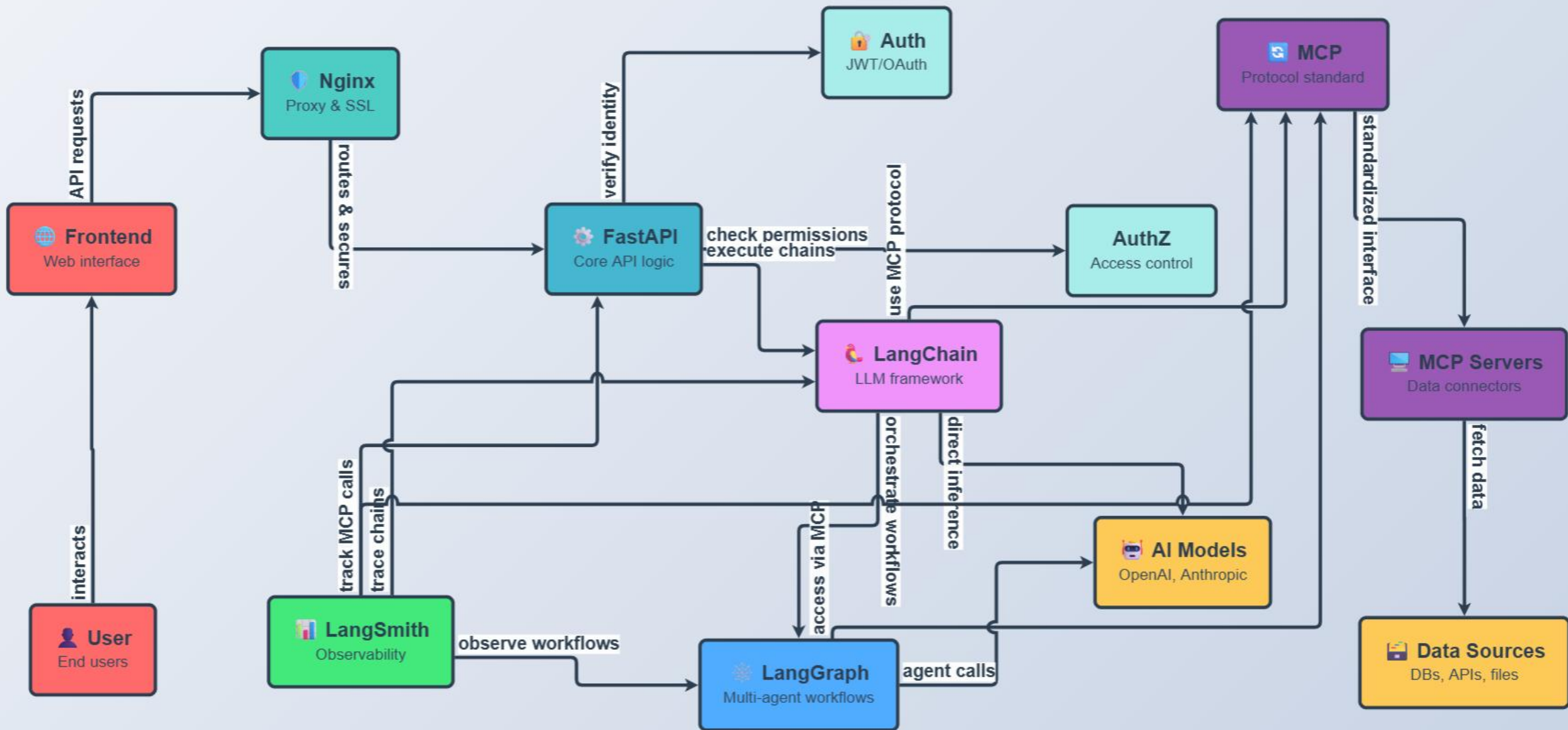
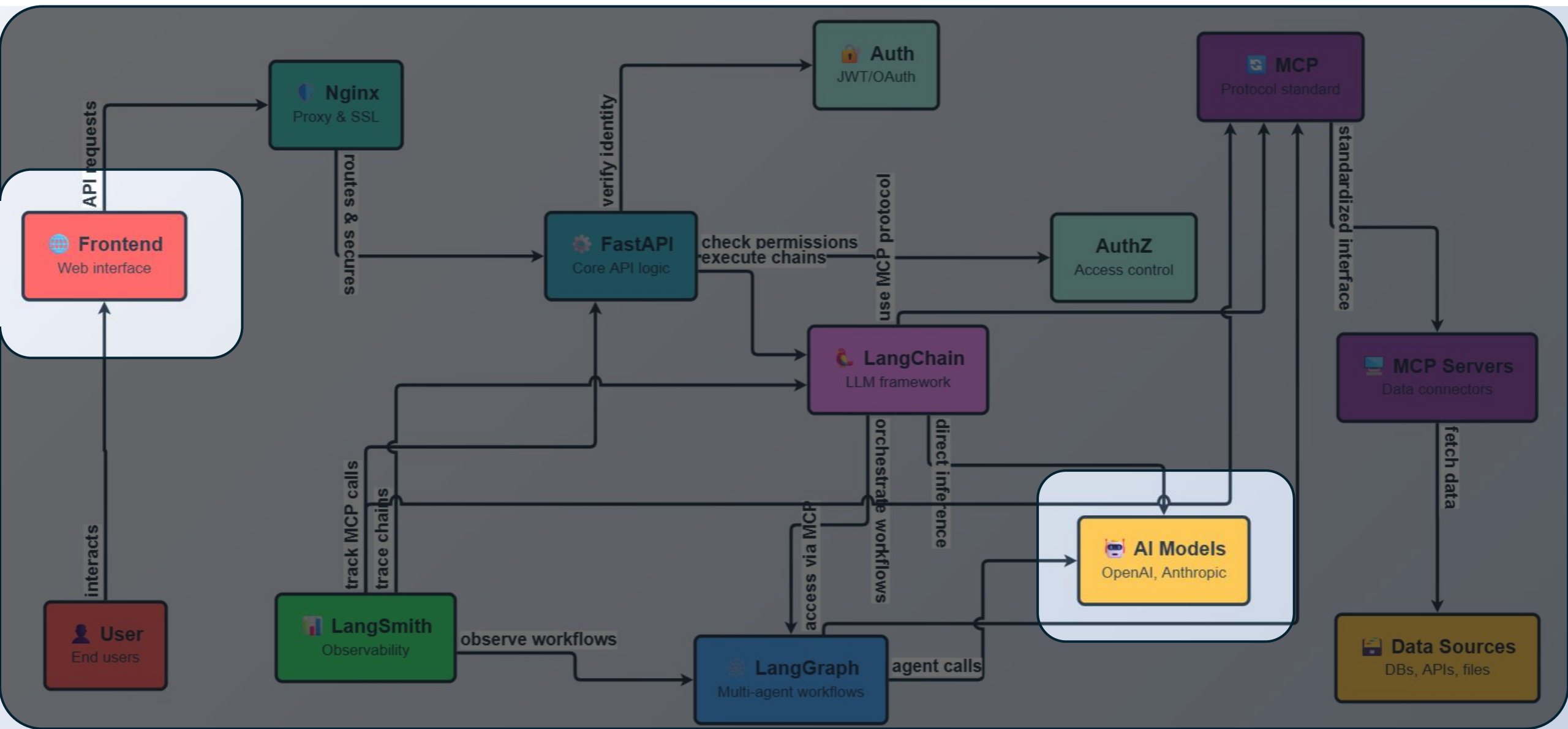


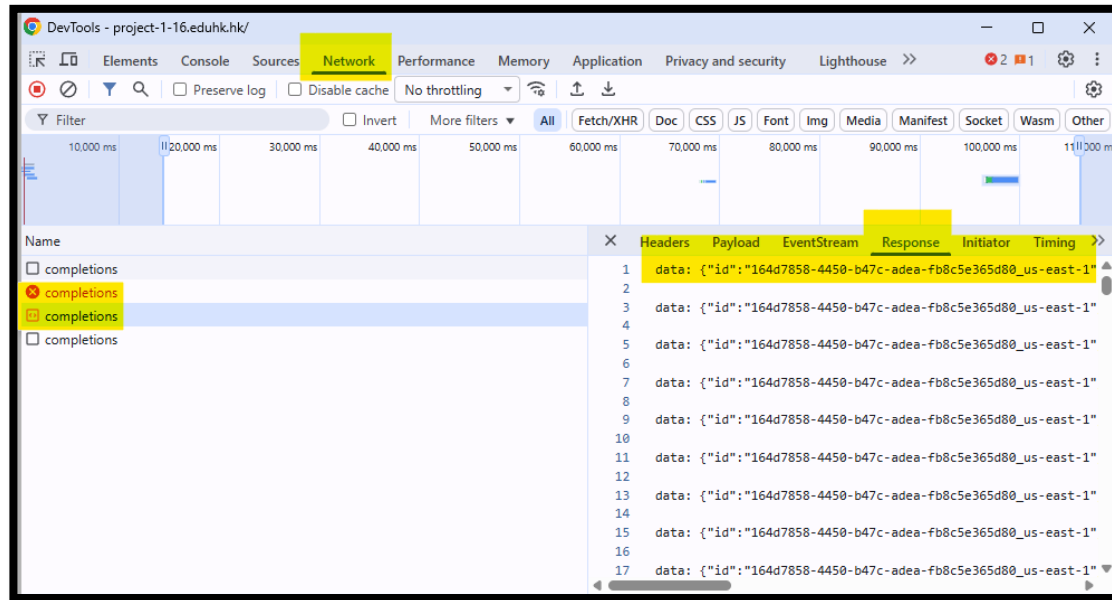
Review and AI Agent Design III: Evaluation





Chatbot UI

1. API endpoint is stateless
2. You need to sent full chat history to API endpoint
3. You can find the responses in the network tab



Why do you need to get the responses?

Name	Description	URL
Base44	AI platform for building full-stack apps from prompts with auto-deployment. Free starter plan.	https://base44.ai/
Bolt.new	Browser-based AI for generating and deploying full-stack apps. Free tier for personal projects.	https://bolt.new/
Cline	Local-first AI coding agent for VS Code with task planning. Free open-source plan.	https://cline.app/
Windsurf	AI-native IDE with agentic editing and multi-model support. Free tier for individuals.	https://windsurf.com/
Cursor	AI-first code editor for generation and refactoring. Free tier with limited AI usage.	https://cursor.sh/
GitHub Copilot	AI for real-time code suggestions and PR reviews in IDEs. Free for individuals and OSS.	https://github.com/features/copilot
Replit	An AI-powered platform for building professional web apps and websites.	https://replit.com/

