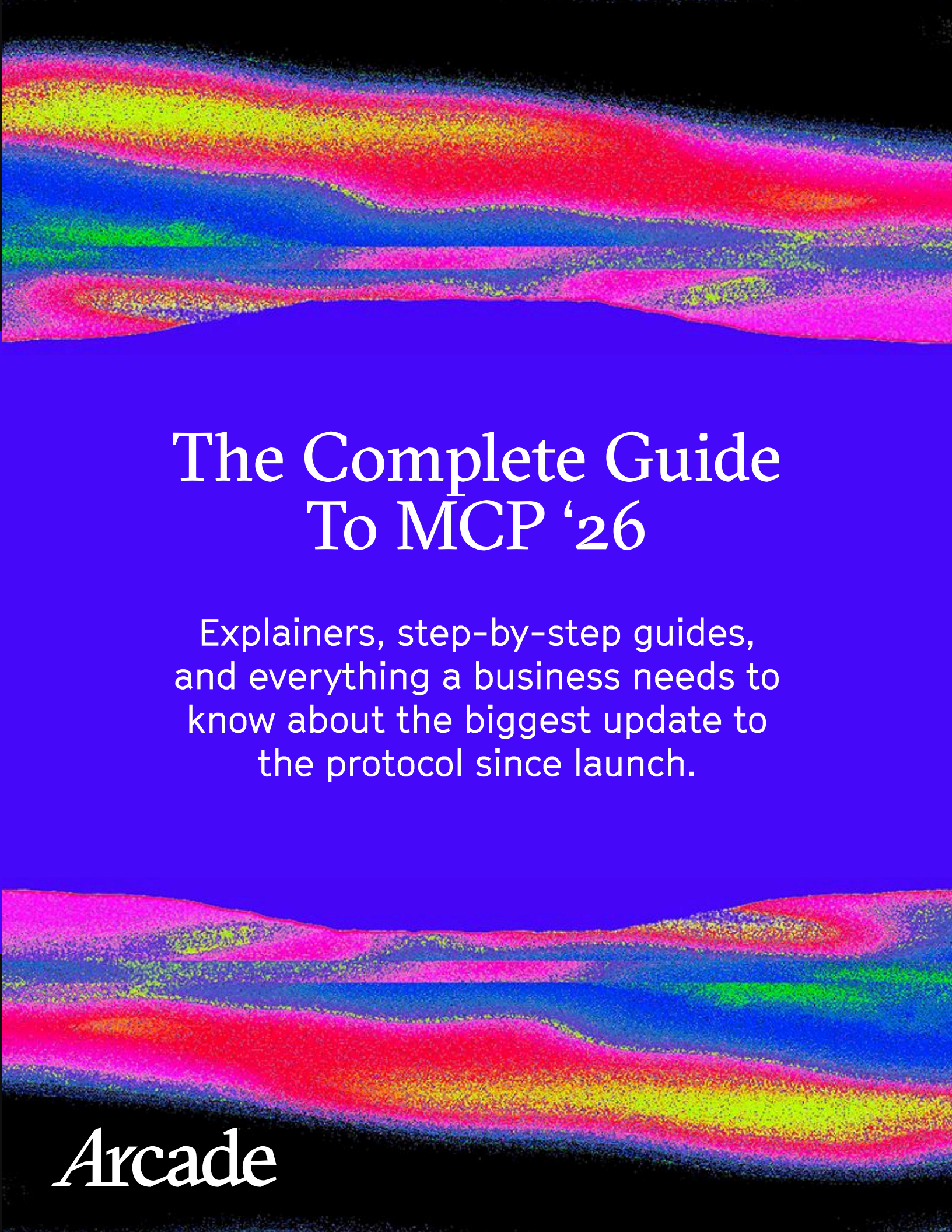




The Complete Guide To MCP '26

Explainers, step-by-step guides,
and everything a business needs to
know about the biggest update to
the protocol since launch.

Arcade

Index

Welcome

Explainers

MCP Is Going Stateless. Here's What That Means.....	Page 2
Why Closing the Refresh Token Gap in the New MCP Specification Matters.....	Page 4
MCP Extensions Are Now Official. So Is Fragmentation.....	Page 6

How-Tos

Migration Checklist. What to Do Before Your Server Meets the New Protocol.....	Page 8
The Servers & Gateways You Can Finally Build Without Session IDs.....	Page 10
Supporting Stateless MCP: A Detailed, Step-by-Step Guide for Clients.....	Page 11

How Arcade Helps

Welcome to MCP' 26

The Model Context Protocol (MCP) was developed as an open standard for connecting AI agents to external tools and data sources. Instead of every agent and every tool negotiating their own bespoke integration, MCP provides a shared protocol that any compliant client and server can use to discover tools, share context, and coordinate action. This protocol is the critical foundation that Arcade.dev contributes to, and builds on, to enable agents to touch real users and systems.

