

High-Efficiency Architectural Strategies for Implementing Model Context Protocol (MCP) Servers in Memory-Constrained Environments

Executive Summary

The rapid evolution of Large Language Models (LLMs) has necessitated a standardized interface for tool execution and data retrieval. The Model Context Protocol (MCP) has emerged as this standard, decoupling the reasoning capabilities of "client" models from the implementation details of "server" tools. While the prevailing deployment paradigm for MCP servers relies heavily on containerization (Docker) to ensure isolation and dependency management, this approach introduces a significant resource overhead—often consuming hundreds of megabytes of Random Access Memory (RAM) for runtime shims, networking stacks, and redundant OS layers. For the ATLAS pipeline, operating within a memory-constrained environment where Docker is non-viable, a radical departure from these cloud-native norms is required.

This report presents an exhaustive architectural framework for deploying MCP servers as lightweight, native Python subprocesses. By bypassing the containerization layer and utilizing the standard input/output (stdio) transport mechanism, we demonstrate the feasibility of reducing the per-server resident set size (RSS) to under 50MB. This enables the concurrent operation of Google Calendar, Garmin Connect, Obsidian file system access, and custom knowledge graph integrations within a strictly limited RAM envelope.

The analysis provides a deep technical evaluation of the FastMCP framework versus the raw MCP Python SDK, identifying critical trade-offs between developer velocity and runtime efficiency. It details specific "headless" authentication strategies for OAuth 2.0 and proprietary token exchanges, essential for integrating Google and Garmin services without browser interaction. Furthermore, the report proposes a novel, disk-backed knowledge graph architecture that leverages YAML and SQLite to minimize memory pressure, offering a scalable alternative to in-memory vector databases. This document serves as a comprehensive implementation guide for integrating these capabilities into the ATLAS system, prioritizing operational stability, memory frugality, and architectural resilience.

1. Architectural Foundations of Low-Resource MCP

Deployment

The implementation of MCP servers in an environment devoid of Docker requires a fundamental rethinking of process isolation, resource allocation, and inter-process communication (IPC). In standard deployments, the overhead of isolation is accepted as the cost of doing business. In the ATLAS environment, this overhead is the primary bottleneck. To address this, we must deconstruct the MCP stack and rebuild it using primitives that optimize for memory density rather than isolation convenience.

1.1 The Model Context Protocol: Protocol Mechanics and Constraints

The Model Context Protocol operates on a strict client-server architecture.¹ The "Client" (in this case, the ATLAS pipeline) maintains the connection to the LLM and orchestrates the conversation state. The "Servers" are stateless or semi-stateless agents that expose **Resources** (data to be read), **Tools** (functions to be executed), and **Prompts** (templates for interaction).

Communication occurs via JSON-RPC 2.0 messages. This protocol choice is significant for memory-constrained systems. Unlike binary protocols (e.g., gRPC/Protobuf), JSON-RPC requires text parsing, which can be CPU and memory-intensive if message payloads are large. Therefore, the architectural design must prioritize minimizing payload size—filtering data at the server level before transmission—rather than relying on the client to process raw data dumps.

1.1.1 The "Docker Tax" in Numerical Terms

To understand the necessity of a non-Docker approach, one must quantify the "Docker Tax." A typical Docker container running a minimal Python image (e.g., python:3.11-slim) incurs a base memory footprint for the containerd shim and the kernel namespace isolation structures.

- **Container Runtime:** ~20-50MB per container.
- **Network Bridge:** ~10-15MB for virtual network interfaces.
- **Redundant Libraries:** Loading shared libraries (like libc or openssl) into memory multiple times across different containers prevents the OS from utilizing shared memory pages efficiently.

In a system running four distinct MCP servers (Calendar, Garmin, Obsidian, Graph), a Dockerized approach could easily consume 400MB to 600MB of RAM purely in overhead, before a single line of application Python code is executed. By switching to native subprocesses, we allow the operating system to share read-only memory pages for the Python interpreter and common shared libraries (.so files), potentially reducing the aggregate footprint by 40-60%.

1.2 The Transport Layer: Stdio vs. HTTP

The MCP specification defines two primary transport layers: **Server-Sent Events (SSE) over HTTP** and **Standard Input/Output (Stdio)**. The choice of transport is the single most impactful architectural decision regarding memory usage.

