The Modern Software Developer

CS146S
Stanford University, Fall 2025
Mihail Eric

To MCP and Beyond

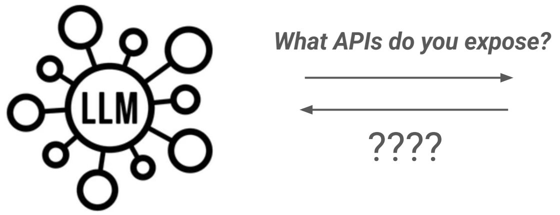
Why

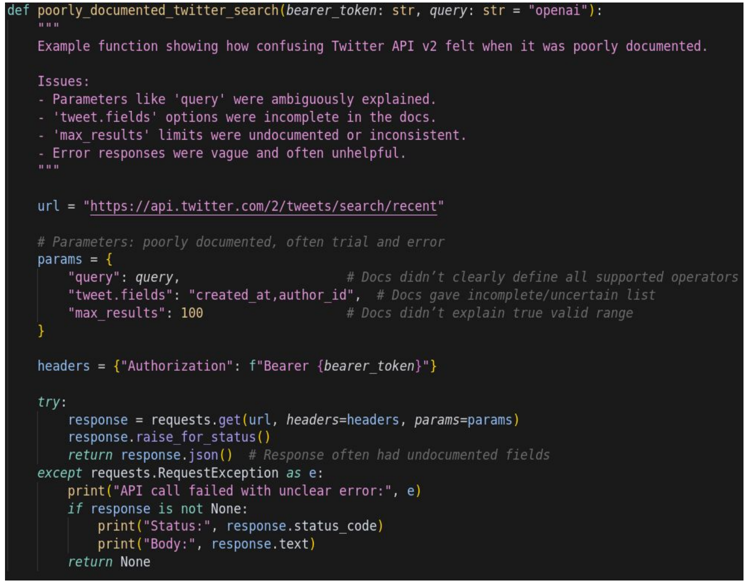
- LLMs have vast (but static) world knowledge that only updates when we retrain
- To build fully autonomous systems we need robust ways to feed dynamic data in
  - What’s the weather today
  - Who’s president
  - What’s the price of Bitcoin
  - Who’s the narrator in Nike’s latest ad campaign
- RAG and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

Basics

- **M**odel **C**ontext **P**rotocol
    - 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…

