ANTHROP\C

Running an MCP proxy at scale

Camila Rondinini | Anthropic

About me

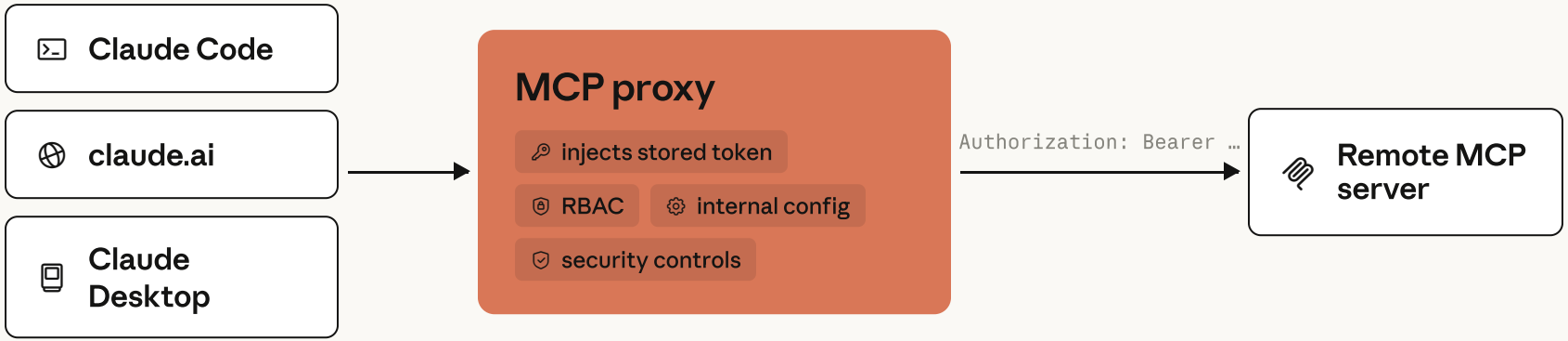
Camila Rondinini

カミラ・ロンディニーニ

- From Brazil 🇧🇷 Based in London 🇬🇧
- First time in Japan 🇯🇵, very excited to explore the country and enjoy the amazing food
- I work as a Backend Engineer on the Connectivity team at Anthropic
- My team owns an MCP proxy that some of our products use to talk to MCP remote servers, which is what I'm going to talk about today

Our MCP Proxy

A forward proxy between Anthropic product surfaces and remote MCP servers.



What I'll cover today

Three themes we had to deal with when running this proxy at scale

01 Keeping users connected

02 Cutting upstream requests

03 Understanding server health

Part one

Keeping users connected

Whenever a user needs to authenticate to use a server, they should ideally only need to do it once

1. **Server says it needs auth**

   `401 Unauthorized` with a pointer to its authorization server.

2. **Authorization flow**

   We build an authorization URL, the user is redirected, consents once, comes back with a code.

3. **Token exchange**

   We trade the code at the token endpoint and store what comes back.

↻ **Refresh, without the user**

When a `refresh_token` is provided, we can use it to get a new access token and it expires. Repeat.

token endpoint → proxy

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store

{
  "access_token": "eyJhbGciOi...",
  "token_type": "Bearer",
  "expires_in": 3600,
  "refresh_token": "8xLOxBtZp8...",
  "scope": "read write offline_access"
}
```

Getting a refresh token in the first place

Getting a refresh token in the first place, when possible

Getting the refresh right under concurrency

Knowing when a refresh failure is recoverable or not

Getting a refresh token in the first place

You can know it's supported

The authorization server metadata tells you: if `refresh_token` is in `grant_types_supported`, the server can do the refresh flow.