Since the initial MCP release in November, 2024, tracked MCP servers have gone from a few hundred to 10,000-plus, with over 97 million monthly downloads of the official SDK and growth accelerating.

In response to the evolution from early-stage prototyping to enterprise production, on July 28, 2026, MCP shipped the largest revision to its spec since launch. In the long-term this update will mean real benefit to businesses building on MCP, particularly in regards to scale. But in the short-term, there are breaking changes you'll need to respond to.

We've assembled this guide to help you through all of it: what's new, what's changing, what breaks, and how to migrate efficiently and effectively.

MCP Is Going Stateless. Here's What That Means

By Nate Barbettini

Building a MCP server on your laptop has become relatively easy. Running one for 10,000 employees has not.

That gap is driving a significant portion of the new MCP release. The headline is a rework of the core protocol from stateful to stateless, with session IDs going away.

If you've only run MCP servers locally, this might sound like irrelevant plumbing. But if you build or operate servers and gateways at scale, the breaking change in this release will force a rewrite in the short term and provide a lot of benefit in the long term.

Let's walk through what's changing and why.

Where stateful sessions came from

Early on, Model Context Protocol grew from the assumption that servers would be local. MCP's early design was inspired by the Language Server Protocol that powers code editors like Sublime and VSCode, which also run locally. On your local machine, having two processes talk to each other statefully is no big deal.

Here's what it looks like in MCP today (or, yesterday):

The first time an MCP client like Claude connects to a server, it sends a "hello": I'm Claude, here's my version, here are my capabilities. The server replies with its own capabilities and hands back a session ID. In MCP, this handshake is called `initialize`.

From then on, the client sends that session ID on every request so the server knows it's the same conversation. Sometimes the ID expires, so the client has to notice, request a new one, and carry on.

On one machine, that's trivial. Store the session ID in memory and you're done. At enterprise scale, it becomes a problem.

Why stateful sessions break at scale

Picture a real deployment. You're running a server for millions of users, behind a load balancer whose entire job is to route each request to whatever server in the farm is free, sometimes in a different region. Now every one of those machines has to know about a session ID that some other machine handed out. It's not impossible, but it's a serious pain, and it fights the load balancer instead of working with it.

The web solved this long ago. A long time ago, you configured "sticky sessions," pinning a browser to the one server that held its state. But as the web grew, we preferred stateless applications, where the server itself doesn't have to remember you between requests. That's the pattern MCP is finally adopting.

This scale problem is one reason why first-party MCP servers have been slow to arrive. They exist, but not everywhere! Plenty of big companies can stand up a demo. Getting to full production and serving millions of users runs straight into the session problem and frequently stalls there.

What stateless actually means

"Stateless" is easy to misread, so let's be precise!

It does not mean there's no state, and it doesn't mean that MCP is now limited to simple requests. It means the server doesn't have to remember anything between requests, because the client sends everything the server needs every time.

In practice, that opening handshake I mentioned before disappears. The client no longer says hello and negotiates a session up front. Instead, it sends a request, and everything important now travels on each request instead. In practice, that's just a few extra fields and headers. Instead of identifying itself once, the client identifies itself on every call the same way a browser sends a user-agent header.

The short version is that MCP is starting to behave the way web already has for decades. That's good!

Why it's a hard break

In protocol terms, this is a breaking change. A client built for version **2025-11-25** expects to initialize and carry a session ID, so it can't talk to a server expecting **2026-07-28**. The reverse is true, too.

MCP already has an escape hatch, though: version negotiation. A client can declare that it only speaks the **2025-11-25** protocol, and a server that's capable of speaking either can speak to that client but still support newer clients using today's protocol. This means the ecosystem doesn't fall apart (good!), but supporting multiple versions at once takes real work and is difficult to sustain.

For a server or client developer, this is a significant rewrite. The new protocol is actually simpler to implement, with far less tracking of where you are in the conversation, but there's just no getting around that it will be new work for anyone who's already written all the stateful code.

The payoff is scale and stability

The short-term pain buys a real long-term gain, and it comes back to scale.

Deleting session handling makes it far easier to build and maintain high-quality servers at scale, which is exactly what has held back official first-party servers from the companies best positioned to run them.

My hope is that this release finally gets more of them to plant a flag: this is the official, supported way for agents to reach our service.

