Tool Use for Language Model Agents

How models turn token predictions into actions

Carnegie Mellon University
Language Technologies Institute

Graham Neubig
Language Technologies Institute

Basic idea

Tool definition

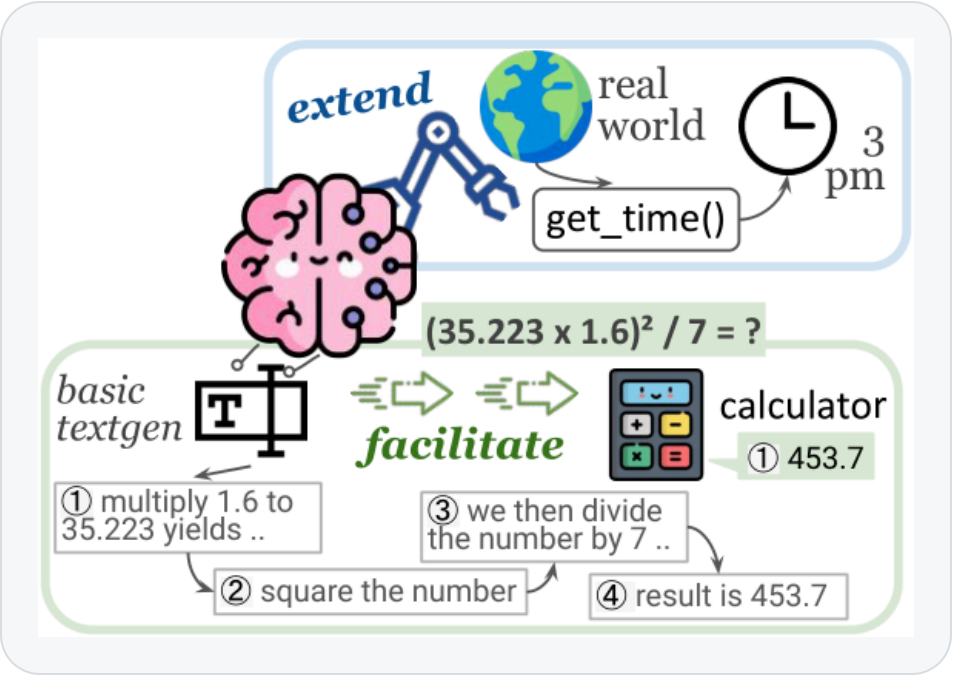
> **A tool is an interface through which a language model can invoke an external computer program.**

Search	Calculator	Browser	Python	API

The model proposes; external software decides whether and how to execute.

What Are Tools Anyway? · Wang et al., COLM 2024 ↗

Benefits of tools



Extend

Access information or actions unavailable in the model's parameters.

Facilitate

Delegate operations external programs perform more reliably or efficiently.

What Are Tools Anyway? · Wang et al., COLM 2024 ↗

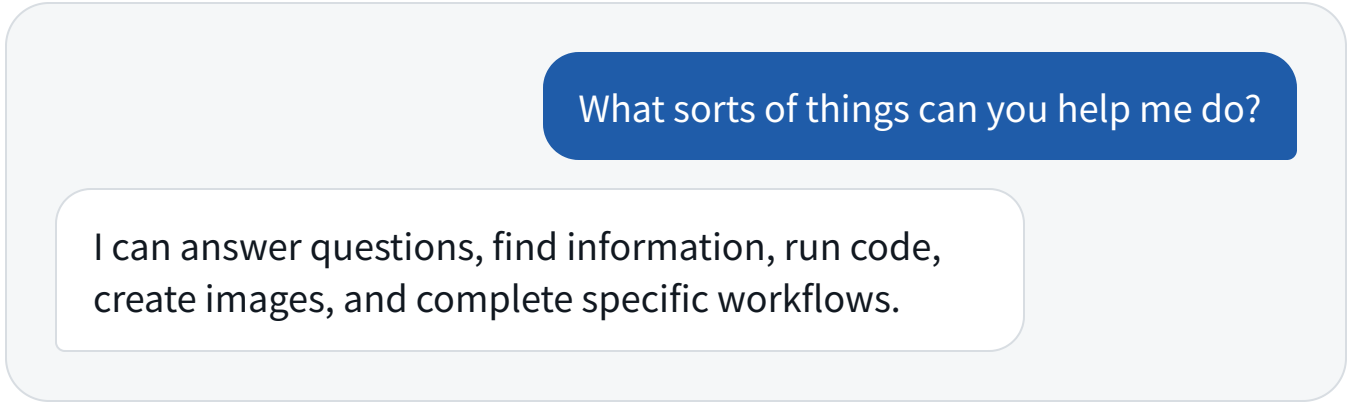
Tool-use scenarios

Need	Example tool	Benefit
Current information	Search	Fresh evidence
Exact computation	Calculator or Python	Reliable execution
Private state	Account API	Authorized application data
External action	Browser or API	Changes the environment

Call tools only when their benefit outweighs latency, cost, failure, and risk.

Examples of tools

Building ChatGPT



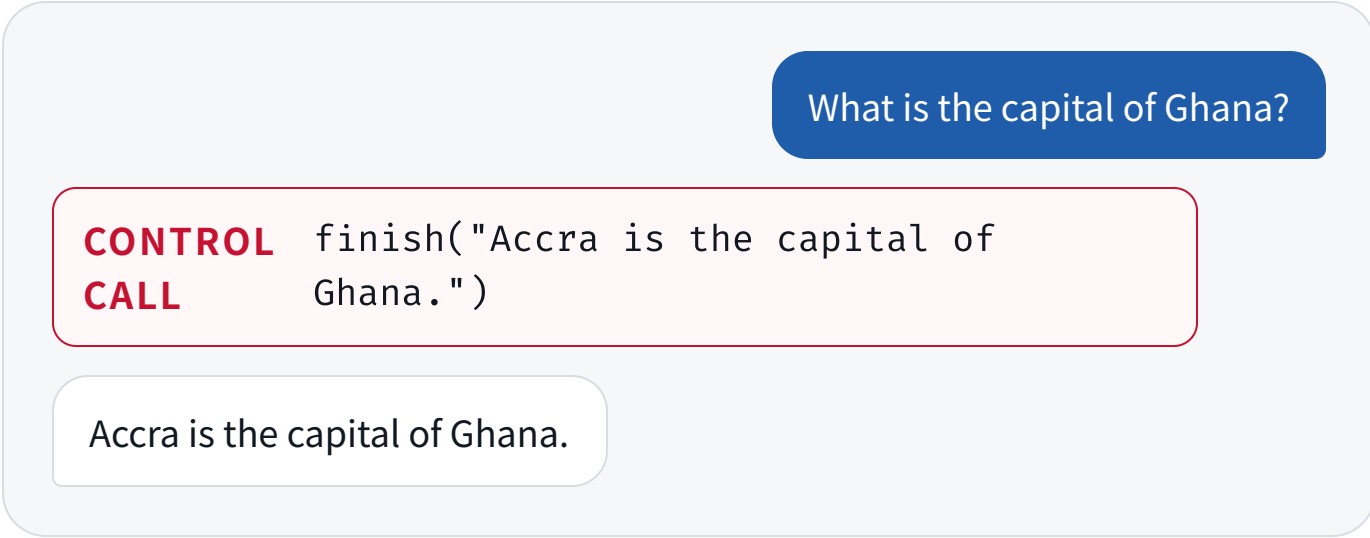
What calls are necessary?

