key or by prompting the user to sign in an existing crypto wallet) and then retries the same tool call with the payment signature attached (header or metadata). The server verifies the signature and, through an X402 provider or facilitator, settles the transaction on-chain or schedules settlement according to the configured confirmation policy; only after the facilitator reports the transaction as settled under that policy does the server execute EXEC(`args_snapshot`) and return the tool result.

Guarantees and failure modes.

- **Safety:** stronger argument binding than generic RESUBMIT when the X402 payment request commits to an `args_snapshot` hash, amount, and destination address. EXEC only runs after a valid payer signature has been verified and the provider reports PAID for the corresponding request.
- **Liveness:** depends on the availability of an X402-aware client, the responsiveness of the underlying chain, and wallet UX. Mempool congestion, confirmation delays, or facilitator outages can stall execution even when the MCP session remains healthy.
- **Transport mismatch:** if the MCP client ignores HTTP 402 status codes or drops server-set headers, it may never surface the X402 payment request to the wallet or user. In such environments X402 silently degrades into a permanent error state rather than a usable payment flow.
- **Signature channel variance:** different clients may choose headers versus metadata for returning the payment signature; misconfiguration or partial adoption can lead to EXEC

never running even though the user has signed the request.

- **Wallet trust and limits:** when an agent-controlled wallet signs automatically, bugs or prompt injection may cause unintended on-chain spend; when a user wallet is involved, complex signing prompts or repeated requests can cause confusion or consent fatigue.
- **Gas funding:** on-chain settlement requires gas. Deployments must decide whether the payer funds gas directly or a facilitator front-runs gas and recovers costs; misconfigured gas policies can cause paid-but-unsettled states.
- **Ecosystem maturity:** as of submission, X402 support in MCP tooling is limited to custom clients or gateways; generic MCP clients do not expose X402 payment requests directly, which constrains interoperability and discoverability.

UX notes. Like RESUBMIT, X402 preserves a single tool surface and avoids introducing extra confirm tools or explicit payment parameters; the visible tool catalog does not change between the pre- and post-payment calls. In practice the pattern works best when payments are largely automatic (agent-to-agent or agent-controlled wallet) and the MCP client takes responsibility for surfacing any human signing prompts and handling retries while keeping the LLM unaware of the underlying payment flow. The coordination logic is implemented in the MCP client and wallet layers: the LLM typically cannot inspect or manipulate the X402 payment request, and generic tool descriptions are insufficient to teach models how to complete the flow. On-chain confirmation may take seconds or longer, so users may perceive the tool as slow or flaky if progress is not surfaced clearly. Because error payloads are protocol-specific rather than natural language, debugging X402 failures can be harder than debugging plain RESUBMIT, especially in custom clients.

Mitigations (recommended).

- **Explicit binding:** include a hash of `args_snapshot`, price, and destination address in the X402 payment request; verify this binding before EXEC to prevent post-payment argument drift.
- **Client gating:** only enable X402 mode for clients known to be X402-aware (for example, via configuration or a declared capability flag); fall back to plain RESUBMIT when X402 headers or metadata are not supported.
- **Header/metadata abstraction:** hide the differences between header-based and metadata-based signatures behind a thin SDK layer so server-side logic sees a single `payment_signature` input; emit structured diagnostics when signatures are missing or invalid.
- **Spend controls:** enforce per-user and per-tool spend limits, daily caps, and amount thresholds that require explicit user confirmation even when an agent-controlled wallet is available.
- **Progress signaling:** consider sending progress notifications or similar status updates where the client reliably displays human-readable messages so users understand that (i) a payment is being prepared, (ii) a wallet prompt is expected, and (iii) on-chain confirmation may take time.
- **Fallback paths:** when settlement fails or times out, return a structured error that explains the failure and, where appropriate, suggests a non-on-chain alternative (for example, a traditional RESUBMIT flow or manual payment).