Chatbot UI vibe coding keywords

1. Single page html (for fast prototyping)
2. React (with Vite: Vite is a modern frontend build tool)
 1. Vibe coding

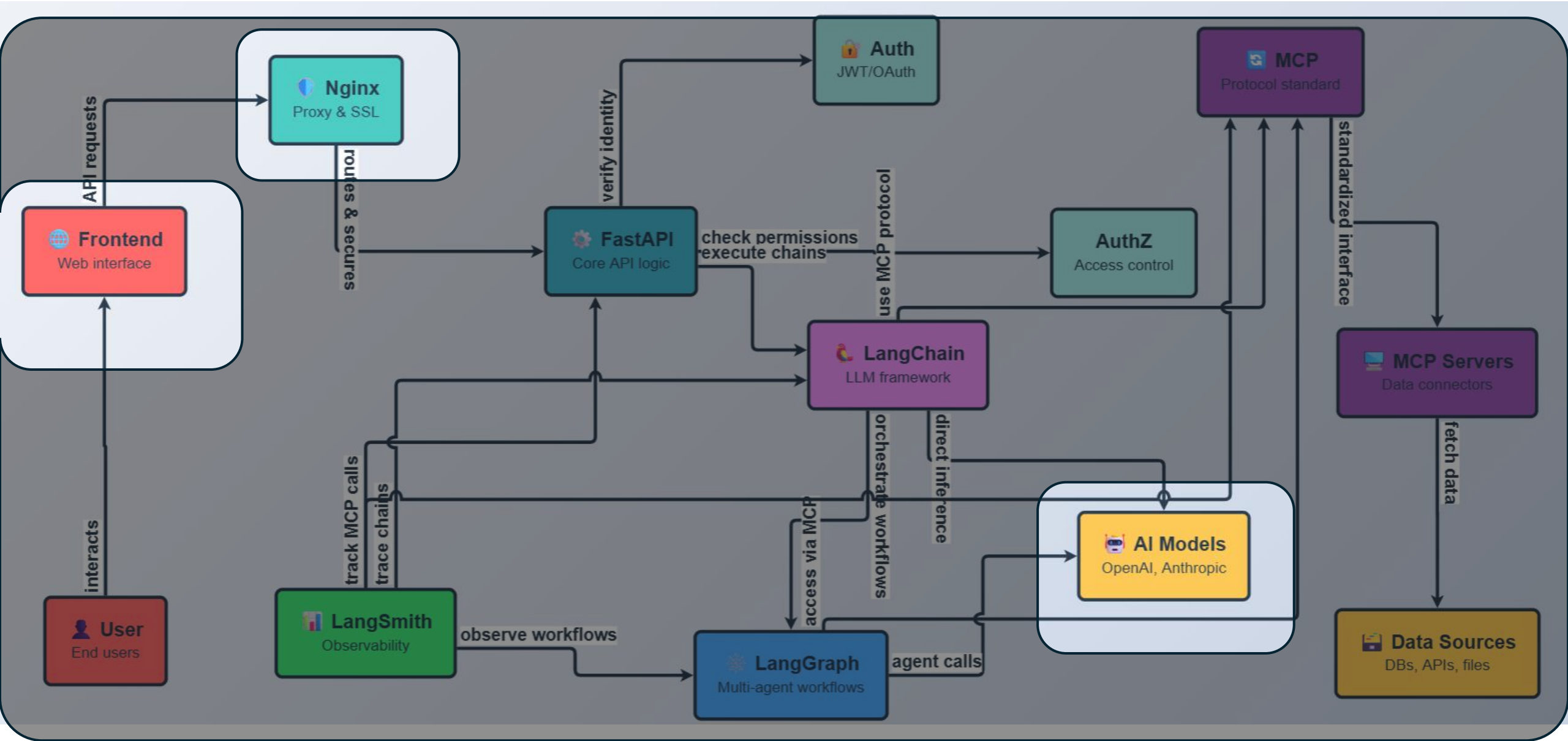
Why Vite?

1. Fast Development Experience
2. Strong Ecosystem and Community Backing
3. Simplicity and Extensibility
 1. ES Modules Facilitate UI Modularization

ES Modules Facilitate UI Modularization

1. Break down UI rendering designs into separate files
2. Frameworks like React, Vue, or vanilla JS

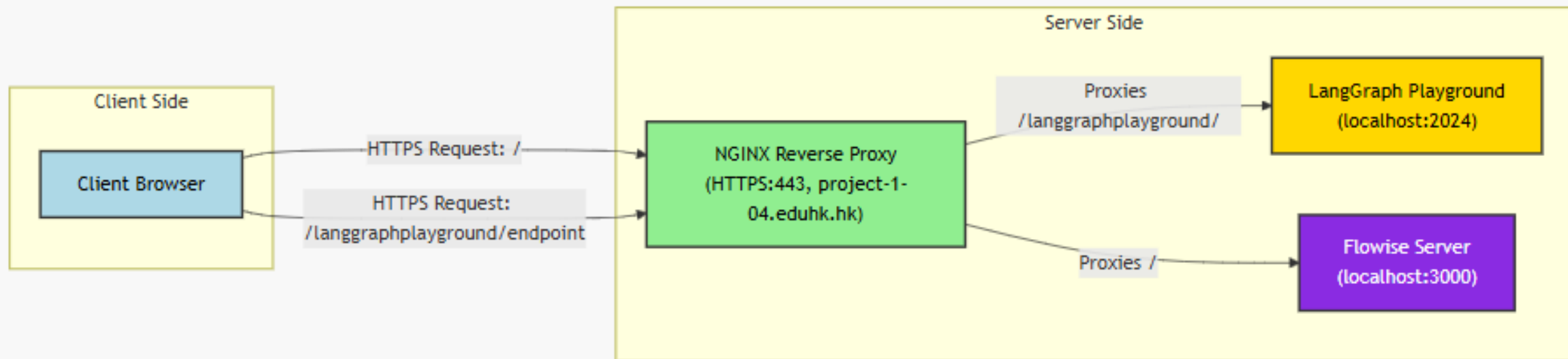
Performance Benefits: Modules enable code-splitting (lazy-loading parts of the UI) and removing unused code, which is crucial for fast-loading UIs.



Nginx

1. Serve static Front-End application
2. Reverse proxy to different web resource

Nginx



Reverse proxy to different web resource

1. Forces everyone to use secure connections (HTTPS)
2. Routes different types of requests to different applications running on the server
3. Ensures secure, encrypted communication

Nginx

```
server{  
    listen 80;  
    server_name project-1-04.eduhk.hk;  
    return 301 https://$host$request_uri;  
}
```

- **listen 80:** Port 80 is the standard "door" for regular HTTP traffic (non-secure). This tells Nginx to watch this door.
- **server_name project-1-04.eduhk.hk:** This says "I handle requests for project-1-04.eduhk.hk"

Nginx(return)

```
server{  
    listen 80;  
    server_name project-1-04.eduhk.hk;  
    return 301 https://$host$request_uri;  
}
```

- **return 301 https://\$host\$request_uri:**

What it does: This is a Nginx directive that immediately stops processing and sends a response back to the browser.

How it works:

When Nginx encounters return, **it doesn't look at any other configuration** below it

- It immediately constructs and sends the response
- No further processing happens for this request
- **Think of it like:** A receptionist who says "Sorry, you need to go to a different building" and gives you the new address, instead of processing your request at the current location.

Nginx(301)

```
server{  
    listen 80;  
    server_name project-1-04.eduhk.hk;  
    return 301 https://$host$request_uri;  
}
```

- **return 301 https://\$host\$request_uri:**

What it is: An HTTP status code meaning "Moved Permanently"

HTTP Status Codes - Quick Overview:

- **2xx** = Success (200 = OK)
- **3xx** = Redirection (301 = Moved Permanently, 302 = Moved Temporarily)
- **4xx** = Client Error (404 = Not Found)
- **5xx** = Server Error (500 = Internal Server Error)

Nginx(301)

```
server{  
    listen 80;  
    server_name example.com www.example.com;  
    return 301 https://$host$request_uri;  
}
```

- **return 301 https://\$host\$request_uri:**

For HTTPS redirects, 301 is used because:

- You always want HTTPS, forever
- You want browsers to remember this
- You want search engines to index the HTTPS version

Nginx(https://)

```
server{  
    listen 80;  
    server_name project-1-04.eduhk.hk;  
    return 301 https://$host$request_uri;  
}
```

- **return 301 https://\$host\$request_uri:**

What it is: The protocol/scheme for the new URL

- **This specifies:**
 - Use the HTTPS protocol (secure, encrypted)
 - Browser should connect on port 443 (HTTPS default port)
 - An encrypted TLS/SSL connection must be established
 - **The redirect changes only the protocol**, not the domain or path.

Nginx(\$host)

```
server{  
    listen 80;  
    server_name project-1-04.eduhk.hk;  
    return 301 https://$host$request_uri;  
}
```

- **What it is:** A Nginx variable containing the hostname from the request
- **How it works:** When a browser makes a request, it includes a "Host" header. Nginx captures this in the \$host variable.
- Does anyone know what a header is?

Nginx(\$host)

```
server {  
    listen 80;  
    server_name project-1-04.eduhk.hk;  
    return 301 https://$host$request_uri;  
}
```

Benefit: Preserves whatever domain the user typed

- www.project-1-04.eduhk.hk → https://www.project-1-04.eduhk.hk
- project-1-04.eduhk.hk → https://project-1-04.eduhk.hk

Nginx(\$request_url)

```
server {  
    listen 80;  
    server_name project-1-04.eduhk.hk;  
    return 301 https://$host$request_uri;  
}
```

What it is: A Nginx variable containing the complete original request path and query string

What it includes:

- The path (everything after the domain)
- Query parameters (the ? and everything after it)
- Fragment identifiers are NOT included (the # part - browsers don't send this to servers)

Nginx(location /langgraphplayground/)

- **location /langgraphplayground/**
- **What it matches:** Any URL starting with /langgraphplayground/
- **Examples:**
- ✓ <https://project-1-04.eduhk.hk/langgraphplayground/>
- ✓ <https://project-1-04.eduhk.hk/langgraphplayground/api/chat>
- ✓ <https://project-1-04.eduhk.hk/langgraphplayground/static/logo.png>
- ✗ <https://project-1-04.eduhk.hk/langGraph/> (case-sensitive)
- ✗ <https://project-1-04.eduhk.hk/other/path>

Nginx(location /langgraphplayground/)

The trailing slash matters:

```
location /langgraphplayground/ { # With trailing slash
    # Matches: /langgraphplayground/*, but NOT /langgraphplayground
}
```

```
location /langgraphplayground { # Without trailing slash
    # Matches: /langgraphplayground and /langgraphplayground/*
}
```

In your case, with the trailing slash:

```
/langgraphplayground    → Goes to location / (caught by second block)
/langgraphplayground/    → Goes to this block ✓
/langgraphplayground/chat → Goes to this block ✓
```

Nginx(SSE Requirements)

```
location /langgraphplayground/{  
    proxy_pass http://localhost:2024;  
    proxy_http_version 1.1;      # ✓ HTTP/1.1 (good for SSE)  
    proxy_buffering off;         # ✓ if you want streams immediately  
    proxy_cache off;            # ✓ No caching  
    proxy_read_timeout 300s;     # ✓ Long timeout for persistent  
connection  
    # ... other headers  
}
```

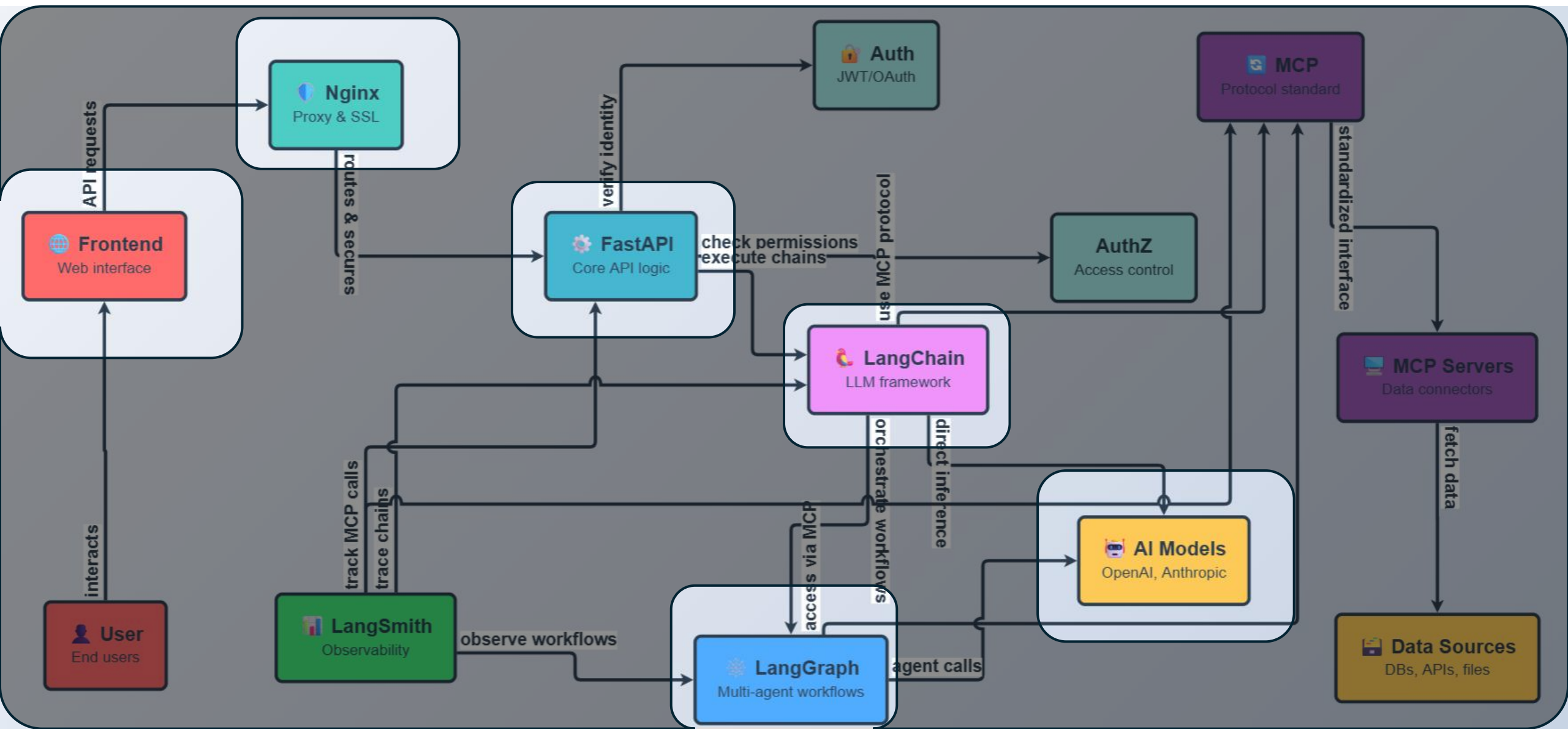
For SSE to work properly, you need:

- **HTTP/1.1** (technically works with 1.0, but 1.1 is better)
- **No response buffering** (if you want streams immediately)
- **Long read timeout** (SSE connections stay open)
- **No caching** (each event stream is unique)

Nginx (if you want streaming)

```
location / {  
    proxy_pass http://localhost:3000;  
  
    # CRITICAL for Flowise streaming  
    proxy_http_version 1.1; # ← ADD THIS  
    proxy_set_header Connection ""; # ← ADD THIS  
    proxy_buffering off;      # ← ADD THIS  
    proxy_cache off;          # ← ADD THIS  
    proxy_read_timeout 300s;  # ← ADD THIS (AI can take time)  
  
    # Standard headers  
    proxy_set_header Host $host;  
    proxy_set_header X-Real-IP $remote_addr;  
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;  
    proxy_set_header X-Forwarded-Proto $scheme;  
}
```

```
sudo nginx -t  
sudo systemctl reload nginx
```

FastAPI and LangChain and LangGraph

LangChain as the Foundation

Handles core LLM logic, such as chaining prompts, managing memory (e.g., conversation history), and integrating external tools/data sources. It's the "brain" for simple interactions.

LangGraph as the Orchestrator

Builds on LangChain to create dynamic, graph-based workflows. In chatbots, it manages multi-turn conversations, agentic behaviors (e.g., deciding when to call tools or end a loop), and state persistence across sessions.

FastAPI as the Interface

Serves as the entry point, turning the LangChain/LangGraph logic into a web service. It handles routing, request validation, and response streaming, making the chatbot accessible via APIs for frontends or integrations and serving the frontend.

The relationship between LangGraph and LangChain and FastAPI

```
from typing import Annotated, Sequence
from fastapi import FastAPI, Request
from fastapi.responses import HTMLResponse
from fastapi.staticfiles import StaticFiles
from pydantic import BaseModel
from langchain_core.messages import BaseMessage, SystemMessage, HumanMessage,
ToolMessage
from langgraph.graph.message import add_messages
from langgraph.graph import StateGraph, END
from langchain_core.tools import tool
from langchain_aws import ChatBedrock
from langgraph.checkpoint.memory import MemorySaver
import json
```

What is Sequence? Like ["1", "2", "3"]

```
from typing import Sequence
# ✗ Too restrictive - only accepts lists
def process_items(items: list[str]) -> int:
    return len(items)

# ✓ Better - accepts any sequence type
def process_items(items: Sequence[str]) -> int:
    return len(items)

# Now all of these work:
process_items(["a", "b", "c"])      # list ✓
process_items(("a", "b", "c"))      # tuple ✓
process_items({"a", "b", "c"})      # ✗ set is NOT a sequence (unordered)
```

What is Annotated? For documentation

```
from typing import Annotated

def send_email(
    recipient: Annotated[str, "Email address of the recipient"],
    subject: Annotated[str, "Email subject line, max 100 chars"],
    body: Annotated[str, "HTML or plain text email body"]
) -> None:
    ...
```

What is FastAPI?

```
from fastapi import FastAPI

app = FastAPI(
    title="My API", # API name (shows in docs)
    description="A cool API", # API description
    version="1.0.0", # API version
)
```

What is FastAPI?

```
from fastapi import FastAPI, Request
from fastapi.responses import HTMLResponse
from fastapi.staticfiles import StaticFiles
```

Request

1. **Purpose:** Represents HTTP request data
2. **Use Case:** Access request headers, body, query parameters, etc.

HTMLResponse

1. **Purpose:** Returns HTML content in HTTP responses
2. **Use Case:** Serve web pages directly from FastAPI

StaticFiles

1. **Purpose:** Serves static files (CSS, JavaScript, images)
2. **Use Case:** Mount static file directories to the application

What is Pydantic?

What is Pydantic?

Data validation and settings management using Python type annotations

BaseModel: Base class for creating data models with automatic validation

```
from pydantic import BaseModel

class ChatRequest(BaseModel):
    message: str
    thread_id: str
```


The relationship between LangGraph and LangChain and FastAPI: FastAPI code

```
# Serving Static files
app = FastAPI()
app.mount("/static", StaticFiles(directory="static", html=True), name="static")

# LangChain and LangGraph logic starts
#...
# LangChain and LangGraph logic ends

# Frontend calling these endpoints
@app.get("/", response_class=HTMLResponse)
async def root(request: Request):
    return HTMLResponse(open("static/index.html").read())

@app.post("/chat")
async def chat(input: ChatInput):
    config = {"configurable": {"thread_id": input.thread_id}}
    state = {"messages": [HumanMessage(content=input.message)]}
    result = graph.invoke(state, config)
    return {"response": result["messages"][-1].content}
```

The relationship between LangGraph and LangChain and FastAPI: LangChain code

```
# LLM: AWS Nova Lite via Bedrock
llm = ChatBedrock(model_id="amazon.nova-lite-v1:0", region_name="us-east-1")

# Simple tool
@tool
def multiply(a: int, b: int) -> int:
    """Multiply two numbers."""
    return a * b

tools = [multiply]
llm = llm.bind_tools(tools)
tools_by_name = {tool.name: tool for tool in tools}
```

The relationship between LangGraph and LangChain and FastAPI: LangGraph code

```
# State
class AgentState(dict):
    messages: Annotated[Sequence[BaseMessage], add_messages]

# Nodes
def tool_node(state: AgentState):
    outputs = []
    for tool_call in state["messages"][-1].tool_calls:
        tool_result = tools_by_name[tool_call["name"]].invoke(tool_call["args"])
        outputs.append(ToolMessage(content=json.dumps(tool_result), name=tool_call["name"],
tool_call_id=tool_call["id"]))
    return {"messages": outputs}

def call_model(state: AgentState):
    system_prompt = SystemMessage("You are a helpful AI. Use tools if needed.")
    response = llm.invoke([system_prompt] + state["messages"])
    return {"messages": [response]}
```

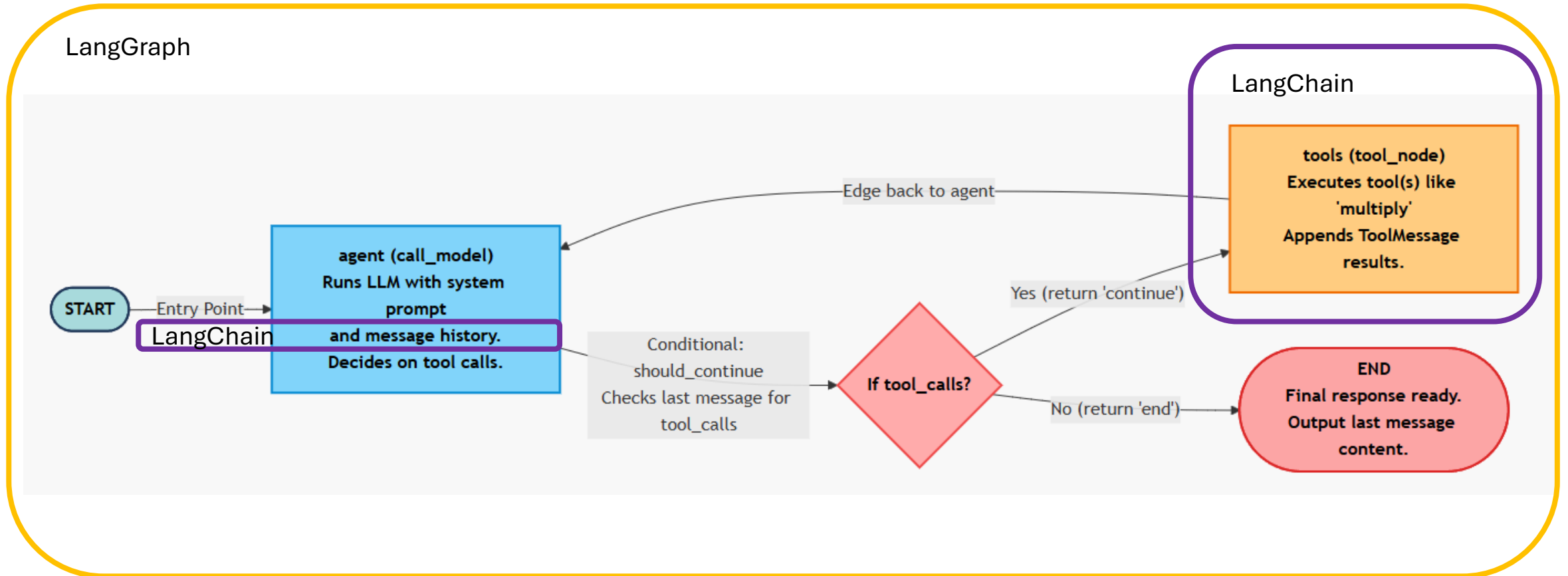
The relationship between LangGraph and LangChain and FastAPI: LangGraph code

```
# Condition
def should_continue(state: AgentState):
    last_message = state["messages"][-1]
    if not last_message.tool_calls:
        return "end"
    return "continue"

# Graph
workflow = StateGraph(state_schema=AgentState)
workflow.add_node("agent", call_model)
workflow.add_node("tools", tool_node)
workflow.set_entry_point("agent")
workflow.add_conditional_edges("agent", should_continue, {"continue": "tools", "end": END})
workflow.add_edge("tools", "agent")
memory = MemorySaver()
graph = workflow.compile(checkpointer=memory)

class ChatInput(BaseModel):
    message: str
    thread_id: str = "1" # Default for simplicity
```

The relationship between LangGraph and LangChain and FastAPI: LangGraph code



The relationship between LangGraph and LangChain and FastAPI: LangGraph doc demo

Constructors		
C constructor		
Properties		
P branches	P channels	P compiled
P edges	P entryPoint?	P Node
P nodes	P waitingEdges	
Accessors		
A allEdges		
Methods		
M _addSchema	M addConditionalEdges	M addEdge
M addNode	M addSequence	M compile
M validate		
Deprecated		
M setEntryPoint	M setFinishPoint	

<https://langchain-ai.github.io/langgraphjs/reference/classes/langgraph.StateGraph.html>

Frontend

```
<!DOCTYPE html>
<html><body>
<div id="messages"></div>
<input id="input" placeholder="Type message"><button onclick="send()">Send</button>
<script>
const threadId = crypto.randomUUID(); // Unique per instance
async function send() {
  const inp = document.getElementById('input');
  const msg = inp.value; inp.value = '';
  document.getElementById('messages').innerHTML += `<p>You: ${msg}</p>`;
  const res = await fetch('/chat', {method: 'POST', headers: {'Content-Type': 'application/json'}, body:
JSON.stringify({message: msg, thread_id: threadId})});
  const data = await res.json();
  document.getElementById('messages').innerHTML += `<p>Bot: ${data.response}</p>`;
}
</script>
</body></html>
```

```
config = {"configurable": {"thread_id":  
input.thread_id}}
```

- 1) The config dict passes the actual thread_id (from the request input) to LangGraph's checkpointer during graph.invoke().
- 2) This overrides the default and enables the MemorySaver to load/save state (including message history) specific to that thread_id, isolating conversations across different users or sessions.

```
const res = await fetch('/chat', {method: 'POST', headers: {'Content-Type': 'application/json'}, body:  
JSON.stringify({message: msg, thread_id: threadId})});
```