As users, we all get a second benefit, which is stability.

A lot of the sharp edges with MCP today trace back to how brittle session handling is. At Arcade, we routinely file bug reports on clients that mishandle expired sessions and on connections that die and won't come back. Delete that code and a whole class of bugs goes away with it.

The MCP team has already done great work here, getting the official SDKs compliant and building out the official conformance suite that tells you whether you're speaking the protocol correctly. Once everyone clears the rebuild, the ecosystem should land somewhere far more stable.

What this means if you build servers or agents

Short term, expect fragmentation.

Any breaking change to an ecosystem this size opens a window where some clients have auto-updated to the new protocol and some servers haven't, and vice versa. Maybe your clients and servers line up. Maybe they don't. If you operate an agent fleet, or roll agents out to a workforce, the last thing you want is a version-compatibility matrix where version 1.5 works, version 1.7 doesn't, and you're the one debugging why at 2 a.m.

For more on how to navigate this fragmentation, see our How-To section.

Why Closing the Refresh Token Gap in the New MCP Specification Matters

By Wils Dawson

I've contributed to the Model Context Protocol (MCP) since the earliest days, starting with our proposal for User Interaction when we saw people in the community talking about the need to bridge text-based AI with actionable systems. That's what we'd spent a year building in-house at Arcade.dev and we were excited to contribute to the open standard and help securely accelerate the adoption of real production agents across systems, vendors, and applications.

At the same time, I kept an eye on the basic authentication and authorization stories, helping contribute to the large directions of various improvements, and authoring other small improvements of my own.

My biggest individual contribution was [URL mode elicitation](#), which became the basis for secure tool authorization from a downstream MCP server. It lets MCP Servers pause tool execution to interact directly with the user. In a secure, external browser, you can complete sensitive actions, preventing data like passwords, payment info, or OAuth tokens from passing through the client or LLM context. It was one piece of the larger MCP puzzle, but an important one and I'm proud of it.

My contribution to this upcoming release is less flashy but will have a real impact on how agents work in production. It clarifies how MCP clients should handle refresh tokens.

What a refresh token actually does

When an MCP client like Claude Code or Cursor connects to an MCP server, it has to authenticate and get authorized. The client receives an access token, and that token stays short-lived by design. It might last 15 minutes, maybe an hour.

That's a critical security practice, but you also don't want a user signing in again every 15 minutes. That experience is miserable. So alongside the access token, the client receives a refresh token, which it uses to quietly get new access tokens without pulling the user back in.

Why clients were handling refresh tokens inconsistently

While building MCP capabilities at Arcade I kept watching clients handle refresh tokens differently. Some weren't using refresh tokens at all. Some never requested a refresh token, which meant our runtime never issued them one. The result was inconsistent behavior across the ecosystem.

The standards MCP authentication builds on, mainly OAuth, already document this behavior well, but MCP itself stayed relatively quiet on what refresh token behavior should look like in its own environment. I suspected this silence was the cause of the inconsistent behavior. Without clear guidance in the spec for people or agents to read, clients drifted.

What changes about refresh tokens in this MCP release

My thinking was straightforward: carry the established refresh token guidance from OAuth and OpenID Connect into MCP and write it down plainly, so anyone reading the spec, whether a casual observer, an auth expert, or a coding agent generating an implementation, lands on the right behavior more often.

This matters because MCP is open, and interoperability is the entire point. When the behavior isn't written down, consistency gets left to chance.

The gap only shows up in production

Refresh tokens are the kind of problem that hides during a demo. The first time you connect, everything works. You sign in, the tool call succeeds, you close the laptop. The refresh token doesn't matter yet. Then you open the laptop the next morning, the client needs that refresh token, and it breaks. It worked yesterday and fails today.

That distance between the demo and the day after is exactly where we focus at Arcade. A new p97 tool can break in the most mundane, frustrating ways. Roll it out, tell employees to use it, and the moment someone can't log back in, you've lost them. The refresh token is one part of the security story, but an important one.

What interoperability means for the enterprise

For an enterprise, this clarification shows up as consistency. You can build an MCP server, maintain one, or buy one, then connect to it from any client your organization allows, and the authentication, the user experience, and the security behavior stay the same. Most companies run a mix: many clients and many servers, deployed across many teams. Interoperability keeps that mix coherent, which is one reason why MCP is so powerful as a standard.

On the governance side, that means one consistent security story across every touch point. On the experience side, it means nobody gets asked to type a password more than once and single sign-on works the way it should. Both depend on refresh tokens behaving correctly.

