In the AI world, **MCP** generally refers to the **Model Context Protocol**.

What is the Model Context Protocol (MCP)?

- **Overview** The Model Context Protocol, introduced by Anthropic in November 2024, is an **open standard** designed to enable seamless, secure, and bidirectional integration between AI models (particularly LLMs) and external tools, services, or datasets. It effectively acts as the "USB-C" of AI—providing a universal interface so that once a tool or data source is connected via MCP, any compliant AI system can interact with it without custom glue code. ([Wikipedia][1], [Anthropic][2], [The Verge][3])
- **How It Works** MCP uses a **client-server architecture**:
  - **MCP Clients** (e.g., AI agents like Claude Desktop) send requests.
  - **MCP Servers** expose data, tools, or workflows. Requests and responses follow a structured format (typically JSON-RPC 2.0) to allow AI models to dynamically discover and invoke capabilities across platforms. ([Replit Blog][4], [Stytch][5], [Descope][6])
- **Purpose & Benefits** MCP resolves the so-called **NxM integration problem**—where numerous LLMs and tools need pairwise connections—by enabling standard interoperability. This replaces the tedious process of building custom integrations for every combination. ([Descope][6], [Andreessen Horowitz][7], [Replit Blog][4])
- **Industry Adoption** Since its launch, MCP has seen widespread adoption. OpenAI, Google DeepMind, Replit, Block, Sourcegraph, and more have integrated MCP into their platforms. Microsoft has also embraced MCP within Windows, branding it the "USB-C of AI apps" and embedding support in its Windows AI Foundry ecosystem. ([Wikipedia][1], [Axios][8], [The Verge][9])

Security & Safety Considerations

With growing adoption, security has emerged as a critical concern:

- **Prompt Injection & Tool Poisoning** Researchers have identified risks like malicious MCP servers performing prompt injection, stealing credentials, or executing unwanted code. ([Wikipedia][1], [arXiv][10])
- **Safety Auditing Tools** Tools such as the **MCPSafetyScanner** have been developed to evaluate MCP servers for vulnerabilities before deployment. ([arXiv][10])

Quick Analogy

Think of MCP as a universal connector for AI models—much like USB-C connects devices to accessories. Instead of writing custom code for each AI-tool pairing, you connect using the MCP standard and your AI can access a wide variety of tools seamlessly.

TL;DR

- **MCP stands for Model Context Protocol**, an open, standardized protocol for connecting AI models with tools and data sources.

- It enables portability and interoperability across systems, reducing fragmentation and simplifying development.
- Widely adopted across major AI players.
- Comes with associated security risks that are being actively researched and mitigated.

What MCP Would Add to Our Actus Agent

1. Plug-and-Play Access to External Tools & Data

- Our Actus Agent could **connect to any MCP-compliant tool or dataset** without building a custom integration.
- Examples:
  - MCP server providing **video metadata search**.
  - MCP server connected to **broadcast schedule databases**.
  - MCP server with **speech-to-text or translation APIs**.
  - MCP server for **face recognition or keyword detection pipelines**.

---

2. Real-Time AI + Data Fusion

- MCP can let our AI query **live or stored data** during a job:
  - While processing a broadcast file, the AI could directly pull **context** from an MCP data source (like “who is speaking” or “what’s trending now”).
  - Could cross-reference broadcast content with **news feeds, sports results, or ad schedules** in real time.

---

3. Easier Multi-AI Interoperability

- If in the future we want our Actus Agent to talk to **Claude, GPT, Gemini, or other LLMs**, MCP acts as a **universal bridge**.
- Instead of building a custom API for each, our Actus Agent would expose itself as an MCP server (or use an MCP client) and any MCP-capable AI could use it immediately.

---

4. Reduced Development Effort

- Instead of “N x M” integrations between our tools and each AI model, MCP means **write once, connect everywhere**.
- This is useful if we plan to integrate:
  - Our **SpeechmatixSvc** transcription.
  - Our **Real-Time Job Scheduler**.
  - Our **media insight modules**.

---

5. Safer, More Controlled Tool Usage

- MCP includes permission and capability negotiation:

- Our Actus Agent can **control exactly what external AIs are allowed to do** (e.g., read data, trigger a job, stop a recording).
- The protocol logs all calls for auditing.

Example in Our System

Right now we have:

- Real-time jobs (transcription, keyword detection, face detection).
- Stored metadata in MongoDB.
- Custom REST/WebSocket APIs.

With MCP:

1. We'd write a small **MCP server wrapper** around our Actus Agent's API.
2. An MCP-capable AI model could then:
    - List available Actus commands (`detect_keywords` , `get_transcripts` , `start_job`).
    - Execute them with parameters.
    - Get results in a standard JSON format.

Result: We could control our media processing from *inside* an LLM chat without custom API coding for each model vendor.

