NATURAL LANGUAGE ACCESS TO VEHICLE TELEMETRY VIA THE MODEL CONTEXT PROTOCOL

Valer Bocan[1] Bianca Gușită[2]

Abstract: Fleet management systems generate continuous GPS telemetry, yet operators access it through specialized dashboards that handle planned queries well but ad-hoc, cross-vehicle questions poorly. The Model Context Protocol (MCP), introduced by Anthropic in 2024, standardizes how large language models invoke external tools, but its use for IoT telemetry is largely unexplored. This paper describes an architecture that bridges a production GPS tracking system to LLMs through an MCP server exposing four atomic telemetry tools. The Go-based tracking backend receives data from two vehicles over the H02 and TLT2H protocols and stores it in MongoDB behind a REST API; a .NET 9 MCP server translates between this REST surface and the MCP protocol. A structured evaluation over 30 queries against multi-year per-vehicle history shows the LLM autonomously chains multiple tool calls without custom prompting. The server is publicly reachable at https://gpstracker-mcp.dataman.ro/ for reproducibility.

Key words: Model Context Protocol, GPS tracking, fleet management, large language models, natural language interface, Internet of Things, cyber-physical systems.

1. INTRODUCTION

Commercial vehicle fleets generate vast amounts of location data. A mid-sized fleet of 500 trucks, each reporting GPS coordinates every 10 seconds, produces over 4.3 million data points per day. Fleet operators rely on this data for route optimization, regulatory compliance, and driver-behavior monitoring. The standard interface to such data is a web dashboard backed by a geographic information system (GIS), where operators filter by vehicle and date range to inspect routes and statistics. These dashboards have matured over two decades and remain effective for their designed purpose.

Dashboards work well for planned queries: an operator who knows which vehicle to inspect and which date to examine can navigate to the relevant view in a few clicks. They work poorly for ad-hoc questions that span multiple vehicles, require aggregation, or combine location with contextual information. “Which vehicle spent the most time stationary last Thursday?” or “Where was the red car when the white car arrived at the warehouse?” demand multiple manual lookups and mental composition of results. As fleets scale, the gap between the data available and the data accessible through point-and-click interfaces widens.

Large language models have transformed how users interact with structured data in other domains, from natural language database access [1] to tool-augmented question answering [2]. The missing ingredient for fleet telemetry is a standardized bridge between the LLM and the tracking system. Custom API integrations are brittle and model-specific; each new LLM provider requires a separate adapter, and each new data source demands prompt or fine-tuning changes. What is needed is a protocol-level solution that decouples the domain data from the AI model consuming it.

The Model Context Protocol (MCP) [3] fills this gap. Released by Anthropic in November 2024, MCP provides a vendor-neutral standard for LLMs to discover and invoke tools exposed by external servers. The key property for our purposes is that the server describes its tools once, and every connected model reasons over those descriptions independently. This separation of concerns maps naturally onto IoT architectures, where device infrastructure and application logic already operate as independent subsystems.

This paper is an applied systems contribution rather than a new algorithm. Our contributions are:

- The first reported integration of MCP with live GPS fleet telemetry, separating the GPS backend, the MCP interface layer, and the AI client as independently deployable components.

[1] PhD, CSSLP, Lecturer, Department of Computer and Information Technology, Politehnica University of Timișoara, Vasile Pârvan Blvd. 2, 300223 Timișoara, Romania, valer.bocan@upt.ro

[2] Eng., PhD Candidate, Teaching Assistant, Department of Computer and Information Technology, Politehnica University of Timișoara, Vasile Pârvan Blvd. 2, 300223 Timișoara, Romania, bianca.gusita@cs.upt.ro

- A production implementation comprising a Go-based GPS tracker receiving data from two hardware devices over the H02 and TLT2H protocols, and a .NET 9 MCP server exposing four telemetry tools, publicly reachable at https://gpstracker-mcp.dataman.ro/.
- An empirical evaluation on real telemetry spanning multiple years of per-vehicle history, showing that the LLM autonomously chains multiple tool calls to answer compositional fleet queries, and a quantitative measurement of tool selection accuracy, answer correctness, and end-to-end latency across query categories.
- A reusable design pattern that embeds domain metadata directly into MCP tool descriptions and is in principle applicable to other IoT telemetry domains, with explicit scope limits.