```
"grant_types_supported": [
  "authorization_code",
  "refresh_token"
]
```

~4% of servers in our data don't advertise the `refresh_token` grant but issue one anyway, so treat it as a hint.

But how to make the server return a `refresh_token` varies widely

How to make the server return it is entirely up to the auth server.

- **Some require specific parameters on the authorization URL**
- **Some require the client metadata to list it**
  `refresh_token` in the client's `grant_types`.
- **Some require you to enable it in their own console**
  Nothing the client sends matters.
- **Some don't require anything**
  You just get one back.

To illustrate · 1 of 3

Some require specific parameters on the authorization URL

A scope or an extra query parameter, and each provider picked a different one. You learn which by reading that provider's docs, or by noticing no refresh token ever comes back.

proxy → authorization endpoint

```
https://auth.example.com/authorize
  ?response_type=code&client_id=…
  &redirect_uri=…&state=…
  &code_challenge=…&code_challenge_method=S256
```

…plus, depending on the provider, one of:

- `&scope=… offline_access` — Entra, Auth0, Okta, Atlassian (OIDC)
- `&scope=… refresh_token` — Salesforce
- `&prompt=consent` — OIDC with offline_access; Google
- `&access_type=offline` — Google
- `&token_access_type=offline` — Dropbox
- `&duration=permanent` — Reddit

To illustrate · 2 of 3

Some auth servers enforce the registered grant types strictly and require client metadata to declare `refresh_token` in `grant_types`

Dynamic Client Registration (DCR) and Client ID Metadata Document (CIMD) share the same client metadata fields, defined in RFC 7591. `grant_types` is one of them, and it defaults to `authorization_code` when omitted.

Strict servers take that literally and won't issue a refresh token to a client that didn't declare it.

proxy → authorization endpoint

```
https://auth.example.com/authorize
  ?client_id=https://claude.ai/oauth/mcp-oauth-client-metadata
  &…
```

↓ The client_id is the CIMD URL. The authorization server fetches it:

GET https://claude.ai/oauth/mcp-oauth-client-metadata

