

The Modern Software Developer

CS146S
Stanford University, Fall 2025
Mihail Eric

To MCP and Beyond

Why

- LLMs have vast (but static) world knowledge that only updates when we retrain
- To build fully autonomous systems we need robust ways to feed dynamic data in
 - What's the weather today
 - Who's president
 - What's the price of Bitcoin
 - Who's the narrator in Nike's latest ad campaign
- RAG and tool-calling are the best answer we have today

Basics

- **Model Context Protocol**
 - Open protocol that allows systems to provide context to AI models in a manner generalizable across integrations
 - In English: standard format for exposing tools to LLMs
- History: in the distant past pre-November 2024 when MCP was introduced...

Imagine integrating with a questionable 3rd party API



What APIs do you expose?



```
def poorly_documented_twitter_search(bearer_token: str, query: str = "openai"):
    """
    Example function showing how confusing Twitter API v2 felt when it was poorly documented.

    Issues:
    - Parameters like 'query' were ambiguously explained.
    - 'tweet.fields' options were incomplete in the docs.
    - 'max_results' limits were undocumented or inconsistent.
    - Error responses were vague and often unhelpful.
    """

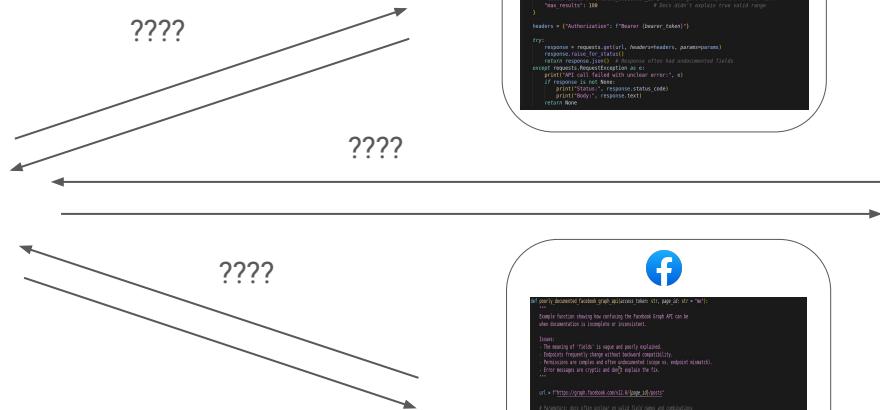
    url = "https://api.twitter.com/2/tweets/search/recent"

    # Parameters: poorly documented, often trial and error
    params = {
        "query": query,                                     # Docs didn't clearly define all supported operators
        "tweet.fields": "created_at,author_id",             # Docs gave incomplete/uncertain list
        "max_results": 100                                  # Docs didn't explain true valid range
    }

    headers = {"Authorization": f"Bearer {bearer_token}"}

    try:
        response = requests.get(url, headers=headers, params=params)
        response.raise_for_status()
        return response.json()  # Response often had undocumented fields
    except requests.RequestException as e:
        print("API call failed with unclear error:", e)
        if response is not None:
            print("Status:", response.status_code)
            print("Body:", response.text)
        return None
```

Now many APIs



```
## poorly documented twitter search (source: https://git.io/vfpmA)
# Example function showing how confusing Twitter API v2 felt when it was poorly documented.

Issues:
  - Parameters like 'query' were ambiguously explained.
  - 'count' fields' options were incomplete in the docs.
  - max_results' limits were undocumented or inconsistent.
  - Many responses were vague and often ambiguous.

...
url = "https://api.twitter.com/2/tweets/search/recent"

params = {
    # poorly documented, often trial and error
    'query': query,
    'start_time': start_time, # Doesn't clearly define all supported operators
    'end_time': end_time, # Doesn't clearly define all supported operators
    'max_results': 100 # Doesn't explain what valid range
}

headers = {'Authorization': f"Bearer {beaver_token}"}

try:
    response = requests.get(url, headers=headers, params=params)
    response.raise_for_status()
    return response.json()
except requests.exceptions.HTTPError as e:
    print(f"HTTP error! (Detailed with unclear errors).", e)
    print(f"Status: {response.status_code}")
    print(f"Body: {response.text}")
    return None
```

```
## poorly documented Facebook graph api (source: https://git.io/vfpmB)
# Example function showing how confusing the Facebook Graph API can be when documentation is incomplete or inconsistent.

Issues:
  - The use of 'fields' is vague and poorly explained.
  - Returns frequently change without backwards compatibility.
  - Returns are often missing required fields (e.g., 'id', 'name', 'request_params').
  - Error messages are cryptic and don't explicitly explain the fail.

...
url = "https://graph.facebook.com/v6.0/page_id?fields"

# Parameters don't often include all valid field names and combinations
# (e.g., 'id,message,reaction_count,likes')
# Fields: 'id,message,reaction_count,likes', a user often fails silently or returns an object instead of a regular JSON object if they miss one of these fields.
# 'access_token': access_token # If access token is missing, it's likely expected as 'missing' vs. 'empty' scope

try:
    response = requests.get(url, params=params)
    response.raise_for_status()
    return response.json()
except requests.exceptions.HTTPError as e:
    print(f"HTTP error! (Detailed with unclear errors).", e)
    if response is not None:
        print(f"Status: {response.status_code}")
        print(f"Body: {response.text}")
    return None
```

