ChatGPT

YouTube MCP Server on Cloud Run – Step-by-Step Guide

Tools and Prerequisites

- **Languages/Frameworks:** Python 3.10+, along with the MCP Python SDK (`mcp[cli]`) or a similar MCP library (e.g. FastMCP) [1] . You will also need an HTTP client library (e.g. `httpx` or `requests`) to call the YouTube Data API.
- **YouTube Data API:** Enable **YouTube Data API v3** in Google Cloud Console and obtain an *API key* for your project [2] . For read-only operations like search or metadata lookup, an API key in query parameters is sufficient. (For user-specific data or uploads you would use OAuth2, but that's beyond this guide.)
- **Development Tools:** Install the Google Cloud CLI (`gcloud`) for deployment, and Docker if you plan to build a container manually. (Alternatively, Cloud Build can deploy directly from source.)
- **MCP Knowledge:** The Model Context Protocol uses JSON-RPC (over STDIO or streamable HTTP/SSE) to expose "tools" and "resources" to LLMs [3] . We will expose each YouTube function as an MCP *tool*.

1. Setup the MCP Server Project

1. **Create a Python environment:**

```
python3 -m venv venv
source venv/bin/activate
pip install "mcp[cli]" httpx
```

The `mcp[cli]` package installs the FastMCP server framework [4] . Ensure Python ≥3.10.

2. **Project layout:** In your project directory (say `youtube-mcp-server/`), create a main server file (e.g. `server.py`) and a `requirements.txt` listing at least:

```
mcp[cli]
httpx
```

3. **YouTube API key:** Obtain an API key via Google Cloud Console (Enable YouTube Data API v3 → **Credentials** → **API Key**) [2] . Store this key in an environment variable (e.g. `YOUTUBE_API_KEY`) or Secret Manager. In Cloud Run, you will set the same env var.

2. Build the MCP Server and Schema

We use FastMCP to define tools. Below is skeleton code to start the server:

```
from mcp.server.fastmcp import FastMCP
import os, httpx

# Load YouTube API key from env
YOUTUBE_API_KEY = os.getenv("YOUTUBE_API_KEY")

# Initialize MCP server; json_response=True returns raw JSON (optional)
mcp = FastMCP("YouTubeMCP", json_response=True)
```

The `FastMCP` class will auto-generate tool schemas from type hints and docstrings [5].

Define a tool (endpoint): Use `@mcp.tool()` to declare each callable function. For example, a search tool:

```
@mcp.tool()
async def search_videos(query: str, maxResults: int = 5) -> str:
    """
    Search YouTube videos by keyword.
    """
    params = {
        "part": "snippet",
        "q": query,
        "maxResults": maxResults,
        "key": YOUTUBE_API_KEY
    }
    async with httpx.AsyncClient() as client:
        resp = await client.get("https://www.googleapis.com/youtube/v3/search",
params=params, timeout=15.0)
        data = resp.json()
    # Format output (e.g., list titles)
    if "items" not in data:
        return "No results."
    titles = [item["snippet"]["title"] for item in data["items"]]
    return "\n".join(f"{i+1}. {t}" for i, t in enumerate(titles))
```

This snippet uses FastMCP to create a tool named `search_videos` (MCP will expose it with that name) [6]. You would similarly define other tools, e.g.:

```
@mcp.tool()
async def get_video_details(id: str) -> str:
    """Get details (title, views, etc.) for a video ID."""
    params = {"part": "snippet,statistics,contentDetails", "id": id, "key": YOUTUBE_API_KEY}
    async with httpx.AsyncClient() as client:
        resp = await client.get("https://www.googleapis.com/youtube/v3/videos", params=params, timeout=15.0)
        data = resp.json()
    if "items" not in data or not data["items"]:
        return f"Video ID {id} not found."
    info = data["items"][0]
    title = info["snippet"]["title"]
    stats = info["statistics"]
    return f"Title: {title}\nViews: {stats.get('viewCount','?')}, Likes: {stats.get('likeCount','?')}"
```

```
@mcp.tool()
async def get_channel_info(channelId: str = None, forUsername: str = None) -> str:
    """Get channel info by ID or username."""
    params = {"part": "snippet,statistics", "id": channelId, "forUsername": forUsername, "key": YOUTUBE_API_KEY}
    async with httpx.AsyncClient() as client:
        resp = await client.get("https://www.googleapis.com/youtube/v3/channels", params=params, timeout=15.0)
        data = resp.json()
    if "items" not in data or not data["items"]:
        return "Channel not found."
    ch = data["items"][0]
    name = ch["snippet"]["title"]
    subs = ch["statistics"].get("subscriberCount","?")
    return f"Channel: {name}, Subscribers: {subs}"
```