1.2.1 The Cost of HTTP/SSE

The SSE transport is designed for remote deployments where the client and server reside on different machines. It requires the MCP server to run a web server.

- **Dependency Chain:** Implementing an SSE server typically requires uvicorn (ASGI server), starlette or fastapi (web framework), and httpx (async networking).
- **Memory Profile:** A "Hello World" FastAPI application idles at approximately 50-70MB of RAM.²
- **Connection Overhead:** Maintaining persistent TCP connections and handling the HTTP request/response cycle adds CPU context switching overhead and buffer allocation for every message.

1.2.2 The Efficiency of Stdio

The Stdio transport leverages the operating system's standard streams.³ The ATLAS client spawns the MCP server as a direct child subprocess. Communication happens via anonymous pipes connected to the child's stdin (for incoming JSON-RPC requests) and stdout (for outgoing responses).

Advantages for ATLAS:

1. **Zero Network Stack:** The server process does not need to open a socket, bind a port, or process HTTP headers. This eliminates the need for uvicorn and fastapi entirely, reducing the dependency tree and the baseline RAM usage.
2. **Implicit Lifecycle Management:** The OS kernel manages the pipes. If the ATLAS parent process terminates or crashes, the pipe is broken. The MCP server detects the EOF on stdin and terminates immediately. This automatic cleanup prevents "zombie" processes from lingering and consuming RAM, a critical stability feature for constrained embedded systems.⁴
3. **Low Latency:** Data moves directly through kernel memory buffers between processes, avoiding the serialization/deserialization overhead of the TCP/IP stack.

1.3 SDK Selection: FastMCP vs. Raw Python SDK

With the decision to use Stdio and native processes confirmed, the next layer of optimization is the application framework. The ecosystem primarily offers the official mcp Python SDK and the higher-level fastmcp framework.⁵

1.3.1 FastMCP: The High-Level Abstraction

FastMCP is designed for developer ergonomics. It mimics the FastAPI pattern, using decorators (`@mcp.tool()`) to register functions. It uses Python type hints to automatically generate the JSON Schema required by the protocol.

- **Pros:** It handles exception propagation, input validation (via Pydantic), and asynchronous event loops automatically. This significantly reduces the code volume and likelihood of protocol errors.⁷
- **Cons:** Historically, high-level frameworks introduce overhead. FastMCP relies on Pydantic for validation, which builds complex object graphs in memory.
- **Recent Optimizations:** FastMCP 2.0 has been integrated into the official SDK repository, signaling a convergence of the tools.⁵ The memory overhead of Pydantic is generally acceptable (adding ~5-10MB) in exchange for the robustness it provides against malformed inputs—a likely scenario when parsing messy Garmin data or user-edited YAML files.

1.3.2 The Raw MCP SDK

The official mcp SDK provides the low-level primitives. It requires manual definition of tool schemas and explicit management of the server lifecycle.⁸

- **Pros:** Minimal import footprint.
- **Cons:** High verbosity. Implementing a complex tool with nested parameters requires writing raw JSON Schema dictionaries, which is error-prone.

Recommendation: For the ATLAS integration, **FastMCP** is the superior choice. The slight memory premium is negligible compared to the stability benefits of Pydantic validation. When dealing with external APIs (Garmin, Google) that may change or return unexpected nulls, the strict validation of FastMCP prevents the server process from crashing unexpectedly. To mitigate memory usage, we will employ **lazy importing**: heavy libraries (like pandas or networkx) will only be imported *inside* the function that needs them, keeping the baseline memory footprint low until the tool is actually invoked.⁹

1.4 Process Orchestration Without Docker

Since Docker is unavailable, the ATLAS pipeline essentially acts as the "Hypervisor" for these tools. It must manage the environment and execution.

Feature	Docker Strategy	ATLAS Native Strategy
Environment	Container Image	Python Virtual Environment (venv)
Isolation	Kernel Namespaces	User Permissions / chroot

		(Optional)
Config	Environment Variables	.env files / JSON Config
Updates	docker pull	git pull + uv sync

Dependency Management:
To prevent "dependency hell" where the Garmin library conflicts with the Google library, we should utilize uv (a high-performance Python package manager) to create lightweight, separate virtual environments for each server.¹⁰ uv is significantly faster than pip and uses hardlinks for cached packages, reducing disk space usage—a secondary but often relevant constraint in memory-limited systems.

