MCP Dev Summit Nairobi

MCP at The Edge

Lessons from Running Agentic AI on KubeEdge

Israel Olatunji Tijani
Head of Research | ChatVE, Nigeria.

Scan to save contact

Session Roadmap

1

Why the Edge

The integration gap for local agents

2

Architecture

KubeEdge, SLM agent, local MCP server

3

Implementation

MCP server, GraphQL, PostgreSQL, YAML

4

Results

Latency heatmap, success rate ρ

5

Lessons

Trade-offs, benefits, open questions

The Integration Gap at the Edge

MCP is moving to production

but its use at the network edge remains largely unexplored

SLM agents can reason locally

inference agents on KubeEdge nodes, next to the devices

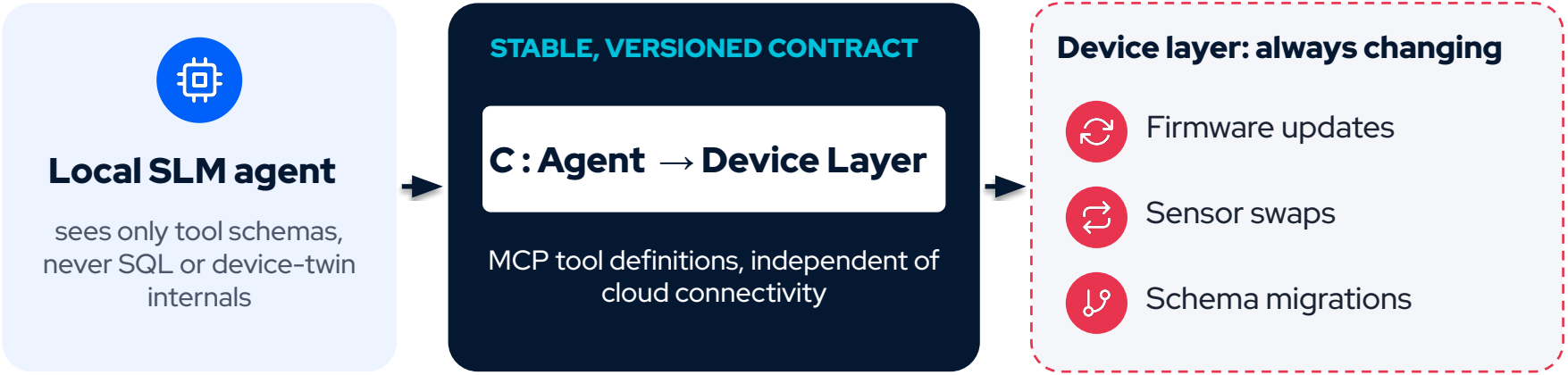
Device integration is ad hoc

data, metrics and management tools wired one by one

OUR APPROACH

Treat MCP as the **integration boundary** between a local agent and a constantly changing device layer.

