

# 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.
    """


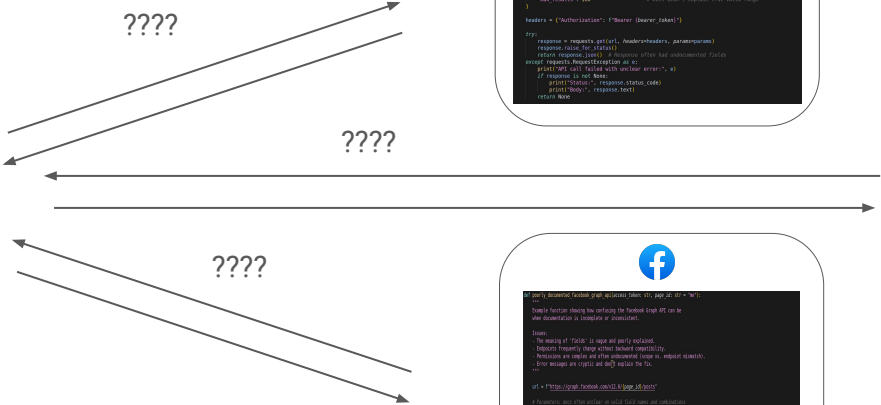
    url = "https://api.twitter.com/2/tweets/search/recent"

    # 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



```
#!/usr/bin/perl
# poorly documented twitter api (twitter.com/docs)
# Example function showing how confusing Twitter API v2 fails when it was poorly documented.

use strict;
use warnings;
use LWP::UserAgent;

my $ua = LWP::UserAgent->new;

my $url = "https://api.twitter.com/2/tweets/search/request";

my $headers = {
    "Authorization" => "Bearer (bearer token)"
};

my $response = $ua->get($url, $headers, {
    "Accept" => "application/json",
    "Content-Type" => "application/json",
    "User-Agent" => "perl"
});

my $body = $response->content;
my $status = $response->status_line;

print "Status: $status\n";
print "Body: $body\n";
```



```
#!/usr/bin/perl
# poorly documented, generic API (example.com/api)
# Example function showing how confusing a poorly documented API can be.

use strict;
use warnings;
use LWP::UserAgent;

my $ua = LWP::UserAgent->new;

my $url = "http://example.com/api";

my $headers = {
    "Content-Type" => "application/json",
    "Accept" => "application/json"
};

my $response = $ua->get($url, $headers, {
    "User-Agent" => "perl"
});

my $body = $response->content;
my $status = $response->status_line;

print "Status: $status\n";
print "Body: $body\n";
```



```
#!/usr/bin/perl
# poorly documented facebook api (facebook.com/developers)
# Example function showing how confusing the facebook api can be
# due to its complexity and lack of documentation.

use strict;
use warnings;
use LWP::UserAgent;

my $ua = LWP::UserAgent->new;

my $url = "https://graph.facebook.com/v2.8/me";


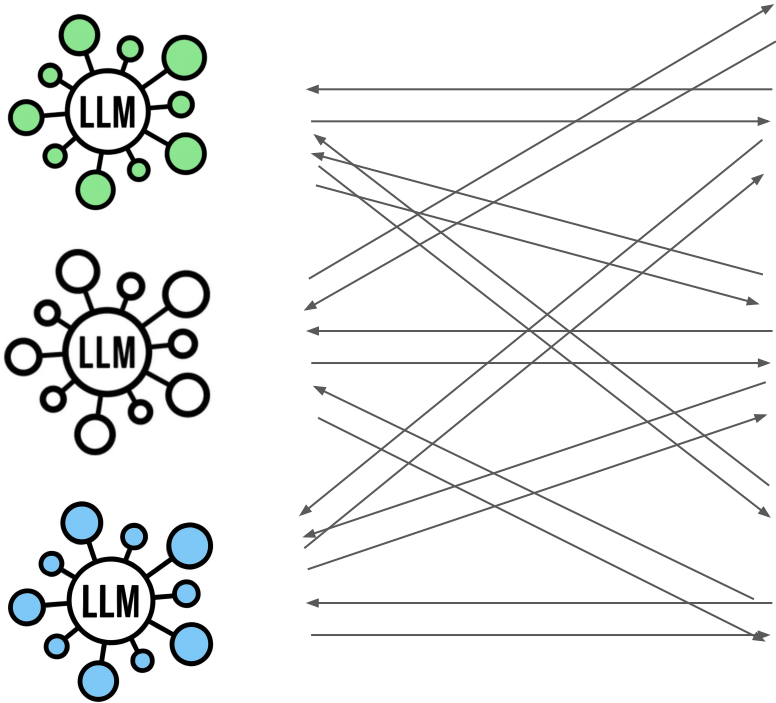
my $headers = {
    "Authorization" => "OAuth (access token)",
    "Content-Type" => "application/json"
};

my $response = $ua->get($url, $headers, {
    "User-Agent" => "perl"
});

my $body = $response->content;
my $status = $response->status_line;

print "Status: $status\n";
print "Body: $body\n";
```