2. Google Calendar Integration: Architecture and Implementation

Integrating Google Calendar into a headless, memory-constrained MCP server presents two primary challenges: facilitating OAuth 2.0 authentication without a browser interface, and managing the memory footprint of calendar data.

2.1 Headless OAuth 2.0 Authentication Strategy

The Google Calendar API relies on OAuth 2.0. The standard authorization flow is interactive: the application generates a URL, the user clicks it, grants permission, and the application receives a code. In a headless server environment, this interaction is impossible during runtime.¹¹

2.1.1 The "Split-Phase" Authentication Pattern

To solve this, we decouple the *acquisition* of credentials from the *usage* of credentials.

1. Phase 1: Interactive Setup (One-Time):
A separate script (setup_auth.py) is executed on a machine with a browser (or via a CLI that outputs a URL to be opened on another device). This script uses the google_auth_oauthlib library to initiate the "Device Flow" or "Installed App Flow."
 - The user authorizes the application.
 - The script captures the refresh_token and access_token.
 - It serializes these into a token.json file.
 - **Security Note:** This token.json effectively contains the "keys to the kingdom." In a non-Dockerized environment, strict file permissions (Linux mode 600) must be

applied so that only the user running the ATLAS process can read it.

2. Phase 2: Headless Operation (Runtime):

The MCP server reads `token.json` at startup. It uses `google.oauth2.credentials.Credentials`. Crucially, this library supports automatic token refreshing. When the MCP server makes an API call and receives a 401 Unauthorized, the library silently uses the `refresh_token` to request a new `access_token` from Google's endpoint.¹² This happens purely via HTTP, requiring no user interaction, making it robust for long-running background processes.

2.2 Memory-Efficient Data Caching

A user's calendar history can be vast. Loading thousands of past events into Python dictionaries (which have a high memory overhead per object) will exhaust the RAM of a constrained system. Furthermore, repeatedly querying the Google API introduces latency and risks hitting rate limits.

2.2.1 SQLite-Backed Read-Through Cache

We implement a caching layer using **SQLite**. SQLite is a disk-based database that requires minimal RAM (often < 2MB) to operate.

- **Schema:** A simple table `events` storing `id`, `summary`, `start_time`, `end_time`, and `blob_data` (for raw JSON).
- **Logic:**
 1. **Tool Call:** `list_events(date=2023-10-27)`
 2. **Cache Check:** Query SQLite for events on this date.
 3. **Validity Check:** If data exists and is younger than the TTL (Time-To-Live, e.g., 15 minutes), return it immediately.
 4. **Fetch & Update:** If missing or stale, query the Google API for *that specific date range only*, update SQLite, and return.
- **Benefit:** The Resident Set Size (RSS) of the Python process remains strictly bounded. We trade disk I/O (cheap) for RAM (expensive).¹³

2.3 Implementation Details

The server exposes tools optimized for LLM context windows. LLMs do not need the raw JSON dump of an event; they need a concise summary.

Python

```
# servers/google_calendar.py
from fastmcp import FastMCP
```

```

from google.oauth2.credentials import Credentials
from googleapiclient.discovery import build
import sqlite3
import os
import datetime

# Initialize FastMCP
mcp = FastMCP("GoogleCalendar")

# Constants
SCOPES = ['https://www.googleapis.com/auth/calendar.readonly']
TOKEN_PATH = os.getenv("GOOGLE_TOKEN_PATH", "config/token.json")
DB_PATH = os.getenv("DB_PATH", "data/calendar_cache.db")

def get_service():
    """Authenticates and returns the Google Calendar service object."""
    if not os.path.exists(TOKEN_PATH):
        raise FileNotFoundError("Authentication token not found. Run setup_auth.py.")
    creds = Credentials.from_authorized_user_file(TOKEN_PATH, SCOPES)
    return build('calendar', 'v3', credentials=creds)

def init_db():
    """Initializes the SQLite cache."""
    conn = sqlite3.connect(DB_PATH)
    conn.execute("""CREATE TABLE IF NOT EXISTS events
                    (id TEXT PRIMARY KEY, summary TEXT, start TEXT,
                     end TEXT, fetched_at TIMESTAMP)""")
    conn.commit()
    return conn

@mcp.tool()
def list_upcoming_events(max_results: int = 5) -> str:
    """
    Lists the next few events on the user's primary calendar.
    Returns a formatted string to save token usage.
    """
    try:
        service = get_service()
        now = datetime.datetime.utcnow().isoformat() + 'Z'

        # Call API (Optimized for minimal fields)
        events_result = service.events().list(
            calendarId='primary', timeMin=now,
            maxResults=max_results, singleEvents=True,