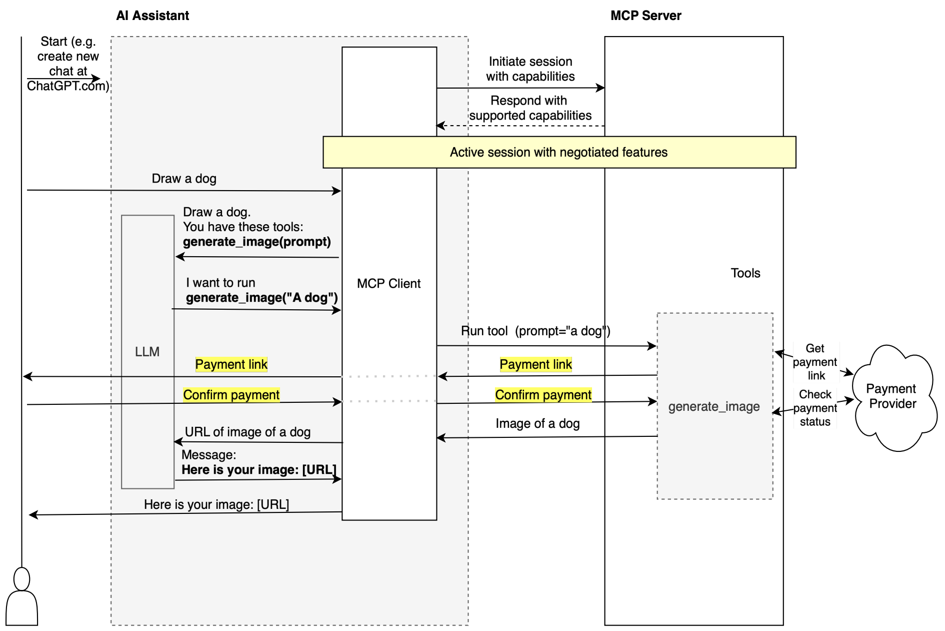
AI Assistant
MCP Server
Start (e.g. create new chat at ChatGPT.com)
Initiate session with capabilities
Respond with supported capabilities
Active session with negotiated features
Draw a dog
Draw a dog. You have these tools: generate_image(prompt)
I want to run generate_image("A dog")
MCP Client
Tools
LLM
Run tool (prompt="a dog")
Payment link
Payment link
Confirm payment
Confirm payment
Get payment link
Check payment status
Payment Provider
generate_image
Image of a dog
URL of image of a dog
Message: Here is your image: [URL]
Here is your image: [URL]

Figure 5: ELICITATION coordination pattern: the server pauses execution, prompts the user to complete payment, and resumes once payment is verified.

4.4 ELICITATION (in-flow request and confirm)

Intent. Before executing the main logic, create a payment intent and link, pause via elicitation, and present the user a minimal form (payment link plus “I've paid” button). After the user confirms and the server verifies PAID (webhook or poll), resume and run the original logic. This keeps a single-tool surface and tends to reduce LLM routing risk.

Preconditions. Client supports elicitation (server to client prompt during tool execution) and renders it inline; recommended: extendable or long timeouts and session restore across reconnects.

Mechanics. Tool invoked → server captures `args_snapshot` and creates {`payment_link`, `payment_id`} → server elicits a form {`url`, “I've paid” button} → client surfaces it; user pays and clicks confirm → server checks `PAID(payment_id)` via webhook or poll → (PAID? → resume EXEC(`args_snapshot`) : return `PENDING` and re-elicit).
The overall elicitation flow is shown in Figure 5.

Guarantees and failure modes.

- Safety: strong if the resume path re-checks PAID and executes against the stored `args_snapshot`.
- Liveness: high when the elicitation UI is rendered and timeouts are adequate.
- Uneven client support: some clients do not yet implement elicitation, so the flow may silently degrade.
- Elicitation timeout or abandonment: many clients impose short inactivity timeouts; payment may complete after the tool has errored. The user returns with “I've paid,” but the model has no in-flight context.

- Session restore: interrupted elicitations are common; when restore or reattach is unavailable, users and servers face recovery complexity and may lose context.
- Backgrounding or navigation: mobile or tab background throttling or navigation can drop the elicitation UI; if redisplay is not possible, users may miss the link and be unable to confirm.

UX notes. True single-tool surface (no extra confirm tool) with fewer routing errors than TWO_STEP, but portability depends on client maturity. Set expectations that users can pay and return; on resume, show clear states (awaiting payment → checking → resumed).