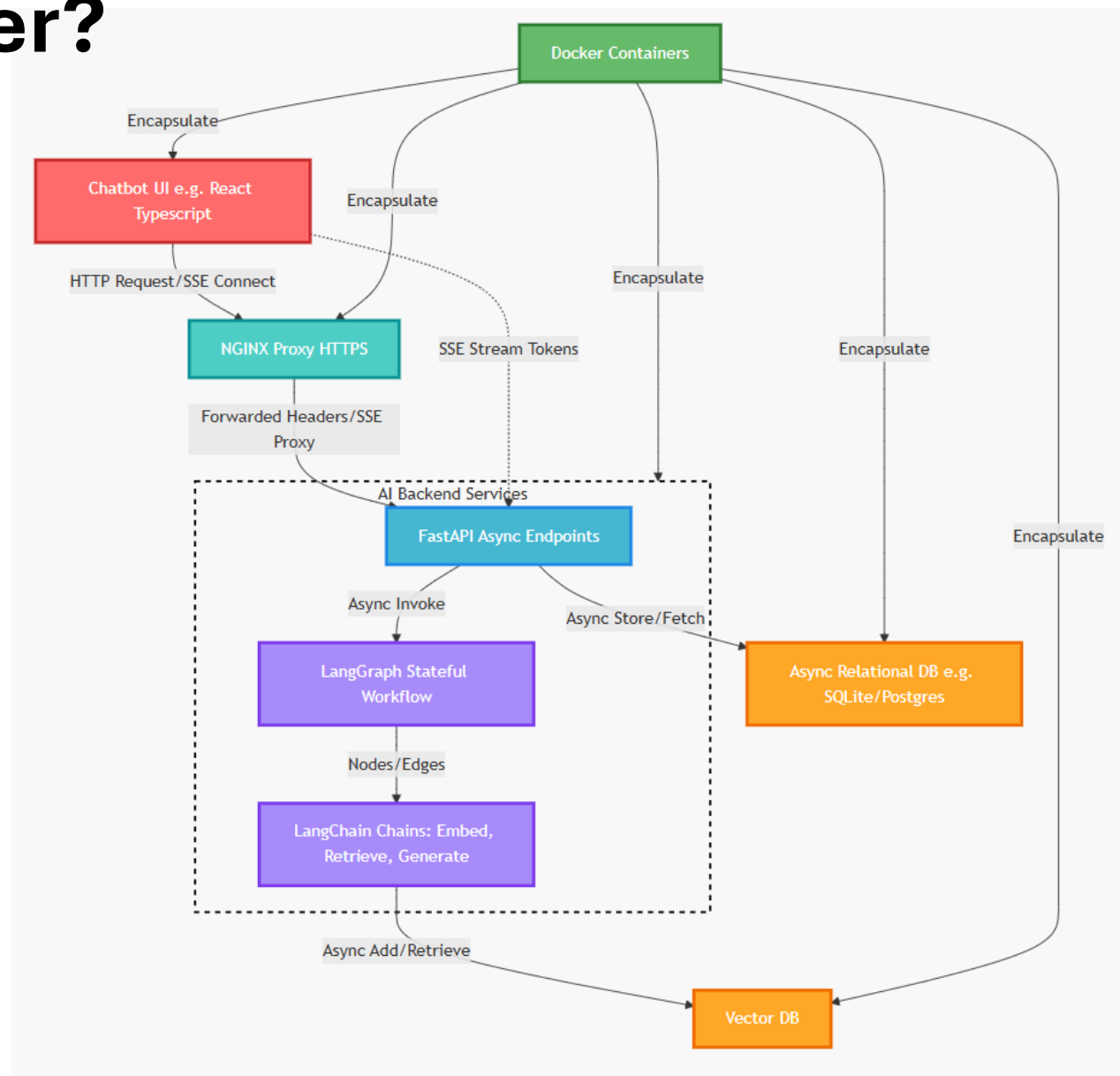

Why thread_id: str = "1" # Default for simplicity?

- 1) This is in the ChatInput Pydantic model for request validation.
- 2) It sets a default value of "1" for thread_id if the frontend doesn't provide one in the POST body.

This ensures the API doesn't fail on missing fields (e.g., for quick testing), while allowing overrides for unique sessions.

```
class ChatInput(BaseModel):  
    message: str  
    thread_id: str = "1" # Default for simplicity
```

Why Docker?



Why Docker?

Consistency Across Environments:

Docker ensures that applications run the same way in development, testing, staging, and production.

Portability:

Containers can be easily moved and deployed across different machines, clouds, or operating systems without reconfiguration.

Isolation and Security:

Each container runs in its own isolated environment, preventing conflicts between applications (e.g., different versions of libraries).

Resource Efficiency:

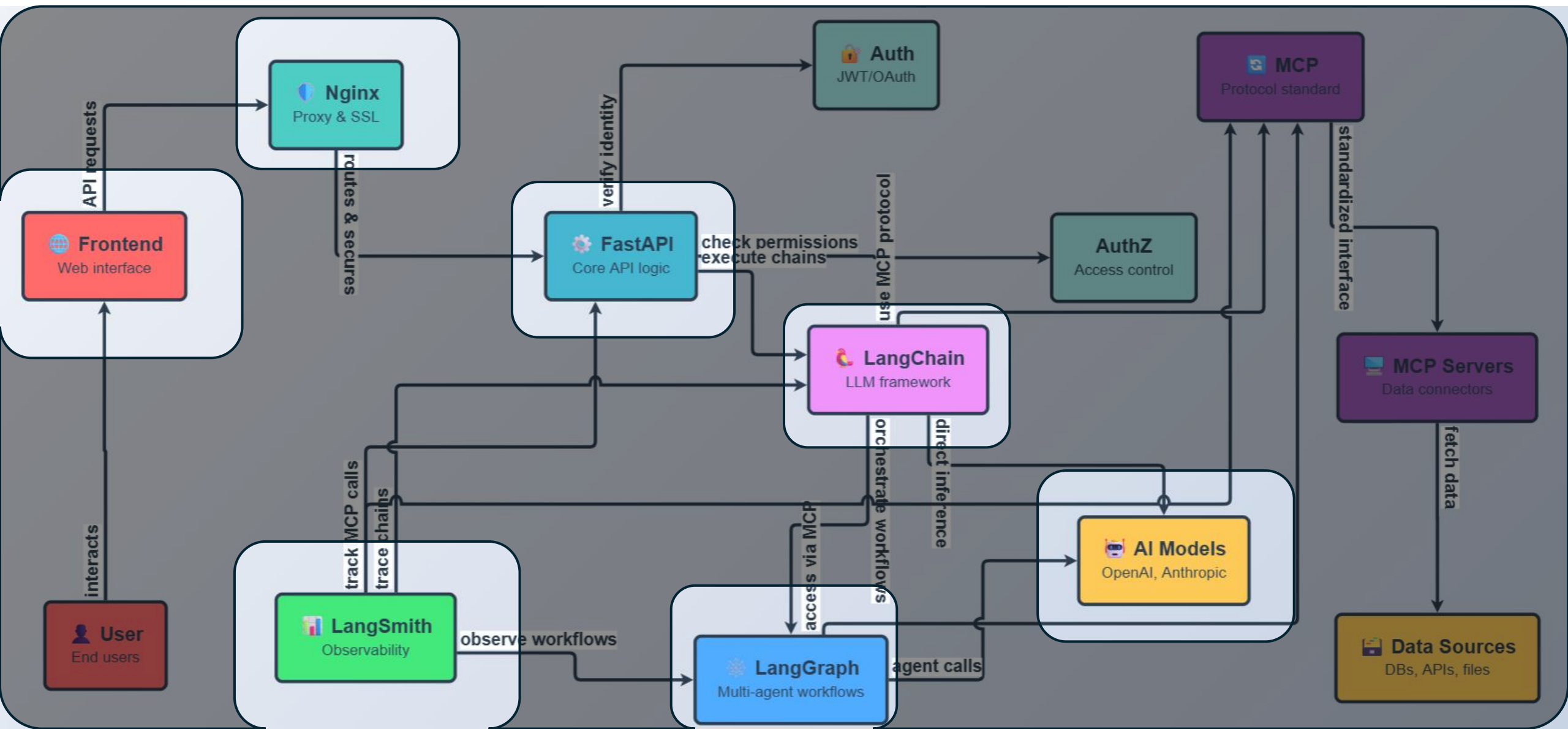
Unlike virtual machines, Docker containers share the host OS kernel, making them lightweight and fast to start.

Scalability and Speed:

Containers can be scaled up or down quickly, supporting microservices architectures.

Improved Development Productivity:

Developers can work in identical environments, simplifying collaboration and onboarding.



What is LangSmith?

- 1) LangSmith is an evaluation and monitoring platform for AI agents and LLM applications.
- 2) It helps developers systematically test, track, and improve their AI systems through structured evaluation frameworks.
- 3) The platform provides tools for creating datasets, running experiments, and analyzing agent performance.

What are Traces in LangSmith?

- **Traces** are comprehensive logs of your agent's execution runs.
- They **capture** inputs, outputs, latencies, and any errors that occur during runtime.
- These traces provide **visibility** into how your agent behaves and help identify issues or optimization opportunities.

```
# Enable tracing in your .env file
LANGCHAIN_TRACING_V2=true
LANGCHAIN_API_KEY=lsv2_pt
LANGCHAIN_PROJECT=your-project-name

# Traces are automatically captured when you
run your agent
from langchain_aws import ChatBedrock

llm = ChatBedrock(model_id="amazon.nova-
lite-v1:0")
response = llm.invoke("Hello") # This run
is traced
```

Why Do We Need Datasets?

- Datasets are collections of test examples with inputs and optional expected outputs.
- They enable repeatable testing across different versions of your agent.
- By maintaining consistent test cases, you can objectively measure improvements or regressions over time.

How Do We Create Datasets?

- Create datasets using the **LangSmith Client** with `create_dataset()` and add examples with `create_examples()`.
- You can batch-create multiple examples at once with inputs and expected outputs.
- **Dataset versioning** allows you to track changes and evaluate against specific versions.

```
from langsmith import Client

client = Client()

# Create a dataset
dataset = client.create_dataset(
    dataset_name="MTR Simple QA",
    description="Simple Q&A examples for MTR"
)

# Batch create examples
examples = [
    {
        "inputs": {"question": "What is MTR?"},
        "outputs": {"answer": "Mass Transit Railway"}
    },
    {
        "inputs": {"question": "How many lines?"},
        "outputs": {"answer": "10 lines"}
    },
]

client.create_examples(
    inputs=[ex["inputs"] for ex in examples],
    outputs=[ex["outputs"] for ex in examples],
    dataset_id=dataset.id,
)
```


What are Evaluators?

- Evaluators are functions that score your agent's outputs against quality criteria.
- They can use string matching, LLM-as-judge patterns, or custom logic for metrics like correctness or tool usage.
- Multiple evaluators can be combined to assess different aspects of agent performance.

What Types of Evaluators Exist?

- **Row-level evaluators** score individual examples (exact match, keyword presence, conciseness).
- **Summary evaluators** aggregate metrics across entire datasets.
- **Comparative evaluators** judge which of two outputs is better, useful for A/B testing different agent versions.

```
# 1. Built-in String Evaluator (LLM-as-judge)
correctness_evaluator = LangChainStringEvaluator(
    "labeled_score_string",
    config={
        "criteria": {
            "correctness": "Is the response
factually correct?"
        },
        "normalize_by": 10  # Score out of 10
    }
)

# 2. Custom Evaluator
@run_evaluator
def exact_match(run, example):
    prediction = run.outputs["answer"]
    reference = example.outputs["answer"]
    return {"key": "exact_match", "score":
prediction == reference}
```

What is Trajectory Evaluation?

- Trajectory evaluation examines the sequence of actions an agent takes, not just the final output.
- It validates that tools are called in the correct order and with appropriate parameters.
- This is crucial for multi-step reasoning and complex agent workflows.

```
@run_evaluator
def tool_usage_evaluator(run, example):
    # Check if tool was called when expected
    tool_calls = [
        msg for msg in run.outputs["messages"]
        if isinstance(msg, ToolMessage)
    ]
    expected_tool = "multiply" in example.inputs["question"].lower()
    actual_tool = len(tool_calls) > 0
    score = 1 if expected_tool == actual_tool else 0
    return {
        "key": "tool_usage_correct",
        "score": score,
        "comment": f"Expected tool: {expected_tool}, Used: {actual_tool}"
    }
```

What are Experiments?

