MCP Dev Summit

2026

Protocol Evolution:

Adapting MCP for SLMs and the Edge

From cloud-native to local

Prepared by
Kierra Dotson

MCP + The Cloud Native Problem

The development of MCP is cloud-centric. Most current Model Context Protocol (MCP) implementations focus on connecting LLMs to remote SaaS tools, cloud databases, and centralized APIs

(1)

High bandwidth

Assumes network connections are high speed with large data-carrying capacity

(2)

Persistent connectivity

Assumes that the connection between the host and the server will stay alive for the duration of the session

(3)

Cheap compute

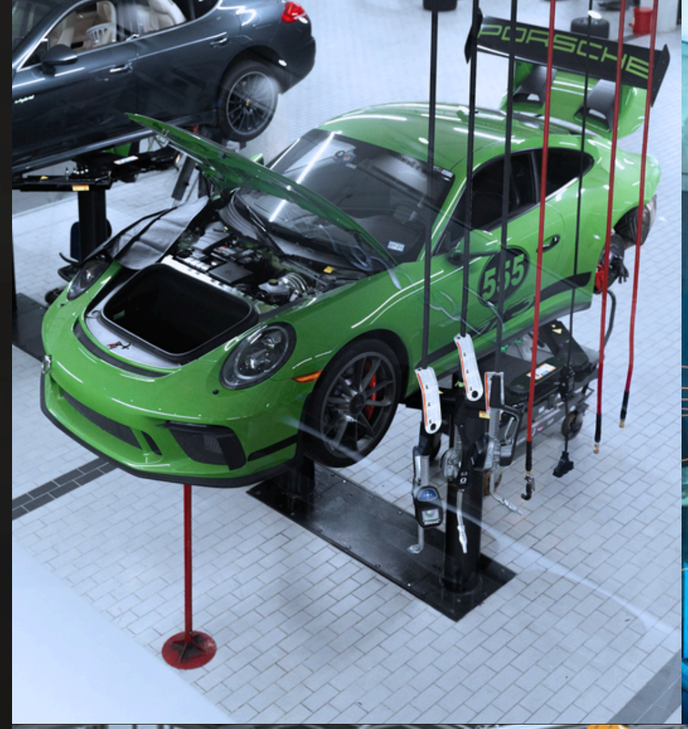
Assumes companies have a massive token budget + tokens are cheap

Use Case: Autonomous Vehicle Manufacturing

The goal is to build out an AI powered production line that can detect product defects in real time.

CONSTRAINTS:

- need to use small language models
- itermittent wifi
- network deadzones
- power management



Introducing bMCP

(1)

Idea

Replace JSON-RPC on the wire with Protocol Buffers.

- Step 1: Client advertises binary support during handshake
- Step 2: Protobuf schema compiled once at startup — not interpreted at runtime
- Step 3: Tool call serialized to bytes — no field name strings, no JSON structure
- Step 4: Server responds in binary — client deserializes to native struct

(2)

JSON

human-readable text

```
{
  "tool":"check_weld",
  "unit_id":"MFG-4471",
  "sensor":"ultrasonic"
}
```

(3)

Protobuf

compact sequence of bytes

```
[0x0A 0x0B 0x12 0x08...]
```

Semantic Tool Compression

Problem

Tool schemas are incredibly verbose and token consuming

Solution

Let the client decide when to pay the “context tax” for tool schemas durng discovery time.

Add a new field to the capabilities object:

- ClientSignal: capabilities.experimental.semanticCompression: true

Full returns the standard complete JSON schema
Semantic returns a compressed one-liner

Regular description

```
{
  "name": "trigger_rejection_arm",
  "description": "Activates the pneumatic
rejection ....",
  "parameters": {
    "type": "object",
    "required": ["position", "severity"],
    "properties": {
      "position": {
        ...
      },
}
```

Semantic description

```
trigger_rejection_arm | Diverts flagged
unit off conveyor via pneumatic arm |
required: position(int 1-9999),
severity(low|moderate|critical) |
optional:
defect_type(weld_crack|panel_misalignme
nt|paint_void|foreign_object),
confidence_score(float 0-1) |

                ,,,
```

Dynamic Schema Pruning

Problem

‘tools/list’ returns more information than necessary for the task at hand

Solution

Let the client pass context to the server so that it will recieve a filtered list of tools that are relevant to the task

MCP + Network Connectivity

MCP assumes the host will always be available and networks will be stable, where there will always be a cloud server to handle all MCP server requests

(1)

SSE

Requires a consistent HTTP connection. If network has a blip, client has to re-establish the connection

(2)

STDIO

a 1:1 relationship between the MCP client and server but if that process crashes or the device goes to sleep, the server dies

(3)

Either way

both are incapable of handling the intermittent network connection issues that come with edge devices

Store-and-Forward with MQTT

Proposal

move from a synchronous request-response model to a more Resilient Event-Driven Architecture via Message Queues. This way the network can drop without the data/message dropping as well.