```
{
  "client_id": "https://claude.ai/oauth/mcp-oauth-client-metadata",
  "client_name": "Claude",
  "client_uri": "https://claude.ai",
  "redirect_uris": ["https://claude.ai/api/mcp/auth_callback"],
  "grant_types": ["authorization_code", "refresh_token"],
  "response_types": ["code"],
  "token_endpoint_auth_method": "none"
}
```

To illustrate · 3 of 3

And sometimes nothing the client sends makes a difference

Either someone flips a setting on the provider's side, or the server hands one back no matter what.

outside the request

A switch in the provider's console

Someone has to enable refresh tokens (or token rotation, or expiry) on the app, server-side. Until they do, no parameter we add changes the response.

Slack token rotation · GitHub Apps token expiry · Auth0 "Allow Offline Access"

Always sent

The server always returns a refresh token alongside the access token. No scope, no parameter, no setting.

HubSpot · Asana · Zoom · Box

Takeaway

LLM clients are the first “universal” OAuth clients, and they’re expected to handle every server’s quirks

None of these quirks are new, but the difference is the expectation that an MCP client should just work with any MCP server. To do so, you should, as much as possible, ensure you are handling all of the quirks from the different providers.

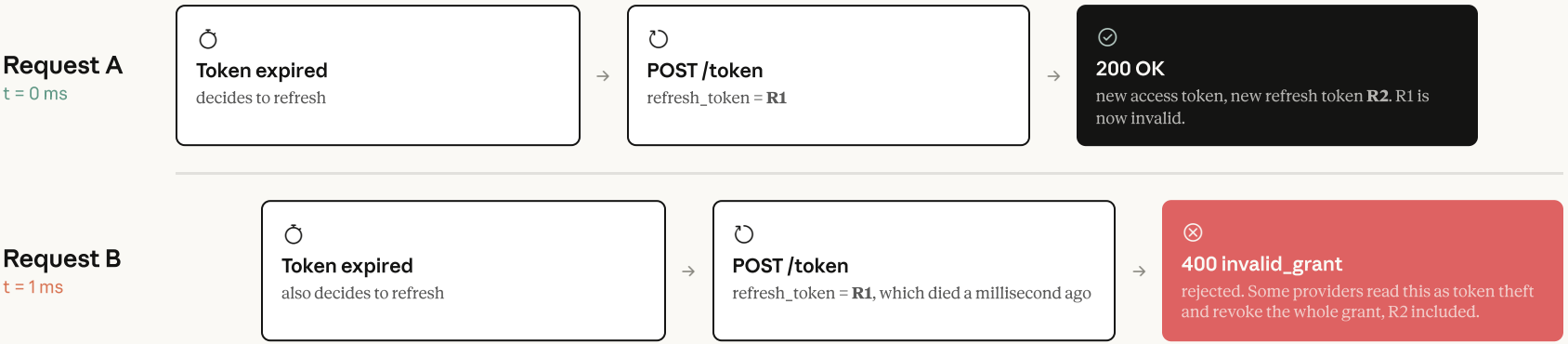
Getting the refresh right under concurrency

Getting a refresh token in the first place, when possible

Getting the refresh right under concurrency

Knowing when a refresh failure is recoverable or not

Concurrent refreshes can kill the token they're trying to save

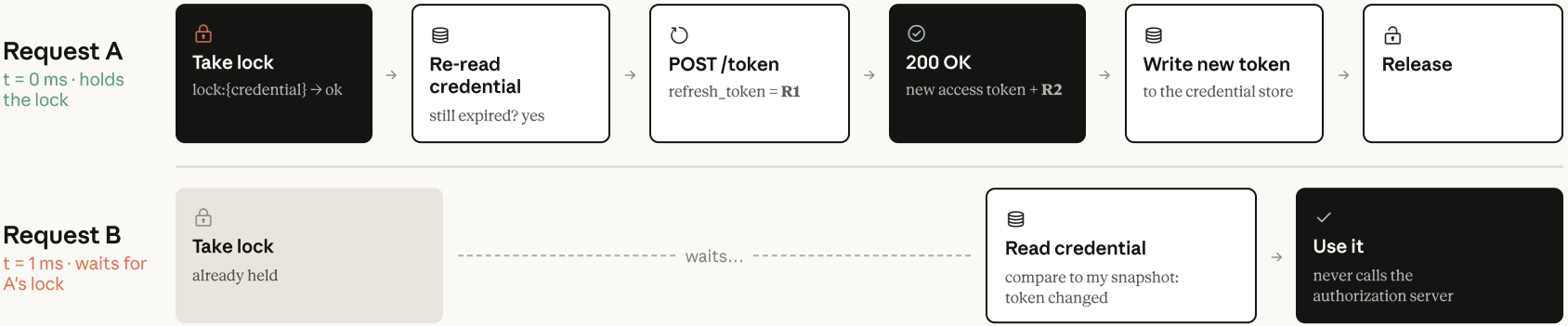


A user who was happily connected gets "please reconnect", caused entirely by the proxy.

It scales with activity: the heavier the user, the more certain the failure.

Use a distributed lock to ensure one refresh at a time

A short-lived lock on the credential. Whoever holds it talks to the authorization server; everyone else reads the store.



↔ Ten concurrent requests, one call to the authorization server.

Same if A's refresh had failed: A writes the failure, B returns it. Still one call.

The lock prevents invalidations caused by concurrent requests

Refresh early

In the last ~5% of the token's lifetime, with a sensible minimum and maximum, so everyone else still holds a working token while one request refreshes.

Two waiter timeouts

Still holding a valid token → wait a few seconds, then carry on with it.
Token already dead → wait up to the full lease; it's the only option.

Waiters read failures too, not just success

If they only looked for a new token, a failed refresh would send every waiter to refresh again. The holder records the error; waiters see it and back off.

Knowing when a refresh failure is recoverable or not

Getting a refresh token in the first place, when possible

Getting the refresh right under concurrency

Knowing when a refresh failure is recoverable or not

The theory

The OAuth RFCs define exactly what a failed refresh looks like, and what each error means

RFC 6749 §5.2: the token endpoint answers `400` (or `401` for a bad client) with a JSON body whose `error` is one of six values. RFC 6750 does the same for resource requests.

If every server did this, “retry or re-auth?” would be a lookup table.

RFC 6749 §5.2 · token endpoint error