- Experiments are **tracked runs** of your agent on specific **datasets** with associated evaluators.
- Each experiment captures **metrics**, **metadata** (**model version**, **prompts**), and **timestamps**.
- Experiments enable systematic comparison of different agent configurations or improvements.

How Do We Run Evaluations?

- Use the `evaluate()` function with your target function, dataset, and evaluators.
- For better performance, use `aevaluate()` for async evaluation with higher concurrency.
- Results include `scores`, metadata, and detailed traces for each test example.

```
from langsmith.evaluation import evaluate

def my_agent(inputs: dict) -> dict:
    """Your system to evaluate."""
    response = llm.invoke(inputs["question"])
    return {"answer": response.content}

# Run evaluation
results = evaluate(
    my_agent,                                # Your target function
    data="MTR Simple QA",                   # Dataset name
    evaluators=[exact_match, correctness_evaluator],
    experiment_prefix="baseline-v1",
    description="First baseline evaluation",
    metadata={
        "model": "amazon.nova-lite-v1:0",
        "temperature": 0.0,
    },
    max_concurrency=4,
)

print(f"Average score: {results['results'][0]['score']}")
```

How Can We Compare Different Versions?

- Use `evaluate_comparative()` to compare two experiments with pairwise evaluators.
- The platform can randomize comparison order to reduce bias.
- Results show which version performs better across your test dataset with statistical metrics.

```
from langsmith.evaluation import evaluate_comparative

# Run two experiments
exp1 = evaluate(agent_v1, data="MTR Simple QA",
evaluators=[...])
exp2 = evaluate(agent_v2, data="MTR Simple QA",
evaluators=[...])

# Compare with pairwise evaluator
def preference_evaluator(inputs: dict, runs: list) -> dict:
    """LLM judges which output is better."""
    prompt = f"""Question: {inputs['question']}
    Answer A: {runs[0].outputs['answer']}
    Answer B: {runs[1].outputs['answer']}
    Which answer is better? Return score for A and B."""

    # Use LLM to judge (simplified)
    return {"key": "preference", "scores": [0.3, 0.7]}

results = evaluate_comparative(
    experiments=(exp1.experiment_name, exp2.experiment_name),
    evaluators=[preference_evaluator],
    randomize_order=True, # Reduce bias
)
```

What is Async Evaluation?

- Async evaluation uses `aevaluate()` for better performance with concurrent requests.
- It allows higher concurrency levels than synchronous evaluation.
- This significantly reduces total evaluation time for large datasets.

```
from langsmith import aevaluate

async def async_agent(inputs: dict) -> dict:
    """Async version of your agent."""
    response = await async_llm.ainvoke(inputs["question"])
    return {"answer": response.content}

# Run async evaluation with higher concurrency
results = await aevaluate(
    async_agent,
    data="MTR Simple QA",
    evaluators=[exact_match, correctness_evaluator],
    max_concurrency=10, # Process 10 examples simultaneously
    experiment_prefix="async-baseline",
)
```

What Metadata Should We Track?

- Track model versions, temperature settings, prompt templates, and timestamps in experiment metadata.
- This enables filtering and grouping results in the LangSmith UI.
- Metadata makes results reproducible and helps identify which changes improved performance.

```
# Add metadata to experiments
results = evaluate(
    my_agent,
    data="MTR Simple QA",
    evaluators=[evaluators],
    metadata={
        "model": "amazon.nova-lite-v1:0",
        "temperature": 0.7,
        "prompt_version": "v2.1",
        "date": "2025-10-24",
        "engineer": "team-ai"
    }
)

# Version your datasets too
client.update_dataset(
    dataset_id=dataset.id,
    metadata={
        "version": "v1.0",
        "created_date": "2025-10-24"
    }
)
```


What are Best Practices?

- Always tag experiments with metadata and establish baselines before making changes.
- Use async evaluation for better performance and run multiple repetitions for statistical significance.
- Include edge cases in datasets and use multiple evaluator types beyond just exact matching.

```
# Best practice: Run with repetitions
results = evaluate(
    my_agent,
    data="MTR Simple QA",
    evaluators=[exact_match, correctness, conciseness],
    num_repetitions=3, # Run each example 3 times
    metadata={
        "model": "amazon.nova-lite-v1:0",
        "version": "baseline"
    }
)

# Best practice: Include diverse evaluators
evaluators = [
    exact_match,           # Exact string match
    keyword_match,         # Contains key terms
    correctness_llm,       # LLM judges correctness
    conciseness,           # Response length
    tool_usage_correct,    # Trajectory validation
]
```

What Can We Do in the LangSmith UI?

- The web interface at smith.langchain.com provides visual experiment comparison and detailed trace inspection.
- You can analyze **aggregate** metrics, filter by **metadata**, and add **human feedback**.
- The UI makes it easy to share results and collaborate on improvements.

How Does This Improve AI Agents?

- **Systematic evaluation** reveals strengths and weaknesses in agent behavior objectively.
- Experiment tracking shows whether changes actually improve performance.
- The iterative cycle of test-measure-improve leads to more reliable and capable AI systems over time.

```
# Complete evaluation workflow
from langsmith import Client
from langsmith.evaluation import evaluate

# 1. Create dataset
client = Client()
dataset = client.create_dataset("Production Tests")

# 2. Define evaluators
evaluators = [correctness, conciseness, tool_usage]

# 3. Run baseline
baseline = evaluate(agent_v1, data=dataset,
                    evaluators=evaluators)

# 4. Make improvements to your agent
# ... modify prompts, add tools, tune parameters ...

# 5. Run new experiment
improved = evaluate(agent_v2, data=dataset,
                    evaluators=evaluators)

# 6. Compare results in UI or programmatically
print(f"Baseline: {baseline.metrics}")
print(f"Improved: {improved.metrics}")
```

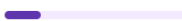
Langsmith

Personal

Setup resource ti

Personal ID

Traces



989/5000 ↗



Studio quickstart



Tracing quickstart

Observability

Tracing Projects 6

Custom Dashboards 0

Name	Most Recent Run (7D)	Feedback (7D)	Run Count (7D)	Error Rate (7D)	P50 Latency (7D)	P99 Latency (7D)
evaluators	10/24/2025, 4:08:49 PM		302	1%	🕒 0.00s	🕒 1.49s
pr-mcp-quick-test	10/24/2025, 2:14:46 PM		2	0%	🕒 1.13s	🕒 1.17s
pr-jaunty-gosling-55	10/24/2025, 10:31:09 AM		41	0%	🕒 3.50s	🕒 7.73s
Eng Demo Split			0	0%		
Engdemo			0	0%		

Showing 5 most active projects.

[View all](#)

Math Chabot evaluation by LangSmith

- LangSmith Evaluation Example
- 1. Dataset creation
- 2. LLM-as-judge evaluators
- 3. A/B testing with different system prompts
- 4. Trajectory evaluation with tool usage
- 5. Comparative experiments

Math Chatbot Datasets

- Datasets are collections of test examples with inputs and expected outputs that enable repeatable testing.

```
from langsmith import Client

client = Client()
dataset_name = "Math Calculator QA"

# Create dataset
if not client.has_dataset(dataset_name=dataset_name):
    dataset = client.create_dataset(
        dataset_name=dataset_name,
        description="Math questions requiring calculator tool usage"
    )
else:
    dataset = client.read_dataset(dataset_name=dataset_name)
```

Math Chatbot: How Do We Create Dataset Examples?

- Our Math Calculator QA dataset includes 4 examples covering math operations and edge cases (conversational queries).

-

```
examples = [  
    {  
        "inputs": {"question": "What is 15 plus 27?"},  
        "outputs": {  
            "answer": "42",  
            "should_use_tool": True,  
            "expected_tool": "add"  
        }  
    }, ...  
]  
  
client.create_examples(  
    inputs=[ex["inputs"] for ex in examples],  
    outputs=[ex["outputs"] for ex in examples],  
    dataset_id=dataset.id,  
)
```

Math Chatbot: **How Do We Create Dataset Examples?**

- Use batch creation with `create_examples()` to add multiple test cases at once with inputs and outputs.
- Each example includes the question, expected answer, and metadata about tool usage expectations.
- Include edge cases like conversational queries that shouldn't trigger tool usage to validate agent behavior.

Math Chatbot Datasets

- Our Math Calculator QA dataset includes 4 examples covering math operations and edge cases (conversational queries).

Personal > Datasets & Experiments > Math Calculator QA

Math Calculator QA

ID

Experiments

Examples

Evaluators

Pairwise Experiments

Few-shot search

latest

+ Evaluator

+ Experiment

Add resource tags

DEVELOPER

Filters

Select split

Compact

Full

JSON

YAML

Columns

+ Example

	Inputs	Reference Outputs	Created At ↓	Modified At ↑↓	Splits	
<input type="checkbox"/>	Hello, how are you?	conversational response	10/24/2025, 3:49:04 PM	10/24/2025, 3:49:04 PM	base	⋮
<input type="checkbox"/>	What is 100 divided by 4?	25	10/24/2025, 3:49:04 PM	10/24/2025, 3:49:04 PM	base	⋮
<input type="checkbox"/>	What is 15 plus 27?	42	10/24/2025, 3:49:04 PM	10/24/2025, 3:49:04 PM	base	⋮
<input type="checkbox"/>	Calculate 8 times 7	56	10/24/2025, 3:49:04 PM	10/24/2025, 3:49:04 PM	base	⋮

4 examples in total

Page 1<>Show 15

How Do We Define Tools for Agents? (LangChain Stuff)

- Tools are Python functions decorated with @tool that agents can call to perform operations.
- Clear docstrings are crucial - the LLM uses them to decide when to call each tool.
- Our calculator has three tools: add, multiply, and divide with error handling for edge cases.

How Do We Define Tools for Agents? (LangChain Stuff)

```
from langchain_core.tools import tool

@tool
def add(a: float, b: float) -> float:
    """Add two numbers together."""
    return a + b

@tool
def multiply(a: float, b: float) -> float:
    """Multiply two numbers together."""
    return a * b

@tool
def divide(a: float, b: float) -> float:
    """Divide first number by second number."""
    if b == 0:
        return "Error: Division by zero"
    return a / b

tools = [add, multiply, divide]
tools_by_name = {t.name: t for t in tools}
```

A/B testing and Agent Factory Pattern

- A factory function creates agents with different configurations while maintaining the same core logic.
- It enables easy A/B testing by generating variants with different system prompts or parameters.
- The factory captures trajectories (execution history) and extracts tool calls for evaluation.

```

def create_agent(system_prompt: str):
    """Factory function to create agents with different system prompts."""
    def agent_with_tools(inputs: dict) -> dict:
        question = inputs["question"]
        trajectory = []
        llm_with_tools = agent_llm.bind_tools(tools)
        messages = [SystemMessage(content=system_prompt), HumanMessage(content=question)]
        trajectory.append({"step": "initial", "type": "user_message", "content": question})
        max_iterations = 3
        for iteration in range(max_iterations):
            response = llm_with_tools.invoke(messages)
            messages.append(response)
            trajectory.append({
                "step": f"llm_response_{iteration}", "type": "ai_message",
                "content": response.content,
                "tool_calls": len(response.tool_calls) if hasattr(response, 'tool_calls') else 0
            })

            if hasattr(response, 'tool_calls') and response.tool_calls:
                for tool_call in response.tool_calls:
                    tool_result = tools_by_name[tool_call["name"]].invoke(tool_call["args"])
                    trajectory.append({
                        "step": f"tool_call_{iteration}", "type": "tool_call",
                        "tool": tool_call["name"], "args": tool_call["args"], "result": tool_result
                    })
                    messages.append(ToolMessage(content=str(tool_result), tool_call_id=tool_call["id"], name=tool_call["name"]))
            else:
                break
        return {
            "answer": messages[-1].content if messages else "No response",
            "trajectory": trajectory,
            "tool_calls": [t for t in trajectory if t["type"] == "tool_call"]
        }
    return agent_with_tools

# Usage
agent_a = create_agent(SYSTEM_PROMPT_A)
agent_b = create_agent(SYSTEM_PROMPT_B)

```

Rule base evaluator with Math Chatbot


- Rule-based evaluators use simple logic like string matching or keyword detection for fast, deterministic scoring.
- The correctness evaluator checks if the expected numerical answer appears in the agent's response.
- They're ideal for clear success criteria but less flexible than LLM-as-judge approaches.