Start from the abilities we want, then define a narrow interface for each kind of interaction.

Textual responses

`finish(text)`

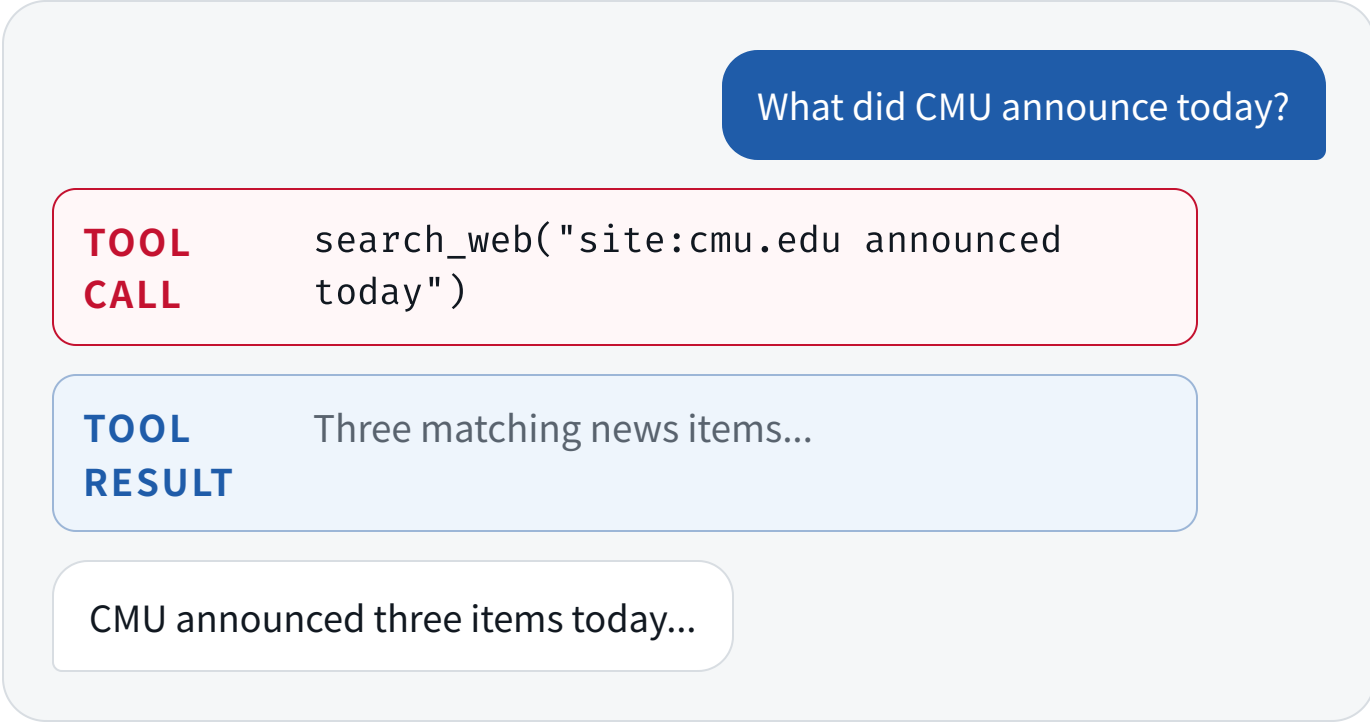
Return the final text and stop the agent loop. This is a harness control action, not an external program.



Information retrieval

`search_web(query)`

Retrieve information from the web.



Code execution

```
execute_code(code)
```

Run a program in an isolated environment and return its output.

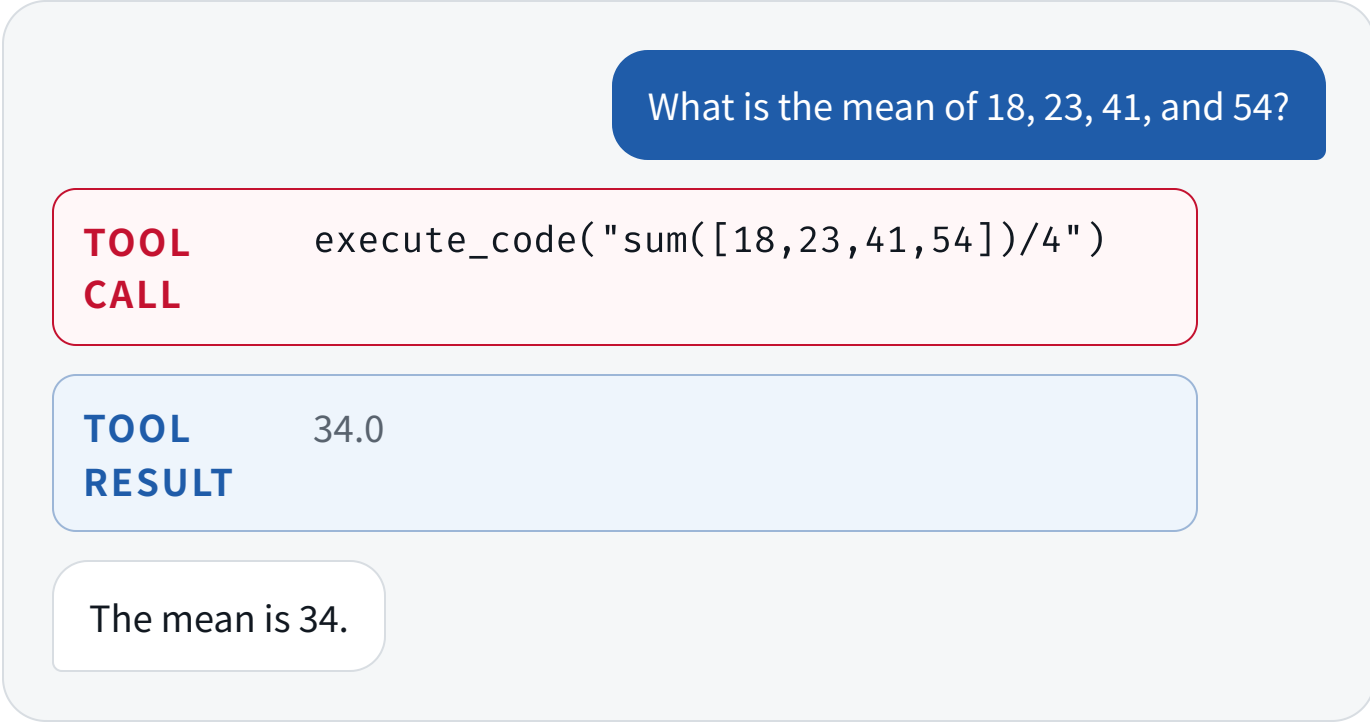


Image generation

`generate_image(query)`

Generate an image from a textual description.

Create a watercolor robot studying in a library.