After defining your tools, **run** the MCP server. For HTTP streaming transport (SSE), use:

```
if __name__ == "__main__":
    mcp.run(transport="streamable-http")
```

FastMCP will start an HTTP server (by default on port 8000) that listens for MCP JSON-RPC calls. Cloud Run supports *streamable HTTP* (SSE) transport for MCP servers [3], so this configuration is compatible. The MCP server's entrypoint is the `/mcp` or `/sse` route (depending on defaults), which clients will use.

3. YouTube Data API Authentication

For public data (search, stats), append the API key as a query parameter `key=YOUR_API_KEY` [2]. In the code above, we passed `YOUTUBE_API_KEY` from the environment. Ensure to set this environment variable before running or deploying. For example, in Cloud Run you can go to **Container > Environment variables** and add `YOUTUBE_API_KEY` with your key.

YouTube's quotas are limited (default 10,000 units/day). Each search costs 100 units, video lookup 1 unit, etc. (Monitor usage in Google Cloud Console.)

4. Structuring MCP Endpoints

Each `@mcp.tool` function becomes an endpoint that clients can invoke via the MCP protocol [6]. FastMCP handles the JSON-RPC boilerplate. Under the hood, when an LLM agent calls a tool, it sends a JSON-RPC request like:

```
{"jsonrpc":"2.0","method":"tools/call","params":
{"name":"search_videos","arguments":{"query":"cats","maxResults":3}},"id":1}
```

and FastMCP returns a JSON-RPC response with the string output of your function.

Because we used Python type hints and docstrings, FastMCP auto-generates the schema of each tool. You can view or customize the generated OpenAPI/JSON schema if needed, but FastMCP does this automatically.

(No extra manual "schema file" is needed beyond the decorated functions, since FastMCP produces the spec. If you did want an OpenAPI spec for a GPT plugin manifest, you could export it by adding a prompt or resource, but this is optional.)

5. Containerize and Deploy to Cloud Run

1. **Create a Dockerfile:** For Cloud Run, you can build a container. Example `Dockerfile`:

```
FROM python:3.11-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
# Expose port (Cloud Run uses $PORT or default 8000)
ENV PORT 8000
CMD ["uv", "run", "server.py"]
```

This uses the `uv` runner installed with `mcp[cli]`. Alternatively, use `uvicorn server:app` or `python server.py` if you adapt the entrypoint. The key is that the container listens on `$PORT` (Cloud Run sets `PORT=8080` or uses the default).

1. **Build & Push:** Tag and push the image to Container Registry or Artifact Registry:

```
docker build -t gcr.io/$PROJECT_ID/youtube-mcp .
docker push gcr.io/$PROJECT_ID/youtube-mcp
```

2. **Deploy with gcloud:**
   You can deploy directly from the image:

   ```
   gcloud run deploy youtube-mcp --image gcr.io/$PROJECT_ID/youtube-mcp --
   platform managed --region us-central1 --allow-unauthenticated
   ```

   Or deploy from source (no Dockerfile needed explicitly):

   ```
   cd /path/to/youtube-mcp-server
   gcloud run deploy youtube-mcp --source . --platform managed --region us-
   central1 --allow-unauthenticated
   ```

   (The `--allow-unauthenticated` flag makes the endpoint public; omit if you require IAM auth.) After deployment, Cloud Run gives you an HTTPS URL for the service, e.g. `https://youtube-mcp-xxxx-uc.a.run.app`.

3. **Configure Cloud Run:** In the Cloud Run console, set the `YOUTUBE_API_KEY` environment variable to your API key. Ensure the container port matches your code (usually 8000). Cloud Run automatically handles HTTPS and will use SSE on the `/sse/` or `/mcp` endpoints for streaming.