Mitigations (recommended).

- Timeouts: set generous client or server time budgets; send lightweight keepalives during the pause; allow extension.
- Resume or reattach: persist {`payment_id`, `args_snapshot`}. On a subsequent "I've paid" message, detect PAID and resume; expose a "Resume payment" control.
- Lost context fallback: if resume is impossible, re-issue the canonical link and confirmation instructions, or fall back to TWO_STEP or RESUBMIT for completion.
- Structured links: when available, use a dedicated `url` field in the elicitation form; otherwise include the link in the UI body.

4.5 PROGRESS (notify and poll)

Intent. On invocation, emit a progress update that carries the payment link and instructions; wait for PAID via webhook or polling and only then execute the original logic. This preserves a single-tool surface but relies on the client displaying progress message text.

Preconditions. Client renders progress notifications *and* displays human-readable message text; clickable links are not guaranteed. Long or extendable timeouts recommended.

Mechanics. Tool invoked → server captures `args_snapshot` and creates {`payment_link`, `payment_id`} → server emits periodic progress messages containing the canonical link (often non-clickable; users may need to copy or paste) → server polls or awaits `PAID(payment_id)` → (PAID? → EXEC(`args_snapshot`) and finish : continue reporting or timeout).
Figure 6 depicts this streaming coordination pattern.

Guarantees and failure modes.

- Safety: strong if EXEC only runs post-PAID and against the stored `args_snapshot`.
- Liveness: fragile when clients hide progress message text or throttle updates; the user will not learn that payment is required.
- Non-clickable links: even where text is visible, in several clients URLs are displayed as plain text rather than actionable links; users must copy or paste, increasing drop-off.

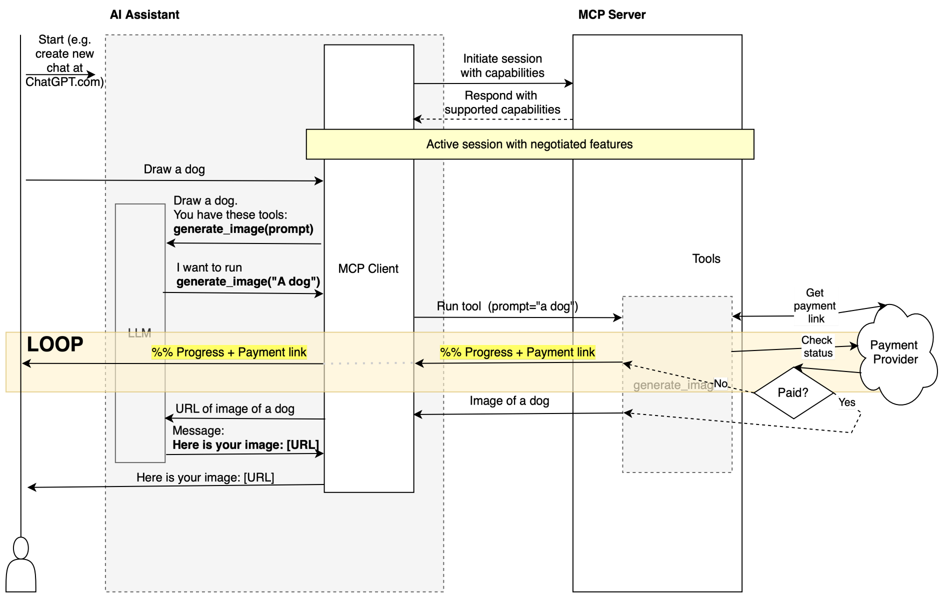
AI Assistant
MCP Server
Start (e.g. create new chat at ChatGPT.com)
Initiate session with capabilities
Respond with supported capabilities
Active session with negotiated features
Draw a dog
Draw a dog. You have these tools: generate_image(prompt)
I want to run generate_image("A dog")
MCP Client
Tools
Run tool (prompt="a dog")
Get payment link
LLM
LOOP
%% Progress + Payment link
%% Progress + Payment link
Check status
Payment Provider
generate_image
No
Paid?
Yes
URL of image of a dog
Image of a dog
Message: Here is your image: [URL]
Here is your image: [URL]

Figure 6: PROGRESS coordination pattern: the server streams status updates that include payment instructions while awaiting confirmation.