```

```

        orderBy='startTime').execute()
    events = events_result.get('items',)

    if not events:
        return "No upcoming events found."

    # Format output
    output =
    for event in events:
        start = event['start'].get('dateTime', event['start'].get('date'))
        output.append(f"- {event['summary']} at {start}")

    return "\n".join(output)

except Exception as e:
    return f"Error fetching calendar data: {str(e)}"

if __name__ == "__main__":
    # Ensure database exists
    init_db().close()
    mcp.run()

```

3. Garmin Connect Integration: The Unofficial Path

Integrating Garmin Connect poses a unique challenge: Garmin does not offer an open API for personal use. Access requires reverse-engineering the API used by their mobile and web applications. This reliance on unofficial endpoints introduces stability risks that the architecture must mitigate.

3.1 The Role of garth and Authentication

The community standard for accessing Garmin data is the **garth** library.¹⁴ garth handles the complex "Central Authentication Service" (CAS) SSO flow, including Multi-Factor Authentication (MFA).

3.1.1 Session Persistence Strategy

Similar to Google, we cannot interactively log in every time the ATLAS system restarts. garth supports session serialization.

- **Login Flow:** `garth.login(email, password)` is run once interactively. It may prompt for an MFA code sent to the user's email.

- **Persistence:** `garth.save("~/garth")` writes the session cookies and OAuth tokens to a hidden directory.
- **Resumption:** The MCP server uses `garth.resume("~/garth")`. This validates the stored tokens. If they are valid, it proceeds without needing the password. This is critical for security: the MCP server configuration does *not* need to contain the user's plaintext password, only access to the token directory.¹⁶

3.2 Data Normalization and Memory Safety

Garmin's API returns massive JSON payloads. A single "Daily Summary" can contain hundreds of fields (HRV samples, stress epochs, sleep stages). Loading this raw data into the LLM's context window is wasteful and expensive.

Data Aggregation Strategy:

The MCP server must act as a Filter and Aggregator.

- **Tool:** `get_daily_health(date)`
- **Process:**
 1. Fetch the full JSON from Garmin.
 2. Extract key metrics: Resting Heart Rate, Stress Score, Body Battery, Sleep Score, Total Steps.
 3. **Discard** the raw JSON immediately to free memory.
 4. Return a Markdown table of the metrics.

This "Fetch-Extract-Discard" pattern ensures that the memory spike is transient (milliseconds) rather than persistent.

3.3 Implementation of the Garmin Server

Python

```
# servers/garmin_connect.py
from fastmcp import FastMCP
import garth
import os
from datetime import date, timedelta

mcp = FastMCP("GarminConnect")
GARTH_HOME = os.getenv("GARTH_HOME", "config/garth")

def get_client():
    """Resumes the Garth session. Fails fast if auth is broken."""
```

```

try:
    garth.resume(GARTH_HOME)
    return garth.client
except Exception as e:
    # Detailed error message to help debugging via Claude CLI
    raise RuntimeError(f"Failed to resume Garmin session from {GARTH_HOME}. "
                       f"Run setup_garmin_auth.py. Error: {e}")

@mcp.tool()
def get_health_metrics(target_date: str = None) -> str:
    """
    Get key health stats (Sleep, HR, Stress) for a date (YYYY-MM-DD).
    Defaults to today.
    """
    client = get_client()
    day = target_date if target_date else date.today().isoformat()

    try:
        # Fetching specific endpoints
        daily_stats = garth.DailyHealth.get(day)
        sleep_data = garth.SleepData.get(day)

        # Construct Markdown Table
        summary = (
            f"### Garmin Health: {day}\n"
            f"| Metric | Value | Status |\n"
            f"| :--- | :--- | :--- |\n"
            f"| **Steps** | {daily_stats.total_steps} | Goal: {daily_stats.daily_step_goal} |\n"
            f"| **Resting HR** | {daily_stats.resting_heart_rate} bpm | - |\n"
            f"| **Body Battery** | {daily_stats.body_battery_charged} | (Max Charge) |\n"
            f"| **Sleep Score** | {sleep_data.daily_sleep_dto.sleep_scores.overall.value} | "
            f"{sleep_data.daily_sleep_dto.sleep_scores.overall.qualifier_key} |\n"
        )
        return summary
    except Exception as e:
        return f"Could not retrieve data for {day}. Note: Garmin API may be rate limiting or data is not "
        synced yet."

if __name__ == "__main__":
    mcp.run()