Rule base evaluator with Math Chatbot

```
def correctness_evaluator(outputs: dict, reference_outputs: dict) -> dict:
    """Check if answer contains the expected numerical result."""
    answer = outputs["answer"].lower()
    expected = str(reference_outputs["answer"]).lower()

    # Check if expected answer is in the response
    score = 1 if expected in answer else 0

    return {
        "key": "correctness",
        "score": score,
        "comment": f"Expected '{expected}' in answer"
    }

# Example usage:
# Input: "What is 15 plus 27?"
# Expected: "42"
# Agent output: "The answer is 42"
# Score: 1 (correct) 
```

Math Chatbot Trajectory Evaluation

- Trajectory evaluation examines the sequence of actions an agent takes, not just the final output.
- It validates that the correct tools were called when needed and that conversational queries didn't trigger tools.
- This is crucial for multi-step reasoning and ensuring agents follow expected behavior patterns.
-

Trajectory Evaluation

```
def tool_usage_evaluator(outputs: dict, reference_outputs: dict) -> dict:
    """Check if correct tool was used when needed."""
    should_use_tool = reference_outputs.get("should_use_tool", False)
    expected_tool = reference_outputs.get("expected_tool")

    tool_calls = outputs.get("tool_calls", [])
    tools_used = [tc["tool"] for tc in tool_calls]

    if should_use_tool:
        if expected_tool in tools_used:
            score = 1
            comment = f"Correctly used {expected_tool}"
        else:
            score = 0
            comment = f"Should use {expected_tool}, but used {tools_used}"
    else:
        if len(tools_used) == 0:
            score = 1
            comment = "Correctly did not use tools"
        else:
            score = 0
            comment = f"Should not use tools, but used {tools_used}"

    return {
        "key": "tool_usage",
        "score": score,
        "comment": comment
    }
```

math-agent-formal-8e04b1af

mtr-mc...

Inputs

Reference Outputs

Hello, how are you?	#9aa0 →	conversational respo
What is 100 divided by 4?	#9db4 →	25
What is 15 plus 27?	#a0f2 →	42
Calculate 8 times 7	#c4d2 →	56

TRACE

Waterfall

Target

1.71s 1,261

ChatBedrockC... amazon.nova-...

0.82s 594

ChatBedrockC... amazon.nova-...

0.77s 667

add 0.00s

Target ID

Playground

Run Feedback Metadata

Input

Inputs

Question What is 15 plus 27?

Output

USER MESSAGE

What is 15 plus 27?

AI MESSAGE

<thinking>The User has asked to add 15 and 27. This is a simple arithmetic operation that can be performed using the 'add' tool.</thinking>

AI MESSAGE

<thinking>The 'add' tool has returned the result of 15 plus 27 as 42.0.</thinking>

The sum of 15 and 27 is 42.0

ADDITIONAL FIELDS 2

Answer

<thinking>The 'add' tool has returned the result of 15 plus 27 as 42.0.</thinking>

FEEDBACK

correctness 1.00

helpfulness_... 0.50

response_len... 1.00

tool_usage 1.00

START TIME

10/24/2025, 04:08:26 PM

END TIME

10/24/2025, 04:08:27 PM

STATUS

Success

TOTAL TOKENS

1,261 tokens

LATENCY

1.71s

TYPE

Chain

FROM EXAMPLE

#a0f2 @ Math Calculator QA

LLM-as-Judge Evaluation

- LLM-as-judge uses another LLM to assess subjective qualities like helpfulness, clarity, or tone.
- It's more flexible than rule-based evaluators but slower and more expensive due to additional LLM calls.
- Use **low temperature (0.0)** for consistency and clear scoring criteria in the judge prompt.

```
def llm_judge_helpfulness(outputs: dict, reference_outputs: dict) -> dict:
    """Use LLM to judge the helpfulness of the response."""

    judge_prompt = f"""You are evaluating an AI assistant's response for helpfulness.
```

```

    Question: {reference_outputs.get('question', 'N/A')}
    Response: {outputs['answer']}
```

```

    Rate the helpfulness on a scale of 0-1:
    - 1.0: Very helpful, clear, and answers the question well
    - 0.5: Somewhat helpful but could be clearer
    - 0.0: Not helpful or confusing
```

```

    Respond with ONLY a number between 0 and 1 (e.g., 0.8)"""
    judge_response = judge_llm.invoke([HumanMessage(content=judge_prompt)])
    score_text = judge_response.content.strip()
```

```

    # Extract number from response
    import re
    match = re.search(r'0\.\d+|1\.\d|0|1', score_text)
    if match:
        score = float(match.group())
    else:
        score = 0.5 # Default if can't parse
```

```

    return {
        "key": "helpfulness_llm_judge",
        "score": score,
        "comment": f"LLM judged: {score_text[:50]}"
    }
```

Heuristic Evaluation

- Heuristic evaluators apply simple rules like checking response length, format, or structure.
- The response_length evaluator ensures answers are concise (20-200 characters) without being too brief.
- They're fast, free, and useful for quick sanity checks but don't assess content quality.

```
def response_length_evaluator(outputs: dict) -> dict:
    """Check if response is appropriately concise."""
    answer = outputs["answer"]
    length = len(answer)

    # Good range: 20-200 characters
    if 20 <= length <= 200:
        score = 1
    elif length < 20:
        score = 0.5 # Too short
    else:
        score = 0.7 # A bit long but acceptable

    return {
        "key": "response_length",
        "score": score,
        "comment": f"Length: {length} chars"
    }
```

Run an Experiment for Math Chatbot

- Use the `evaluate()` function with `your agent`, `dataset name`, `evaluators`, and `metadata` for tracking.
- Each experiment processes all examples, runs all evaluators, and captures metrics, latency, and traces.
- Results are automatically uploaded to LangSmith with a unique experiment name for comparison.

```
from langsmith.evaluation import evaluate
results_a = evaluate(
    agent_a, # Your target function
    data=dataset_name,
    evaluators=[correctness_evaluator, tool_usage_evaluator,
                llm_judge_helpfulness, response_length_evaluator],
    experiment_prefix="math-agent-formal",
    description="Agent with formal, precise system prompt",
    metadata={
        "model": os.getenv("BEDROCK_MODEL"),
        "system_prompt": "formal",
        "variant": "A",
        "temperature": 0.7
    },
    max_concurrency=2,
)
```

```
print(f"✅ Experiment A Complete: {results_a.experiment_name}")
```

What gets tracked: Input: The question, Output: Agent's response, Trajectory: Full execution trace, Evaluator Scores: All 4 evaluator results, Latency: How long it took Metadata: Model, variant, temperature, etc.



Math Calculator QA



Experiments

Examples

Evaluators

Pairwise Experiments

Few-shot search

All Versions

+ Evaluator

+ Experiment



Search by name...

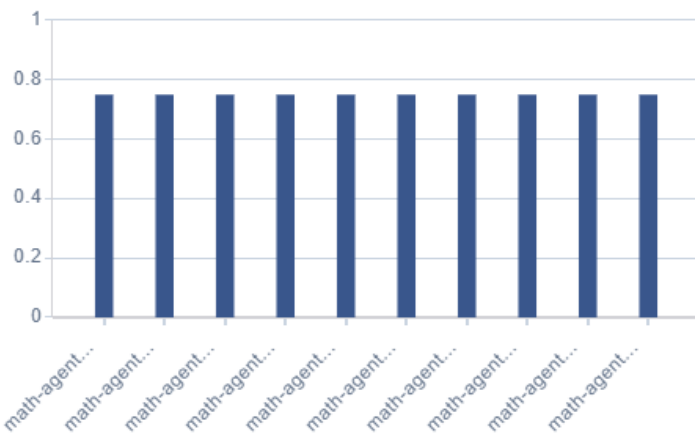
Charts

x-axis Experiment

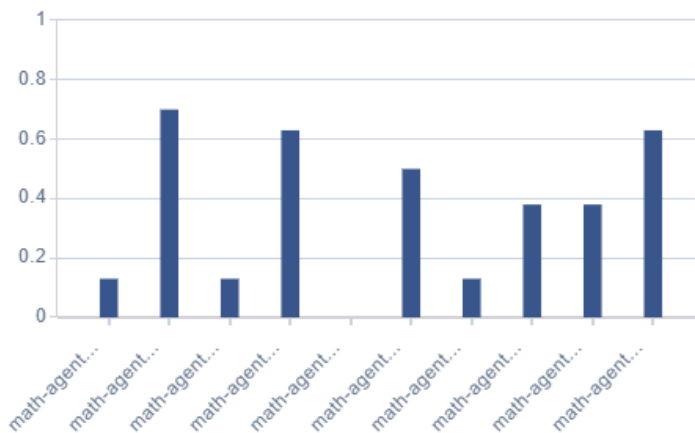
Filters



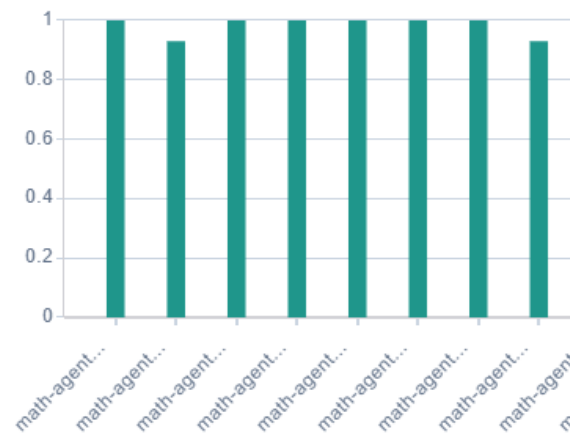
Correctness



Helpfulness_llm_judge



Response_length



<input type="checkbox"/>	Experiment ↑↓	Splits 🕒	Repetitions 🔄	Correctness ↑↓	Helpfulness_llm_judge ↑↓	Response_length ↑↓	Tool_usage ↑↓	
#12	math-agent-friendly-d5cc... Agent with friendly, encou...	base	1	0.75 <div></div>	0.63 <div></div>	1.00 <div></div>	1.00 <div></div>	⋮
#11	math-agent-formal-dd1e4... Agent with formal, precise...	base	1	0.75 <div></div>	0.38 <div></div>	0.93 <div></div>	1.00 <div></div>	⋮
#10	math-agent-friendly-328b... Agent with friendly, encou...	base	1	0.75 <div></div>	0.38 <div></div>	0.93 <div></div>	1.00 <div></div>	⋮
#9	math-agent-formal-8e04... Agent with formal, precise...	base	1	0.75 <div></div>	0.13 <div></div>	1.00 <div></div>	1.00 <div></div>	⋮
#8	math-agent-friendly-301e... Agent with friendly, encou...	base	1	0.75 <div></div>	0.50 <div></div>	1.00 <div></div>	1.00 <div></div>	⋮
#7	math-agent-formal-1a699... Agent with formal, precise...	base	1	0.75 <div></div>	0.00 <div></div>	1.00 <div></div>	1.00 <div></div>	⋮



A/B Testing with Math Chatbot

- A/B testing compares two agent variants to determine which configuration performs better.
- Our example tests a formal prompt (concise, directive) vs. a friendly prompt (warm, explanatory).
- Both agents use the same tools and dataset but differ in tone and instruction style.

A/B Testing with Math Chatbot

```
# Variant A: Formal system prompt
SYSTEM_PROMPT_A = """You are a precise mathematical assistant.
When asked to perform calculations, you MUST use the available calculator tools.
Always use tools for arithmetic operations. Be formal and concise."""

# Variant B: Friendly system prompt
SYSTEM_PROMPT_B = """You are a friendly and helpful math tutor!
When someone asks you to calculate something, use your calculator tools to help them out.
Use tools for math operations and explain your steps in a warm, encouraging way."""

agent_a = create_agent(SYSTEM_PROMPT_A)
agent_b = create_agent(SYSTEM_PROMPT_B)

# What's being tested?
# Tone: Formal vs. Friendly
# Length: Concise vs. Explanatory
# Instruction: "MUST use" vs. "use your tools to help"
```

How Do We Compare Two Experiments?

- Run both variants on the same dataset with identical evaluators to ensure fair comparison.
- Tag each experiment with metadata (variant: A/B) to easily identify and filter results.
- In the LangSmith UI, select both experiments and click Compare for side-by-side analysis.