- Timeout or abandonment: clients often enforce short inactivity or runtime limits; payment may complete after the tool has errored. The user returns with "I've paid," but there is no in-flight context to resume.
- Resume requirement: in practice, servers accept a subsequent message, re-check `PAID(payment_id)`, and complete EXEC; without resume or reattach, users can be stuck despite having paid.
- Backgrounding or navigation: tab or mobile backgrounding can suppress progress rendering, further reducing discoverability of the link.

UX notes. Works where message text is actually shown; if it is hidden, users typically will not realize payment is required. Even when shown, links may be non-clickable (copy or paste). Communicate clearly that payment happens outside the tool and that completion may take a moment to be detected.

Mitigations (recommended).

- Capability check: only choose PROGRESS when the client reliably displays progress message text; otherwise select ELICITATION or TWO_STEP or RESUBMIT.
- Keepalive and budgets: send periodic progress to keep sessions alive; negotiate longer client or server timeouts where possible.
- Resume or reattach: persist {`payment_id`, `args_snapshot`}; on a later "I've paid" message, detect PAID and complete EXEC; optionally expose a one-shot confirm tool as a fallback.
- Canonical link: include the same URL in every progress message; avoid per-message links to reduce mismatch.

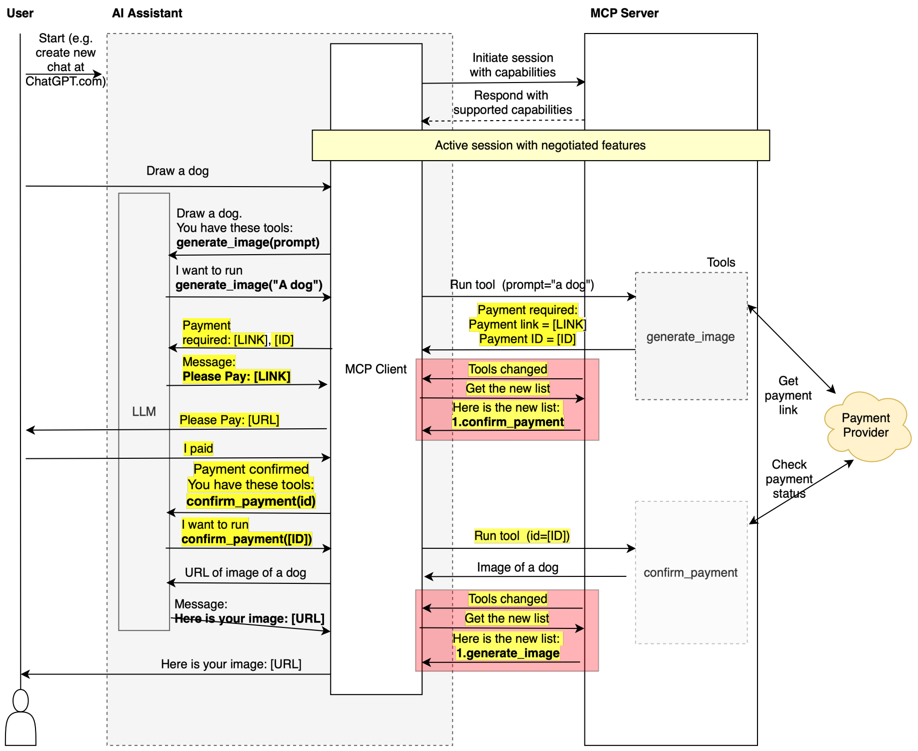

Figure 7: DYNAMIC_TOOLS coordination pattern: the server adjusts the advertised tool set so that only the next valid payment action is visible.

- Prefer structured URLs: when a client supports URL_ELICITATION or a dedicated `url` field, use that instead of free-text links.

4.6 DYNAMIC_TOOLS (via `tools/listChanged`)

Intent. Guide the next valid action by changing the visible tool set at specific points (for example, temporarily expose a `confirm_payment_{short_id}` tool), so the model and user are steered toward payment confirmation before execution.