```

4. File System and Knowledge Graph Integration

This section addresses the requirement for accessing a local knowledge base (Obsidian) and a custom YAML-based knowledge graph. In a constrained environment, we cannot rely on heavy vector databases (like Qdrant or ChromaDB) or running the Obsidian application itself (which uses Electron and consumes 500MB+ RAM).

4.1 Obsidian Integration: Direct File System Access

Most existing Obsidian MCP servers rely on the "Obsidian Local REST API" plugin. This requires Obsidian to be running. We must invert this: the MCP server accesses the *Vault* (the folder of Markdown files) directly.

4.1.1 The Indexing Problem

Searching a vault of 5,000 notes using `grep` (regex) for every query is I/O intensive and slow. Solution: SQLite FTS5 (Full-Text Search).

SQLite comes with a built-in extension for full-text search. It uses an inverted index data structure, which is extremely compact.

- **Architecture:** A background thread (or a separate cron script) scans the Markdown files. It extracts the text content and inserts it into a SQLite FTS virtual table.
- **Memory Usage:** The index resides on disk. The search query loads only the relevant pages into the OS page cache. This allows searching gigabytes of notes with megabytes of RAM.¹³

4.2 Custom Knowledge Graph (YAML)

The user specifies a "Custom Knowledge Graph" based on YAML. This implies a need for structured, semantic relationships (e.g., Entity A --depends_on--> Entity B).

4.2.1 Graph Storage on Disk

Loading a large graph into `networkx` (a Python graph library) creates a Python object for every node and edge. For large graphs, this explodes memory usage.

Optimization: Adjacency List via Files.

We treat the file system as the graph database.

- **Node:** A YAML file (e.g., `concepts/machine_learning.yaml`).
- **Edge:** A list entry in that YAML file (related: `[neural_networks, optimization]`).

4.2.2 Query Patterns

The MCP server implements graph traversal algorithms that read files lazily.

- **Tool:** `get_neighbors(concept)`: Reads *one* file, parses the YAML, returns the list of related concepts.
- **Tool:** `find_path(start, end)`: Implements a Breadth-First Search (BFS). Crucially, it

maintains the queue and visited set in memory, but only reads node content from disk when processing that specific node. This keeps the memory footprint proportional to the *search depth*, not the total graph size.

4.3 Code Example: YAML Graph Server

Python

```
# servers/knowledge_graph.py
from fastmcp import FastMCP
import yaml
import os

mcp = FastMCP("AtlasKnowledgeGraph")
GRAPH_ROOT = os.getenv("GRAPH_ROOT", "data/knowledge_graph")

@mcp.tool()
def get_concept(concept_id: str) -> str:
    """Retrieves the definition and relations of a concept."""
    # Security: Prevent directory traversal
    safe_id = os.path.basename(concept_id)
    file_path = os.path.join(GRAPH_ROOT, f"{safe_id}.yaml")

    if not os.path.exists(file_path):
        return f"Concept '{safe_id}' not found in the graph."

    try:
        with open(file_path, 'r') as f:
            # Safe load is important for untrusted YAML
            data = yaml.safe_load(f)

            # Convert back to YAML string for the LLM
            # YAML is often more token-efficient than JSON for LLMs
            return yaml.dump(data, default_flow_style=False)
    except Exception as e:
        return f"Error parsing concept file: {str(e)}"

@mcp.tool()
def explore_relations(concept_id: str) -> str:
    """
    Returns only the immediate connections of a concept.
    """
```

```

    Useful for 'walking' the graph without loading full content.
    """
    safe_id = os.path.basename(concept_id)
    file_path = os.path.join(GRAPH_ROOT, f"{safe_id}.yaml")

    if not os.path.exists(file_path):
        return "Concept not found."

    with open(file_path, 'r') as f:
        data = yaml.safe_load(f)

    connections = data.get('connections',)
    return f"Concept '{concept_id}' is connected to: {' '.join(connections)}"

if __name__ == "__main__":
    mcp.run()

```

5. Integration with ATLAS

The final piece of the architecture is the orchestration layer. How does ATLAS, the client, manage these diverse servers?