6. OpenAPI/Plugin Metadata (Optional)

To let OpenAI's systems (like the Agent Builder or ChatGPT plugins) *discover* your MCP tools, you can provide a **.well-known/ai-plugin.json** manifest and an OpenAPI spec. This is the same format used by ChatGPT plugins. For example, host a file at `https://your-domain.com/.well-known/ai-plugin.json` such as:

```
{
  "schema_version": "v1",
  "name_for_human": "YouTube MCP Server",
  "name_for_model": "youtube_mcp",
  "description_for_human": "Search YouTube and get video info.",
  "description_for_model": "Use this to search YouTube videos and retrieve
metadata.",
  "auth": { "type": "none" },
  "api": {
    "type": "openapi",
    "url": "https://your-domain.com/openapi.json",
    "is_user_authenticated": false
  },
```

```
  "logo_url": "https://your-domain.com/logo.png",
  "contact_email": "admin@your-domain.com"
}
```

You would also generate (or let FastMCP generate) an OpenAPI spec (e.g. at `/openapi.json`) that describes the `search_videos`, `get_video_details`, etc. tools and their parameters. The manifest tells the LLM to fetch the spec and auto-learn your API. *(In practice, Agent Builder can accept the endpoint directly, so hosting the plugin manifest is optional unless you want ChatGPT to auto-add it as a plugin.)*

7. Using in OpenAI Agent Builder

Once your MCP server is running and reachable, you can add it as a tool in Agent Builder:

- In the **Agent Builder** UI on platform.openai.com/agent-builder, go to your workflow and click **"+ Add Node" → "Tools" → "MCP"**.
- Click **"+ Servers"** and enter a **Server Label/Name** (e.g. "YouTubeSearch"), an optional description, and the **Server URL** of your Cloud Run endpoint. Use the streaming endpoint (e.g. `https://youtube-mcp-xxxx-uc.a.run.app/sse/`) [7] . If your Cloud Run service requires no auth, set "No auth" or leave token blank.
- Save the server. Agent Builder will list the tools available (derived from your MCP schema). You can now drag an **MCP tool node** into your workflow and select one of your YouTube tools (e.g. `search_videos`).

Alternatively, when using the Responses API or Agents SDK, include your MCP server in the `tools` array. For example, using `openai.Responses.create` in Python:

```
client = OpenAI()
response = client.responses.create(
    model="gpt-4o-mini",
    tools=[{
        "type": "mcp",
        "server_label": "YouTube",
        "server_url": "https://youtube-mcp-xxxx-uc.a.run.app/sse/",
        "require_approval": "never"
    }],
    input="Search for the top 3 cats videos on YouTube."
)
print(response.output_text)
```

This tells the model about your MCP server [8] . In the JSON above, `type: "mcp"` and `server_url` point to your service; `server_label` is a friendly name. The model can then autonomously invoke your `search_videos` tool when answering.

Finally, in Agent Builder's workflow, use an **MCP node** (connect it to an Agent node) and select the YouTube tool. The agent will call your Cloud Run MCP endpoint under the hood to perform searches and lookups.

Summary: By combining a Python MCP server (FastMCP) that wraps YouTube Data API calls with Google Cloud Run hosting, you provide an LLM-accessible toolset. Be sure to set up the correct API key and endpoints, containerize your app (Docker or `gcloud run deploy`), and register the server in Agent Builder or as a ChatGPT plugin manifest [1] [8]. This allows your AI agents to call YouTube search and metadata functions via the MCP standard.

Sources: Official MCP and Cloud Run documentation [3] [1]; OpenAI examples of MCP usage [8] [6]; YouTube Data API setup guide [2].

---

[1] [3] Host MCP servers on Cloud Run | Google Cloud Documentation
https://docs.cloud.google.com/run/docs/host-mcp-servers

[2] How to Create YouTube MCP Server - API to MCP Documentation
https://apitomcp.ai/docs/examples/youtube

[4] [5] GitHub - modelcontextprotocol/python-sdk: The official Python SDK for Model Context Protocol servers and clients
https://github.com/modelcontextprotocol/python-sdk

[6] Build an MCP server - Model Context Protocol
https://modelcontextprotocol.io/docs/develop/build-server

[7] Connectors and MCP servers | OpenAI API
https://developers.openai.com/api/docs/guides/tools-connectors-mcp

[8] Building MCP servers for ChatGPT Apps and API integrations
https://developers.openai.com/api/docs/mcp/