TOOL CALL `generate_image("watercolor robot studying in a library")`

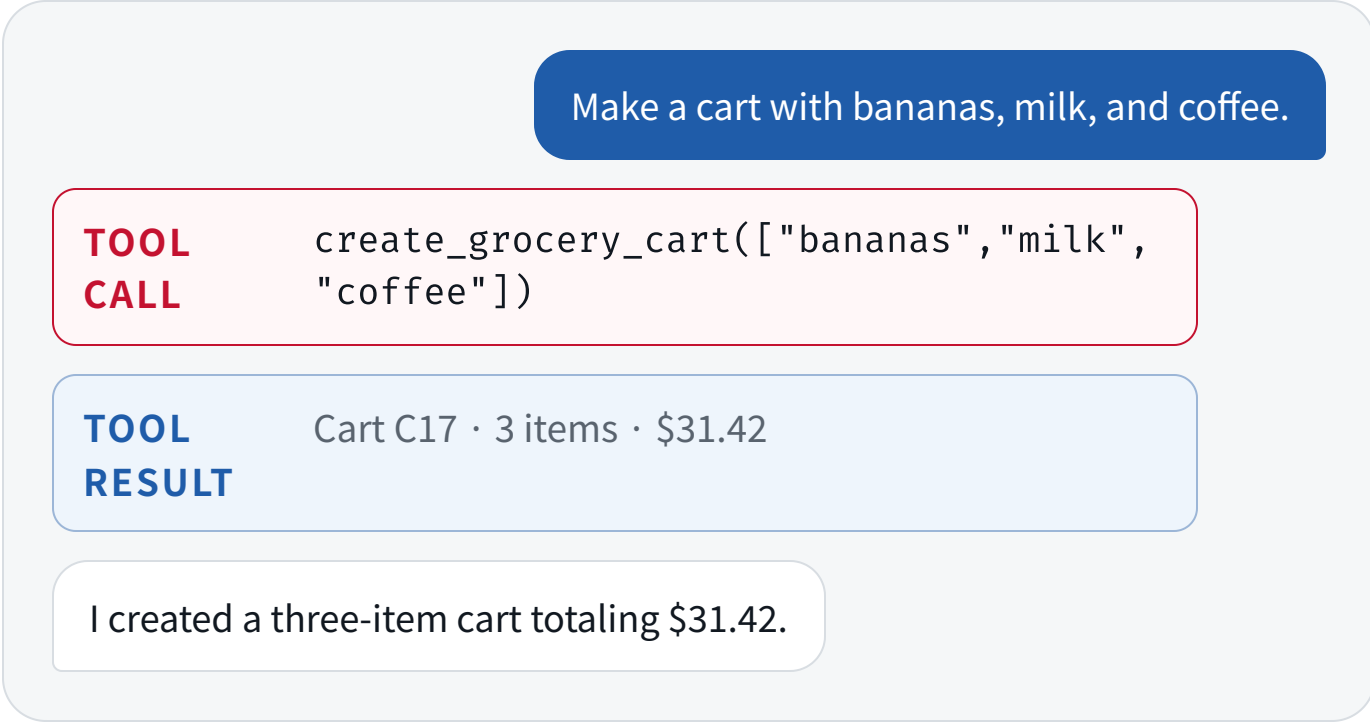
TOOL RESULT generated_image.png

Here is the generated image.

Custom functions

`create_grocery_cart(items)`

Expose an application-specific capability through one typed call.



Programmatic tool calling

Code as a meta-tool

build-in functions → tools

Q: The bakers baked 200 loaves of bread ... How many loaves of bread did they have left?

LM: The bakers started with 200 loaves. They sold 93 in the morning …The answer is 62.

```
loaves_baked, loaves_returned = 200, 6
sold_morning, sold_afternoon = 93, 39
answer = loaves_baked - loaves_sold_morning
- loaves_sold_afternoon + loaves_returned
```

CodeLM

external libraries → tools

Q: The table shows how many... What is the max number of vacation days across years?

Year	2013	2014	2015
Vacation days	23	18	11

```
import pandas as pd
df = pd.DataFrame({"Year": [2013, 2014, 2015],
"Vacation days": [23, 18, 11]})
max_days = df["Vacation days"].max()
```

CodeLM

utility functions → tools

Q: Who is wearing the dress?

```
dress_box = locate_objects(image, "dress")
dress_region = crop_region(image, dress_box)
answer = visual_qa(dress_region,
question="Who is wearing the dress?")
```

CodeLM

Code composes control flow, libraries, and multiple operations in one action.

What Are Tools Anyway? · Wang et al., COLM 2024 ↗

Programmatic composition

Instruction: Determine the most cost-effective country to purchase the smartphone model "CodeAct 1". The countries to consider are the USA, Japan, Germany, and India.

Available APIs
[1] lookup_rates(country: str) -> (float, float)
[2] convert_and_tax(price: float, exchange_rate: float, tax_rate: float) -> float
[3] estimate_final_price(converted_price: float, shipping_cost: float) -> float
[4] lookup_phone_price(model: str, country: str) -> float
[5] estimate_shipping_cost(destination_country: str) -> float

LLM Agent using [Text/JSON] as Action

Think I should calculate the phone price in USD for each country, then find the most cost-effective country.

Action
```
Text: lookup_rates, Germany
JSON: {"tool": "lookup_rates", "country": "Germany"}
```

Environment 1.1, 0.19

Action
```
Text: lookup_phone_price, CodeAct 1, Germany
JSON: {"tool": "lookup_phone_price", "model": "CodeAct 1", "country": "Germany"}
```

Environment 700

Action
```
Text: convert_and_tax, 700, 1.1, 0.19
JSON: {"tool": "convert_and_tax", "price": 700, "exchange_rate": 1.1, "tax_rate": 0.19}
```

Environment 916.3

[... interactions omitted (look up shipping cost and calculate final price) ...]

Action
```
Text: lookup_rates, Japan
JSON: {"tool": "lookup_rates", "country": "Japan"}
```

[... interactions omitted (calculate final price for all other countries)...]

Response The most cost-effective country to purchase the smartphone model is Japan with price 904.00 in USD.

> **Fewer Actions Required!**

CodeAct: LLM Agent using [Code] as Action

Think I should calculate the phone price in USD for each country, then find the most cost-effective country.

Action

```
countries = ['USA', 'Japan', 'Germany', 'India']
final_prices = {}

for country in countries:
    exchange_rate, tax_rate = lookup_rates(country)
    local_price = lookup_phone_price("xAct 1", country)
    converted_price = convert_and_tax(
      local_price, exchange_rate, tax_rate
    )
    shipping_cost = estimate_shipping_cost(country)
    final_price = estimate_final_price(converted_price, shipping_cost)
    final_prices[country] = final_price

most_cost_effective_country = min(final_prices, key=final_prices.get)
most_cost_effective_price = final_prices[most_cost_effective_country]
print(most_cost_effective_country, most_cost_effective_price)
```

Control & Data Flow of Code Simplifies Complex Operations

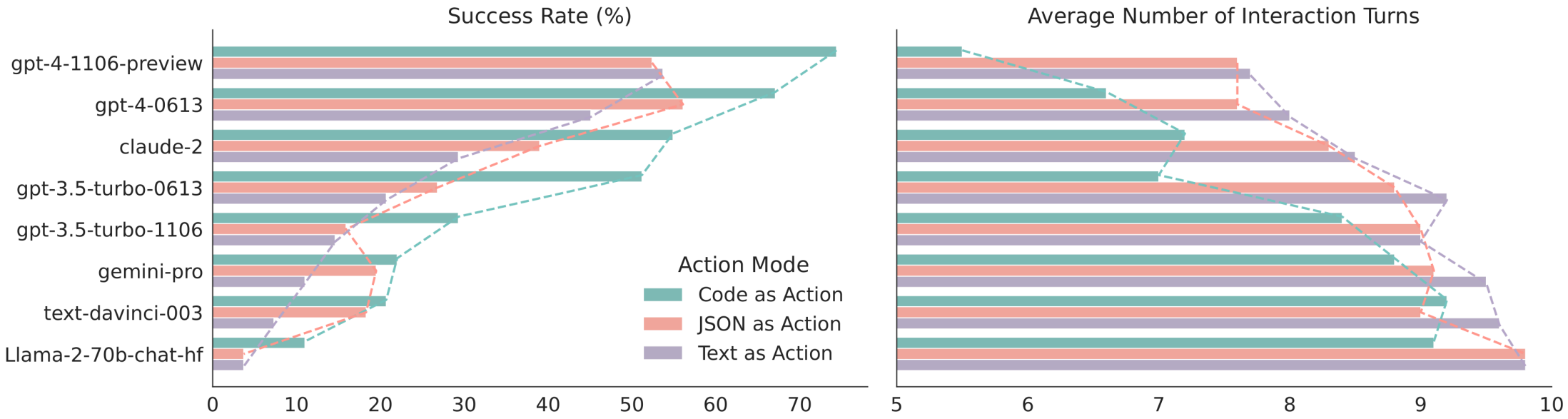
Re-use `min` Function from Existing Software Infrastructures (Python library)

Environment 1.1, 0.19

Response The most cost-effective country to purchase the smartphone model is Japan with price 904.00 in USD.