2. BACKGROUND AND RELATED WORK

GPS-based fleet management is a mature industry segment. Commercial platforms such as Geotab and Samsara combine hardware trackers with cloud-hosted analytics dashboards into end-to-end solutions [4]. On the open-source side, Traccar supports over 200 GPS protocols and runs as a self-hosted server [5]. All of these systems expose data through graphical interfaces designed for known queries.

Bridging natural language to structured data is not a new problem. Text-to-SQL systems translate English questions into database queries, and recent LLM-based approaches have pushed accuracy on benchmarks like Spider and BIRD past 85% [1]. Fleet telemetry, however, does not sit in a relational database that SQL can reach directly. The data lives behind proprietary APIs encoded in domain-specific protocols, spread across time-series collections optimized for geospatial queries rather than ad-hoc joins. A different integration strategy is needed: one where the LLM invokes domain tools rather than writing database queries.

Schick et al. introduced Toolformer, showing that LLMs can learn to call external APIs as part of their text generation process [2]. Subsequent work by Qin et al. generalized this into multi-tool orchestration, where the model plans a sequence of API calls to satisfy a complex request [6]. Commercial LLM providers now ship native tool-use capabilities: OpenAI's function calling, Anthropic's tool use, and Google's function declarations each allow the model to emit structured invocations during inference. What these capabilities lack is a vendor-neutral protocol for tool discovery and invocation; each provider defines its own schema format and calling convention.

MCP [3] addresses this fragmentation by defining a client-server architecture in which an MCP server advertises tools with typed parameters and natural language descriptions, and an MCP client (typically an LLM runtime) discovers and invokes them via JSON-RPC. The protocol supports multiple transport layers, including HTTP with Server-Sent Events and standard I/O streams. Each tool definition carries a description that the LLM reads at inference time to decide when and how to invoke it. Hou et al. provide a systematic analysis of the MCP ecosystem and identify security considerations for production deployments, including input validation and rate limiting [7].

Connecting natural language to IoT data specifically has attracted growing interest. Xu et al. built IoT-NLI, a natural language query interface for IoT platforms that uses hierarchical semantic parsing to map questions to platform API calls across agriculture and smart home domains [8]. Their system requires hand-crafted grammars for each new domain. Zong et al. surveyed the broader integration of LLMs with IoT applications and identified sensor data processing as a key use case where LLMs can interpret and reason over time-series readings [9]. Closest to our work, Rayfield et al. applied ReAct-style LLM agents to query historical and real-time sensor data from industrial SCADA systems [10]. Their agents use custom prompts that hard-code the available API endpoints; adding a new data source requires rewriting the agent's prompt template.

Our approach differs from IoT-NLI and ReAct-based agents in two respects. First, MCP replaces hand-crafted grammars and hard-coded prompt templates with runtime tool discovery, so new data sources require no client-side changes. Second, we target mobile GPS telemetry rather than stationary industrial sensors, which introduces challenges around continuous location streams with timezone handling and the resolution of informal vehicle references ("the red car") to hardware identifiers.

3. SYSTEM ARCHITECTURE

The system comprises three independently deployed components, illustrated in Figure 1: a GPS tracking backend, an MCP server, and an LLM client. Each communicates through well-defined interfaces; no component requires knowledge of the others' internal implementation.