How Do We Compare Two Experiments?

```
# Run Experiment A: Formal
results_a = evaluate(
    agent_a,
    data=dataset_name,
    evaluators=evaluators,
    experiment_prefix="math-agent-formal",
    metadata={"variant": "A", "system_prompt": "formal"},
)

# Run Experiment B: Friendly
results_b = evaluate(
    agent_b,
    data=dataset_name, # Same dataset!
    evaluators=evaluators, # Same evaluators!
    experiment_prefix="math-agent-friendly",
    metadata={"variant": "B", "system_prompt": "friendly"},
)

# Navigate to LangSmith UI:
# 1. Go to "Math Calculator QA" → "Experiments" tab
# 2. Click checkboxes for both experiments
# 3. Click "Compare" button
# 4. View side-by-side results with differences highlighted
```

math-agent-formal-8e04b1af

mtr-mc.../16733d



Compact

Full

Diff

Default

Group by

Display



+ Compare

Inputs	Reference Outputs	Outputs	Correctness	Helpfulness...	Response_L...	Tool_usage	Latency
			0.75 AVG	0.125 AVG	1.00 AVG	1.00 AVG	1.661 P
Hello, how are you? #9aa0 →	conversational response	Hello! I'm doing well, thank you for asking	0.00	0.00	1.00	1.00	1.61s
What is 100 divided by 4? #9db4 →	25	The result of dividing 100 by 4 is 25.0.	1.00	0.00	1.00	1.00	2.17s
What is 15 plus 27? #a0f2 →	42	<thinking>The 'add' tool has returned the	1.00	0.50	1.00	1.00	1.71s
Calculate 8 times 7 #c4d2 →	56	The result of multiplying 8 by 7 is 56.	1.00	0.00	1.00	1.00	1.48s

Inputs	Reference Outputs	math-agent-friendly-... <div><div>Baseline</div><div>≡</div><div>:</div></div>	math-agent-form... <div><div>0 ↑</div><div>0 ↓</div><div>≡</div><div>:</div></div>	
Hello, how are you? <div>#9aa0 →</div>	conversational response	Hello! I'm doing well, thank you for asking. How ca <div><div>correctness</div><div>0.00</div><div>:</div><div>Show 3 more...</div></div> <div><div>0.86s</div><div>586</div></div>	Hello! I am here to assist you with any mathematic <div><div>correctness</div><div>0.00</div><div>:</div><div>Show 3 more...</div></div> <div><div>1.66s</div><div>576</div></div>	
What is 100 divided by 4? <div>#9db4 →</div>	25	<thinking>The result from the 'divide' tool is 25.0. <div><div>correctness</div><div>1.00</div><div>:</div><div>Show 3 more...</div></div> <div><div>1.70s</div><div>1,280</div></div>	The result of 100 divided by 4 is 25.0. <div><div>correctness</div><div>1.00</div><div>:</div><div>Show 3 more...</div></div> <div><div>2.23s</div><div>1,257</div></div>	
What is 15 plus 27? <div>#a0f2 →</div>	42	<thinking>The 'add' tool has calculated the sum o <div><div>correctness</div><div>1.00</div><div>:</div><div>Show 3 more...</div></div> <div><div>1.59s</div><div>1,266</div></div>	The sum of 15 and 27 is 42. <div><div>correctness</div><div>1.00</div><div>:</div><div>Show 3 more...</div></div> <div><div>1.42s</div><div>1,228</div></div>	
Calculate 8 times 7 <div>#c4d2 →</div>	56	<thinking>The 'multiply' tool has returned a result <div><div>correctness</div><div>1.00</div><div>:</div><div>Show 3 more...</div></div> <div><div>1.55s</div><div>1,245</div></div>	The result of 8 times 7 is 56. <div><div>correctness</div><div>1.00</div><div>:</div><div>Show 3 more...</div></div> <div><div>1.36s</div><div>1,192</div></div>	

What Results Did We Get from A/B Testing?

- Formal variant: More concise (1.00 length score), faster (fewer tokens), but lower helpfulness (0.13).
- Friendly variant: Higher helpfulness (0.50), more engaging, but slightly longer responses (0.93 length score).
- Both variants: Equally correct (0.75), perfect tool usage (1.00) - the trade-off is conciseness vs. engagement.

Comparing 2 Experiments

Outputs

Charts



Baseline

correctness

Compact

Full

Diff

Default

Display

+ Compare

Inputs	Reference Outputs	math-agent-friendly-... Baseline	math-agent-form... 0 ↑ 0 ↓
Hello, how are you? #9aa0 →	conversational response	<p>Hello! I'm doing well, thank you for asking. How ca</p> <p>correctness 0.00 : Show 3 more...</p> <p>0.86s 586</p>	<p>Hello! I am here to assist you with any mathematic</p> <p>correctness 0.00 : Show 3 more...</p> <p>1.66s 576</p>
What is 100 divided by 4? #9db4 →	25	<p><thinking>The result from the 'divide' tool is 25.0.</p> <p>correctness 1.00 : Show 3 more...</p> <p>1.70s 1,280</p>	<p>The result of 100 divided by 4 is 25.0.</p> <p>correctness 1.00 : Show 3 more...</p> <p>2.23s 1,257</p>
What is 15 plus 27? #a0f2 →	42	<p><thinking>The 'add' tool has calculated the sum o</p> <p>correctness 1.00 : Show 3 more...</p> <p>1.59s 1,266</p>	<p>The sum of 15 and 27 is 42.</p> <p>correctness 1.00 : Show 3 more...</p> <p>1.42s 1,228</p>
Calculate 8 times 7 #c4d2 →	56	<p><thinking>The 'multiply' tool has returned a result</p> <p>correctness 1.00 : Show 3 more...</p> <p>1.55s 1,245</p>	<p>The result of 8 times 7 is 56.</p> <p>correctness 1.00 : Show 3 more...</p> <p>1.36s 1,192</p>

LangSmith UI: Experiment Results

- The Experiments tab shows aggregate charts for all metrics (correctness, helpfulness, tool usage, length).
- Each experiment row displays scores, latencies (P50, P99), error rates, and metadata columns.
- Click any experiment to see detailed results for each test example with color-coded scores (green = good, red = bad).



Math Calculator QA



Experiments

Examples

Evaluators

Pairwise Experiments

Few-shot search

All Versions

+ Evaluator

+ Experiment



Search by name...

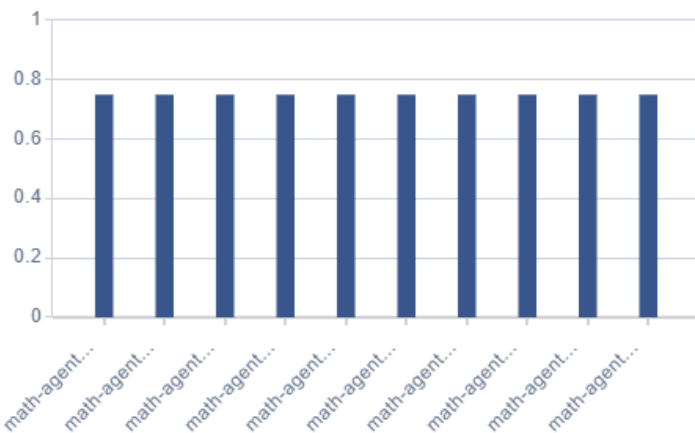
Charts

x-axis Experiment

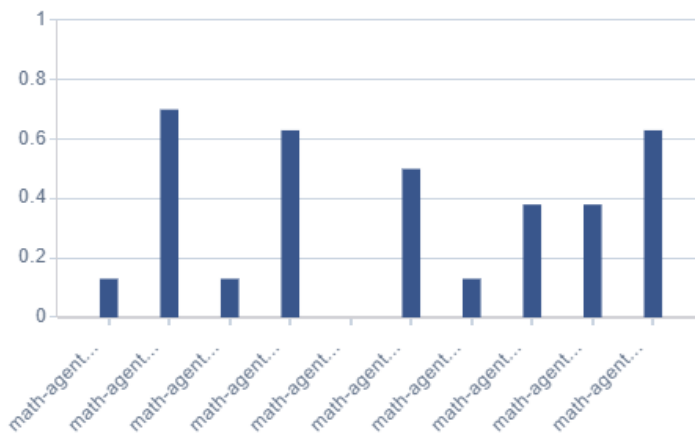
Filters



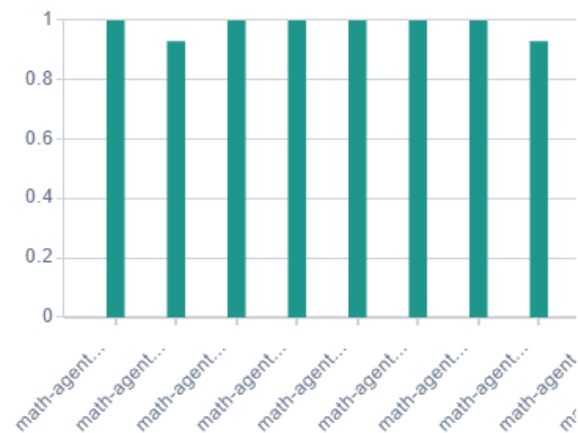
Correctness



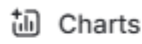
Helpfulness_llm_judge



Response_length



<input type="checkbox"/>	Experiment ↑↓	Splits 🕒	Repetitions 🔄	Correctness ↑↓	Helpfulness_llm_judge ↑↓	Response_length ↑↓	Tool_usage ↑↓	
#12	math-agent-friendly-d5cc... Agent with friendly, encou...	base	1	0.75 <div></div>	0.63 <div></div>	1.00 <div></div>	1.00 <div></div>	⋮
#11	math-agent-formal-dd1e4... Agent with formal, precise...	base	1	0.75 <div></div>	0.38 <div></div>	0.93 <div></div>	1.00 <div></div>	⋮
#10	math-agent-friendly-328b... Agent with friendly, encou...	base	1	0.75 <div></div>	0.38 <div></div>	0.93 <div></div>	1.00 <div></div>	⋮
#9	math-agent-formal-8e04... Agent with formal, precise...	base	1	0.75 <div></div>	0.13 <div></div>	1.00 <div></div>	1.00 <div></div>	⋮
#8	math-agent-friendly-301e... Agent with friendly, encou...	base	1	0.75 <div></div>	0.50 <div></div>	1.00 <div></div>	1.00 <div></div>	⋮
#7	math-agent-formal-1a699... Agent with formal, precise...	base	1	0.75 <div></div>	0.00 <div></div>	1.00 <div></div>	1.00 <div></div>	⋮



Charts

x-axis

Experiment



Filters



Search by name...