CodeAct · Wang et al., ICML 2024 ↗

CodeAct results



In the displayed subset, code had the best success rate for **8/8** models.

Code used the fewest turns for **7/8**; across all 17 models, both counts are **12/17**.

CodeAct · Wang et al., ICML 2024 ↗

Code is a high-power tool

Expressive

Loops · variables · libraries

Good for multi-step data manipulation and composition.

Harder to constrain

Broad action space

Validation is less precise than for a narrow typed function.

Higher impact

Files · network · processes

Use sandboxing, resource limits, permissions, and audit logs.

Mechanics of providing tools

Tool calls as tokens

Text continuation

```
The weather is sunny.
```

Tool continuation

```
<tool_call>
get_weather( ... )
```

The model requests an action. The harness executes it.

Quick review: messages become model input

API messages

system	Be concise.
user	Weather in Pittsburgh?
assistant	I need current data.

chat template

→

Qwen input sequence

```
<|im_start|>system
Be concise.<|im_end|>
<|im_start|>user
Weather in Pittsburgh?<|im_end|>
<|im_start|>assistant
I need current data.<|im_end|>
```

Roles structure the API; the template serializes one model-input sequence.

Tools add structure to the request

Tool definition

```
{
  "type": "function",
  "function": {
    "name": "get_weather",
    "parameters": {
      "type": "object",
      "properties": {"city": {"type": "string"}},
      "required": ["city"]
    }
  }
}
```

API request

```
response =
client.chat.completions.create(
    model=model,
    messages=messages,
    tools=[weather_tool],
)
```

The harness sends messages and tool schemas; the template exposes both to the model.

Definition in, tool call out

Prompt contains

```
<tools>
{"name":"get_weather", ...}
</tools>
```

model →

Model emits

```
<tool_call>
{"name":"get_weather",
 "arguments":{"city":"Pittsburgh"}}
</tool_call>
```

The model emits the call content; the API response attaches a unique call ID.

Same tool call, different model protocols

Qwen 3.8

Function + parameter tags

```
<|im_start|>assistant
<tool_call>
<function=get_weather>
<parameter=city>
Pittsburgh
</parameter>
</function>
</tool_call><|im_end|>
```

Mistral Small 4

Control token + call ID

```
[TOOL_CALLS] [{"name":
"get_weather",
"arguments": {"city":
"Pittsburgh"}, "id":
"abc123xyz"}]</s>
```

DeepSeek V3.2

DSML invocation block

```
<｜DSML｜function_calls>
<｜DSML｜invoke
name="get_weather">
<｜DSML｜parameter
name="city"
string="true">Pittsburgh<
/｜DSML｜parameter>
</｜DSML｜invoke>
</｜DSML｜function_calls>
```

One schema; model-specific serialization and parsing.

How OpenHands dispatches a tool call

1 Register at initialization

Tool spec
`BashTool`

→

Resolve + configure
`resolve_tool(spec, state)`

→

Name → definition
`tools_map[name]`
ToolDefinition: schema · Action · executor

2 Dispatch each model call

Parse
`{ name, args }`

→

Lookup
`tools_map[name]`

→

Validate
`action_from_arguments(args)`

→

Execute
`executor(action)`

→

Return
`ObservationEvent(call_id)`

Results return by call ID

Assistant tool call

```
{
  "id": "call_123",
  "name": "get_weather",
  "arguments": {"city":
"Pittsburgh"}
}
```

↔

Tool-result message

```
{
  "role": "tool",
  "tool_call_id": "call_123",
  "content": "72°F · sunny"
}
```

`tool_call_id` matches results to calls, including parallel calls.

Constrained decoding

Malformed tool calls

Model output

```
{"name":"get_weather",
 "arguments":{"city":"Pittsburgh"}
```

Missing `}`

The harness cannot parse it.

Tool-call constraints

Syntax	Shape	Types	Values
valid JSON	expected fields	city is a string	units is C or F

Constraints ensure form—not truth, permissions, tool choice, or task success.

JSON Schema

- Vocabulary for describing JSON data
- Types, required fields, allowed values
- Machine-readable validation rules

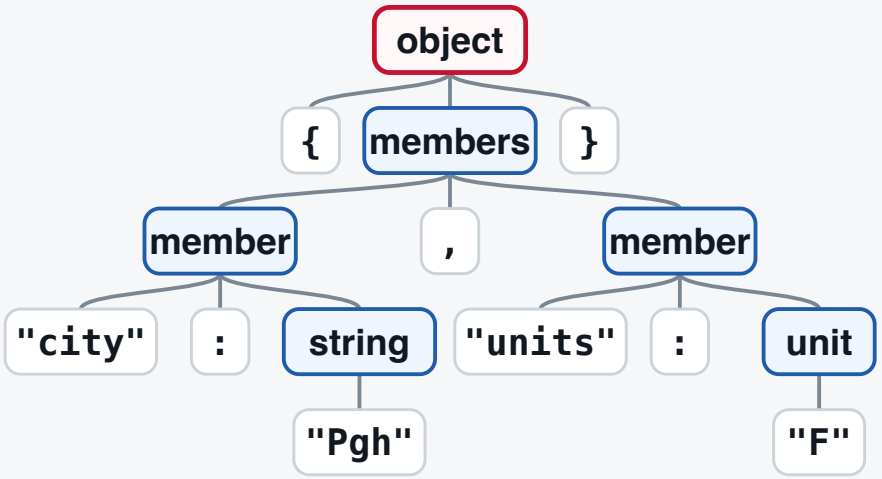
```
{
  "type": "object",
  "properties": {
    "city": {"type": "string"},
    "units": {"enum": ["C", "F"]}
  },
  "required": ["city"],
  "additionalProperties": false
}
```