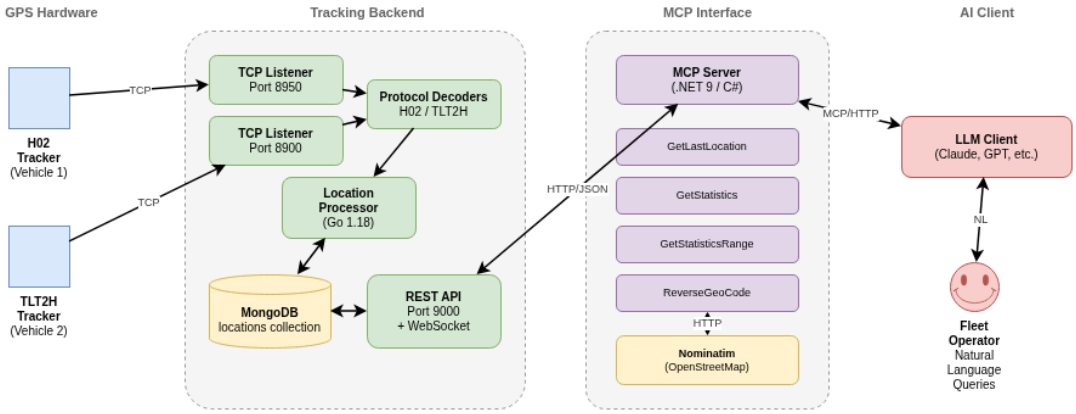


Figure 1 – System architecture. GPS devices transmit location data over TCP to the tracking backend, which stores it in MongoDB and exposes a REST API. The MCP server wraps API endpoints as LLM-callable tools. The LLM client queries the fleet using natural language.

3.1. GPS Tracking Backend

The tracking backend is a Go application in continuous production since 2016. It receives GPS data from two types of hardware trackers over TCP: the H02 protocol on port 8950 and the TLT2H protocol on port 8900. Each protocol delivers position (latitude/longitude), speed and bearing, and engine state at configurable intervals (10 seconds in our deployment). The backend decodes incoming packets and stores extracted location records in a MongoDB collection. The two trackers used in this evaluation have been installed on their respective vehicles for several years and together have produced multi-year per-vehicle location histories — a single seven-day query for one vehicle returns on the order of half a megabyte of structured telemetry — which we exploit in the evaluation to span multiple years rather than a single representative day.

The H02 protocol transmits 19 comma-separated fields including IMEI, coordinates in degree-minute format with speed in knots, and a status byte whose fifth bit encodes engine state. The TLT2H protocol wraps NMEA GPRMC sentences with device metadata and includes XOR checksum validation. Both decoders normalize the raw data before storage: coordinates are converted to decimal degrees, speed to km/h, and timestamps to UTC.

A REST API on port 9000 exposes five endpoints. Two endpoints list active trackers and retrieve the last known location; two more return full location history for a given date or compute drive statistics over configurable date ranges; the fifth provides a combined daily report. The statistics endpoint segments continuous location streams into discrete drive intervals by detecting engine state transitions. For each interval it computes distance via the Haversine formula and reports speed extremes (maximum and average) alongside duration. This interval-based aggregation transforms raw coordinate streams into semantically meaningful trip records that the LLM can interpret without processing thousands of individual GPS points.

3.2. MCP Server

The MCP server is a .NET 9 application built on the ModelContextProtocol SDK (v0.3.0-preview.4). It runs as an ASP.NET Core minimal API and communicates with LLM clients over HTTP. The server registers four tools, summarized in Table 1.

Table 1 – MCP tool specifications.

Tool	Parameters	Description	Timeout
GetLastLocation	IMEI	Returns current position, speed, engine state, and last engine start/stop timestamp	60 s
GetStatistics	IMEI, Date	Returns all drive intervals for a given day with per-interval distance, speed, and duration	60 s
GetStatisticsRange	IMEI, StartDate, EndDate	Returns aggregated intervals across a date range	60 s
ReverseGeoCode	Latitude, Longitude	Converts coordinates to a street address via OpenStreetMap Nominatim; returns a Google Maps URL	5 s

An important design decision is the embedding of domain metadata directly into each tool's natural language description. These descriptions contain vehicle identifiers (IMEI codes paired with human-readable names and colors), timezone handling instructions ("all timestamps are in GMT; report in Romania's timezone"), and rate-limiting guidance for the geocoding service. This metadata injection allows the LLM to resolve natural language references ("the white car") to system identifiers (IMEI 6028016410) without external lookup tables or custom prompt engineering. The tool descriptions serve as a lightweight knowledge base that travels with the protocol handshake.