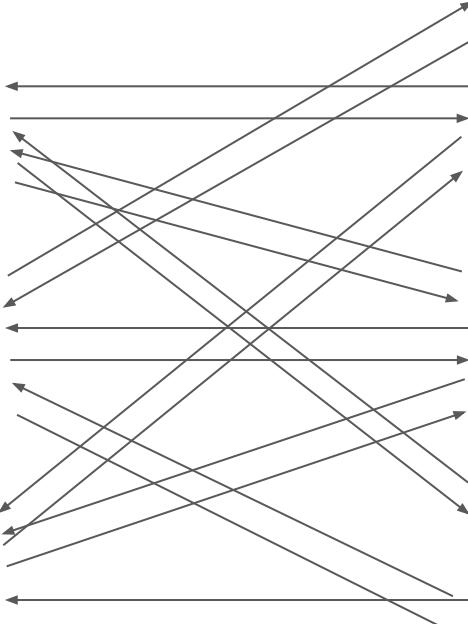
```
## poorly documented linkedin api (source: https://git.io/vfpmC)
# Example function showing how confusing a poorly documented LinkedIn API can be.

Issues:
  - No body: body structure is unclear without a XML and Fields are password.
  - Headers are required but not explained or documented.
  - The 'SOPAuth' header is sometimes required, sometimes not - docs inconsistent.
  - Error messages often return a generic SOAP faults which are hard to decode.

Headers = {
    'Content-Type': 'text/xml; charset=utf-8',
}

try:
    # The XML body structure is vague and often undocumented
    body = """<!DOCTYPE message><?xml version='1.0' encoding='UTF-8'?>
<message>
  <methodCall>
    <methodName>http://example.com/service/</methodName>
    <params>
      <param>
        <name>username</name>
        <value>user12345</value>
      </param>
    </params>
  </methodCall>
</message>"""
    ...
    response = requests.post(url, headers=Headers, data=body)
    response.raise_for_status()
    return response.json()
except requests.exceptions.HTTPError as e:
    print(f"HTTP error! (Detailed with unclear errors).", e)
    if response is not None:
        print(f"Status: {response.status_code}")
        print(f"Body: {response.text}")
    return None
```

Now many LLM apps



```
curl -X POST https://api.twitter.com/2/tweets/create.json  
Content-Type: application/json  
Authorization: Bearer <REDACTED>  
  
{  
  "text": "Hello from the LLM API!"  
}
```

```
curl -X GET https://api.github.com/repos/octocat/Hello-World  
Authorization: Bearer <REDACTED>  
  
{  
  "id": 1,  
  "name": "Hello-World",  
  "full_name": "octocat/Hello-World",  
  "owner": {  
    "login": "octocat",  
    "id": 1,  
    "node_id": "MDQ6VXNlcjE=",  
    "avatar_url": "https://github.com/images/error/octocat_happy.gif",  
    "gravatar_id": "",  
    "url": "https://api.github.com/users/octocat",  
    "html_url": "https://github.com/octocat",  
    "followers_url": "https://api.github.com/users/octocat/followers",  
    "following_url": "https://api.github.com/users/octocat/following{/other_user}",  
    "gists_url": "https://api.github.com/users/octocat/gists{/gist_id}",  
    "starred_url": "https://api.github.com/users/octocat/starred{/owner}",  
    "subscriptions_url": "https://api.github.com/users/octocat/subscriptions",  
    "organizations_url": "https://api.github.com/users/octocat/orgs",  
    "repos_url": "https://api.github.com/users/octocat/repos",  
    "events_url": "https://api.github.com/users/octocat/events{/privacy}"  
  },  
  "private": false,  
  "html_url": "https://github.com/octocat/Hello-World",  
  "description": "This repository is for my first GitHub repo.",  
  "fork": false,  
  "url": "https://api.github.com/repos/octocat/Hello-World",  
  "forks": 0,  
  "forks_count": 0,  
  "stargazers": 0,  
  "stargazers_count": 0,  
  "watchers": 0,  
  "watchers_count": 0,  
  "language": null,  
  "has_issues": true,  
  "has_projects": true,  
  "has_downloads": true,  
  "has_wiki": true,  
  "topics": [],  
  "size": 0,  
  "stargazer_count": 0,  
  "created_at": "2011-04-13T16:20:42Z",  
  "updated_at": "2011-04-13T16:20:42Z",  
  "pushed_at": "2011-04-13T16:20:42Z",  
  "git_url": "https://api.github.com/repos/octocat/Hello-World/git",  
  "ssh_url": "git@github.com:octocat>Hello-World.git",  
  "clone_url": "https://github.com/octocat>Hello-World",  
  "svn_url": "https://github.com/octocat>Hello-World",  
  "permissions": {}  
}
```