JSON Schema Core · Draft 2020-12 ↗

Schema-valid vs. schema-invalid

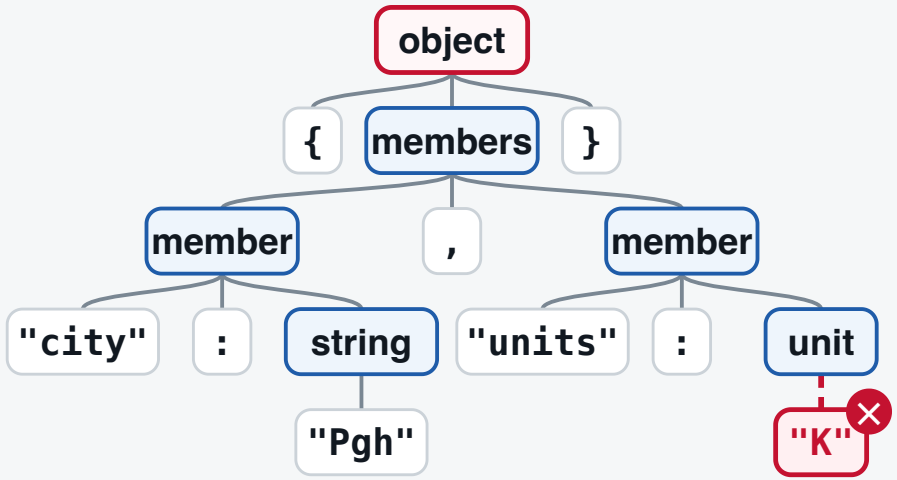
Schema-valid

```
{"city":"Pgh","units":"F"}
```



Schema-invalid

```
{"city":"Pgh","units":"K"}
```



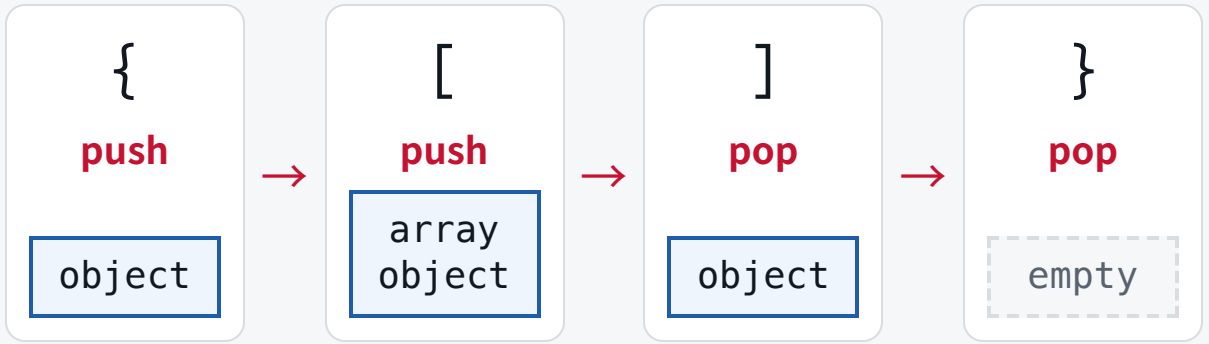
Both are JSON; only `"F"` derives from `unit → "C" | "F"`.

A pushdown automaton adds a stack

Finite control

State records the local position in a grammar rule.

Stack memory



The stack remembers arbitrarily deep nesting.

State tracks the current rule; the stack remembers where nested rules return.

Grammar checking happens below tokenization

Compile once

```
unit → "C" |
       "F"
```

→ **byte-level PDA**

Check every candidate token

Token `t₁: "F"}` → **Decoded bytes** `22 46 22 7D` → **Run all bytes** `advance(bytes)` → **Keep or mask** `allowed ✓`

A token may cross several grammar terminals—or end partway through one—so token boundaries cannot be treated as grammar boundaries.

XGrammar · Dong et al., MLSys 2025 ↗

Example: cached token-mask generation

1 · Parse the prefix

Advance persistent PDA stacks over the bytes already emitted.

2 · Fetch cached decisions

The stack-top node indexes precomputed validity for context-independent tokens.

3 · Check the minority

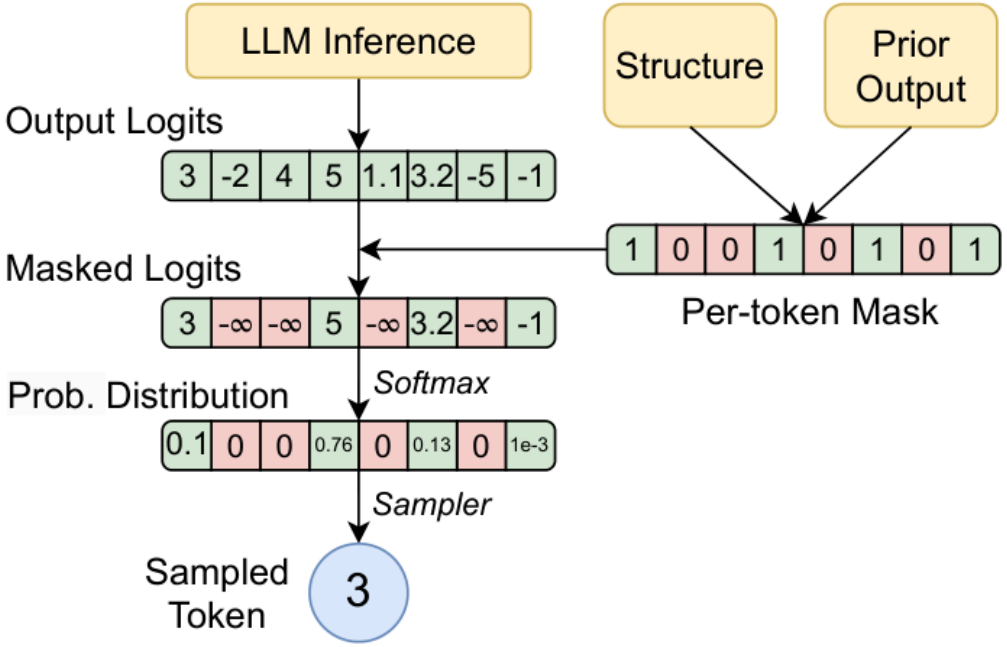
Run context-dependent candidate tokens against the full stack; union results across matching stacks.

4 · Mask and continue

Combine the decisions, mask the logits, sample a token, and update the PDA stacks.