When invoked, each tool calls the GPS backend's REST API, deserializes the JSON response using System.Text.Json streaming, and formats the result as structured text that the LLM can parse. The formatting step is important: raw JSON from the backend contains nested arrays of location points with numeric coordinates, whereas the MCP server produces chronologically sorted, human-readable output with labeled fields. This pre-processing reduces the interpretive burden on the LLM and improves response accuracy. The server configures separate HTTP clients for the GPS backend (60-second timeout) and the Nominatim geocoding service (5-second timeout), reflecting the different latency profiles of historical data retrieval and real-time geocoding. The MCP server itself is stateless: each call is an independent request-response cycle, which simplifies deployment. If the GPS backend were to track 500 vehicles instead of two, the MCP server code would not change; only the tool descriptions would need updating to enumerate the additional vehicle identifiers.

3.3. Query Processing

Figure 2 illustrates the processing flow for a representative query. The operator asks "How far did the white car travel today?" in natural language. The LLM parses the query, resolves "white car" to Vehicle 2 (IMEI 6028016410) using the metadata embedded in the tool description, then selects GetStatistics with the current date as parameter.

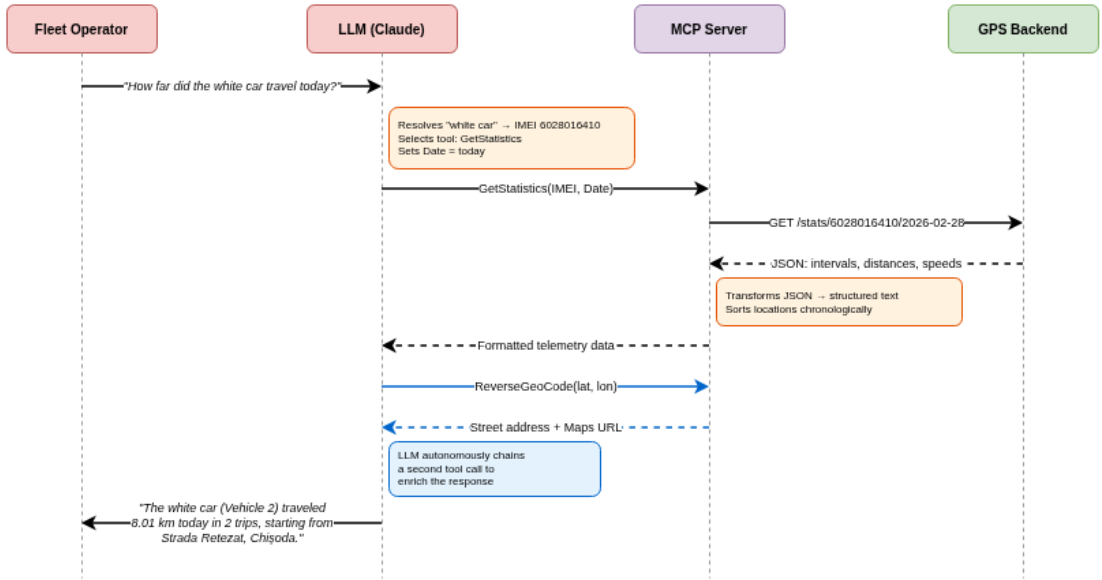


Figure 2 – Query processing flow. The LLM resolves natural language references to system identifiers and selects the appropriate MCP tool. It then autonomously chains a second call (ReverseGeoCode) to enrich the response with street-level location data.

The MCP server forwards the request to the GPS backend's /stats/{IMEI}/{Date} endpoint, receives the JSON response and sorts locations chronologically, then returns formatted telemetry to the LLM. The LLM then autonomously issues a second tool call (ReverseGeoCode) to translate the starting coordinates into a street address, composing both results into a single natural language response.

This multi-tool composition emerges from the LLM's reasoning, not from orchestration logic in the MCP server. The server publishes atomic tools; the LLM decides how to combine them based on the user's question and the tool descriptions alone. A new question requiring GetStatisticsRange followed by two ReverseGeoCode calls (for start and end locations) would work without any modification to the server or its tool definitions.