What We Now Understand Differently



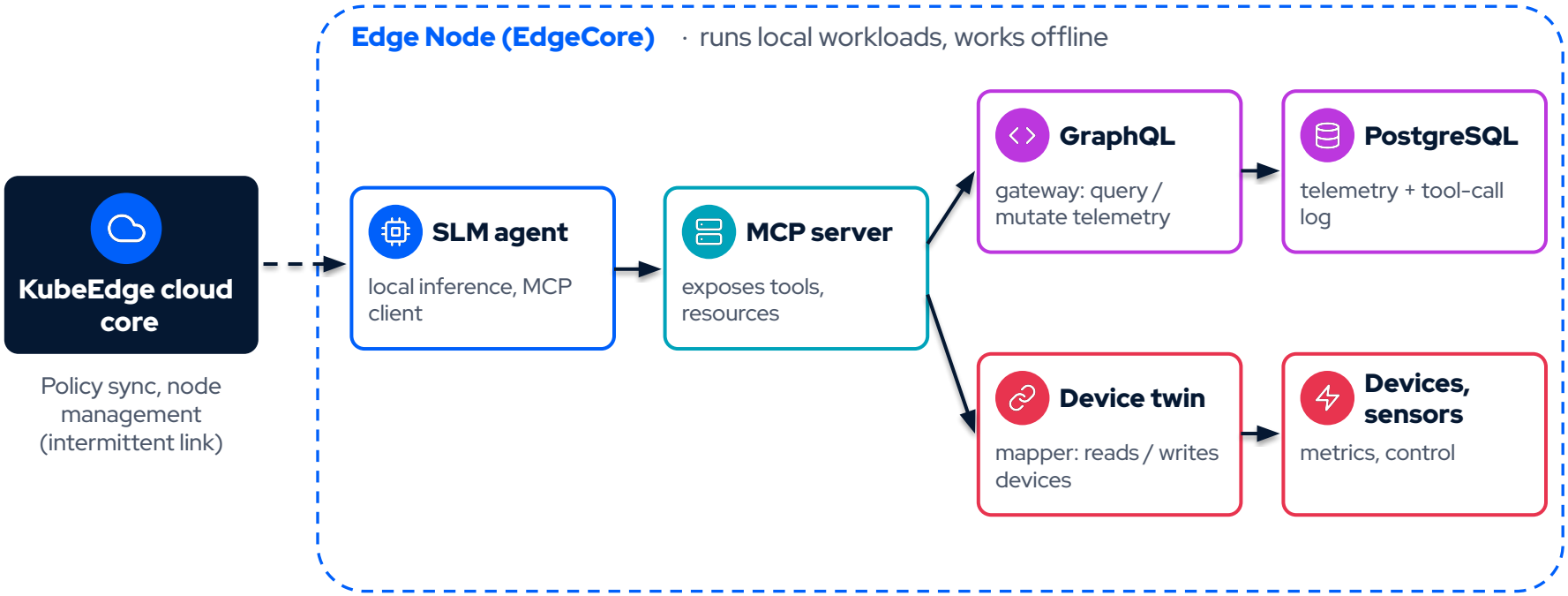
MCP became a **decoupling boundary** more than an AI-enablement layer: the contract survives change below it.

Architecture and Implementation



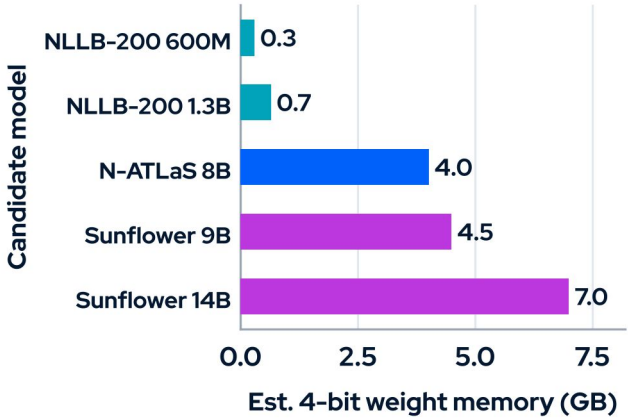
MCP
Dev Summit
Nairobi

End-to-End Architecture on KubeEdge



Choosing the On-device Language Stack

Option	Type	Sizes	Role at the Edge
N-ATLaS	Multilingual LLM (Llama-3 8B base), NCAIR Nigeria	8B	Agent SLM: Hausa, Igbo, Yoruba, Nigerian-accented English
Sunflower	Multilingual LLMs, Sunbird AI Uganda	9B, 14B, 32B	Agent SLM; 4-bit and GGUF builds for small nodes
NLLB-200	Translation model, Meta	0.6B, 1.3B, 3.3B	MCP tool: translate_text across 200 languages
WAXAL	African speech dataset, Google	Dataset	Fine-tune on-device speech-to-text and text-to-speech



$M \approx P \times 0.5$ bytes

4-bit weights only; excludes KV cache

Memory bars are estimates (P = parameters), not measurements. Sizes from each model's Hugging Face release.

A thin, Stateless MCP Server in Front of PostgreSQL

edge_mcp_server.py (Python MCP SDK)

```
@app.list_tools()
async def list_tools():
    return [Tool(
        name="read_device_metric",
        description="Read latest metric from a KubeEdge device twin",
        inputSchema={"type": "object",
            "properties": {"device_id": {"type": "string"}},
            "required": ["device_id"]})]

@app.call_tool()
async def call_tool(name: str, arguments: dict):
    if name == "read_device_metric":
        cur = psycopg2.connect(dsn=PG_DSN).cursor()
        cur.execute(
            "SELECT metric, value, recorded_at FROM device_metrics "
            "WHERE device_id = %s ORDER BY recorded_at DESC LIMIT 1",
            (arguments["device_id"],))
        return [TextContent(type="text",
                text=json.dumps(cur.fetchone(), default=str))]
```