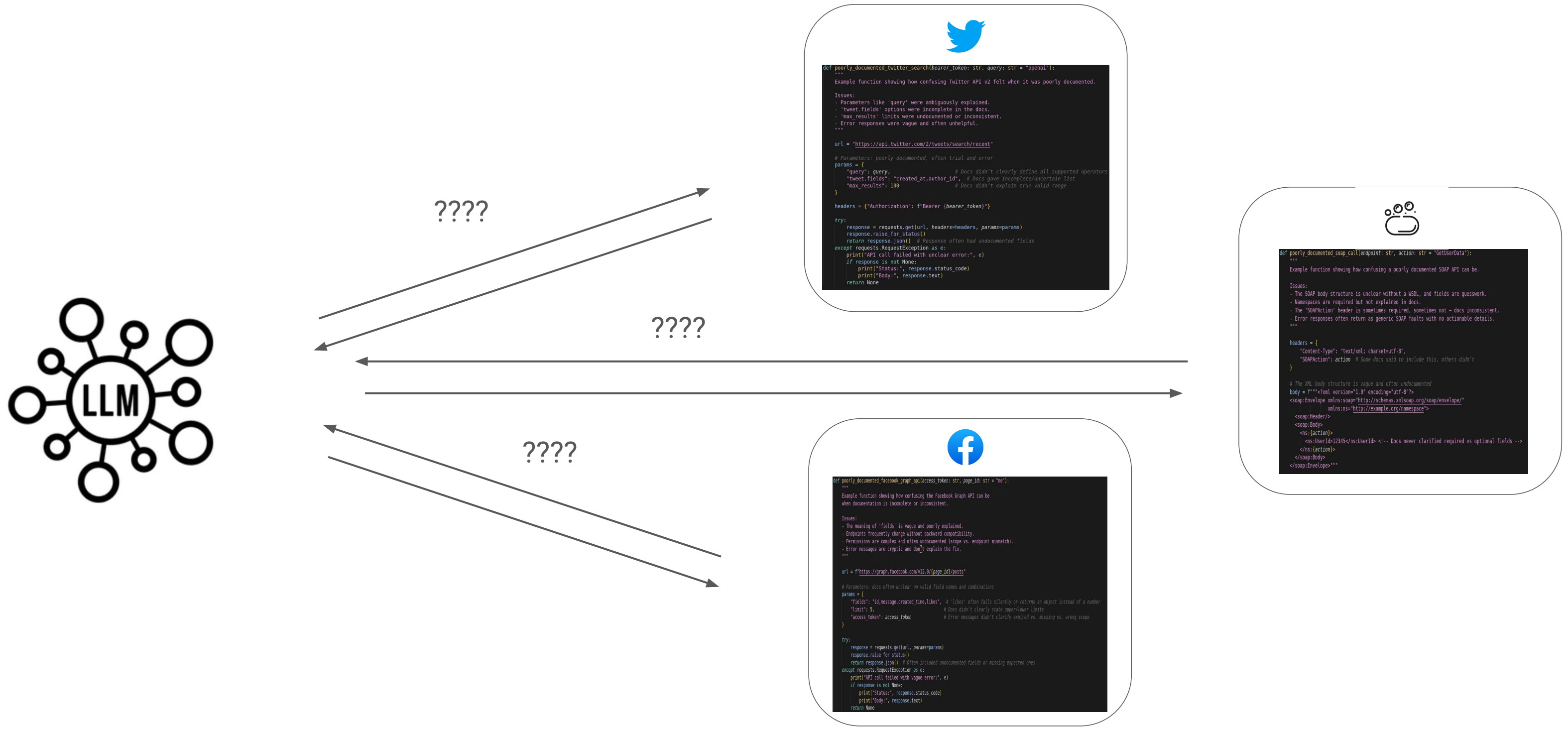
Imagine integrating with a questionable 3rd party API





```python
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

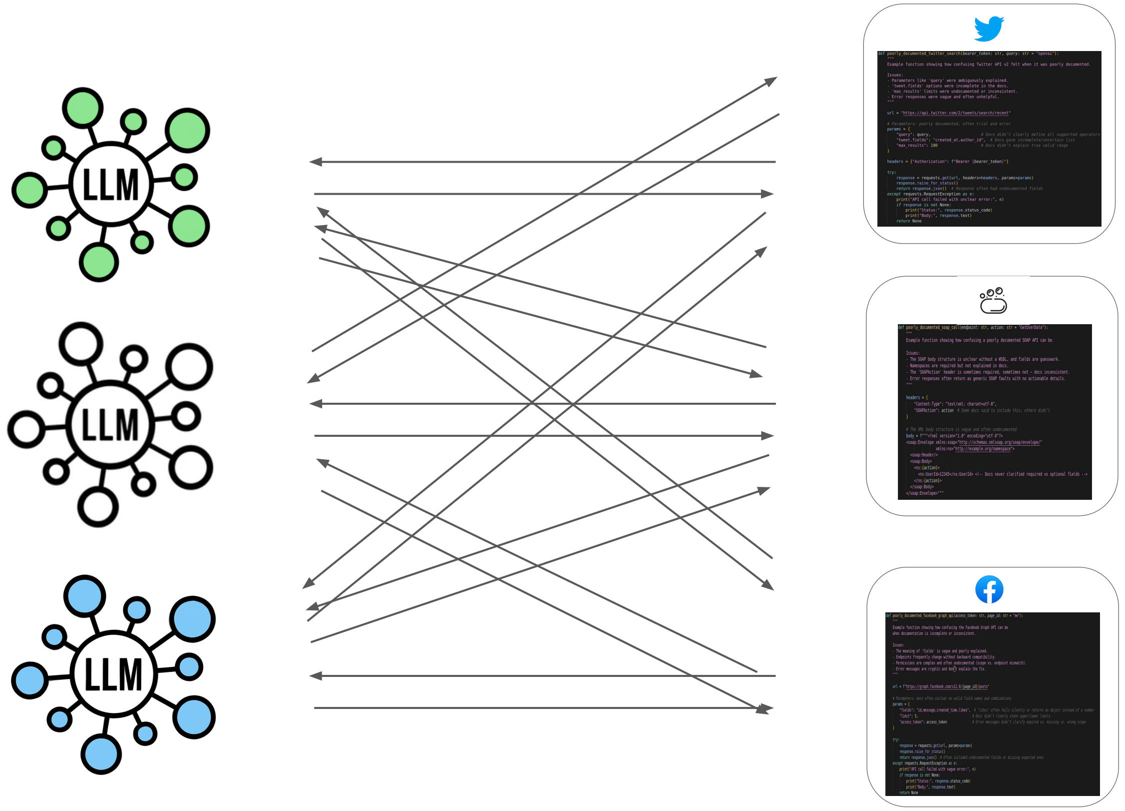


```python
def poorly_documented_twitter_search(bearer_token: str, query: str = "openai"):
    """
    Example function showing how confusing Twitter API v2 felt when it was poorly documented.