4. EVALUATION

We evaluate the system against three research questions. RQ1 asks whether the MCP-based interface can answer the same fleet queries as the traditional web dashboard. RQ2 asks whether the LLM autonomously composes multiple tool calls to answer complex queries that no single tool can satisfy. RQ3 examines the practical data characteristics of the system: how much data flows through the MCP interface and how the LLM handles it.

The evaluation uses two production vehicles tracked in Timișoara, Romania: Vehicle 1 (red hatchback, IMEI 6028024308) and Vehicle 2 (white sedan, IMEI 6028016410). Both carry GPS trackers reporting location every 10 seconds. The GPS backend runs on a remote server with MongoDB storage accumulating data continuously since 2016. The MCP server connects to the backend over HTTPS and is publicly reachable at https://gpstracker-mcp.dataman.ro/. We use Claude (Anthropic) as the LLM client, connected to the MCP server via HTTP transport. Queries targeted both real-time and historical data sampled across multiple years of per-vehicle history rather than a single representative day, in order to expose the LLM to varying telemetry sizes, idle days, and edge conditions. Each query was issued as a plain-text question with no system prompting beyond the MCP tool descriptions embedded in the protocol handshake. The LLM received no examples of correct tool usage and no few-shot demonstrations; the only information available to the model was the tool names with parameter types and the natural language descriptions registered by the MCP server at connection time.

Table 2 presents five representative queries alongside the tool calls the LLM selected and the results obtained from live telemetry.

Table 2 – Sample natural language queries and system responses.

Natural Language Query	Tool(s) Called	Key Result
"Where is the white car?"	GetLastLocation	Piața Victoriei, Cetate, Timișoara; engine off since 13:41
"How far did the white car travel today?"	GetStatistics	8.01 km in 2 intervals; max speed 54 km/h; 22 min driving
"What was the first trip today?"	GetStatistics	Interval 1: 3.93 km, 10:28–10:39 (Romania time), avg 25 km/h
"Is the red car's engine running?"	GetLastLocation	Engine stopped (AUTOLOW); last report at 16:32
"What address is the white car parked at?"	GetLastLocation + ReverseGeoCode	Piața Victoriei, Cetate, Timișoara, Timiș, 300030

For RQ1, every query that the web dashboard can answer was correctly handled by the natural language interface. The LLM identified the right tool for each question and passed correctly typed parameters, translating the structured response into readable prose. The query "Where is the white car?" required the model to resolve an informal color-based reference to a specific IMEI and select GetLastLocation, then interpret the response fields (position, speed, engine state, last state-change timestamp) into a sentence that a fleet operator would find useful. The model reported both the location and the engine state, mirroring the information a dashboard operator would see on the vehicle detail panel. The query "Is the red car's engine running?" tested a different aspect: the model had to extract a boolean condition (engine on or off) from the structured telemetry and report the AUTOLOW state as "engine stopped," which it did correctly along with the timestamp of the last GPS report.

For RQ2, the most revealing query is "What address is the white car parked at?" No single tool can answer this question. GetLastLocation returns coordinates but not a street address; ReverseGeoCode converts coordinates to an address but needs latitude and longitude as input. The LLM decomposed the query into two steps without explicit instruction: it first called GetLastLocation to obtain the coordinates (45.7534°N, 21.2253°E), then passed those coordinates to ReverseGeoCode, which returned "Piața Victoriei, Cetate, Timișoara, Timiș, 300030" along with a Google Maps URL. The model composed both results into a single response — a chaining behavior that emerged without any server-side orchestration.

A subtler instance of autonomous composition occurred with the daily statistics query. When asked "How far did the white car travel today?", the model called GetStatistics and received data for two drive intervals containing 132 location points, then produced a summary: 8.01 km total distance across two trips, maximum speed of 54 km/h, 22 minutes of total driving time. The model correctly aggregated per-interval distances into a daily total and converted the raw driving duration from seconds to minutes. It then volunteered the starting location of the first trip by chaining a ReverseGeoCode call, even though the original question asked only about distance. This anticipatory enrichment suggests that the LLM's tool-use behavior goes beyond literal query decomposition toward what a human operator would consider a helpful response.