This release is about making MCP work in production

Plenty of people in the OAuth world were already doing this correctly. However, many people building on MCP haven't come from OAuth or OpenID Connect, and a lot of the implementation runs through coding agents. Putting best practice directly in front of them is the point.

This wasn't so much an oversight from the previous release as it was something people didn't think about while MCP was mostly demos, when the question was "does this work today?" instead of "will this hold up in production?" That question has shifted. MCP is here to stay. That's settled. We're in production mode now, and the work has moved to making it hold up in the real world.

An unglamorous clarification about refresh tokens is part of that evolution, and I'm glad to see it land.

MCP Extensions Are Now Official. So Is Fragmentation.

By Eric Gustin

One of the key changes in the new MCP release may be easy to overlook, but it's one that's likely to have real long-term consequences. This release defines an official track for what the spec calls extensions.

Extensions are the capability layer of MCP. The core protocol defines the primitives (tools, resources, prompts) and how a client communicates with a server.

Extensions sit on top of that foundation and define capabilities, what a client and server can agree to do together beyond the basics, like displaying a UI inside the client.

This new release formalizes extensions, and I understand why the core maintainers decided to do it. But it also formalizes a consideration that's been with MCP since the beginning - fragmentation. This release doesn't create that problem. It puts a name on it, gives it a process, and potentially sets it up to grow.

What extensions looked like in the previous version of MCP

Extensions have technically existed since the beginning, they just weren't defined. MCP always anticipated the idea of capabilities, every connection started with an initialization handshake where the client and server negotiated what each could do. But there was no way to define an extension itself.

MCP provides a meta field, which is a container where developers can put any arbitrary thing, and that's exactly what developers were using it for. If you wanted to extend the protocol, you stuffed your custom behavior into meta and hoped the client on the other end knew what to do with it.

None of it went through an approval process from the core maintainers.

It was an unofficial, off-to-the-side way for anybody to extend the protocol in any direction. And because none of it was labeled, it was hard for anyone else to know an extension even existed, let alone support it.

What's changing

The new release gives extensions a process. There are now three types: official, experimental, and unofficial. Anything official has to get an explicit sign-off from the core maintainers.

How clients and servers agree on capabilities has changed too. The initialization handshake, where that negotiation used to happen once per connection, is gone in the new stateless protocol. Instead, client capabilities, including extension support, travel in `\_meta` on every request, and servers expose their capabilities via a new `server/discover` method whose results clients can cache. There's no longer a negotiated session state that both sides commit to. Instead, what's active is computed fresh from what each side claims.

But here's the important part; that official sign-off only covers version one. After that initial approval, the core maintainers don't need to review each update and extension is on its own release cadence. An extension author can ship breaking changes whenever they want. The core protocol releases every few months, but an extension could ship a breaking change every single day.

The maintainers want the core protocol thin and stable, and moving capabilities out to their own track accomplishes that. But the trade off is an always-changing capability layer sitting on top of a stable core.

Two more details matter. First, in specification language — "must", "should", "may" — extensions are a "may." No client is obligated to support any of them.

Second, extensions are identified by a reverse-DNS string, something like "com.company/mcp-app" or "com.company/memory." When an extension ships a breaking change, the identifier changes with it. "com.company/memory" becomes "com.company/memory-v2". Your code has to keep up.

The release also moves tasks, long-running tools that report progress back to the client, from an experimental feature in the core spec to an extension. Anyone who implemented tasks now has to migrate.

The implications for developers

As someone with tools on an MCP server, you now have to think about graceful degradation. If this client supports the extension, do this. If that one doesn't, do that. Multiply that across every client and every extension and it gets complex in a hurry.

Here's what that looks like in practice, using an extension that exists today. Take [MCP Apps](#). Say you build a tool that drafts an email. You call the Gmail API, create the draft, and build a UI so the end user sees the actual email: the subject, the recipient, the body, and a send button. They can edit it before it goes out. That preview is the whole point, so the user trusts the email because they saw it first.

You test it in Claude, running your server locally, and it works perfectly. You ship it. A day later, a user complains: "I can't see my draft email before I send it. Can you add that feature?" You did add that feature. They're just using a client that doesn't support the extension. The clients aren't obligated to implement it, you don't control the client, and the thing you spent all that time building only works for some of your users.

The immediate problem is migration, with tasks moving out of core. The problem on the horizon is that within six months, I could see popular clients pushing extensions specific to their own use cases. Anthropic ships a user-memory extension, and Cursor ships its own. Two extensions, same capability, and you have to implement both. Versioning adds another dimension to the same matrix: the client supports version three of an extension, you support version two, and now that's your problem too. There aren't many extensions yet. Once this track is defined, it's going to explode.