```
HTTP/1.1 400 Bad Request
Content-Type: application/json

{ "error": "invalid_grant", "error_description": "…" }
```

the six values, and what they tell a client

`invalid_grant`	refresh token is dead → ask the user again
`invalid_client`	your registration is the problem
`unauthorized_client`	this client may not use this grant
`invalid_request`	you built the request wrong
`unsupported_grant_type`	server doesn’t do refresh at all
`invalid_scope`	asked for more than was granted

In practice

~25% of refresh failures don't look like anything in the oauth-compliant error

74%

Spec-compliant

72.5% 400 with a real §5.2 code

1.4% 401 with `invalid_client`, the one 401 the spec allows

10%

HTTP server errors

9.3% 5xx with a non-OAuth body or HTML

0.9% Cloudflare 52x / 530 edge errors

16%

Everything else

6.5% 404, endpoint gone or moved

5.4% 401 / 403 with the wrong or a made-up code

3.3% 400 / 429 whose body isn't an OAuth error, or a vendor code

0.6% 2xx with no token, and stray 405 / 410 / 415…

Knowing when a refresh failure is recoverable or not

Ideally, the refresh error tells you whether it's recoverable

When the server is compliant, it mostly does:

`invalid_grant`	The refresh token is no longer valid. You will not refresh it again.	⊗ Dead. Re-auth.
`invalid_client`	Depends on how the client was registered:	
DCR	The registration is gone, and the token with it.	⊗ Dead. Re-auth.
CIMD	Most likely a blip fetching the metadata document.	↻ Retry.
Pre-registered	Secret rotated or app disabled. The credential's owner can fix it.	🔧 Recoverable, not by the user.
5xx	A plain HTTP server error, sometimes with a `Retry-After` header.	↻ Retry with backoff.
Anything non-compliant	Back to the same world of quirks as getting the refresh token: learn each one, interpret it.	? It depends.

For authorization server developers

Use OAuth-compliant errors, and mean them literally. It's what lets a client tell recoverable from not.

Knowing when a refresh failure is recoverable or not

Ideally, the error tells you whether it's recoverable

When the server is compliant, it mostly does:

`invalid_grant`	The refresh token is no longer valid. You will not refresh it again.	⊗ Dead. Re-auth.
`invalid_client`	Depends on how the client was registered:	
DCR	The registration is gone, and the token with it.	⊗ Dead. Re-auth.
CIMD	Most likely a blip fetching the metadata document.	↻ Retry.
Pre-registered	Secret rotated or app disabled. The credential's owner can fix it.	Recoverable, not by the user.
5xx	A plain HTTP server error, sometimes with a `Retry-After` header.	↻ Retry with backoff.
Anything non-compliant	Back to the same world of quirks as getting the refresh token: learn each one, interpret it.	? It depends.

For authorization server developers: use OAuth-compliant errors, and mean them literally. It's what lets a client tell recoverable from not.

Knowing when a refresh failure is recoverable or not

Will retrying work in this case?

Example of a real error we got when refreshing

```
# POST /oauth/token  grant_type=refresh_token

HTTP/1.1 400 Bad Request
Content-Type: application/json
Content-Length: 67

{
  "error": "server_error",
  "error_description": "..."
}
```

Status: compliant
RFC 6749 §5.2 says 400

Shape: compliant
JSON body with an `error` field

Value: not on the list for /token, only for /authorize
§5.2 allows exactly these at the token endpoint:

`invalid_request`	malformed request
`invalid_client`	client authentication failed
`invalid_grant`	token expired, revoked, or already used
`unauthorized_client`	client may not use this grant
`unsupported_grant_type`	server doesn't do refresh
`invalid_scope`	scope not allowed

In this case, yes: retrying worked

But non-compliant error values make it hard to know when a token is really gone and when the server just had a bad moment.