    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
```

```python
def poorly_documented_soap_call(endpoint: str, action: str = "GetUserData"):
    """
    Example function showing how confusing a poorly documented SOAP API can be.

    Issues:
    - The SOAP body structure is unclear without a WSDL, and fields are guesswork.
    - Namespaces are required but not explained in docs.
    - The 'SOAPAction' header is sometimes required, sometimes not – docs inconsistent.
    - Error responses often return as generic SOAP faults with no actionable details.
    """

    headers = {
        "Content-Type": "text/xml; charset=utf-8",
        "SOAPAction": action  # Some docs said to include this, others didn’t
    }

    # The XML body structure is vague and often undocumented
    body = f"""<?xml version="1.0" encoding="utf-8"?>
    <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
                   xmlns:ns="http://example.org/namespace">
      <soap:Header/>
      <soap:Body>
        <ns:{action}>
          <ns:UserId>12345</ns:UserId> <!-- Docs never clarified required vs optional fields -->
        </ns:{action}>
      </soap:Body>
    </soap:Envelope>"""
```

```python
def poorly_documented_facebook_graph_api(access_token: str, page_id: str = "me"):
    """
    Example function showing how confusing the Facebook Graph API can be
    when documentation is incomplete or inconsistent.

    Issues:
    - The meaning of 'fields' is vague and poorly explained.
    - Endpoints frequently change without backward compatibility.
    - Permissions are complex and often undocumented (scope vs. endpoint mismatch).
    - Error messages are cryptic and don't explain the fix.
    """

    url = f"https://graph.facebook.com/v12.0/{page_id}/posts"

    # Parameters: docs often unclear on valid field names and combinations
    params = {
        "fields": "id,message,created_time,likes",  # 'likes' often fails silently or returns an object instead of a number
        "limit": 5,                                 # Docs didn’t clearly state upper/lower limits
        "access_token": access_token                # Error messages didn’t clarify expired vs. missing vs. wrong scope
    }

    try:
        response = requests.get(url, params=params)
        response.raise_for_status()
        return response.json()  # Often included undocumented fields or missing expected ones
    except requests.RequestException as e:
        print("API call failed with vague error:", e)
        if response is not None:
            print("Status:", response.status_code)
            print("Body:", response.text)
        return None
```

Now many LLM apps



```python
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
```

```python
def poorly_documented_soap_call(endpoint: str, action: str = "GetUserData"):
    """
    Example function showing how confusing a poorly documented SOAP API can be.

    Issues:
    - The SOAP body structure is unclear without a WSDL, and fields are guesswork.
    - Namespaces are required but not explained in docs.
    - The 'SOAPAction' header is sometimes required, sometimes not – docs inconsistent.
    - Error responses often return as generic SOAP faults with no actionable details.
    """

    headers = {
        "Content-Type": "text/xml; charset=utf-8",
        "SOAPAction": action  # Some docs said to include this, others didn't
    }