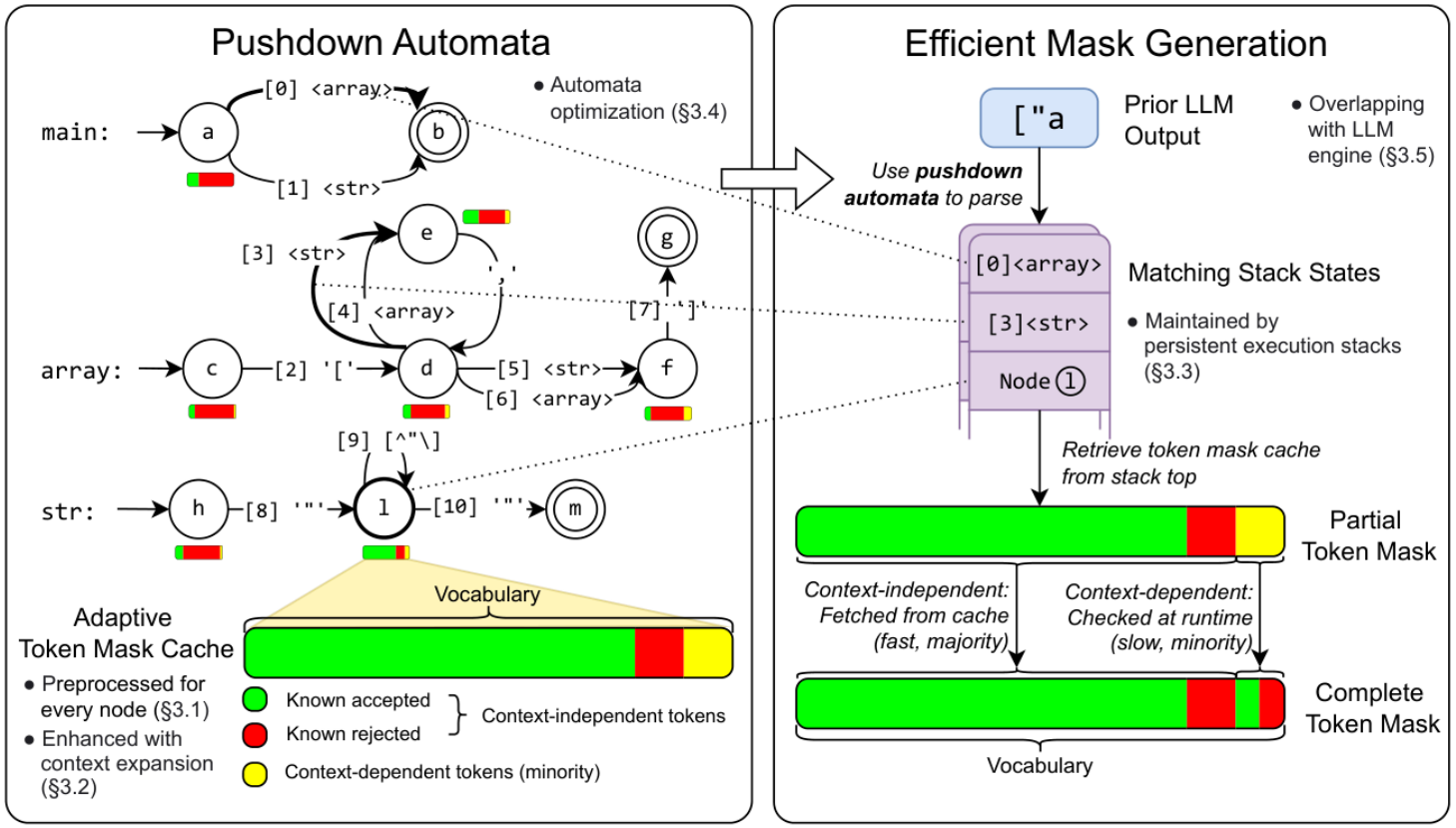
XGrammar · Dong et al., MLSys 2025 ↗

Token masking



XGrammar · Dong et al., MLSys 2025 ↗

Case study: where the speedup comes from



Caching handles most tokens; persistent stacks support branching; CPU work overlaps GPU inference.

XGrammar · Dong et al., MLSys 2025 ↗

REST API Calling

Libraries expose in-process APIs

```python
# weather_lib.py
from typing import Literal
from pydantic import BaseModel

class Weather(BaseModel):
    summary: str
    temperature: float

def get_weather(
    city: str,
    units: Literal["C", "F"] = "F",
) -> Weather:
    return provider.lookup(city, units)
```

Caller and function share one process

Python supplies the name, arguments, types, and return type.

Not remotely callable yet

A network client still needs an address, protocol, serialization, and authentication.

A library API is a typed programming interface—not a network service.

REST APIs expose functions over HTTP

```python
# api.py
from typing import Annotated, Literal
from fastapi import FastAPI, Security
from fastapi.security import APIKeyHeader
from weather_lib import Weather, get_weather

app = FastAPI(title="Weather API")
key_header = APIKeyHeader(name="X-API-Key")

@app.get("/weather", operation_id="get_weather")
def weather(
    city: str,
    key: Annotated[str, Security(key_header)],
    units: Literal["C", "F"] = "F",
) -> Weather:
    verify(key)
    return get_weather(city, units)
```

GET /weather

`city=Pittsburgh`

`units=F`

Header: `X-API-Key`

Response: `Weather` as JSON

FastAPI binds the library function to a REST endpoint and documents its API-key header.

FastAPI security and OpenAPI ↗

REST API servers publish OpenAPI descriptions

```
{
  "paths": {"/weather": {"get": {
    "operationId": "get_weather",
    "parameters": [
      {"name": "city", "in": "query", "required": true,
       "schema": {"type": "string"}},
      {"name": "units", "in": "query",
       "schema": {"type": "string", "enum": ["C", "F"], "default": "F"}}
    ],
    "security": [{"APIKeyHeader": []}]
  }}},
  "components": {"securitySchemes": {"APIKeyHeader": {
    "type": "apiKey", "in": "header", "name": "X-API-Key"
  }}}
}
```

OpenAPI describes the HTTP operation; embedded **JSON Schema** describes its values.

FastAPI · generated OpenAPI ↗

Coding agents often call REST APIs with curl

```
curl -sS -G https://weather.example/weather \
  --data-urlencode "city=Pittsburgh" \
  --data-urlencode "units=F" \
  -H "X-API-Key: $WEATHER_API_KEY"
```

1 · Read the API description

Select the operation and required parameters.

2 · Construct the command

The shell expands the environment variable; curl sends HTTP.

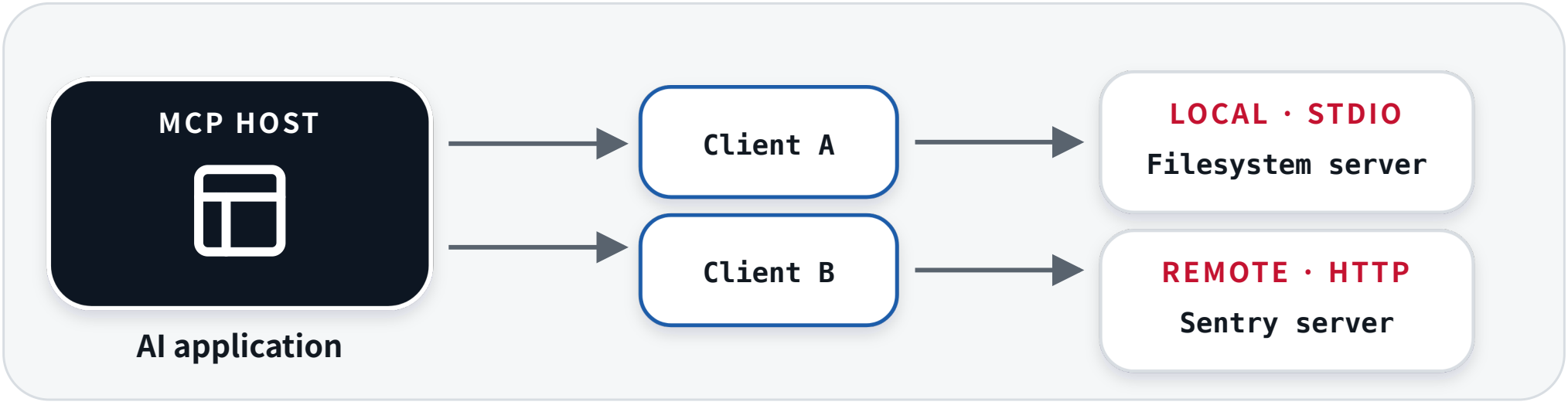
3 · Limit the credential

A coding agent with shell access may also read or misuse that secret.

Direct REST access is simple, but the agent's shell receives the API credential.