Here is a **practical MCP integration design for Actus Agent**: an architecture, concrete capabilities (what MCP unlocks), example tool definitions / JSON shapes, security & safety controls, an implementation checklist with a minimal code sketch, and testing/deployment notes.

1) High-level summary — what MCP gives Actus Agent

- **Plug-and-play access**: Expose Actus features (start/stop jobs, get transcripts, search metadata, run detectors) as *MCP tools* so any MCP-capable LLM (Claude, other agents) can discover & call them. ([Model Context Protocol][1])
- **Context on demand**: LLMs can request precise context (e.g., the last 5 minutes of transcript for channel X) instead of you having to build custom natural-language scrapers. ([Model Context Protocol][4])
- **Bidirectional control**: Agents can both *read* state (job status, transcripts, metadata) and *invoke side effects* (start a new real-time job, pause recording). ([GitHub][2])
- **Faster multi-model integration**: write one MCP server for Actus; many LLM clients can use it without bespoke adapters. ([The Verge][5])

2) Recommended MCP capabilities (mapped to Actus features)

Each capability becomes one or more MCP *tools*.

1. `list_jobs` — returns jobs summary (id, name, status, start/end, channelIds).
2. `get_job` — full job metadata + operations + results.
3. `start_job` — create & schedule a new real-time job (returns job id).
4. `stop_job` / `pause_job` / `resume_job` — control endpoints.
5. `get_transcript_segment` — request transcript for channel/time range (with pagination).
6. `detect_keywords` — run keyword detection on a specified time range or return precomputed results.
7. `get_insight` — fetch computed insights (face detection, scene labels, alerts).
8. `subscribe_notifications` — register a webhook / push subscription for job events.
9. `health` / `capabilities` — server health and metadata (version, rate limits).

(You can combine some of these or break them into more granular endpoints depending on permission needs.)

3) Example MCP tool descriptor (conceptual JSON)

This is the metadata the MCP client will read during discovery (trimmed):

```
{
  "id": "actussvc.jobs",
  "name": "Actus Job Manager",
  "description": "Manage real-time and batch media jobs: start, stop, query status,
fetch transcripts and insights.",
  "commands": [
    {
      "name": "list_jobs",
      "description": "List jobs with basic metadata.",
      "input_schema": {"type":"object","properties":{"status":{"type":"string"}}},
      "output_schema": {"type":"array","items":{"$ref":"#/definitions/JobSummary"}}
    },
    {
      "name": "start_job",
      "description": "Create & start a job. Returns job id.",
      "input_schema": {"type":"object","properties":{"name":
{"type":"string"},"channelIds":{"type":"array","items":
{"type":"string"}},"operations":{"type":"array"}}},
      "output_schema": {"type":"object","properties":{"jobId":{"type":"string"}}}
    }
  ]
}
```

When an LLM invokes `start_job`, the MCP client will send a JSON-RPC style call per the spec; your server performs the action and returns JSON.

(Use the MCP schema.ts/source spec as authoritative when implementing.) ([Model Context Protocol][1])

4) Security & safety — MUST HAVES

MCP allows agentic LLMs to run side effects — so lock this down:

- **Authentication**: accept OAuth2 client credentials or signed API keys for MCP clients; support short-lived tokens. Use mTLS for server-to-server if possible. ([Anthropic][6])
- **Scoped permissions**: define granular scopes, e.g. `jobs.read`, `jobs.write`, `transcript.read`, `transcript.write`, `notifications.manage`. Deny default — explicit grant only.
- **Least privilege in tool descriptions**: keep descriptions concise and avoid long embedded prompts or examples (these get injected into LLM context and can be abused). Sanitize tool descriptions to prevent prompt-injection style attacks. ([Stytch][7], [WRITER][8])
- **Action confirmation**: for destructive operations (`delete_job`, `force_stop`), require additional confirmation parameters or two-step actions (issue intent token first, then execute).
- **Rate-limits & quotas**: per-client and per-tool rate limits to prevent abuse.
- **Input validation**: strict JSON schemas and server-side validation (reject unexpected fields).
- **Output sanitization**: truncate extremely long model requests and strip any embedded control sequences before returning to an LLM.
- **Audit logs**: immutable logs of every MCP call (who, when, input, output hash). Use WORM logs for high-sensitivity environments.
- **Automated security scanning**: run something like **McpSafetyScanner** before exposing an MCP server publicly. ([arXiv][3])

5) Minimal architecture / components

- **MCP Server (Actus MCP adapter)** — exposes MCP endpoints. Thin wrapper over existing Actus internal APIs / microservices.
- **Auth Layer** — OAuth2 / mTLS gateway in front of MCP server (can be same app).
- **Command Executor** — maps MCP tool calls to Actus internal service calls (JobService, TranscriptionService, InsightService).
- **Schema validation + sandbox** — validate inputs & outputs against JSON schemas.
- **Event pub/sub** — to support `subscribe_notifications` (webhooks or server-sent events).
- **Audit & monitoring** — logging, Prometheus metrics, rate-limits.
- **CI / Security tests** — run MCP spec conformance and McpSafetyScanner as part of CI.