Preconditions. Client implements dynamic tool listings and reacts to the `notifications/tools/list_changed` notification by refreshing the tool list (caching disabled or cache busted).

Mechanics. Tool invoked → server captures `args_snapshot` and creates {`payment_link`, `payment_id`} → server returns the link or instructions and emits `notifications/tools/list_changed` to reveal a scoped `confirm_payment_{short_id}` tool → client refreshes tool list → model calls `confirm_payment_{short_id}(payment_id)` → server verifies `PAID(payment_id)` and runs EXEC(`args_snapshot`) → server emits `notifications/tools/list_changed` to hide the temporary tool and restore the original set.

This dynamic adjustment of the tool set is illustrated in Figure 7.

Guarantees and failure modes.

- Safety: strong when EXEC is only reachable via the confirm tool, which re-checks PAID and executes against `args_snapshot`.

- Liveness: depends on the client honoring `list_changed` and actually refreshing the tool list; stale caches silently break the flow.
- Flicker or caching: tools appearing or disappearing can confuse users and models; aggressive caching can leave phantom or missing tools.
- Concurrency: multiple pending intents can expose multiple confirm tools; without a one-to-one association to `payment_id` and single-consumption semantics, the wrong job may execute.
- Naming or ambiguity: a generic confirm tool name invites misrouting; lack of namespacing (for example, `confirm_payment_{short_id}`) and concise instructions increases error rates.
- Notification loss or reordering: missed or out-of-order `list_changed` events strand the user.

UX notes. Explicit confirm tools reduce wrong-tool calls and make the required next step obvious, but flicker and list refreshes can feel jumpy.

Mitigations (recommended).

- Namespace and bind: expose `confirm_payment_{short_id}` and require `payment_id`; store `args_snapshot`; verify at confirm; hide on completion.
- TTL and cleanup: set expirations for temporary tools; always emit a final `list_changed` to restore the base set.
- Capability probe: detect dynamic-tools support at init; fall back to TWO_STEP when unavailable.

4.7 URL_ELICITATION (emerging)

Status (January 2026). URL-mode elicitation was added in the MCP 2025-11-25 specification update as a standardized way (SEP-1036) for servers to request that clients handle outbound URLs explicitly during elicitation. As of January 2026, SDK and client adoption remain limited: only a subset of clients declare or honor `mode: "url"` in elicitation flows. For payment scenarios, this pattern is a nearly ideal refinement of ELICITATION where supported, but implementers should assume that many deployed clients will not yet expose URL-mode elicitation capabilities.

Intent. Increase reliability of payment or OAuth hand-offs by using elicitation in `mode: "url"` so the client presents a dedicated URL prompt instead of relying on free-text links in forms or progress messages.

Preconditions. Client declares `capabilities.elicitation.url` (explicit mode negotiation) and correctly handles URL prompts; the server still learns completion via webhook or polling.

Mechanics. Server issues an elicitation with `mode: "url"`, providing an `elicitationId`, a `url`, and a human-readable `message`. The client prompts the user and may launch the browser. The server observes completion via webhook or polling, re-checks PAID, and then resumes or confirms and executes EXEC.

Guarantees and failure modes.

- Link reliability: higher than free text; the URL is conveyed via a structured field that clients are expected to handle.
- Dependency on backend confirmation: requires robust webhook or polling and idempotent confirm; otherwise the server may stall awaiting PAID.
- Compatibility: even though URL-mode elicitation is now specified, many clients and SDKs do not yet implement `mode: "url"` or do so only partially; fall back to plain ELICITATION (form) or non-elicitation patterns such as RESUBMIT where necessary.
- Timeouts and backgrounding: the hand-off happens outside the client; users may return after client or server time budgets expire.

UX notes. More predictable than free-text links and, where supported, one of the cleanest options for handing off to external payment providers. The underlying message sequence is the same as ELICITATION in Figure 5; the main difference is that the URL is carried in a dedicated `url` field and clients may treat it specially (for example, by opening a browser or rendering a focused prompt). Pair this mode with security controls (origin pinning for outbound URLs, signed webhooks with replay protection), and provide clear fallback behavior when the client does not declare URL elicitation support.