```
# POST /oauth/token  grant_type=refresh_token

HTTP/1.1 400 Bad Request
Content-Type: application/json
Content-Length: 67

{
  "error": "server_error",
  "error_description": "..."
}

↻ retry after 2s  →  HTTP/1.1 200 OK
```

Status: compliant
RFC 6749 §5.2 says 400

Shape: compliant
JSON body with an `error` field

Value: not on the list for /token, only for /authorize
Not one of the six §5.2 values, so the response alone can't tell us.

Knowing when a refresh failure is recoverable or not

Handle each kind of error on its own terms, and let the server's `401` have the final say

Compliant errors, known non-compliant quirks and everything else each get their own failure handling. Whatever the refresh said, the token stays valid until the MCP server rejects it.

Compliant errors

Read with confidence

`invalid_grant` is final. `invalid_client` depends on how the client was registered. A 5xx is transient. The response tells us which.

Non-compliant server quirk errors

Learn them, then treat them as terminal

Some authorization servers return their own error for a revoked or expired grant. Once we know a server's error, we can recognise it and stop retrying immediately.

Everything else

Back off, keep trying, then give up

Retry with backoff for a bounded period. If refreshing still fails and the server still answers 401, mark the connection dead and sign the user out.

What would help

More compliant errors, fewer wasted refreshes

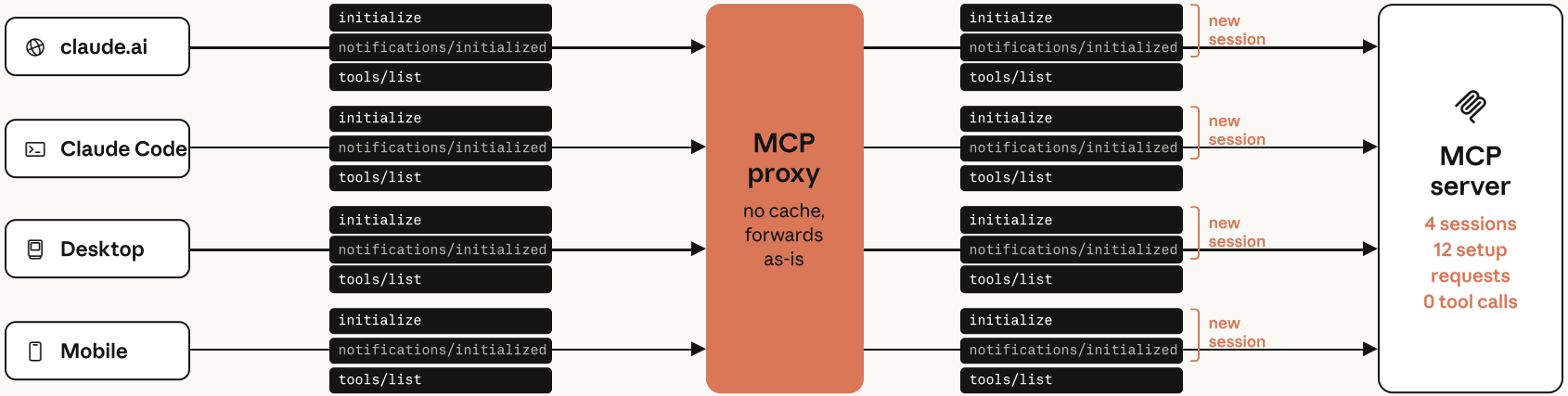
If more authorization servers used the spec's error codes literally, clients could decide immediately and stop sending refresh requests that were never going to work.

Part two

Cutting upstream requests

Most of what we sent upstream was wasteful setup requests

Every surface repeats `initialize` → `notifications/initialized` → `tools/list` per user, per server. On top of that, the same `tools/list` was re-sent constantly, and we forwarded all of it.

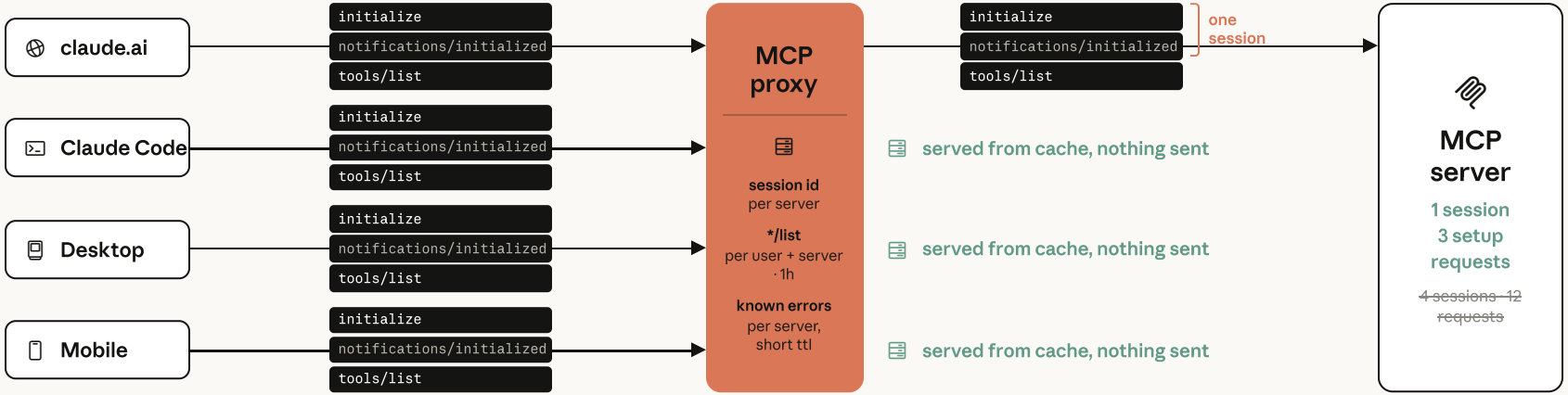


One user, one server. None of it does anything for them, and a partner pointed it out before our dashboards did.

So we started caching at the proxy

Servers gave no hint how long a list stays valid, so one hour was our in-between: worth caching, still picks up changes the same day.

MCP session id	per user + server
Every `*/list` result	per user + server · 1h TTL
Known-broken servers	per server · short TTL



Nothing changed for the clients. Everything changed for the server.

But this is still not good enough

We cache lists per user because the spec hints that tools *may* differ per user. In practice they almost never do.

We took a deterministic hash of every `tools/list` response in our logs and compared it across users of the same server.

So per-user keys mostly multiply cache entries and upstream calls for a case that doesn't happen. The ideal is to cache per server, and per user only when the server really has user-specific tools.

From our logs

~92%

of servers return the same tool list to every user

Identical `tools/list` hash across all users of the server.

With the 2026-07-28 spec revision, you can have the cache granularity per server!

The server now tells you whether its lists are the same for everyone (public), and for how long you may cache them.

Since the 2026-07-28 spec revision

The server response should now tell you how you can cache it

`ttlMs`
How long you may reuse the response.

`cacheScope: "public"`
Any shared proxy may serve it to any user. The per-server cache.

`cacheScope: "private"`
Not shared across credentials. The per-user cache. We key it on the stored connection, not the token bytes, so it survives a refresh.

tools/list · in the JSON-RPC result body, not headers

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "resultType": "complete",
    [ … ]
    "ttlMs": 300000,
    "cacheScope": "public"
  }
}
```