1. **One tool, one JSON schema**
   The agent's entire view of the device layer

2. **Stateless shim**
   Parameterised SQL; no state held between calls

3. **Standard MCP output**
   TextContent with JSON; timestamps serialised

One Data Contract, Two Consumers

GraphQL: shared by dashboard and MCP server

```
type DeviceMetric {
  deviceId: ID!
  metric: String!
  value: Float!
  recordedAt: String!
}
type Query {
  latestMetric(deviceId: ID!): DeviceMetric
  metricHistory(deviceId: ID!,
                limit: Int = 50): [DeviceMetric!]!
}
type Mutation {
  actuateDevice(deviceId: ID!,
                command: String!): Boolean!
}
```

PostgreSQL: telemetry + tool-call audit log

```
create table device_metrics (
  id          bigserial primary key,
  device_id   text not null,
  metric      text not null,
  value       double precision not null,
  recorded_at timestamptz not null default now());
create index on device_metrics
  (device_id, recorded_at desc);

-- every MCP tool call is audited
create table mcp_tool_calls (
  id          bigserial primary key,
  edge_node   text not null,
  tool_name   text not null,
  latency_ms  integer not null,
  succeeded   boolean not null,
  called_at   timestamptz not null default now());
```

Pinning the agent and MCP server to the edge

edge-mcp-agent.yaml (abridged)