EVALUATION SCORES

Correctness

10 experiments



Helpfulness_llm_judge

10 experiments



Response_length

10 experiments



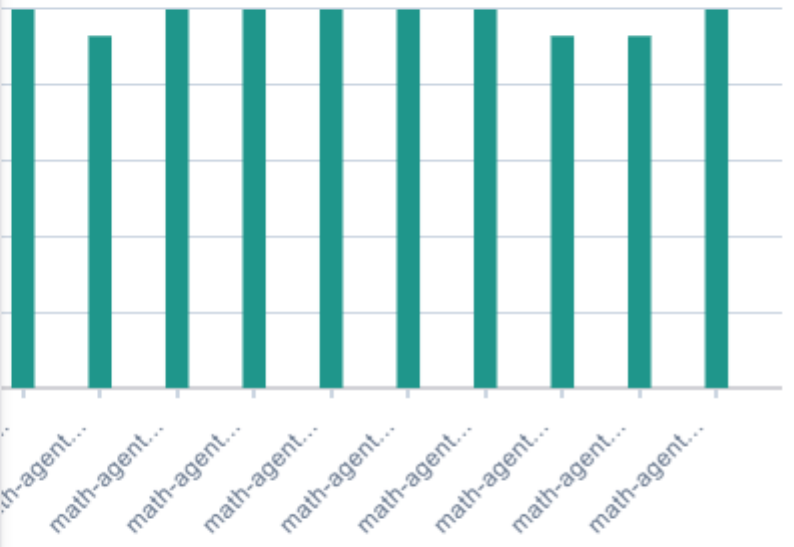
Tool_usage

10 experiments

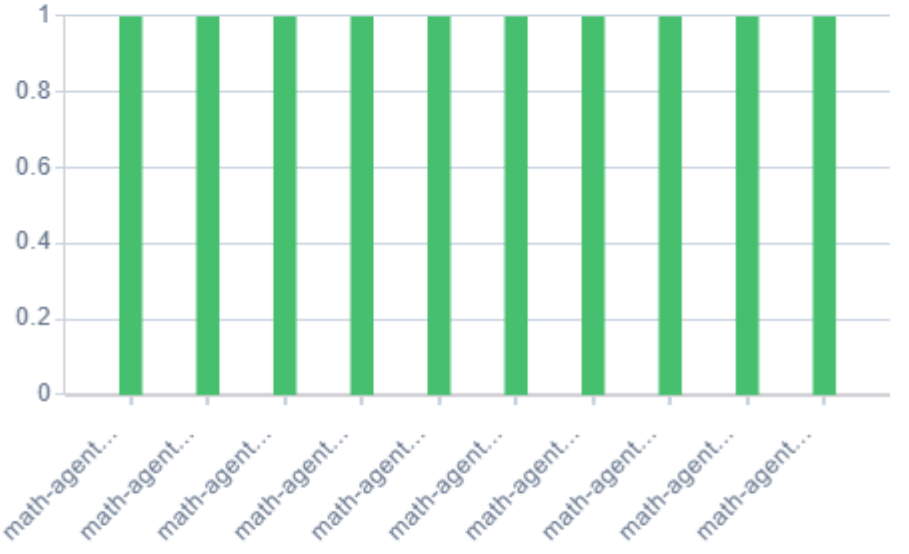


PERFORMANCE

se_length



Tool_usage



Splits



Repetitions

Correctness ↑↓

Helpfulness_llm_judge ↑↓

Response_length ↑↓

Tool_usage ↑↓

Created At ↓

icc... you...	base	1	0.75	0.63	1.00	1.00	10/24/2025, 4:08
e4... ise...	base	1	0.75	0.38	0.93	1.00	10/24/2025, 4:08
8b...	base	1	0.75	0.38	0.93	1.00	10/24/2025, 4:08

How Do We Capture Agent Trajectories in Math Chatbot?

- Build trajectory tracking into your agent by appending each step (user message, LLM response, tool call) to an array.
- Include step type, content, tool names, arguments, and results for complete execution history.
- Return trajectory and extracted tool_calls in the agent output for evaluators to analyze.

```

trajectory = []

# Log user message
trajectory.append({
    "step": "initial",
    "type": "user_message",
    "content": question
})

# In agent loop:
for iteration in range(max_iterations):
    response = llm_with_tools.invoke(messages)
    # Log LLM response
    trajectory.append({
        "step": f"llm_response_{iteration}",
        "type": "ai_message",
        "content": response.content,
        "tool_calls": len(response.tool_calls) if hasattr(response, 'tool_calls') else 0
    })
    # Log tool executions
    if response.tool_calls:
        for tool_call in response.tool_calls:
            tool_result = tools_by_name[tool_call["name"]].invoke(tool_call["args"])

            trajectory.append({
                "step": f"tool_call_{iteration}",
                "type": "tool_call",
                "tool": tool_call["name"],
                "args": tool_call["args"],
                "result": tool_result
            })

# Return trajectory with final answer
return {
    "answer": final_answer,
    "trajectory": trajectory,
    "tool_calls": [t for t in trajectory if t["type"] == "tool_call"]
}

```

What are Best Practices for Evaluation?

- Use multiple evaluator types (rule-based + trajectory + LLM-judge) for comprehensive assessment.
- Always establish a baseline before making changes and run on the same dataset for fair comparison.
- Include edge cases in datasets (conversational queries, error conditions) to test boundary behavior.

```
# Best practice: Multiple evaluator types
evaluators = [
    correctness_evaluator,      # Rule-based (exact match)
    tool_usage_evaluator,       # Trajectory (action validation)
    llm_judge_helpfulness,      # LLM-as-judge (quality)
    response_length_evaluator,  # Heuristic (format check)
]

# Best practice: Baseline first
baseline = evaluate(
    current_agent,
    data=dataset_name,
    evaluators=evaluators,
    experiment_prefix="baseline",
    metadata={"version": "v1.0"}
)

# Best practice: Edge cases in dataset
examples = [
    {"inputs": {"question": "What is 5 + 3?"}, ...}, # Normal case
    {"inputs": {"question": "Hello!"}, ...},        # No tool needed
    {"inputs": {"question": "10 / 0"}, ...},         # Error case
    {"inputs": {"question": "Calculate 2+3*4"}, ...}, # Order of operations
]
```

How Do We Optimize Evaluation Performance?

- Use `max_concurrency` to process multiple examples in parallel (2-4 for safety, up to 10 for speed).
- Skip expensive LLM-as-judge evaluators during rapid iteration, add them back for final validation.
- Consider using smaller dev datasets (10 examples) for quick testing before running full evaluation.

How Do We Optimize Evaluation Performance?

```
# Fast evaluation with higher concurrency
results = evaluate(
    agent,
    data=dataset_name,
    evaluators=[
        correctness_evaluator,      # Fast ⚡
        tool_usage_evaluator,        # Fast ⚡
        # llm_judge_helpfulness,     # Slow 🐌 - skip for iteration
        response_length_evaluator,   # Fast ⚡
    ],
    max_concurrency=10, # Higher for speed
)

# Use dev/prod dataset split
dev_dataset = "Math Calculator QA - Dev"      # 10 examples
prod_dataset = "Math Calculator QA"           # 100 examples

# Iterate on dev
dev_results = evaluate(agent, data=dev_dataset, evaluators=fast_evaluators)

# Final validation on prod
prod_results = evaluate(agent, data=prod_dataset, evaluators=all_evaluators)
```

What is the Complete Evaluation Workflow?

- Create dataset → Define evaluators → Run baseline → Make improvements → Run new experiment → Compare.
- The iterative cycle of test-measure-improve leads to more reliable and capable AI systems over time.
- LangSmith tracks everything: experiments, traces, metrics, and metadata for reproducibility and collaboration.

```
# Complete evaluation workflow
from langsmith import Client
from langsmith.evaluation import evaluate

# 1. Create dataset
client = Client()
dataset = client.create_dataset("Math Calculator QA")
client.create_examples(inputs=[...], outputs=[...], dataset_id=dataset.id)

# 2. Define evaluators
evaluators = [correctness, tool_usage, helpfulness, conciseness]

# 3. Run baseline
baseline = evaluate(
    agent_v1,
    data=dataset,
    evaluators=evaluators,
    experiment_prefix="baseline"
)

# 4. Make improvements to your agent
# ... modify prompts, add tools, tune parameters ...

# 5. Run new experiment
improved = evaluate(
    agent_v2,
    data=dataset, # Same dataset
    evaluators=evaluators, # Same evaluators
    experiment_prefix="improved"
)

# 6. Compare results in UI
print(f"Baseline: {baseline.experiment_name}")
print(f"Improved: {improved.experiment_name}")
# Navigate to LangSmith UI → Select both → Compare
```

What Key Insights Did We Learn?

- Systematic evaluation reveals objective trade-offs: formal prompts are concise but less engaging.
- Multiple evaluator types catch different issues: correctness, tool usage, helpfulness, and format.
- The Math Calculator QA system demonstrates all concepts: datasets, tools, trajectories, evaluators, A/B testing.

Summary: Key Concepts Demonstrated

1. DATASETS

- Created 'Math Calculator QA' with 4 examples
- Includes inputs, expected outputs, and metadata

2. TOOL USAGE & TRAJECTORY

- Defined 3 calculator tools (add, multiply, divide)
- Tracked tool calls in agent trajectory
- Evaluated correct tool usage

3. A/B TESTING

- Variant A: Formal system prompt
- Variant B: Friendly system prompt
- Ran separate experiments for each variant

Summary: Key Concepts Demonstrated

4. MULTIPLE EVALUATORS

- Correctness: String match for numerical answers
- Tool Usage: Trajectory evaluation
- LLM-as-Judge: Helpfulness assessment
- Response Length: Conciseness check

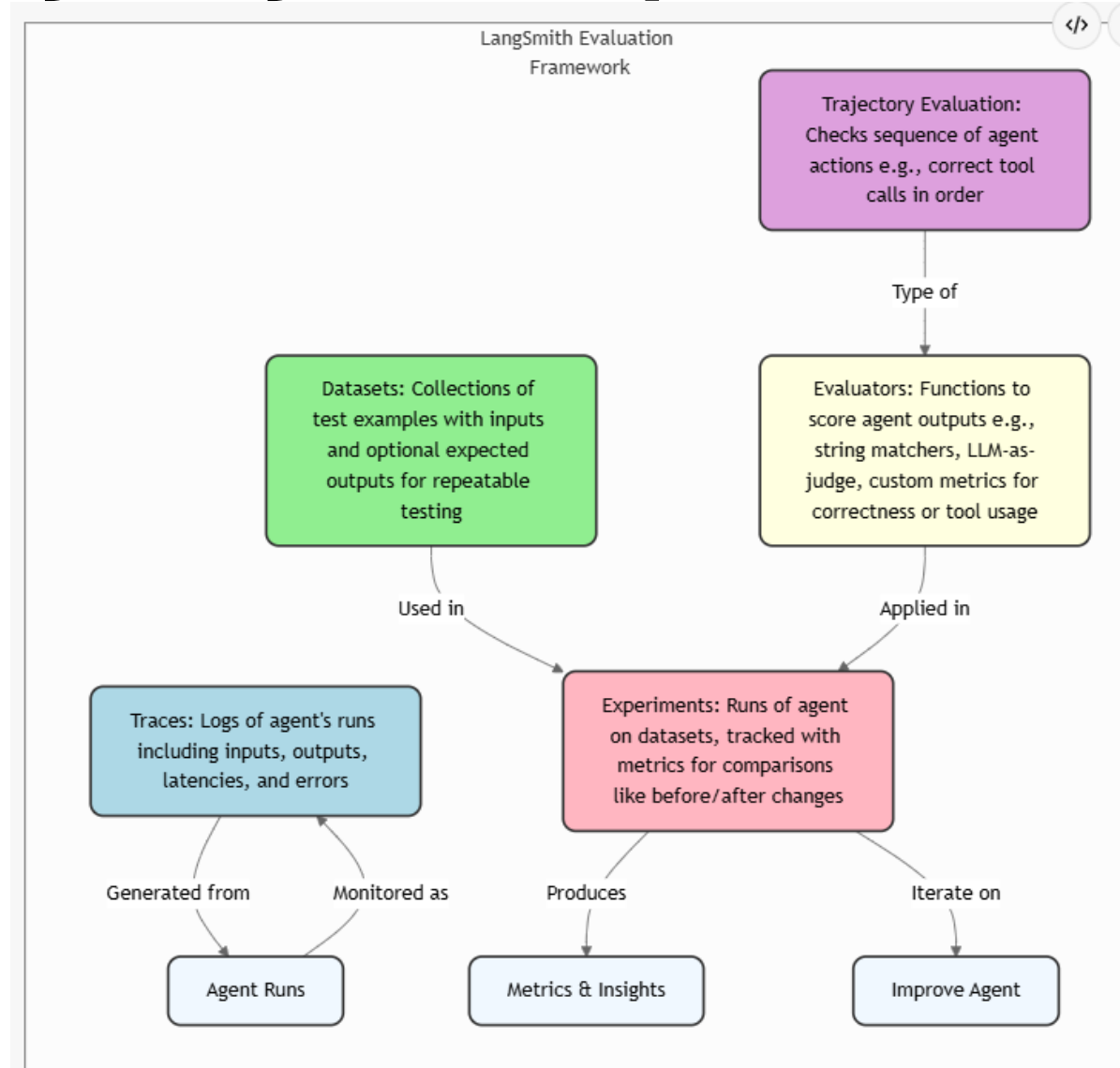
5. EXPERIMENTS

- Two independent experiments (A and B)
- Metadata tracking for comparison
- Full traces and metrics in LangSmith UI

6. RESULTS

- Formal: More concise, faster
- Friendly: More helpful, engaging
- Both: Equally correct, perfect tool usage

Summary: Key Concepts Demonstrated



Colab

https://github.com/enoch-sit/langsmithdemo/blob/main/Tutorial_LangSmith_Evaluation.ipynb