MCP

MCP connects a host to servers



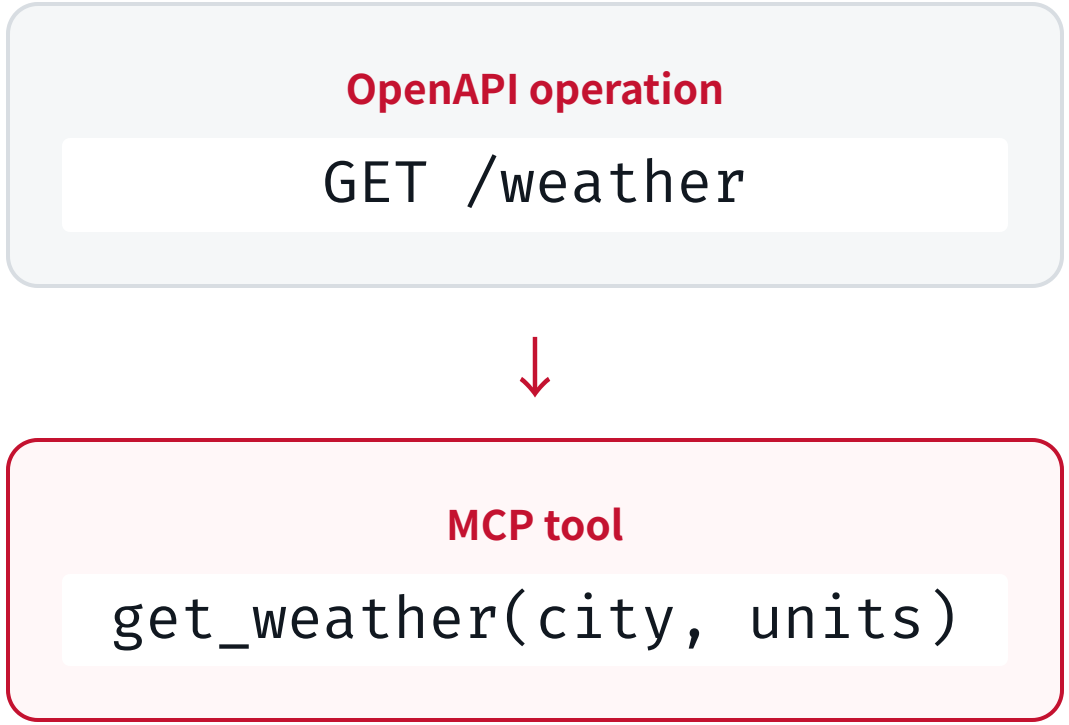
MCP is an alternative agent-facing interface—not a requirement for calling REST APIs.

FastMCP projects OpenAPI operations as MCP tools

```python
import httpx2
from fastmcp import FastMCP

spec = httpx2.get(
    "https://weather.example/openapi.json"
).json()

mcp = FastMCP.from_openapi(
    openapi_spec=spec,
    client=httpx2.AsyncClient(
        base_url="https://weather.example"
    ),
    name="Weather MCP",
)
```



By default, FastMCP converts each OpenAPI route into a discoverable MCP tool.

FastMCP · OpenAPI integration ↗

Start MCP with separate credentials

```python
import os, httpx2
from fastmcp import FastMCP
from fastmcp.server.auth.providers.jwt import StaticTokenVerifier

upstream = httpx2.AsyncClient(
    base_url="https://weather.example",
    headers={"X-API-Key": os.environ["UPSTREAM_API_KEY"]})
mcp_auth = StaticTokenVerifier(tokens={
    os.environ["MCP_API_KEY"]: {"client_id": "course-agent"}})

mcp = FastMCP.from_openapi(
    openapi_spec=spec, client=upstream,
    name="Weather MCP", auth=mcp_auth)
mcp.run(transport="http", port=8001)
```

`UPSTREAM_API_KEY` authorizes API calls; `MCP_API_KEY` protects the MCP endpoint.

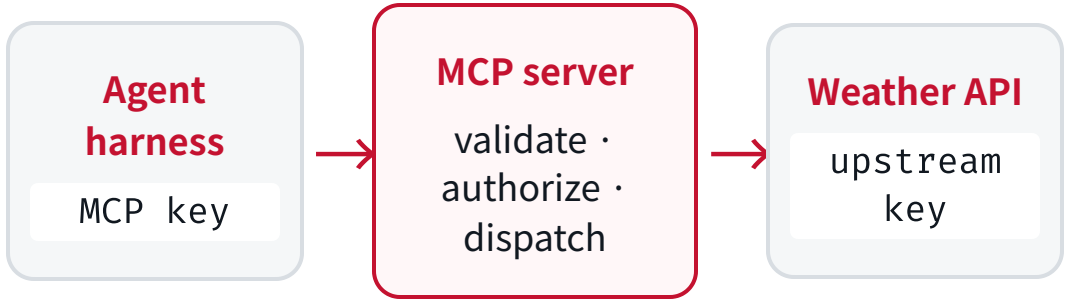
Static tokens are for teaching/development; production deployments should verify JWTs or use OAuth.

FastMCP · token verification ↗

The MCP client uses its own API key

```python
import os
from fastmcp import Client

async with Client(
    "https://mcp.example/mcp",
    auth=os.environ["MCP_API_KEY"],
) as client:
    tools = await client.list_tools()
    result = await client.call_tool(
        "get_weather",
        {"city": "Pittsburgh", "units": "F"},
    )
```



Credential brokering: the server accepts one credential and uses a different credential upstream. The model sees neither secret.

Separate credentials let the MCP server mediate access instead of forwarding its upstream secret.

FastMCP · bearer client authentication ↗

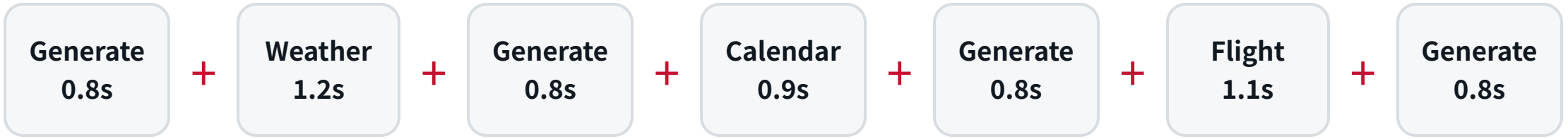
REST APIs and MCP: shared schemas, different contracts

	OpenAPI / HTTP API	MCP
Shared	Names, descriptions, and JSON Schema for structured inputs	
Discovery	Fetch an OpenAPI document	Call `tools/list` at runtime
Invocation	HTTP verb + path + parameters	`tools/call` over an MCP transport
Authentication	Client authenticates to the API	Client authenticates to MCP; upstream auth is separate
Scope	Describes HTTP operations	Also supports resources, prompts, and extensions

OpenAPI describes a web API; MCP standardizes how an AI host discovers and invokes capabilities.