Your responsibility

The extensions track has real benefits: a thin, stable core, and no surprise breaking changes that means you can upgrade when you choose to. But the day-to-day reality for developers is a growing support matrix that carries real implementation weight. The protocol won't resolve that for you. Something in your stack has to.

Migration Checklist. What to Do Before Your Server Meets the New Protocol

By Nate Barbettini

First things first, don't panic. Nothing breaks on day zero (July 28th) with the new MCP release, although there is some real work to be done.

As we've covered extensively, the headline change is that MCP is going stateless, and session IDs are going away. If you've only ever run a server on your laptop, you can mostly ignore this. But if you build or operate MCP servers, the switch to the new protocol is real work.

The good news is that this migration isn't a rebuild-your-life project. We recently migrated our own MCP implementation behind the scenes without too many sharp edges, and have some insights from being involved in writing the new specs. To help orient your mental model, I've put together this checklist, roughly in the order you'll hit each step.

Step 1: Expect some weird traffic

Your server almost certainly speaks the old protocol today. As clients update, they're going to start trying to speak the new protocol at you. The guidance now is that clients assume a server speaks the new protocol until they get an error and fall back to the old one. So you'll see connection attempts in the new format arriving at a server that doesn't understand them yet.

Most of the time, clients will negotiate down gracefully and everything keeps working. But depending on how you built your server, that early traffic could show up as errors, or as harmless noise, or as a flood of log messages. In the worst case, it could blow something up.

So the day-zero task is to make sure your server handles unexpected traffic gracefully, and put a mitigation in place if you're worried about log noise going crazy. You don't have to support the new protocol yet. You just have to respond gracefully.

Step 2: Decide whether you'll augment or rewrite

Once you're ready to actually support the new protocol, you'll face an architecture decision.

You can augment your existing server so it speaks both the old and new protocols, or you can build two separate implementations and run them side by side. The second option is sometimes cleaner, but there's no universal right answer. It depends heavily on how you built your server in the first place.

Either way, you're keeping the old code path around for a while. More on that at the end.

One potentially happy note though: the folks operating some of the largest servers in the world have told us this process ended up deleting more code than it added. The new protocol is simpler. That's always a good thing. But in the short term, you're supporting both, and that's the part that takes work.

Step 3: Check for deprecated and changed features

Before you start rewriting, take inventory of what your server actually uses, because a few things are moving.

Three features are slated for deprecation - roots, sampling, and logging - because frankly, hardly anyone uses them. If you're one of the few whose server does, you're not stuck. There are recommended alternatives, but you need a plan to get off them, because clients will eventually stop supporting them entirely.

Then there's the flip side: features that aren't going away but are changing. Elicitation is the big one. It still exists, but the mechanics are different. It used to be that a server could send an elicitation request at any time, for any reason. Now it can only happen in response to a client request, and there are constraints on when you can send one. Many people aren't using elicitation yet, so this won't hit every server, but if you are, check how the rules changed.

Step 4: Implement the new protocol

Now the core work. If you built on top one of the official MCP SDKs, this step can be close to free. There's a new major release available, you upgrade, and you get most of the new protocol mechanics for free.

The wrinkle is the session ID. It's going away, and if any part of your architecture depended on it, you'll need another way to get what you need about the client's session. A common example is analytics: Servers often leaned on the session ID to figure out which clients were calling them, because the client announced itself once at the start of the session and you had to remember it from then on. Now the client tells you who it is on every request, the same way a browser sends a user-agent header.

That's actually simpler, but if the old behavior was wired into your metrics pipeline or routing logic, you'll need to update that logic.

Step 5: Verify with the conformance suite

Don't guess whether you got it right. Prove it!

The MCP project publishes an open-source conformance suite for exactly this. You download it, point it at your server, and it runs a battery of tests that tell you where you're not sending the right response. It's a deterministic, programmatic way to know you're actually speaking the protocol correctly.

If you're on an official MCP SDK, you'll likely be green already, since the SDKs are tested against the suite. If you're running another open-source implementation or something homegrown, this step is not optional.

Step 6: Plan for the dual-stack era

One thing that spans the whole migration is to plan for a period where you have to speak both protocols at once.

The client side of the ecosystem takes time to catch up. New versions ship, but not everyone auto-updates, and some people will be running an old version of Claude or another client for a while. Nobody knows exactly how long this lasts, but there will be a stretch where, if you don't want to cut older clients off immediately, you're running dual-stack.