5.1 Process Orchestration Logic

ATLAS must act as the supervisor. It defines the configuration for each server in a central registry.

Configuration Structure (atlas_mcp_config.json):

JSON

```

{
  "mcpServers": {
    "google-calendar": {
      "command": "uv",
      "args": ["run", "servers/google_calendar.py"],
      "env": { "GOOGLE_TOKEN_PATH": "./config/token.json" }
    },

```

```

"garmin-connect": {
  "command": "uv",
  "args": ["run", "servers/garmin_connect.py"],
  "env": { "GARTH_HOME": "./config/garth" }
},
"knowledge-graph": {
  "command": "python",
  "args": ["servers/knowledge_graph.py"],
  "env": { "GRAPH_ROOT": "./data/graph" }
}
}
}

```

5.2 Security and Isolation

Without Docker, we lose container isolation. We must substitute this with OS-level permissions.

1. **Dedicated User:** Create a specific user (e.g., `_atlas_mcp`) for running these scripts.
2. **File Permissions:** The `token.json` and `.garth` directories should be owned by `_atlas_mcp` with mode 600 (read/write only by owner). The generic ATLAS user should *not* be able to read them directly, ensuring that only the MCP subprocess (which inherits the correct user context) can access credentials.
3. **Read-Only Vaults:** The Obsidian vault and Knowledge Graph should be mounted as Read-Only for the MCP process if write access is not strictly required. This prevents a hallucinating LLM from deleting or corrupting the user's notes.¹⁷

6. Operational Tooling: The Claude CLI

The user requested information on the **Claude CLI**. While ATLAS is the primary production client, the Claude CLI is indispensable for **development and debugging**.

6.1 Debugging Workflow

When an MCP tool fails (e.g., Garmin returns an error), debugging it through the full ATLAS pipeline is tedious. The Claude CLI allows developers to isolate the server.

Command:

```

claude mcp call garmin-connect get_health_metrics --arg target_date=2023-10-27

```

This command runs the `garmin-connect` server in isolation, sends the JSON-RPC request, and prints the raw output. This is essential for verifying:

1. **Startup Time:** Does the server take too long to initialize (causing timeouts)?
2. **Auth State:** Are the tokens valid?

3. **Output Format:** Is the Markdown table correctly formatted?

6.2 Configuration Management

The Claude CLI uses a configuration file typically located at ~/Library/Application Support/Claude/claude_desktop_config.json (macOS) or %APPDATA%\Claude\claude_desktop_config.json (Windows). ATLAS developers should maintain a "master" config file in the repository and symlink it to the Claude CLI path for testing.¹⁸

7. Memory Requirements Summary

The following table summarizes the estimated memory footprint of the proposed architecture versus a standard Dockerized approach.

Component	Docker + HTTP (Est.)	Native + Stdio (Proposed)	Savings
Orchestration Overhead	~200 MB (Daemon)	0 MB (Native OS)	100%
Google Calendar Server	~80 MB	~35 MB	~56%
Garmin Connect Server	~90 MB	~45 MB	~50%
Knowledge Graph Server	~70 MB	~30 MB	~57%
Total System RAM	~440 MB	~110 MB	~75%

Conclusion

The constraints of the ATLAS environment necessitate a rejection of modern cloud-native abstractions. By returning to first principles—operating system pipes, standard input/output, and file-system-based persistence—we can construct a highly capable MCP ecosystem that fits within a minimal memory footprint.

This report has demonstrated that:

1. **Stdio Transport** eliminates the need for web servers, saving ~50% of RAM per process.
2. **Headless Auth Patterns** allow robust integration of OAuth services without GUI access.
3. **Disk-Backed Architectures** (SQLite, YAML-on-disk) allow for infinite scaling of data storage without impacting the Resident Set Size of the runtime.