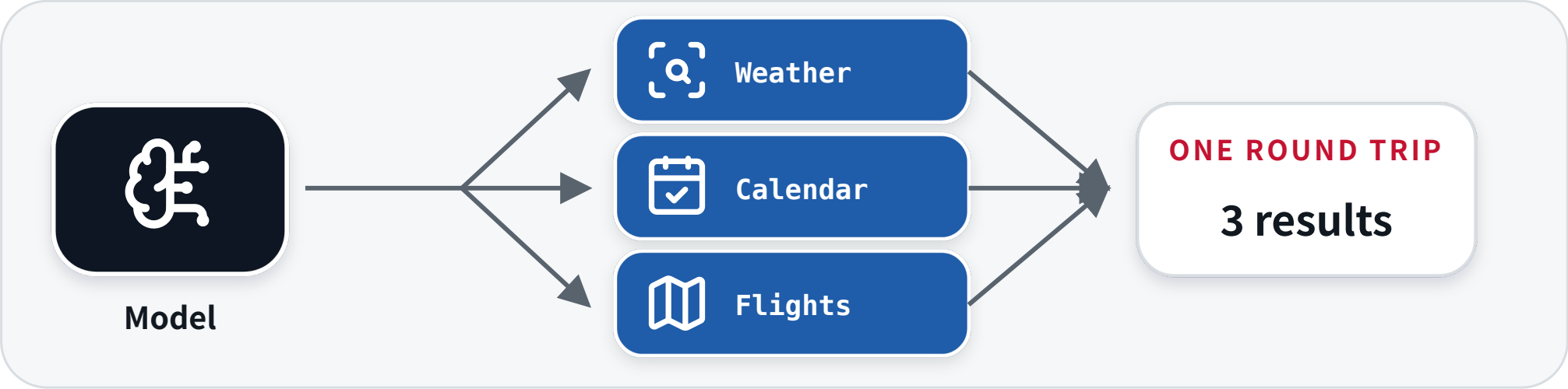
Orchestrating multiple calls

Serial tool-call cost



≈ 6.4 seconds

Parallel tool calls



Independent calls can share one generation and one wait.

Two generation phases plus the slowest tool: $2 \times 0.8 + \max(1.2, 0.9, 1.1) \approx$ **2.8 seconds**.

Parallelize only without dependencies

Safe to parallelize

- Weather for Pittsburgh
- Read today's calendar
- Look up flight status

↔

Must be sequenced

- Find customer ID
- Use ID to fetch orders
- Refund selected order

Parallelize only independent, safely concurrent calls.

Concurrent execution

```
async def execute(call):
    args = json.loads(call.arguments)
    value = await TOOLS[call.name](**args)
    return call.call_id, value

results = await asyncio.gather(
    *(execute(call) for call in function_calls),
    return_exceptions=True,
)
```

Preserve call IDs when results finish out of order.

Evaluating tool use

Berkeley Function Calling Leaderboard

Single turn

simple · multiple · parallel · multiple-parallel

Multi-turn

base · missing function · missing parameter · long context

Agentic

web search · memory

Robustness

hallucination measurement · format sensitivity

Berkeley Function-Calling Leaderboard V4 ↗

Evaluate the whole stack

Level	Question	Example metric	Execution?	Typical failure
Selection	Right tool?	Precision / recall	No	Missing or extra call
Arguments	Right values?	AST / schema match	No	Wrong field or value
Trajectory	Right order?	Sequence success	Usually	Bad dependency
Task	Goal achieved?	End-to-end success	Yes	Plausible wrong answer

Quality has operational dimensions

Efficiency

Latency · calls · tokens · cost

A correct agent can still be unusably slow or expensive.

Reliability

Timeouts · retries · partial failure

Report task success under realistic provider behavior.

Safety

Policy · permissions · side effects

Test adversarial outputs and consequential actions.

Operational tool failures

Model

wrong tool · bad arguments · ignores result

Harness

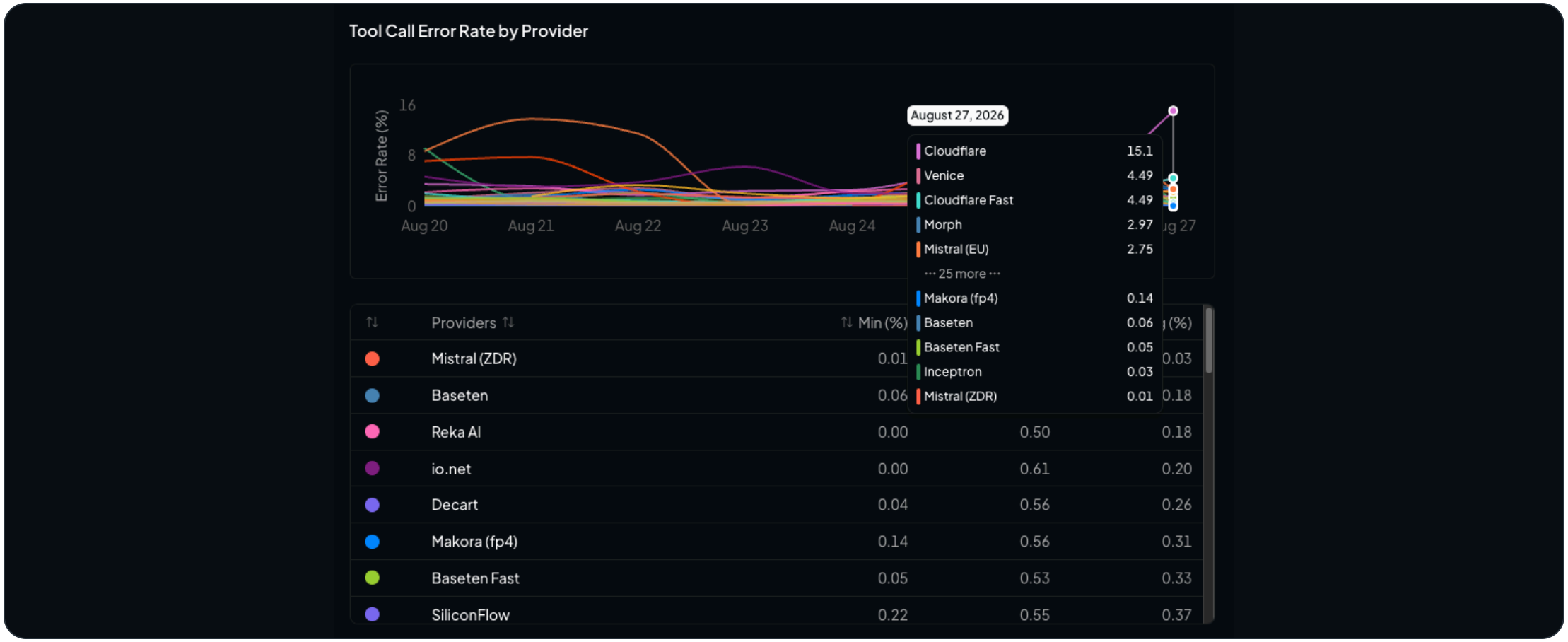
parse · auth · validation · ID mismatch

Provider or tool

timeout · rate limit · execution error

Production evaluation needs benchmark accuracy and operational failure rates.

Provider tool-call error rates



Providers	Min (%)		(%)
Mistral (ZDR)	0.01		0.03
Baseten	0.06		0.18
Reka AI	0.00	0.50	0.18
io.net	0.00	0.61	0.20
Decart	0.04	0.56	0.26
Makora (fp4)	0.14	0.56	0.31
Baseten Fast	0.05	0.53	0.33
SiliconFlow	0.22	0.55	0.37

OpenRouter provider analytics · Aug. 27, 2026 ↗

Takeaways

Tool use is a layered system

1. **Capabilities:** tools add retrieval, execution, generation, and application actions.
2. **Mechanics:** schemas enter the prompt; model protocols emit calls; harnesses validate, dispatch, and match results by ID.
3. **Constraints:** JSON Schema and grammar masks guarantee form—not semantic correctness.
4. **Interfaces:** direct REST calls and MCP are alternative ways to expose capabilities and manage credentials.
5. **Systems:** orchestrate dependencies and concurrency; evaluate selection, arguments, outcomes, cost, and operational failures.

Thank you, Questions?

Next Class: Agent Capabilities 2 — Context Management for Long-Context LLMs