5 Implementation Notes (Reference SDK)

We present a thin wrapper—PayMCP (MIT, open-source) [11]—that transforms a developer tool EXEC into a payment-gated tool without duplicating business logic. The wrapper enforces payment gating (EXEC only after PAID) and supports multiple payment flows configured via a `mode` parameter.

5.1 Wrapper architecture

- **Intercept**: on tool invocation, the wrapper checks configuration (price, currency, provider) and captures an `args_snapshot`.
- **Orchestrate**: depending on the selected pattern, it issues a link (request), elicits confirmation, emits progress, or exposes a confirm tool; when required by the selected mode, the server creates a payment intent to obtain {`payment_link`, `payment_id`}.
- **Gate**: EXEC runs only after `confirm(payment_id)` returns PAID.
- **Resume and finalize**: return the tool's original result; tidy temporary state (for example, dynamic tools) and log outcomes.

5.2 Capability detection

Today, the library supports both explicit `mode` configuration (TWO_STEP | RESUBMIT | X402 | ELICITATION | PROGRESS | DYNAMIC_TOOLS) and an `AUTO` selector. With an explicit mode, developers pin a specific coordination pattern regardless of client behavior. In `Mode.AUTO`, the wrapper inspects the client's declared capabilities during MCP initialization and chooses a pattern accordingly: when an X402-capable client is detected and the server is configured with an X402 provider, it prefers X402; otherwise, if elicitation is supported it prefers ELICITATION; if neither is available, it falls back to RESUBMIT as a conservative default (as a single-tool surface). This selection is based on declared capabilities rather than runtime observation, so misreported client capabilities may still require manual overrides in some deployments.

5.3 Reliability primitives (timeouts, resume, idempotency)

On timeouts or interrupted streams, the wrapper attempts to resume using `Last-Event-ID` when the client supports it. Confirmation `confirm(payment_id)` is idempotent; the server persists a crash-safe binding `payment_id` $\rightarrow$ `args_snapshot` with single-consumption semantics to prevent double execution. Polling uses bounded intervals, and server-side time budgets are configurable.

5.4 Completion channels (polling now; webhooks planned)

The current implementation uses polling to learn `PAID`. We plan to add an optional webhook channel with signature verification and automatic setup where possible; until then, polling remains the default. In all cases, the authoritative payment state is re-checked immediately before running EXEC.

5.5 Status reporting

The wrapper returns concise statuses in tool responses (and, when applicable, in elicitation or progress messages): `paid`, `failed`, `pending`, and `canceled`. Identifiers returned to the client are minimized; logs avoid including sensitive values.

5.6 Minimal example (Python)

```python
from mcp.server.fastmcp import FastMCP, Context
from paymcp import PayMCP, price, Mode

mcp = FastMCP("demo-server")

# Initialize the payment wrapper. ‘mode‘ selects the coordination pattern.
PayMCP(
    mcp,
    providers=[...],
    mode=Mode.AUTO  # TWO_STEP | RESUBMIT | X402 | ELICITATION | PROGRESS
        | DYNAMIC_TOOLS
)

@mcp.tool()
@price(amount=1.00, currency="USD")
def compute_total(a: int, b: int, ctx: Context) -> int:
    """Add two payment parameters and return the result"""
    return a + b
```

Listing 1: Minimal payment-gated tool using PayMCP

For TypeScript usage, see the online documentation or repository examples.

6 Security Considerations and SAFE-MCP Framework

Payment coordination in MCP introduces new attack surfaces that practitioners must address. The SAFE-MCP framework [12] provides a structured taxonomy of threats specific to MCP environments, adapting the MITRE ATT&CK methodology for AI agent security.

6.1 Payment-Relevant Threat Categories

Several SAFE-MCP techniques directly impact payment coordination patterns:

Initial Access Threats.

- **SAFE-T1001 (Tool Poisoning Attack)**: Malicious instructions embedded in tool descriptions could manipulate payment flows—e.g., redirecting `payment_link` or altering `args_snapshot` before confirmation.
- **SAFE-T1007 (OAuth Authorization Phishing)**: Particularly relevant for URL_ELICITATION where OAuth flows hand off to external payment providers.