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: edge-mcp-agent
spec:
  replicas: 1
  template:
    spec:
      nodeSelector:
        node-role.kubernetes.io/edge: ""
      containers:
        - name: slm-agent
          image: registry.local/slm-mcp-client:latest
        - name: mcp-server
          image: registry.local/edge-mcp-server:latest
          env:
            - name: PG_DSN
              value: postgresql://.../telemetry  # localhost
```



Edge-only scheduling

nodeSelector keeps the pod on KubeEdge edge nodes



Agent and server in one pod

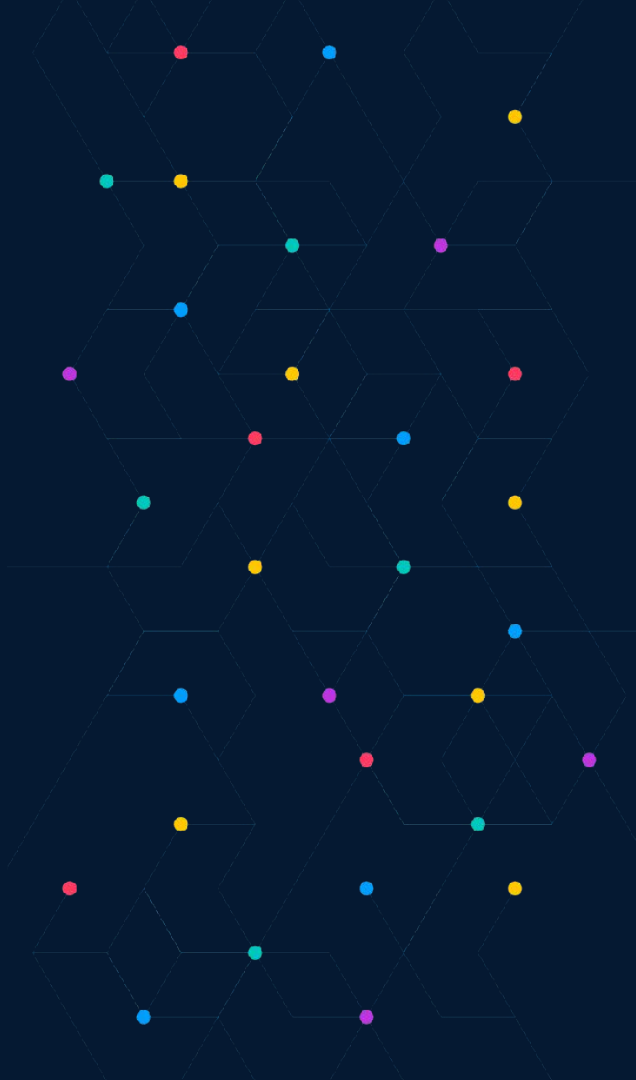
MCP calls stay on localhost, with no network hop



Local telemetry store

PostgreSQL reached on the node; works through outages

Results and Lessons



What We Measured, and How

Mean tool-call latency per node n and tool t

$$\bar{L}_{n,\,t} = \frac{1}{k}\sum_{i=1}^{k} L_i$$

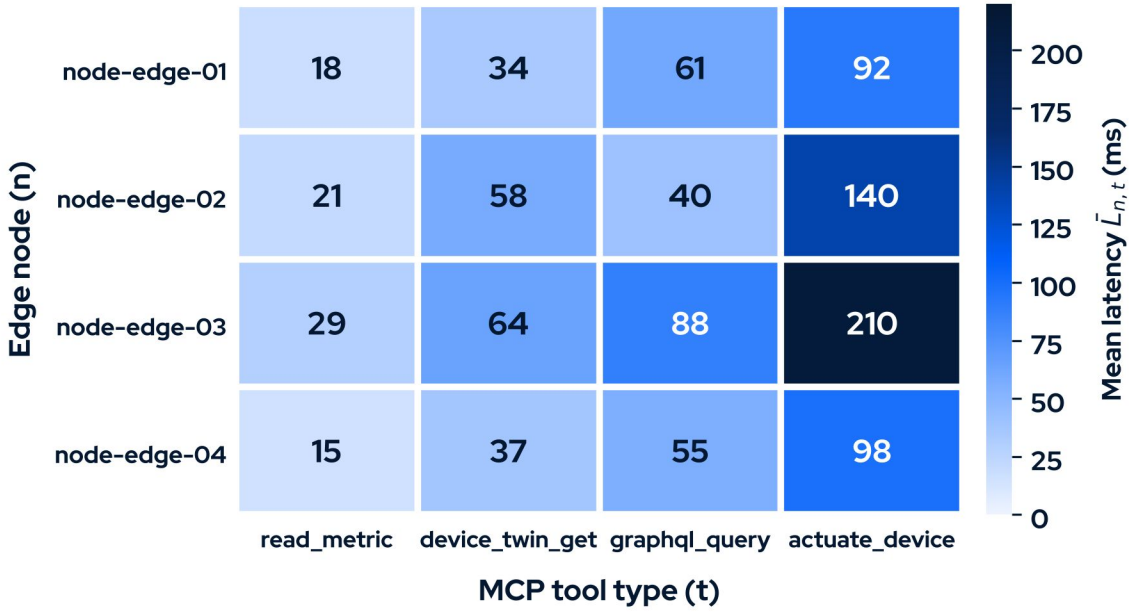
Aggregate tool-call success rate

$$\rho = \frac{\text{calls succeeded}}{\text{calls attempted}} \approx 0.97$$

L_i = latency of call i; k = calls observed for that (node, tool) pair in the collection window

Logged field	Used for
edge_node	node index n
tool_name	tool type t
latency_ms	L_i, then the mean per (n, t)
succeeded	success rate ρ
called_at	collection window
resource use	logged; not reported

Tool-call Latency by Node and Tool



Mean latency per node and tool (ms)

Actuate_device is slowest

Highest mean latency on all four nodes

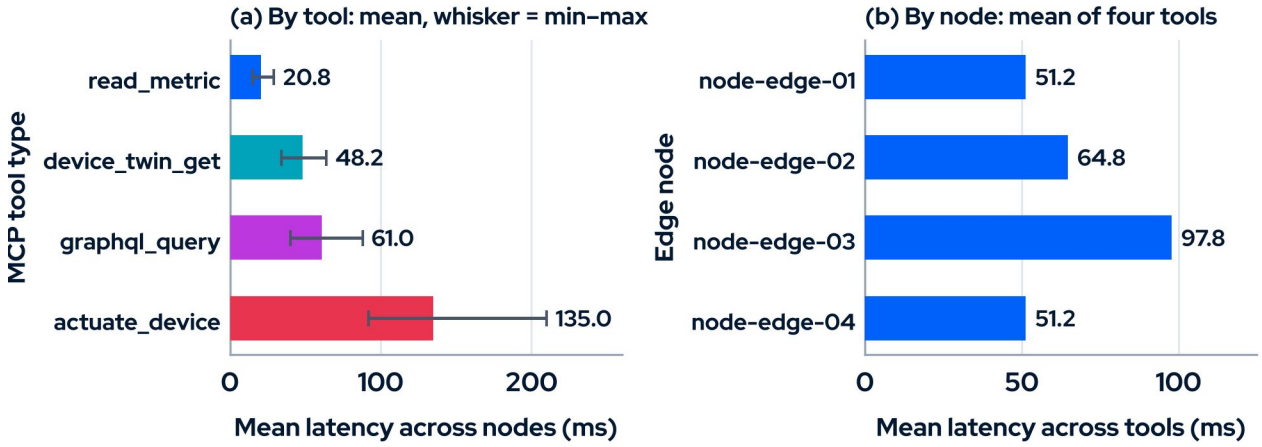
Peak: 210 ms

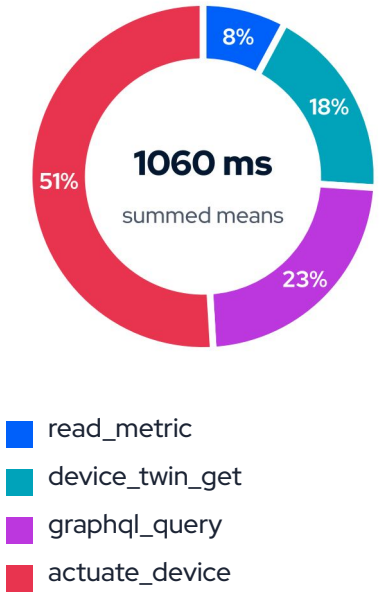
Node-edge-03, the least-cached node

Not protocol overhead

Cold starts and device round trips drive it

Where the Latency Goes

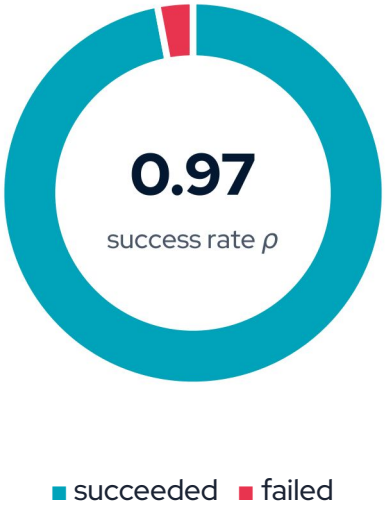




51% of summed mean latency comes from actuate_device: the device round trip, not MCP, dominates

Derived from the paper's Figure 2 values.. Donut: share of the 1,060 ms sum of the 16 cell means.

Reliable Overall, with Three Real Constraints