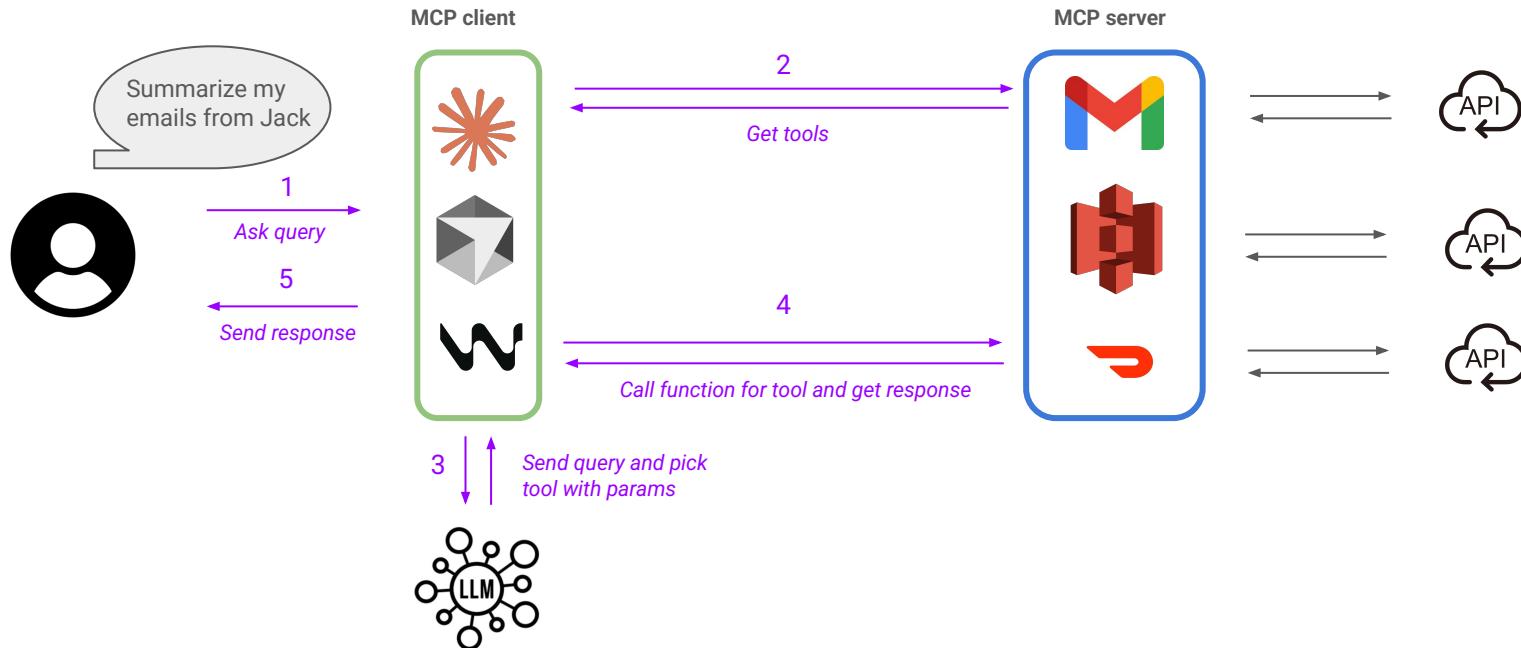
```
curl -X POST https://graph.facebook.com/v10.0/me/messages  
Content-Type: application/json  
Authorization: Bearer <REDACTED>  
  
[  
  {  
    "recipient": {  
      "id": "100000000000000",  
      "name": "Facebook Test User"  
    },  
    "text": "Hello from the LLM API!"  
  }  
]
```

Basics

- MCP
 - Does away with the need to build $M \times N$ connectors from LLM host/agent to underlying tool
 - Don't need to reimplement auth, error handling, rate-limiting, etc
 - Enforces consistent output format using JSON-RPC
 - Extends from Language Server Protocols
 - Allows for proactive agentic workflows rather than purely reactive ones as in LSP
 - Integrating with tools goes from $M \times N \rightarrow M + N$ connectors

MCP A Bit Deeper

- Terminology
 - **Host:** Cursor, Claude Desktop
 - **MCP Client:** Library embedded on host (stateful session per server)
 - **MCP Server:** Lightweight wrapper in front of a tool
 - **Tool:** Callable function (could be data source, API)
- Flow
 - Client calls tools/list to MCP server (what can you do?)
 - Server returns JSON describing each tool (name, summary, JSON schema)
 - Host injects that JSON into model's context
 - User prompt triggers model, emitting a structured tool call
 - MCP server executes and conversation resumes
- MCP provides stdio and SSE transport layer



Let's build a custom MCP server from scratch!

Limitations

- Agents don't handle many tools very well today
- APIs eat up your context window quickly
- Design APIs to be AI-native rather than rigid

Questions?