Temporal

Foundations of GenAI Applications and Temporal

JOIN OUR SLACK CHANNEL

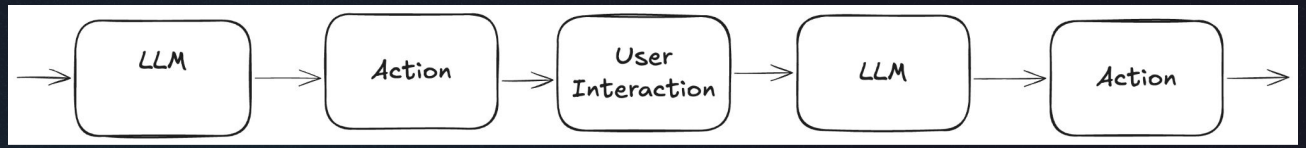
Feel free to ask questions or go there for helpful links!

https://t.mp/nov-18-slack-channel

WHERE WE ARE GOING

- ✅ Foundations of GenAI applications
- ✅ Making GenAI applications durable
- ✅ Humans are still needed
- ✅ Durable AI Agents - putting it all together

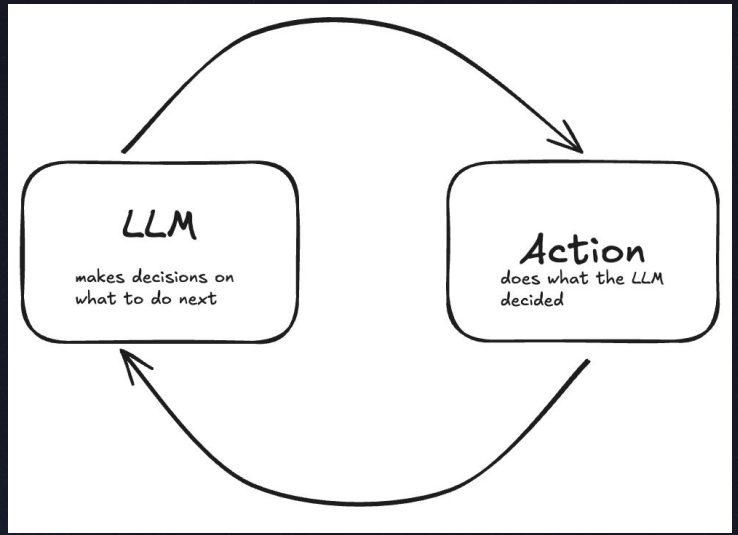
WHAT ARE GENAI APPLICATIONS



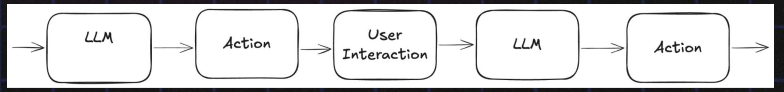
- An application that leverages an LLM to implement a part of the application's functionality
- They can be fixed flow 👆, or...

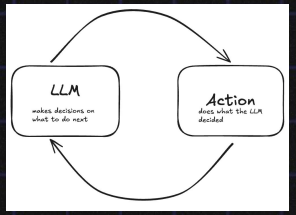
WHAT ARE GENAI APPLICATIONS

- … they may be agentic
- AI agents are GenAI applications where the **LLM has agency** over the functionality and flow of the application.



TWO PRIMARY PIECES





✅ The LLM

✅ What we do with the LLM outputs

Includes calling APIs (aka Tools)

✅ Characteristics of these types of applications ...

✅ ... the challenges these pose.