# Now many LLM apps



```
def generate_tweet(topic: str, sentiment: str) -> str:
    """Generate a tweet about a given topic with a specific sentiment.

    Args:
        topic: The topic to generate a tweet about.
        sentiment: The sentiment to generate a tweet with (e.g., 'positive', 'negative').

    Returns:
        A tweet string.
    """
    # Generate a tweet about the given topic with the given sentiment
    prompt = f"Generate a tweet about {topic} with a {sentiment} sentiment. The tweet should be short and snappy, and include a relevant hashtag."
    response = openai.Completion.create(
        engine="text-davinci-001",
        prompt=prompt,
        max_tokens=150,
        temperature=0.7,
    )
    return response.choices[0].text
```



```
def generate_reply(tweet: str, sentiment: str) -> str:
    """Generate a reply to a given tweet with a specific sentiment.

    Args:
        tweet: The tweet to generate a reply to.
        sentiment: The sentiment to generate a reply with (e.g., 'positive', 'negative').

    Returns:
        A reply string.
    """
    # Generate a reply to the given tweet with the given sentiment
    prompt = f"Generate a reply to the tweet '{tweet}' with a {sentiment} sentiment. The reply should be short and snappy, and include a relevant hashtag."
    response = openai.Completion.create(
        engine="text-davinci-001",
        prompt=prompt,
        max_tokens=150,
        temperature=0.7,
    )
    return response.choices[0].text
```



```
def generate_post(topic: str, sentiment: str) -> str:
    """Generate a post about a given topic with a specific sentiment.

    Args:
        topic: The topic to generate a post about.
        sentiment: The sentiment to generate a post with (e.g., 'positive', 'negative').

    Returns:
        A post string.
    """
    # Generate a post about the given topic with the given sentiment
    prompt = f"Generate a post about {topic} with a {sentiment} sentiment. The post should be short and snappy, and include a relevant hashtag."
    response = openai.Completion.create(
        engine="text-davinci-001",
        prompt=prompt,
        max_tokens=150,
        temperature=0.7,
    )
    return response.choices[0].text
```

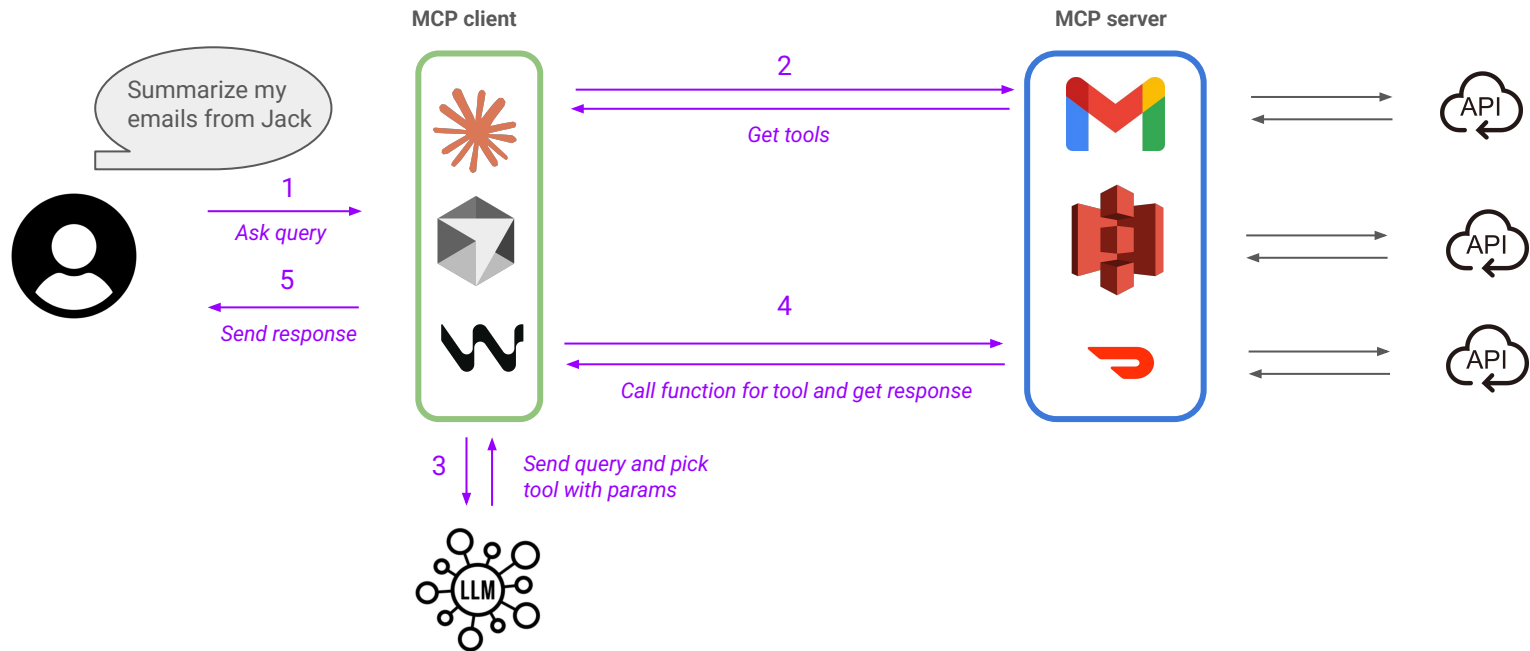
# Basics

- 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?