The same revision removed sessions and the `initialize` handshake. The session cache is now history: a workaround for a cost the protocol no longer has.

Part three

Understanding server health

An MCP server can fail at different layers

Network	**Never answer at all** DNS doesn't resolve, connection refused, TLS fails, timeout. No HTTP response.	`ENOTFOUND` `ECONNREFUSED` `ETIMEDOUT`
HTTP	**Answer with an error status** A real 4xx or 5xx from the server or something in front of it.	`HTTP 502 Bad Gateway`
JSON-RPC	**Return 200 with a JSON-RPC error** The protocol saying the request failed: unknown method, bad params, server threw. No result; the model never sees it.	`200 {"error": {"code": -32603, "message": …}}`
Tool result	**Return 200 with isError set** The call succeeded; the tool is reporting that what it tried to do failed. This one is meant for the model.	`200 {"result": {"content": […], "isError": true}}`

Some error codes have contextual meaning

The same status can be routine or a real problem depending on when it happened and what we sent.

404

Session terminated

The server dropped a session ID it no longer knows. Start a new session and carry on.

or

Endpoint does not exist

Wrong URL, or the server moved. Nothing will work until the config changes.

401

Token no longer valid

We had a token, sent it, and the server rejected it. Refresh, or ask the user to re-auth.

or

No token was sent

We couldn't fetch it internally and the request went upstream without one. That's on us.

Understanding the context is what lets you read the health of the MCP server correctly.

To normalise all of that, every failure maps to an `error_code` and a `fault`

A sample of the mapping we use today. The full list is longer and keeps growing as we add more cases.