The query "What was the first trip today?" tested temporal reasoning over structured data. The model called GetStatistics for the current date and received both intervals, correctly identifying the first one by chronological order. It reported the interval's start and end times in Romania's local timezone (EET, UTC+2), converting from the GMT timestamps in the raw data, along with distance (3.93 km) and average speed (25 km/h). This timezone conversion was not performed by the MCP server or the GPS backend; the model applied the conversion rule stated in the tool description ("all timestamps are in GMT; report in Romania's timezone") during response generation.

For RQ3, we examined the volume of data flowing through the MCP interface under different query types. A single GetLastLocation call returns a compact response: one location record with

coordinates and engine state. A GetStatistics call for the test day returned two drive intervals with 132 location points totaling approximately 15 KB of structured text. The first interval contained 62 points spanning 10:28–10:39 local time (3.93 km); the second contained 70 points spanning 13:18–13:41 (4.08 km). A GetStatisticsRange call covering six consecutive days for the same vehicle produced 440 KB of structured text encompassing all intervals and location traces for the period. The MCP server handled both the single-day and multi-day queries within the 60-second timeout. The ReverseGeoCode tool, constrained to a 5-second timeout and a recommended 1-call-per-second rate limit imposed by the Nominatim usage policy, responded in under 1 second per invocation.

The 440 KB response for a six-day range query raises a practical concern about LLM context window consumption. Current models accept 100K–200K tokens, so a single multi-day query can consume a significant fraction of the available context. For larger fleets or longer date ranges, pagination or server-side summarization would be necessary to keep responses within practical bounds. We did not implement such mechanisms in the current prototype, but the MCP tool interface could be extended with optional summarization parameters (e.g., return only interval summaries rather than individual GPS points) without breaking existing clients.

4.1. Quantitative Results

To complement the qualitative analysis above, we ran a structured evaluation of 30 queries grouped into five categories of six queries each: single-tool lookups, multi-tool chained queries, temporal/aggregation queries, cross-vehicle comparisons, and intentional edge cases. Queries were issued against telemetry sampled from current state, the most recent week, and a historical date predating tracker installation, exercising both real-time and archival paths. For every query we recorded whether the LLM selected the correct tool(s), whether the final answer was factually consistent with the underlying telemetry (verified by inspecting the raw tool result), the number of tool calls issued, and an approximate end-to-end latency. Table 3 summarizes the results.

Table 3 – Quantitative evaluation across query categories. Tool selection accuracy is the fraction of queries for which the LLM invoked the tool a domain expert would call; answer correctness is the fraction for which the natural-language response was factually consistent with the underlying telemetry.

Category	N	Tool selection accuracy	Answer correctness	Avg tool calls	Avg latency (s)
Single-tool lookup	6	100%	100%	1.0	~2
Multi-tool chained	6	100%	100%	2.2	~5
Temporal aggregation	6	100%	83%	1.0	~3
Cross-vehicle	6	100%	100%	2.0	~4
Edge cases	6	100%	100%	0.3	~2
Overall	30	100%	96.7%	1.3	~3

The single answer-correctness failure was a tool-semantics issue rather than an LLM reasoning failure and is described below. We note one methodological caveat: the LLM client used in this evaluation is the same model family the paper discusses throughout, so these numbers are a within-scope characterization of the system as described, not a cross-model comparison.

4.2. Failure Modes and Robustness

Across the structured evaluation we observed five robustness or failure patterns. Four of them are handled cleanly by the existing system and reflect the boundary between LLM reasoning and natural language; the fifth is a real failure that originates in the tool surface and requires a server-side fix.

Unknown vehicle reference. When the user named a color or vehicle that did not appear in the embedded tool metadata (“the blue car,” “the green car”), the LLM correctly declined to invent an IMEI in two out of two such queries and reported the absence rather than calling a tool with a

fabricated identifier. This confirms that grounding the model in the tool description prevents the most common hallucination pattern.