LET'S PROMPT THE LLM

APIs are used to supply an LLM with context and get back a response

```python
from litellm import completion, ModelResponse

def llm_call(prompt: str, llm_api_key: str, llm_model: str) -> ModelResponse:
    response = completion(
      model=llm_model,
      api_key=llm_api_key,
      messages=[{ "content": prompt,"role": "user"}]
    )
    return response

prompt = # TODO Add a prompt here to call your LLM
result = llm_call(prompt, LLM_API_KEY, LLM_MODEL)["choices"][0]["message"]
["content"]

print(result)
```

NOTEBOOK SETUP

Follow along and play with our samples by accessing our Jupyter Notebooks here:

t.mp/temporal-ai-foundations-workshop

LET'S PROMPT THE LLM

APIs are used to supply an LLM with context and get back a response

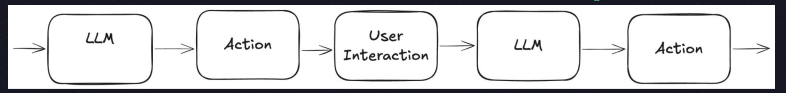
```python
from litellm import completion, ModelResponse

def llm_call(prompt: str, llm_api_key: str, llm_model: str) -> ModelResponse:
    response = completion(
      model=llm_model,
      api_key=llm_api_key,
      messages=[{ "content": prompt,"role": "user"}]
    )
    return response

prompt = # TODO Add a prompt here to call your LLM
result = llm_call(prompt, LLM_API_KEY, LLM_MODEL)["choices"][0]["message"]
["content"]

print(result)
```

NOW, LET'S PROCESS THE LLM RESULT

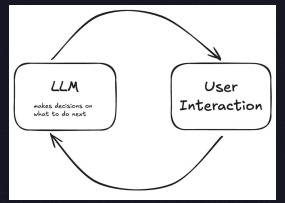
So far, we've only returned the response from the LLM to the user.

We want to do more.

- Pass results into a "next step"



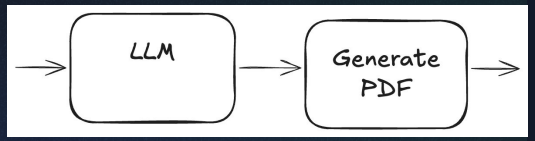
- Add results to conversation history



- Parse the results (LLM are good at parsing 🤔)
- ...

FOR EXAMPLE...

Let's put our research results into a PDF




```python
# Step 1: Call the LLM.
# Step 2: Pass LLM results to PDF generator

# Make the API call
print("Welcome to the Research Report Generator!")
prompt = input("Enter your research topic or question: ")
result = llm_call() # TODO: Call the `llm_call` function with `prompt`, `LLM_API_KEY`, and `LLM_MODEL` as arguments.

# Extract the response content
response_content: str = result["choices"][0]["message"]["content"]

pdf_filename = create_pdf('', "research_report.pdf") # Call the `create_pdf` function with `response_content` as the first argument.
print(f"SUCCESS! PDF created: {pdf_filename}")
```