Execution and Persistence.

- **SAFE-T1201 (MCP Rug Pull)**: Time-delayed changes to tool definitions could alter payment gating logic after initial user approval—critical for TWO_STEP and DYNAMIC_TOOLS patterns.
- **SAFE-T1202 (OAuth Token Persistence)**: Stolen refresh tokens could enable unauthorized payment confirmations.

Defense Evasion.

- **SAFE-T1403 (Consent Fatigue)**: Repeated benign payment prompts desensitize users, hiding a malicious transaction mid-flow.

6.2 Pattern-Specific Security Analysis

Table 1 maps SAFE-MCP techniques to the payment coordination patterns introduced in this paper.

Technique	TWO_STEP	RESUBMIT	ELICIT.	PROGRESS	DYN.	URL_E.	X402
SAFE-T1001	✓	✓	✓	✓	✓	✓	✓
SAFE-T1007	–	–	–	–	–	✓	–
SAFE-T1201	–	–	–	–	✓	–	–
SAFE-T1403	✓	–	✓	✓	✓	✓	✓

Table 1: SAFE-MCP threat applicability by payment pattern. ✓indicates the pattern is susceptible; – indicates limited applicability.

6.3 Recommended Mitigations

For payment-gated MCP tools, we recommend the following mitigations aligned with SAFE-MCP guidance:

1. **Cryptographic binding (SAFE-M-2)**: Associate `payment_id` with a server-signed `args_snapshot` (or its hash) and verify the binding on confirmation to prevent post-payment argument drift.
2. **Unicode sanitization (SAFE-M-4)**: Filter Unicode injection and hidden instructions from tool metadata before display.
3. **Origin validation**: For URL_ELICITATION, verify payment callback URLs against allowlists to prevent OAuth mix-up attacks (SAFE-T1306, SAFE-T1307).
4. **Rate limiting on confirmations**: Implement cooldowns between payment confirmations to mitigate consent fatigue exploitation.

6.4 Limitations

This security analysis has several limitations that practitioners should consider:

- **Taxonomy scope**: Our pattern taxonomy is based on the MCP specification as of January 2026. As the protocol evolves, new patterns may emerge and existing ones may become obsolete.
- **Client implementation variance**: The effectiveness of each pattern depends heavily on client implementation quality. Patterns that work well on one client may fail silently on another due to incomplete feature support.
- **Threat model assumptions**: The SAFE-MCP threat mappings assume adversaries target the MCP layer specifically. Attacks that exploit underlying transport (HTTP, WebSocket) or host system vulnerabilities are outside our scope.
- **No empirical validation**: The compatibility snapshot relies on publicly declared feature matrices rather than controlled experiments. Actual behavior may differ from declared capabilities.
- **Payment provider integration**: We do not address the security of external payment providers (Stripe, PayPal, etc.) or the cryptographic soundness of payment confirmation mechanisms.
- **Model behavior variability**: LLM tool selection is inherently non-deterministic. Our patterns mitigate but cannot eliminate the risk of models bypassing payment gates through unexpected tool sequences.

Future work should include empirical testing across multiple clients, formal verification of payment coordination invariants, and longitudinal studies of pattern effectiveness as the MCP ecosystem matures.

7 Compatibility Snapshot (non-empirical, Jan 5, 2026)

Scope and sources. We report a descriptive (non-experimental) snapshot of client support relevant to payment coordination portability. Data is taken from the official *Example Clients → Feature support matrix* (accessed Jan 5, 2026) and from the capabilities advertised in clients' MCP `initialize` messages captured during routine connections. These sources describe *declared* protocol capabilities; they do not fully characterize end-to-end user experience.

Client selection. Clients were chosen to represent general-purpose chat and IDE integrations with broad distribution, rather than specialized or experimental tooling. The list is illustrative and not a market-share ranking.

Table 2 summarizes declared client support for payment coordination patterns across commonly used MCP clients.