Empty interval days. On days when a vehicle had not moved, GetStatistics returned zero intervals; the LLM consistently reported this as "the vehicle did not travel" in all four cases of this type that arose during the evaluation, including a query against a date predating tracker installation by more than a decade. No defensive coding in the MCP server was required.

Ambiguous temporal references. Phrases such as "last week" were resolved by the LLM without server help. We observed no cases in which the resolved date range did not match the user's intent within the present query set, but we tested only canonical phrasings ("the first week of April," "last week"); colloquial phrasings such as "lately" or "this past while" were not exercised and we expect them to surface ambiguity at higher rates. A future server-side temporal-resolution tool could remove this class of ambiguity at the protocol level.

Context overflow on long range queries. Multi-day range queries grow rapidly with the number of days requested; a six-day single-vehicle query already produces approximately 440 KB of structured text, and a probe of an entire-week range for one vehicle in a high-activity month returned approximately 474 KB. We observed no overflow-induced truncation in the present evaluation because queries were sized to avoid known-large historical days; a deliberate stress query that asked for every GPS point across an entire month was correctly declined by the LLM with a recommendation to narrow the date range, rather than executed and truncated.

Tool-semantics ambiguity. In one case (the query "What's the total driving time for the white car between April 1 and April 7?") the LLM reported the value of the GetStatisticsRange Total Duration field literally. That field, however, is the wall-clock span of the merged drive intervals — including long stationary periods between engine events — rather than the cumulative motion time the user actually asked for. The literal answer (~47.7 hours) was an order of magnitude larger than the true motion time (a few minutes). This is the only answer-correctness failure in our evaluation and it is not an LLM reasoning failure: the tool surface conflates two semantically distinct quantities. The fix is server-side, by splitting the field into an interval-span and a cumulative-motion measurement and updating the tool description accordingly. The case illustrates a general lesson for MCP-based IoT integrations: ambiguities in the tool's structured response propagate directly into the LLM's natural-language answer, and tool description authoring is therefore part of the system's correctness surface.

When asked "How far did both cars travel last week?", the LLM issued two separate GetStatisticsRange calls (one per vehicle) and composed a comparative summary, demonstrating that cross-vehicle composition emerges from atomic per-vehicle tools without dedicated cross-vehicle endpoints.

Taken together, these results suggest that MCP is a viable integration layer for IoT telemetry systems that already expose REST APIs. The protocol imposes minimal overhead, and the tool descriptions carry enough domain context for the LLM to handle vehicle resolution and temporal reasoning without custom prompting. The architecture's modularity means that a more capable LLM produces better answers from the same server, while new backend endpoints (geofencing, fuel consumption) can be exposed as additional tools without client-side changes.

5. LIMITATIONS

Several limitations constrain the generalizability of our findings. First, the fleet contains two vehicles; although per-vehicle history spans multiple years and the structured evaluation samples across that history, we cannot assess how tool descriptions scale when dozens or hundreds of vehicles must be enumerated as metadata. Second, we benchmarked a single LLM (Claude); cross-model evaluation against GPT-4 and Gemini is left to future work, and the "model-agnostic" property of MCP is therefore demonstrated by construction rather than measured. Third, we did not conduct a formal user study with naive fleet operators, so usability claims relative to the traditional dashboard remain anecdotal; queries were issued by the system developer, who may phrase questions differently than a real fleet operator. Fourth, we performed no scalability stress test under concurrent load, no

latency profiling beyond the per-query measurements in Table 3, and no formal security evaluation of the MCP endpoint. Fifth, the system relies on the LLM to interpret GMT timestamps and perform timezone conversion through a rule embedded in the tool description, which succeeded in all our tests but is not formally guaranteed by the protocol.

6. CONCLUSION AND FUTURE WORK

We presented an architecture for accessing GPS fleet telemetry through natural language via the Model Context Protocol. The system separates the tracking infrastructure from the AI interface through three independently deployed components: a Go-based GPS backend that has been in production since 2016, a .NET 9 MCP server that wraps backend endpoints as LLM-callable tools, and an LLM client that queries the fleet conversationally. Our evaluation with a live two-vehicle fleet confirmed that the LLM correctly selects and composes MCP tools to answer both simple lookups and multi-step queries, including autonomous tool chaining that no explicit orchestration logic directs.