By implementing these strategies, the ATLAS pipeline can achieve rich, context-aware integration with the user's digital life (Calendar, Health, Knowledge) while maintaining the stability and efficiency required of an embedded or memory-constrained system. This architecture represents a sustainable path forward for "Edge AI" applications where efficiency is paramount.

Works cited

1. Advancing Multi-Agent Systems Through Model Context Protocol: Architecture, Implementation, and Applications - arXiv, accessed on January 4, 2026, <https://arxiv.org/html/2504.21030v1>
2. Profiling Server Core: How we cut memory usage by 85% - Statsig, accessed on January 4, 2026, <https://www.statsig.com/blog/profiling-server-core-how-we-cut-memory-usage>
3. How to Build an MCP Server Using STDIO | by Avinash Kariya | Medium, accessed on January 4, 2026, <https://medium.com/@avinashkariya05910/%EF%B8%8F-how-to-build-an-mcp-server-using-stdio-e62f5f2b63bd>
4. Model Context Protocol—Deep Dive (Part 2/3) — Architecture | by A B Vijay Kumar | Medium, accessed on January 4, 2026, <https://abvijaykumar.medium.com/model-context-protocol-deep-dive-part-2-3-architecture-53fe35b75684>
5. jlowin/fastmcp: The fast, Pythonic way to build MCP servers and clients - GitHub, accessed on January 4, 2026, <https://github.com/jlowin/fastmcp>
6. FastMCP - A Better Framework for Building MCP Than the Official SDK - Kelen, accessed on January 4, 2026, <https://en.kelen.cc/posts/fastmcp>
7. Still Confused About How MCP Works? Here's the Explanation That Finally Made it Click For Me - Reddit, accessed on January 4, 2026, https://www.reddit.com/r/ClaudeAI/comments/1ioxu5r/still_confused_about_how_mcp_works_heres_the/
8. Simplest MCP Server Example using Python - Tech Easy Blog, accessed on January 4, 2026, <https://techeasyblog.com/ai/simplest-mcp-server-using-python/>
9. MCP Server for DevOps on WSL: Automate Docker, systemd, and System Metrics with Python | by Priyanshu singh | Medium, accessed on January 4, 2026, <https://medium.com/@2003priyanshusingh/mcp-server-for-devops-on-wsl-automate-docker-systemd-and-system-metrics-with-python-298c358573a3>
10. deephaven/deephaven-mcp: Deephaven Model Context Protocol - GitHub, accessed on January 4, 2026, <https://github.com/deephaven/deephaven-mcp>

11. guinacio/mcp-google-calendar: A MCP server that allows Claude and other MCP clients to interact with Google Calendar. This server enables AI assistants to manage your calendar events, check availability, and handle scheduling tasks. - GitHub, accessed on January 4, 2026, <https://github.com/guinacio/mcp-google-calendar>
12. rsc1102/Google_Calendar_MCP: Model Context Protocol (MCP) server that integrates with the Google Calendar API - GitHub, accessed on January 4, 2026, https://github.com/rsc1102/Google_Calendar_MCP
13. CoMfUcloS/obsidian-mcp-sb: An MCP (Model Context Protocol) server that provides intelligent read-only access to your Obsidian vault, enabling it to function as a "second brain" for LLMs. - GitHub, accessed on January 4, 2026, <https://github.com/CoMfUcloS/obsidian-mcp-sb>
14. MCP server with tools to manage workouts on Garmin Connect - GitHub, accessed on January 4, 2026, <https://github.com/st3v/garmin-workouts-mcp>
15. matin/garth: Garmin SSO auth + Connect Python client - GitHub, accessed on January 4, 2026, <https://github.com/matin/garth>
16. garth-mcp-server | MCP Servers · LobeHub, accessed on January 4, 2026, <https://lobehub.com/pl/mcp/matin-garth-mcp-server>
17. Connect to local MCP servers - Model Context Protocol, accessed on January 4, 2026, <https://modelcontextprotocol.io/docs/develop/connect-local-servers>
18. Configuring MCP Tools in Claude Code - The Better Way - Scott Spence, accessed on January 4, 2026, <https://scottspence.com/posts/configuring-mcp-tools-in-claude-code>
19. Connect Claude Code to tools via MCP, accessed on January 4, 2026, <https://code.claude.com/docs/en/mcp>