MQTT is a lightweight messaging protocol designed for environments with high latency and unreliable networks.

Steps

- Step 1: Tool call intent → written to local SQLite outbox (before touching the network)
- Step 2: MQTT client monitors outbox → publishes when connected
- Step 3: Broker holds message in flight
- Step 4: Server processes → result retained by broker
- Step 5: Client reconnects → result delivered immediately

Stateful Sessions

MCP Servers are stateless by default. This makes the server simple and easy to scale, but it also means you cannot support features like user authentication, progress tracking, or personalized responses.

By including a unique Session ID and Sequence Numbers in the MCP header, the client and server can synchronize their state after a long period of disconnection. The server can 'replay' any missed notifications, and the SLM can resume its reasoning exactly where it left off.

Initial connection — device online

```
{
  "mcp_version": "1.0",
  "session_id": "factory-edge-7f3a9c",
  "sequence_number": 1,
  "method": "tools/call",
  "params": {
    "name": "get_sensor_reading",
    "sensor_id": "B7-ULTRA-02"
  }
}
```

Device goes to sleep. Wakes up. Reconnects.

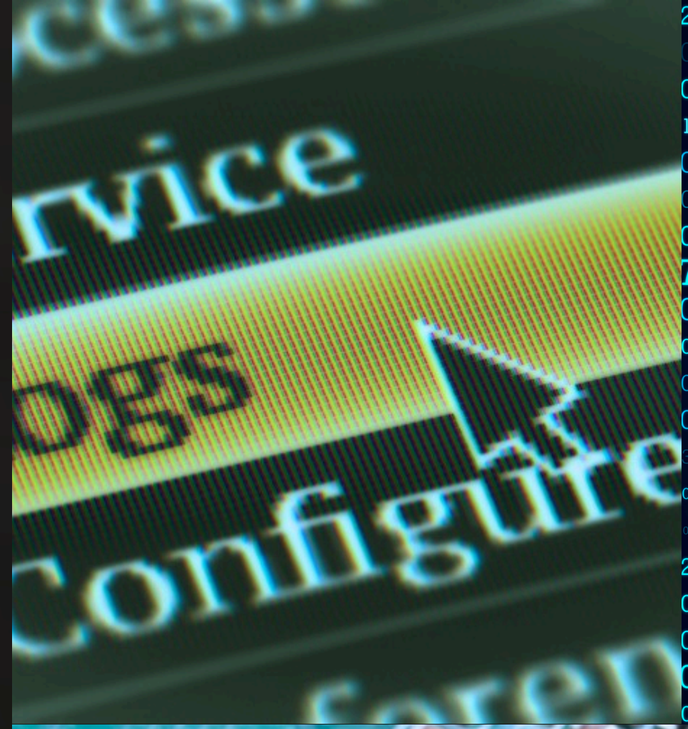
```
{
  "mcp_version": "1.0",
  "session_id": "factory-edge-7f3a9c",
  "sequence_number": 47,
  "method": "session/resume"
}
```