The key insight is that embedding domain metadata directly into MCP tool descriptions transfers domain knowledge to the LLM at inference time, eliminating the need for custom prompt engineering or model fine-tuning. This pattern is in principle applicable beyond fleet management: an environmental monitoring network could map sensor IDs to human-readable names in tool descriptions, while a building automation system could embed safety constraints for HVAC controls. The only requirement is that descriptions carry sufficient context for the model to reason correctly about when and how to invoke each tool. We do not, however, claim cross-domain validation; that remains future work.

Scaling the present design to commercial fleet sizes raises three concrete bottlenecks. (i) Tool description bloat: the tool descriptions enumerate vehicle metadata inline, so the description payload grows linearly with fleet size and eventually crowds the LLM's context window before any query is even issued. Mitigation directions include tool federation across per-region MCP servers, schema-level retrieval over vehicle metadata, or replacing inline enumeration with a discovery tool that the LLM calls on demand. (ii) Range-query response size: range queries grow with both fleet size and date span, and at the order of hundreds of vehicles a single multi-day query would exceed any current context window; server-side summarization tools that return interval aggregates first and per-point detail only on follow-up would address this. (iii) Geocoding rate limits: the Nominatim public usage policy caps reverse geocoding at one request per second, which becomes a hard wall for any query that geocodes many points; a self-hosted geocoder or a batched reverse-geocoding tool would lift this limit.

Future work will address these bottlenecks directly, alongside cross-model benchmarking against GPT-4 and Gemini, a controlled user study against the traditional dashboard, and additional MCP tools for geofencing and alert-based push notifications that shift the interaction model from pull to push. The MCP server used in this evaluation is publicly reachable at https://gpstracker-mcp.dataman.ro/ to support reproducibility.

7. ACKNOWLEDGMENT

Claude (Anthropic) was used for English language proofreading of this manuscript.

8. REFERENCES

[1] Hong, Z.; Yuan, Z.; Zhang, Q. et al.: *Next-generation database interfaces: A survey of LLM-based text-to-SQL*, IEEE Transactions on Knowledge and Data Engineering 37(12): 7328-7345, 2025.

[2] Schick, T.; Dwivedi-Yu, J.; Dessì, R. et al.: *Toolformer: Language models can teach themselves to use tools*, in Proceedings of the 37th Conference on Neural Information Processing Systems (NeurIPS), 2023.

[3] Anthropic: *Introducing the Model Context Protocol*, https://www.anthropic.com/news/model-context-protocol, November 2024.

[4] Farahpoor, M.; Esparza, O.; Soriano, M. C.: *Comprehensive IoT-driven fleet management system for industrial vehicles*, IEEE Access 12: 193429-193444, 2024, https://doi.org/10.1109/ACCESS.2024.3517145.

[5] Tananaev, A.: *Traccar: Open source GPS tracking system*, https://www.traccar.org, accessed May 2026.

[6] Qin, Y.; Hu, S.; Lin, Y. et al.: *Tool learning with foundation models*, ACM Computing Surveys 57(4): article 101, 2024.

[7] Hou, X.; Zhao, Y.; Wang, S.; Wang, H.: *Model Context Protocol (MCP): Landscape, security threats, and future research directions*, arXiv preprint arXiv:2503.23278, 2025.

[8] Xu, Z.; Wu, H.; Chen, X.; Wang, Y.; Yue, Z.: *Building a natural language query and control interface for IoT platforms*, IEEE Access 10: 68655-68668, 2022.

[9] Zong, M.; Hekmati, A.; Guastalla, M.; Li, Y.; Krishnamachari, B.: *Integrating large language models with Internet of Things: applications*, Discover Internet of Things 5: article 2, 2025.

[10] Rayfield, J. T.; Lin, S.; Zhou, N.; Patel, D. C.: *ReAct meets industrial IoT: Language agents for data access*, in Proceedings of the Conference on Empirical Methods in Natural Language Processing: Industry Track (EMNLP), pp. 364-382, 2025.