Diagram (textual): LLM client ↔ (mTLS / OAuth) ↔ MCP Gateway (Actus MCP Server) → Command Executor → Internal Actus services (Job scheduler, MongoDB, Transcription engine) Events → Notification subsystem → MCP push to client

6) Implementation plan (practical steps)

1. **Read the MCP spec & run reference server locally.** (quick start resources exist). ([Model Context Protocol][1])
2. **Design tool set** (start with read-only + a couple write ops: `start_job`, `get_transcript_segment`, `list_jobs`). Keep it small at first.

3. **Auth & scopes**: decide token provider (Azure AD? Keycloak? internal OAuth2) and map scopes to Actus roles.
4. **Implement MCP server adapter** (Node/TypeScript or Python). Use official MCP SDK or reference servers as template. ([GitHub][2], [Medium][9])
5. **Schema and validations**: define JSON schemas for inputs/outputs and integrate a validator.
6. **Unit & integration tests**: include MCP conformance tests; run McpSafetyScanner. ([arXiv][3])
7. **Staging rollout (read-only)**: first expose read-only endpoints to a test LLM client and observe behavior.
8. **Enable write ops behind stricter policies** (manual approval, limited clients).
9. **Monitor & iterate**: watch logs, tune rate limits, refine schemas.

7) Minimal code sketch (Node / TypeScript pseudocode)

This is a tiny conceptual sketch — use official MCP SDKs in production.

```
// pseudo: mcp-server.ts
import express from "express";
import { validate } from "ajv";

const app = express();
app.use(express.json());

// discovery endpoint (MCP clients will fetch tool list)
app.get("/.well-known/mcp/tools", (req, res) => {
  res.json([{
    id: "actussvc.jobs",
    name: "Actus Job Manager",
    commands: [
      { name: "list_jobs", input_schema: {...}, output_schema: {...} },
      { name: "start_job", input_schema: {...}, output_schema: {...} }
    ]
  }]);
});

// invoke endpoint
app.post("/invoke", authenticateMcpClient, async (req, res) => {
  const { tool, command, args } = req.body;
  // validate vs schema
  const ok = validate(argsSchemaFor(tool,command), args);
  if (!ok) return res.status(400).json({error:"invalid input"});
  // map to internal services
  if (tool==="actussvc.jobs" && command==="start_job") {
    const jobId = await JobService.createJob(args);
    return res.json({ jobId });
  }
  res.status(404).json({error:"unknown command"});
});
```

```
app.listen(9000);
```

8) Testing checklist

- Conformance tests vs official MCP spec. ([Model Context Protocol][1])
- McpSafetyScanner run and fix findings. ([arXiv][3])
- Penetration tests: prompt-injection simulation, overflow/large payload tests.
- Rate limit / chaos testing to ensure job scheduler stability.

9) Operational considerations

- **Backpressure**: limit how many concurrent `start_job` calls from MCP clients to preserve capacity.
- **Cost & billing**: if LLMs will trigger heavy processing, track clients for chargeback.
- **Data privacy**: transcripts may contain PII — enforce retention policies and restricted scopes.

Here's a **C# MCP server skeleton** for our **Actus Agent**, plus example **JSON Schemas** for the first few tools (`list_jobs` , `start_job` , `get_transcript_segment`).

This will let you expose your **Job Scheduler**, **Transcription**, and **Insight** features via **Model Context Protocol** so any MCP-compliant LLM can use them.

---

1. Project Structure

```
ActusMcpServer/
  ├── Program.cs
  ├── Controllers/
  │     ├── DiscoveryController.cs
  │     ├── InvokeController.cs
  ├── Models/
  │     ├── JobSummary.cs
  │     ├── StartJobRequest.cs
  │     ├── StartJobResponse.cs
  │     ├── TranscriptSegmentRequest.cs
  │     ├── TranscriptSegmentResponse.cs
  ├── Schemas/
  │     ├── list_jobs_input.json
  │     ├── list_jobs_output.json
  │     ├── start_job_input.json
  │     ├── start_job_output.json
  │     ├── get_transcript_segment_input.json
  │     ├── get_transcript_segment_output.json
```

---

2. Minimal C# MCP Server Skeleton (ASP.NET Core)

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using System.Text.Json;
using System.Text.Json.Serialization;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();
builder.Services.AddEndpointsApiExplorer();

var app = builder.Build();

app.MapControllers();