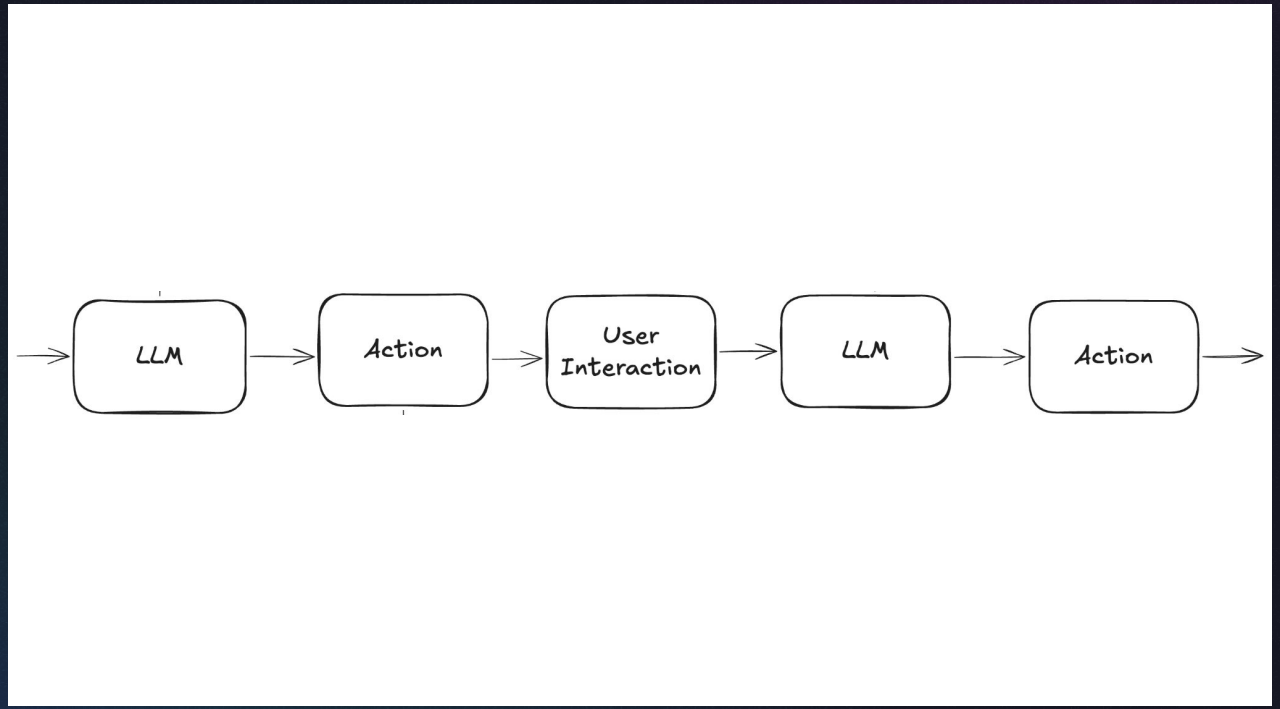
DEMO TIME!