Status	Context	error_code	fault
401	Stored grant is dead, refresh cannot help	`mcp_invalid_oauth_token`	`user_auth_expired`
	The user never connected this server · excluded from the error rate	`mcp_unauthorized_no_token`	`expected_noauth`
404	The URL answers like a website, not an MCP server	`mcp_endpoint_not_found`	`misconfiguration`
400	A stream was opened before any handshake	`mcp_session_id_missing`	`misconfiguration`
502	The server dropped the connection or answered garbage mid-stream	`mcp_connection_error`	`upstream_server`
500	We couldn't fetch the stored token internally · the request never went upstream	`mcp_token_fetch_failed`	`internal`
429	The authorization server rate-limited our refresh · back off, token may still work	`mcp_upstream_auth_rate_limited`	`upstream_auth_server`

We use a single structured log per request

Everything that wouldn't fit in a metric: method, credential state, what the server said, what we said.

Anonymised, keyed by server and by source. Directory servers are curated, so their errors mean real breakage; user-added ones carry configuration noise.

Still missing

Sharing this with server developers. If I were to build an MCP proxy from scratch, I'd include it from day one.

```
{
  // one event per proxied request
  "event": "mcp_proxy_request_completed",
  "mcp_method": "tools/call",
  "server_host": "mcp.example.com",        // hashed or normalised, never the full URL
  "installation_source": "directory",      // directory | user_added
  "client": "web_app",                     // which of your products made the call

  // the verdict
  "outcome": "error",                      // success | error
  "error_code": "mcp_invalid_oauth_token", // typed, stable, also sent to the caller in a header
  "fault_domain": "user_auth_expired",     // none | internal | upstream_server | upstream_auth_server
                                           // | user_auth_expired | misconfiguration | expected_noauth
  "exclude_from_error_rate": false,        // true only for "user never connected" 401s

  // what credential we went out with
  "credential": {
    "outcome": "has_token",                // has_token | no_token | invalid_token | store_unavailable
    "refresh_outcome": "failed",           // skipped_not_needed | succeeded | failed | skipped_backoff …
    "has_refresh_token": true,
    "expires_in_seconds": -31              // already expired when we sent it
  },

  // what the MCP server answered
  "upstream": {
    "status_code": 401,
    "content_type": "application/json",
    "retry_after": null,
    "jsonrpc_error_code": null,            // set when a 200 body is a JSON-RPC error
    "tool_result_is_error": null           // set when tools/call returned isError: true
  },

  // what we told our own caller
  "response": { "status_code": 401 }, // same status; the credential block says why

  // caching
  "cache_result": "skipped"                // hit | stale | miss | skipped; tools/call is never cached
}
```

To close

Wrapping up

What I would tell MCP authorization server developers

Use CIMD or a pre-registered client instead of DCR, as the spec now recommends

With DCR you own a database of clients you didn't create and can't contact.

- Registrations expire or get purged and the client only sees `invalid_client`.
- With CIMD the `client_id` is a URL you fetch and cache. Nothing to store, and you can see who the client is.
- A handful of known clients? A static id and secret is simpler than either.

Be OAuth-compliant with your errors

A proxy can only react as fast as it can recognise the error.

- Use the spec's status codes and the `{"error": …}` JSON shape.
- Mean the codes literally: `invalid_grant` only when the grant is really dead.
- When you're down, say so: `503` or `429` with `Retry-After`.

What I would do if I had to build an MCP proxy today

Get the refresh story right
Mind how you get a refresh token. Refresh behind a distributed lock so your own requests can't race. Store all the OAuth state the refresh needs up front.

Use caching to avoid wasteful calls
Make use of the new caching features of the new spec.

Structured logging
One line per request with everything in it, so you can see what is going on.

Custom, contextual error codes
Each with a fault domain: client, proxy, upstream server or authorization server.

Make server health something available to server developers from day one
Think through every layer an MCP server can fail at, and build it so that view can go back to the people who run the server.

Two more I didn't get to today

Build in support for per-server authorization server metadata overrides
Plenty of servers have their own endpoints or wrong metadata. Bake a per-server option from the start.

Cache the auth discovery
The well-known documents and metadata don't change between users. Key on the server, give it a TTL.

ANTHROP\C

Thank you

ありがとうございました

Camila Rondinini

MCP Connectivity · Anthropic, London