// Discovery endpoint
app.MapGet("/.well-known/mcp/tools", () =>
{
    var tools = new[]
    {
        new
        {
            id = "actussvc.jobs",
            name = "Actus Job Manager",
            description = "Manage real-time and batch media jobs.",
            commands = new object[]
            {
                new {
                    name = "list_jobs",
                    description = "List jobs with optional status filter.",
                    input_schema = "/schemas/list_jobs_input.json",
                    output_schema = "/schemas/list_jobs_output.json"
                },
                new {
                    name = "start_job",
                    description = "Start a new job with given parameters.",
                    input_schema = "/schemas/start_job_input.json",
                    output_schema = "/schemas/start_job_output.json"
                },
                new {
                    name = "get_transcript_segment",
                    description = "Get transcript for a given channel/time range.",
                    input_schema = "/schemas/get_transcript_segment_input.json",
                    output_schema = "/schemas/get_transcript_segment_output.json"
                }
            }
        }
    };
    return Results.Json(tools);
});

// Invocation endpoint
```

```
app.MapPost("/invoke", async (HttpRequest request) =>
{
    using var reader = new StreamReader(request.Body);
    var body = await reader.ReadToEndAsync();
    var call = JsonSerializer.Deserialize<McpInvocation>(body);

    switch (call.Command)
    {
        case "list_jobs":
            // Example stub - replace with your JobService
            var jobs = new[]
            {
                new JobSummary { Id="1", Name="Test Job", Status="Running" }
            };
            return Results.Json(jobs);

        case "start_job":
            var req = JsonSerializer.Deserialize<StartJobRequest>(call.Args);
            // Call your scheduling logic here
            var newJobId = Guid.NewGuid().ToString();
            return Results.Json(new StartJobResponse { JobId = newJobId });

        case "get_transcript_segment":
            var segReq = JsonSerializer.Deserialize<TranscriptSegmentRequest>
(call.Args);
            // Fetch from your Mongo/transcript store
            return Results.Json(new TranscriptSegmentResponse {
                ChannelId = segReq.ChannelId,
                StartTime = segReq.StartTime,
                EndTime = segReq.EndTime,
                Transcript = "Example transcript text..."
            });

        default:
            return Results.BadRequest(new { error = "Unknown command" });
    }
});

app.Run();

public class McpInvocation
{
    [JsonPropertyName("tool")] public string Tool { get; set; }
    [JsonPropertyName("command")] public string Command { get; set; }
    [JsonPropertyName("args")] public JsonElement Args { get; set; }
}

public class JobSummary
{
    public string Id { get; set; }
    public string Name { get; set; }
    public string Status { get; set; }
```

```
}

public class StartJobRequest
{
    public string Name { get; set; }
    public string[] ChannelIds { get; set; }
    public string[] Operations { get; set; }
}

public class StartJobResponse
{
    public string JobId { get; set; }
}

public class TranscriptSegmentRequest
{
    public string ChannelId { get; set; }
    public DateTime StartTime { get; set; }
    public DateTime EndTime { get; set; }
}

public class TranscriptSegmentResponse
{
    public string ChannelId { get; set; }
    public DateTime StartTime { get; set; }
    public DateTime EndTime { get; set; }
    public string Transcript { get; set; }
}
```

---

3. Example JSON Schemas

list_jobs_input.json

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "type": "object",
  "properties": {
    "status": { "type": "string", "enum": ["Pending", "Running", "Completed",
"Failed"] }
  },
  "required": []
}
```

list_jobs_output.json

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "type": "array",
  "items": {
    "type": "object",
    "properties": {
```

```
      "id": { "type": "string" },
      "name": { "type": "string" },
      "status": { "type": "string" }
    },
    "required": ["id", "name", "status"]
  }
}
```

start_job_input.json

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "type": "object",
  "properties": {
    "name": { "type": "string" },
    "channelIds": {
      "type": "array",
      "items": { "type": "string" }
    },
    "operations": {
      "type": "array",
      "items": { "type": "string" }
    }
  },
  "required": ["name", "channelIds", "operations"]
}
```

start_job_output.json

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "type": "object",
  "properties": {
    "jobId": { "type": "string" }
  },
  "required": ["jobId"]
}
```

get_transcript_segment_input.json

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "type": "object",
  "properties": {
    "channelId": { "type": "string" },
    "startTime": { "type": "string", "format": "date-time" },
    "endTime": { "type": "string", "format": "date-time" }
  },
  "required": ["channelId", "startTime", "endTime"]
}
```

get_transcript_segment_output.json

```json { "$schema": "http://json-schema.org/draft-07/schema#", "type": "object", "properties": { "channelId": { "type": "string" }, "startTime": { "type": "string", "format": "date-time" }, "endTime": { "type": "string", "format": "date-time" }, "transcript": { "type": "string" } }, "required": ["channelId", "startTime", "endTime", "transcript"] }
```