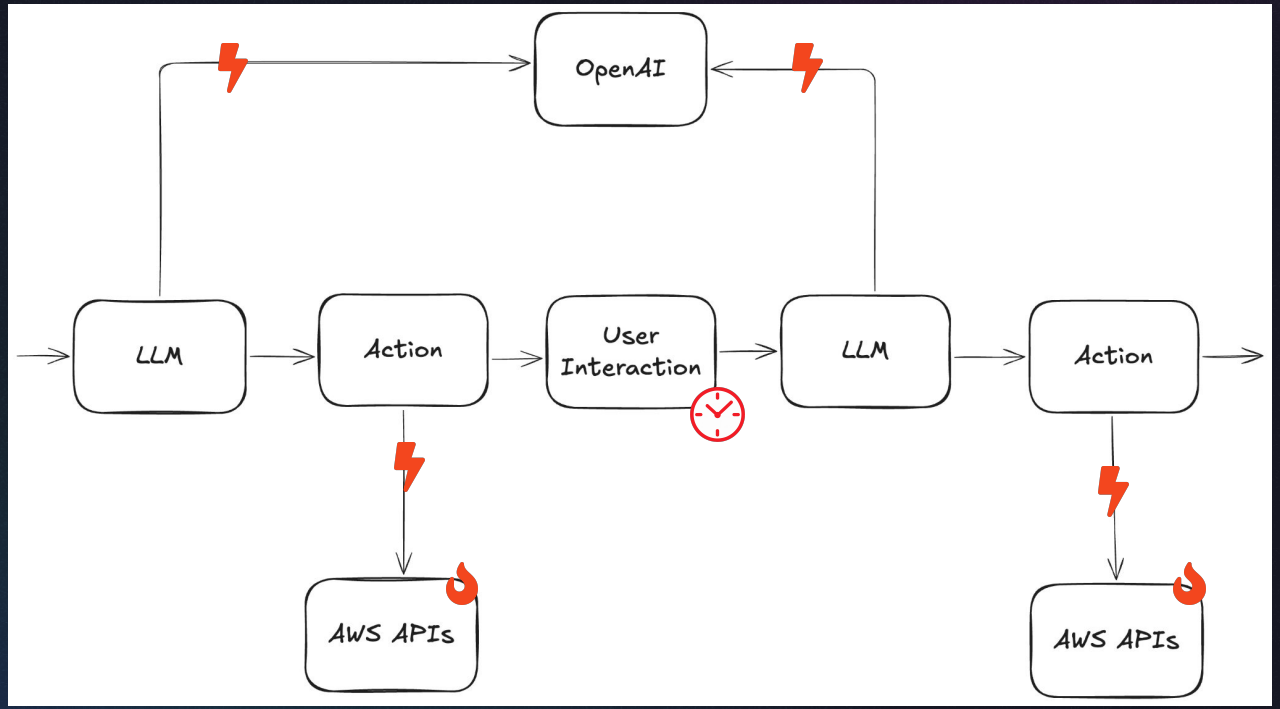
CHALLENGES OF GENAI APPLICATIONS

- Networks can be flakey
- LLMs are often rate limited
- Tool resources (APIs and databases) go down
- LLMs are inherently non-deterministic
- How do we scale these applications?
- What happens when they take a long time to finish?

...

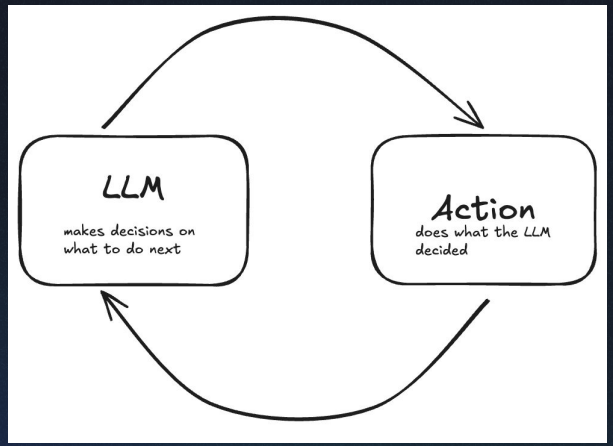
- What else?