If you're an enterprise that controls what's on every machine, maybe you just force the upgrade and skip this. But if you operate an internet-facing service open to the public, you'll speak both for a while, watch the old-protocol traffic taper off, and eventually make a clean decision to cut it off. Plan for that window rather than getting surprised by it.

Step 7: Clean up

This is the fun part. Once you're speaking the new protocol correctly, go back and delete what you no longer need.

You don't need Redis sitting around just to store session IDs. You can drop a lot of session-management and tracking code. This is where that "deleted more than we added" comment comes from, and it's a genuinely satisfying pass. After this, there's no special long-term maintenance burden. The migration ends. What's left is just the ordinary upkeep you'd have had anyway.

The Servers & Gateways You Can Finally Build Without Session IDs

By Sterling Dreyer

Nate Barbettini already walked through the big picture of the new Model Context Protocol (MCP) release, explaining that the core protocol is moving from stateful to stateless, and session IDs are going away. If you want the why, go back and read his article earlier in this guide.

This piece is for the people who have to do the work. I build Arcade.dev's runtime, so I want to get specific about what changes when you run MCP servers or a gateways in production, and what to check before the release lands.

The multi-user session store was the real tax

On a single client, a session ID is harmless. You connect, you get an ID, and you send it back on every request. From the perspective of a single user agent, this can be great for long-running and multistep operations. The cost shows up the moment more than one user is involved.

If you build an agent that handles multiple users, you have to keep a session for each of them, and map requests to the correct one. Then you try to scale the service, and it gets harder. A session is usually tied to the one server instance where the request was initialized, so once you run replicas, any replica that picks up a user's next request needs that same session. At that point you are building a shared session store and wiring every replica into it or using sticky sessions, just to receive requests, even if your server doesn't require state.

That is the hidden tax of statefulness, and new stateless protocol removes it. The client sends everything the server needs on each request, so any replica can answer any request, and the session store goes away.

What actually breaks day one

For simple tool calls, the work gets easier. There is no initialization step anymore, so you just call the tool. That's it. One whole step leaves your process.

The break shows up when your server logic depends on state. Long-running work used to lean on the session. You kicked something off, and the server pushed updates back to you. It's convenient, but increases the complexity of your server. In the stateless model, you get a handle for that long-running work and poll it, asking the server whether it has finished, closer to how a REST call behaves. It is a familiar pattern, but it is a real change. The server no longer notifies you on its own, so that polling logic now lives on the client.

If you built long-running tools against sessions, budget time to migrate them before you upgrade.

The bottom line for builders

None of this is exotic. REST APIs are stateless by design and are the dominant standard due to its ease of use and scalability. People use REST everyday, whether they know it or not, and this change lowers the barrier for MCP to do the same. When the release lands, simple servers get simpler, multi-user scale stops fighting you, and Arcade can help support you as you migrate to the new version. Prepare now and you'll be ready to enter the new MCP era as smoothly as possible.

Supporting Stateless MCP: A Detailed, Step-by-Step Guide for Clients

By Guru Sattanathan

The next revision of the Model Context Protocol, the draft dated 2026-07-28, is the most significant rewrite MCP has had since it appeared, and it lands on the 28th. In one breath: the protocol is going stateless. The initialize handshake is gone. Sessions are gone. Servers can no longer send requests back to clients. Every tool call is now a self-contained HTTP request that carries its own protocol version, identity, and capabilities.

This is a great direction. A stateless protocol is easier to scale, easier to load-balance, and easier to reason about. What changes is the job of the MCP client, the component inside your harness that actually speaks the protocol. A lot of what used to live implicitly in a warm connection now has to be handled explicitly on every request. So if you build a client or a harness, or if you depend on one, this is a good moment to check what “supports the 2026-07-28 spec” actually requires.

Below is that checklist. Arcade.dev will be compatible with the new spec on day one, so you can read this two ways: as a punch list if you’re building the client yourself, or as a set of things to demand from whatever runtime you build on. If you want the full wire-level walkthrough, with every payload that crosses the wire, there’s a companion deconstruction by Arcade’s Mateo Torres: [An MCP Tool Call, Deconstructed](#).

The release in one line

Every tool/call now stands entirely on its own (SEP-2575). No handshake, no session to join, no back-channel for the server to reach you on. Any replica of a server can answer any request, because none of them need to remember anything between calls. That is the whole point, and everything on this list follows from it: if the connection remembers nothing, the client has to carry what matters, explicitly, in every message.

What this unlocks: two workflows that get better

Before the checklist, it’s worth being concrete about why this matters, because the release quietly fixes some of the most annoying failure modes in real tool-using agents. Two examples.