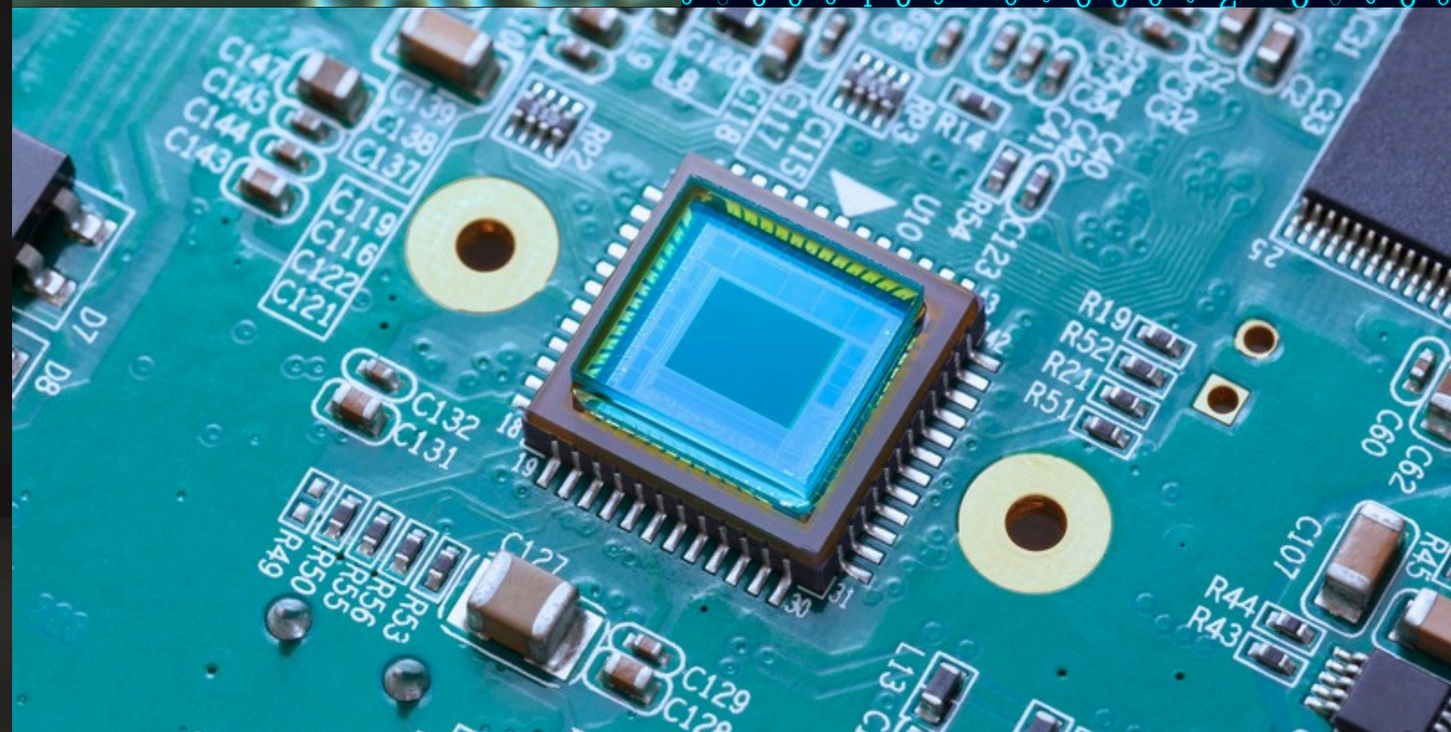
Server response — replaying what was missed

```
{
  "session_id": "factory-edge-7f3a9c",
  "last_server_sequence": 52,
  "client_last_sequence": 47,
  "replaying_missed": 5,
  "status": "resuming"
}
```

MCP + Data Tax

Every single resource request returns the full payload of data, which is useful for data that changes frequently. It becomes a burden when data does NOT change frequently





CAC : Content-Addressable Context

Before sending resource payload, the server sends a hash to the client to verify if the client has a record of the data. If the hash matches, the client sends an acknowledgement. If not, the client requests the full payload and the server will send the full resource payload.

Server sends hash

```
{
  "session_id": "factory-edge-7f3a9c",
  "resource": "mcp://sensors/conveyor-belt-speed",
  "hash": "sha256:b7e2a1f4c9d0e3b6a8c1f5e2d4b9a7c0e3f6d1a4b8c2e5f9a0d3b7e1c4f8a2",
  "hash_only": true
}
```

Client checks cache

Hash is new— requests full payload

```
{
  "session_id": "factory-edge-7f3a9c",
  "resource": "mcp://sensors/conveyor-belt-speed",
  "status": "200_send_payload",
  "cached_hash": "sha256:a3f9c2d81e4b7f0c9d2e5a8b1f4c7e0d3a6b9c2e5f8a1d4b7e0c3f6a9d2e5b8"
}
```

Server sends full payload:

Client updates cache

```
{
  "session_id": "factory-edge-7f3a9c",
  "resource": "mcp://sensors/conveyor-belt-speed",
  "hash": "<updated_hash",
  "data": {
    "sensor_id": "LINE-A-CONV-01",
    "reading": 2.47,
    ...
  }
}
```

Local-First Resource Discovery

(1)

Idea

Bypass the RPC layer and give the server a direct link to a file

- Step 1: Client asks the MCP server what resources it has access to
- Step 2: The client sees the URI and opens direct stream for the vision model
- Step 3: The vision model notices a defect and calls MCP server
- Step 4: Server audits the defect and triggers rejection button

(2)

URI

```
{
  mcp://local/dev/video0
}
```

(3)

MCP Trigger

```
trigger_rejection_arm(position: 847, severity: 'critical')
```

Local-First Resource Discovery

(1)

Idea

Bypass the RPC layer and give the server a direct link to a file

- Step 1: Client asks the MCP server what resources it has access to
- Step 2: The client sees the URI and opens direct stream for the vision model
- Step 3: The vision model notices a defect and calls MCP server
- Step 4: Server audits the defect and triggers rejection button

(2)

URI

```
{
  mcp://local/dev/video0
}
```

(3)

MCP Trigger

```
trigger_rejection_arm(position: 847, severity: 'critical')
```

Conclusion

(1) binaryMCP

Handles commands and metadata.

Small, fast, binary

Great for reducing payload size

(2) Content-Addressable Context

Great for frequently polled yet slowly-changing resources.

Hash once, skip the rest until updates occur

(3) Resource Discovery

High-frequency streams. Data flows direct.

Thank You

Kierra Dotson

AI Strategy & Governance | Bridging the gap between business strategy and technology in...