Client	TS	RS	PR	EL	DT	URL	X
Cursor	✓	✓		✓			
VS Code GitHub Copilot	✓	✓		✓			
Claude Desktop App	✓	✓					
Claude Code	✓	✓					
ChatGPT	✓	✓					
Codex	✓	✓		✓			
Gemini CLI	✓	✓					

TS = TWO_STEP, RS = RESUBMIT, PR = PROGRESS, EL = ELICITATION, DT = DYNAMIC_TOOLS, URL = URL_ELICITATION, X = X402.

Table 2: Declared MCP client support by payment coordination pattern (Jan 5, 2026).

How to read the table. TS (TWO_STEP) and RS (RESUBMIT) require only baseline tool invocation and are therefore broadly portable across MCP clients. By contrast, PR (PROGRESS) and DT (DYNAMIC_TOOLS) depend on client-side behavior that is not captured reliably by declared capabilities alone: PR requires that the client actually surfaces the optional progress `message` field to the user, and DT requires that the client reacts to `notifications/tools/list_changed` by refreshing the tool catalog. URL denotes URL_ELICITATION (SEP-1036), and X denotes the external X402 payment protocol.

Key portability caveats. Although some clients advertise support for progress notifications, in mainstream chat and IDE clients the optional progress `message` text is often not rendered to users (or is rendered in a way that makes URLs unusable). Because the PROGRESS payment pattern relies on that human-visible message channel to convey the payment link, we treat PROGRESS as non-portable and leave PR unmarked unless a specific client is validated end-to-end.

Similarly, DYNAMIC_TOOLS assumes that clients refresh the tool catalog when receiving `notifications/tools/list_changed`. In practice, many clients treat the tool list as effectively static after initial discovery (for example, caching the initial `tools/list` result) and do not reliably react to `list_changed`, which silently breaks dynamic confirmation-tool flows. We therefore leave DT unmarked as a conservative default.

URL_ELICITATION is standardized via SEP-1036 [9], but as of January 2026 it is not broadly implemented by the most widely used clients; deployments should assume limited availability and provide fallbacks (for example, plain ELICITATION or RESUBMIT / TWO_STEP). Finally, X402 [10] is an *external* on-chain payment protocol rather than an MCP specification feature; practical use therefore requires a custom client, gateway, or wallet-integrated environment that explicitly implements the X402 handshake.

Implication. For broad compatibility today, it is preferable to select a coordination pattern based on the specific client's observed behavior and declared capabilities (or to delegate this choice to a wrapper-level `AUTO` mode) rather than hard-coding a single default. In our reference SDK, `Mode.AUTO` prefers X402 only when an X402-aware client and provider are available, otherwise prefers ELICITATION when supported, and falls back to RESUBMIT as a conservative default. TWO_STEP remains a robust explicit fallback when client capabilities are minimal. We recommend treating PROGRESS, DYNAMIC_TOOLS, and URL_ELICITATION as experimental unless validated for the target client.

References

[1] Model Context Protocol Specification (2025-11-25).
https://modelcontextprotocol.io/specification/2025-11-25

[2] MCP Architecture Overview.
https://modelcontextprotocol.io/docs/learn/architecture

[3] MCP Server Tools Capability.
https://modelcontextprotocol.io/specification/2025-11-25/server/tools

[4] MCP Client Elicitation Capability.
https://modelcontextprotocol.io/specification/2025-11-25/client/elicitation

[5] MCP Progress Notifications Utility.
https://modelcontextprotocol.io/specification/2025-11-25/basic/utilities/progress

[6] MCP Protocol Schema (including notifications).
https://modelcontextprotocol.io/specification/2025-11-25/schema

[7] MCP Security Best Practices.
https://modelcontextprotocol.io/specification/2025-11-25/basic/security_best_practices

[8] Model Context Protocol Client Feature Matrix.
https://modelcontextprotocol.io/clients

[9] MCP Specification Changelog (2025-11-25): Added URL mode elicitation (SEP-1036).
https://modelcontextprotocol.io/specification/2025-11-25/changelog

[10] X402 Protocol.
https://www.x402.org

[11] PayMCP SDK (GitHub Repository).
https://github.com/PayMCP/paymcp

[12] SAFE-MCP: Security Analysis Framework for Evaluation of Model Context Protocol (OpenSSF SIG-SAFE-MCP).
https://github.com/SAFE-MCP/safe-mcp

Version 3, January 2026