A scheduling agent that waits on a human. “Book the team offsite and put it on everyone’s calendar” touches Google Calendar and needs a person to confirm and pay somewhere in the middle. Under the current protocol, that pause is awkward: the booking holds an open connection while the server reaches back up the wire to ask for confirmation, and a timeout or a reconnect can wreck the whole thing. In the new release, the server ends the call with an `input_required` result, hands your client a small piece of state to hold, and the booking resumes as a durable task, even if the harness restarts while the user is still on the payment page. The workflow now survives the human taking ten minutes to click “pay.” That’s Multi-Round-Trip Requests (MRTR) and the tasks extension doing real work.

A cross-service ops agent acting as a real person. “Pull the failing PR from GitHub and post a summary to the on-call Slack channel” spans two services, and it should run as the engineer who triggered it, not as one shared bot account with keys to everything. Every request carries its own per-user token, so the agent can act as the real user on each call, and an expired token just refreshes silently instead of derailing the run.

Neither workflow is new. What’s new is that the protocol stops making the hard parts (human-in-the-loop pauses and per-user identity across services) fight you.

The day-one checklist

1. Speak the self-contained request shape

A tools/call is now fully self-describing. The protocol version, client identity, and client capabilities ride in `_meta` on the request itself, and the method and tool name are mirrored into HTTP headers so intermediaries can route without parsing the JSON body (SEP-2243).

```
POST /mcp HTTP/1.1
Host: cabins.example.com
Authorization: Bearer eyJhbGciOiJIJFuzI1NiIs...
Content-type: application/json
Accept: application/json, text/event-stream
MCP-Protocol-Version: 2026-07-28
Mcp-Method: tools/call
Mcp-Name: book_cabin
```

The Accept header lists both content types because the server chooses which one to answer with: a single JSON response, or an SSE stream when it wants to send progress notifications before the result.

Your client needs to build this on every call, and it must keep the headers and body in agreement, because the server is required to reject them if they disagree. The details are in the Streamable HTTP transport section.

2. Discover, don't handshake

There's exactly one discovery RPC now, and it's new: server/discover (SEP-2575). Every server implements it, and a client can call it before anything else to learn supported protocol versions and capabilities up front. It doubles as a backward-compatibility probe: a server that chokes on it is speaking an older, handshake-era MCP, which is how your client tells the two worlds apart on day one.

3. Register by publishing, not by POSTing

The recommended way for a client and a server that have never met to establish identity is now Client ID Metadata Documents (CIMD). Your `client_id` becomes an HTTPS URL pointing at a JSON document you (or the client maintainer) host about yourself, and the authorization server fetches it on demand. Dynamic Client Registration, the old POST-to-register dance, is deprecated. If your client still relies on DCR, today is the day to publish a `client-metadata.json` instead.

4. Own the token lifecycle, per user and per issuer

The draft tightens the guidance around how a client holds credentials, and it's explicit about the shape: tokens are stored keyed by the authorization server's issuer identifier, and that binding is a hard requirement (SEP-2352).

Your client needs to:

- Store an access token and a refresh token per user, per issuer, encrypted at rest.
- Attach the right token to every request.
- Refresh silently on expiry, and re-run the first-contact OAuth flow on a 401.
- Validate the `iss` parameter on the authorization callback, byte-for-byte, against the issuer it recorded during discovery. This closes a mix-up attack where a malicious authorization server tricks the client into sending a code to the wrong endpoint (SEP-2468).
- Keep the token out of the model entirely. It should never enter a prompt, a context window, or a model response.

5. Carry Multi-Round-Trip state

This is the change that may surprise people most. Servers can no longer call clients back, so when a server needs input mid-call, it doesn't send you a request, it ends the call with an interim result (SEP-2322):

```
{
  "resultType": "input_required",
  "inputRequests": { "...": "what the server needs from the user" },
  "requestState": "v1.gcm.9yKX0jPmYQ4kFhk3...hT2w"
}
```

That `requestState` blob is the server's encrypted memory of a half-finished operation, handed to your client for safekeeping. You gather the user's input, then retry the original tools/call with a new id, the answers keyed exactly as the server asked, and the `requestState` echoed back untouched. Your client must not peek inside it, must not lose it, and must be able to resume the operation on a different server replica than the one that issued it. In practice that means durably parking in-flight work, which is a genuinely new responsibility for a client. The full pattern is in Multi-Round-Trip Requests.

6. Poll tasks instead of blocking

Long-running work moved out of the core protocol into a tasks extension (SEP-2663). The old blocking tasks/result is gone. Your client declares the tasks capability, receives a durable task handle, and polls tasks/get at the interval the server suggests, resuming with the same handle if the harness crashes and restarts. One more loop to run, one more handle to persist.