    # The XML body structure is vague and often undocumented
    body = f"""<?xml version="1.0" encoding="utf-8"?>
    <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
                   xmlns:ns="http://example.org/namespace">
      <soap:Header/>
      <soap:Body>
        <ns:{action}>
          <ns:UserId>12345</ns:UserId> <!-- Docs never clarified required vs optional fields -->
        </ns:{action}>
      </soap:Body>
    </soap:Envelope>"""
```

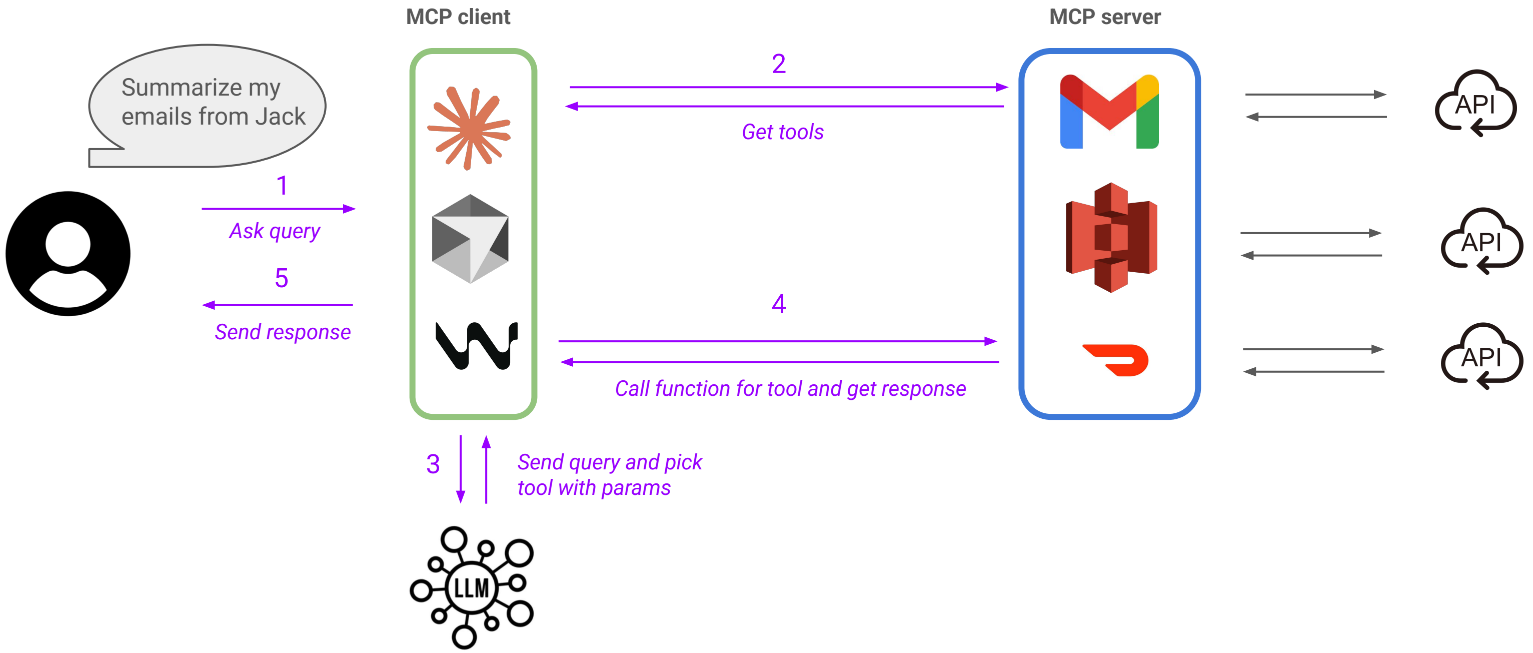
```python
def poorly_documented_facebook_graph_api(access_token: str, page_id: str = "me"):
    """
    Example function showing how confusing the Facebook Graph API can be
    when documentation is incomplete or inconsistent.

    Issues:
    - The meaning of 'fields' is vague and poorly explained.
    - Endpoints frequently change without backward compatibility.
    - Permissions are complex and often undocumented (scope vs. endpoint mismatch).
    - Error messages are cryptic and don't explain the fix.
    """

    url = f"https://graph.facebook.com/v12.0/{page_id}/posts"

    # Parameters: docs often unclear on valid field names and combinations
    params = {
        "fields": "id,message,created_time,likes",  # 'likes' often fails silently or returns an object instead of a number
        "limit": 5,                                 # Docs didn't clearly state upper/lower limits
        "access_token": access_token                # Error messages didn't clarify expired vs. missing vs. wrong scope
    }

    try:
        response = requests.get(url, params=params)
        response.raise_for_status()
        return response.json()  # Often included undocumented fields or missing expected ones
    except requests.RequestException as e:
        print("API call failed with vague error:", e)
        if response is not None:
            print("Status:", response.status_code)
            print("Body:", response.text)
        return None
```

Basics

- MCP
  - Does away with the need to build M x 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 x N →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


MCP client
MCP server
Summarize my emails from Jack
1
Ask query
5
Send response
2
Get tools
4
Call function for tool and get response
3
Send query and pick tool with params
LLM
API
API
API

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 that rigid

Questions?