Intermittent connectivity

the cloud-core link drops; the local loop has to keep working

Schema drift

GraphQL gateway and updated MCP tool definitions fall out of step

Cold-start latency

up to ~210 ms for actuate_device on the least-cached node

Design Trade-offs We Made

Decision	What it gave us	What to watch
MCP server as a thin, stateless shim	Simple agent surface: tool schemas only, no SQL or twin internals	Tool definitions must track gateway changes
GraphQL as the shared data contract	Dashboard and MCP server share one contract; no duplicated queries	Schema drift between gateway and MCP tools
Fully local topology on KubeEdge	Agent keeps working through cloud outages	Cold starts on resource-constrained nodes
MCP as the decoupling boundary	Firmware, sensor and schema changes absorbed below the contract	No spec-level tool versioning or negotiation yet

What this Adds to the MCP Ecosystem

A non-cloud deployment pattern

Fully local, intermittently connected MCP on KubeEdge, beyond the always-connected default

Failure-mode data

ρ ≈ 0.97 and cold starts up to ~210 ms: grounds for timeout, retry and caching conventions

A worked integration

GraphQL, PostgreSQL and Kubernetes/KubeEdge: lower barrier for teams on these stacks

Early spec gaps

Tool versioning and negotiation; connection lifecycle on constrained hosts

Open Questions for the Spec and SDKs

1

Timeouts, retries and caching

What should clients assume when a tool call takes 200 ms on one node and 20 ms on another, or the link drops mid-call?

2

Tool versioning and negotiation

How does a server change a tool schema without silently breaking an agent already deployed in the field?

3

Connection lifecycle on small hosts

When should a constrained host keep connections warm, and when should it release them?

MCP
Dev Summit
Nairobi

Thank you

Israel Olatunji Tijani · israel@chatve.co



MCP at the Edge: Lessons from Running Agentic AI on KubeEdge