7. Broker third-party auth mid-flight

Here's the one that matters most in production, because it's where all of the above shows up at once. Say your agent needs to put an event on a user's Google Calendar. That means calling Google on behalf of that specific user, with a delegated OAuth token, requested in the middle of a tool call, for a service that isn't the MCP server your client is talking to.

The protocol shape is clean: the server ends the call with `input_required` and a URL for the user to complete Google's OAuth out of band, plus a `requestState` blob; the user authorizes on Google's own pages so their credentials never touch your client or the model; you retry with the answer. What your client owns is everything that makes it safe: obtaining and storing a per-user Google token (not a shared service account that would hand every user the keys to every other user's calendar), scoping it minimally, refreshing it weeks later, and doing this for every third-party service every one of your tools might reach.

8. Keep the consent gate, and trust nothing by default

None of statelessness changes the host's core security duties: users must explicitly consent before any tool runs, and tool descriptions are untrusted input unless they come from a trusted server. If your client pre-approves read-only tools, that's a policy decision that lives in the harness, never in the protocol.

Build it, or demand it

Read that list back and it's clear how much of it is auth and lifecycle: per-user tokens, issuer keying, silent refresh, mix-up defense, third-party OAuth, and keeping every credential away from the model. The stateless parts (self-contained requests, discovery, MRTR, tasks) are mechanical and well-scoped. The auth parts are where the real engineering (and the real security surface) lives.

If you're building a client, this is your punch list for the 28th. If you're building on a client or a runtime, this is what to demand from it, because "we support MCP 2026-07-28" should mean all of it, not just the easy half.



How Arcade Helps

As we've covered extensively, for the enterprises shipping agents in production, the MCP release means real work. A stateless rewrite, deprecated features, and a growing list of extensions all create churn, and someone has to absorb it. The protocol won't do it for you. Something in your stack has to.

Our intention with this guide was to break down the ins and outs of the release with Arcade intentionally in the backseat because this moment is about more than any one company, but here we want to be direct about where we fit and then get out of the way.

Arcade Speaks Both Versions of the Protocol

Arcade is the actions runtime for enterprise AI agents, the layer your agent calls through when it needs to do something in a real system. It speaks the [2026-07-28](#) shape on day one, and it still speaks the older versions that use session IDs. So with Arcade, you move your servers and clients to the new stateless protocol on your own schedule instead of coordinating a single cutover.

That matters at scale. Ten clients with different capabilities talking to ten servers with different capabilities is a big matrix, and extensions make it bigger. Arcade sits between clients and servers as an MCP gateway and resolves what each side actually supports. When an extension ships a breaking change and [com.company/memory](#) becomes [memory-v2](#), that's one update in Arcade's control plane, not a change rolled out across every server-client pair. The matrix collapses to a single point you control.

Always On: Authorization, Security, and Governance

The deeper reason the runtime layer matters is the controls you can't afford to fragment - authorization, security policy, and governance - don't live in the protocol at all with Arcade. They live in the runtime, on every action, no matter which protocol version a client speaks.

Arcade brokers the OAuth flow and holds tokens in an encrypted, per-user vault, injecting the credential only at execution and never handing it to the model or the client. A post-prompt authorization check is scoped to the intersection of what the user may do and what the agent may do, evaluated per action rather than granted through a service account or a shared token. And a pre- or post-hook, say one that strips PII, runs every time, regardless of the client on the other end. The transport churns underneath while your controls hold steady.

We can promise that because we first built this runtime independently from MCP, so our governance was never coupled to a single protocol. Enforcement lives at the boundary where an agent acts, in code rather than a wire format, so the same authorization, scoping, and audit model applies over MCP today and agent-to-agent (A2A) tomorrow.

The fragmentation is temporary, and the ecosystem will settle. We can help get you there, and then provide the enforcement and governance infrastructure to help you scale.

Get Started Today

Two paths to deploying production-ready agents with Arcade:

REQUEST A DEMO & SEE IT IN ACTION

Our team will walk through your specific use case, show how Arcade's architecture solves your production barriers, and design a deployment plan for your agents. [Arcade.dev](https://arcade.dev)

SIGN UP & START BUILDING

Developers can start immediately with Arcade's free tier. Build your first agent with production tools, test multi-user auth flows, and validate the platform with your use cases before committing. [Signup here](#)

The difference between agents that demo well and agents that actually work comes down to infrastructure. MCP made the connection possible. Arcade makes it practical.

Arcade