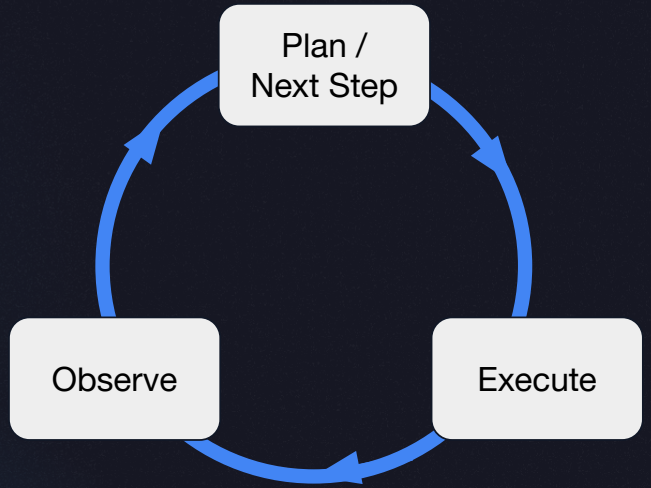

LLM
Action
User Interaction
LLM
Action

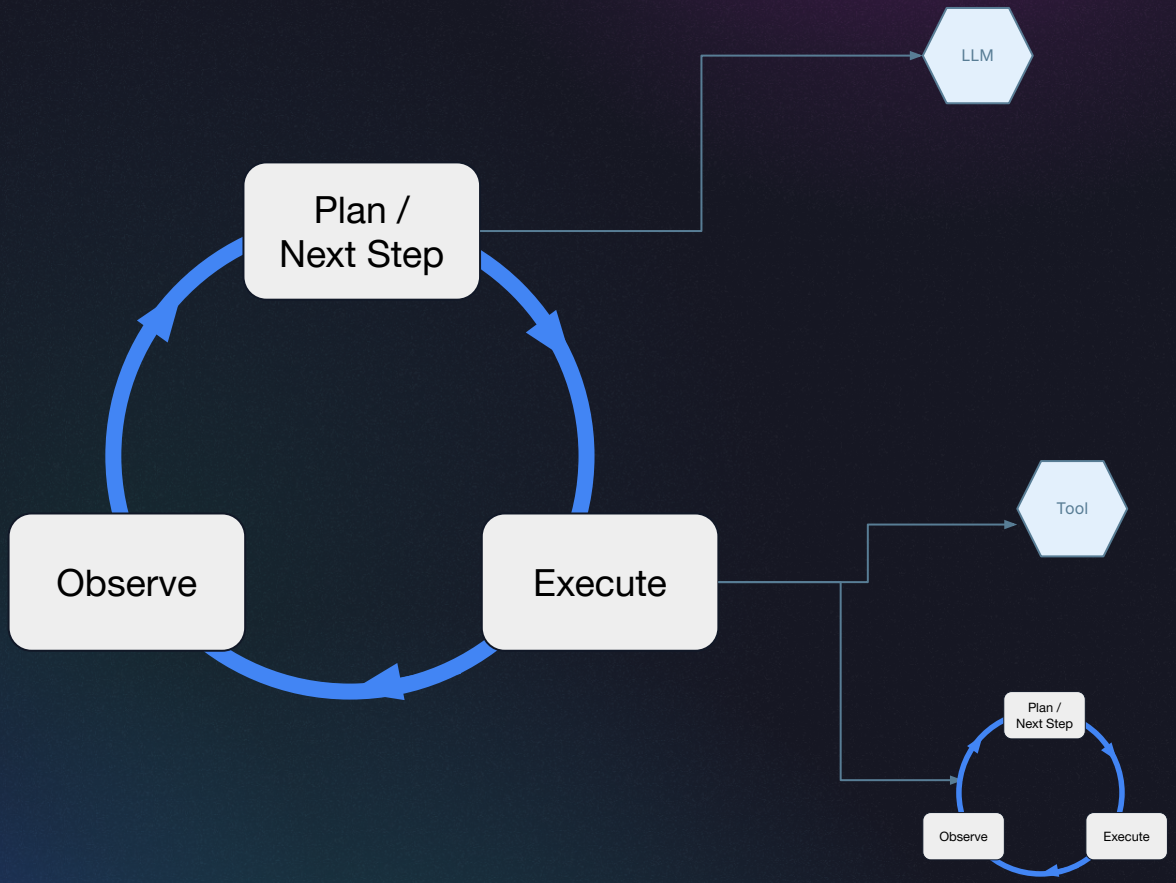

OpenAI
LLM
Action
User Interaction
LLM
Action
AWS APIs
AWS APIs

AGENTS ARE DISTRIBUTED SYSTEMS

THE CHALLENGES ONLY AMPLIFY WITH AGENTS






LLM
Plan /
Next Step
Observe
Execute
Tool
Plan /
Next Step
Observe
Execute


Plan / Next Step
Observe
Execute
LLM
Tool
Plan / Next Step
Observe
Execute
LLM
Tool
Plan / Next Step
Observe
Execute
LLM
Tool
Plan / Next Step
Observe
Execute
LLM
Tool
Plan / Next Step
Observe
Execute


Plan / Next Step
Observe
Execute
LLM
Tool
Plan / Next Step
Observe
Execute
LLM
Tool
Plan / Next Step
Observe
Execute
LLM
Tool
LLM
Plan / Next Step
Observe
Execute
Tool
Plan / Next Step
Observe
Execute

Temporal

Adding Durability

IN THIS SECTION:

- ✅ Introduce Temporal
- ✅ Primary Temporal Abstractions: Activities and Workflows
- ✅ Demonstrate the benefits of durable execution
- ✅ Event-driven Architectures

CHALLENGES OF GENAI APPLICATIONS

- Networks can be flakey
- LLMs are often rate limited
- Tool resources (APIs and databases) go down
- LLMs are inherently non-deterministic
- How do we scale these applications?
- What happens when they take a long time to finish?

...

- What else?

WHAT NORMAL EXECUTION GIVES US

- Every failure means restarting from scratch
- Expensive LLM calls may be repeated unnecessarily
- User experience becomes frustrating and unreliable

WHAT DEVELOPERS ACTUALLY WANT

- “Just fix the disk issue and generate the PDF from the research you already have.”
- “Don’t make me pay for the same LLM call twice!”
- “Don’t lose my work because of a simple file system error!”

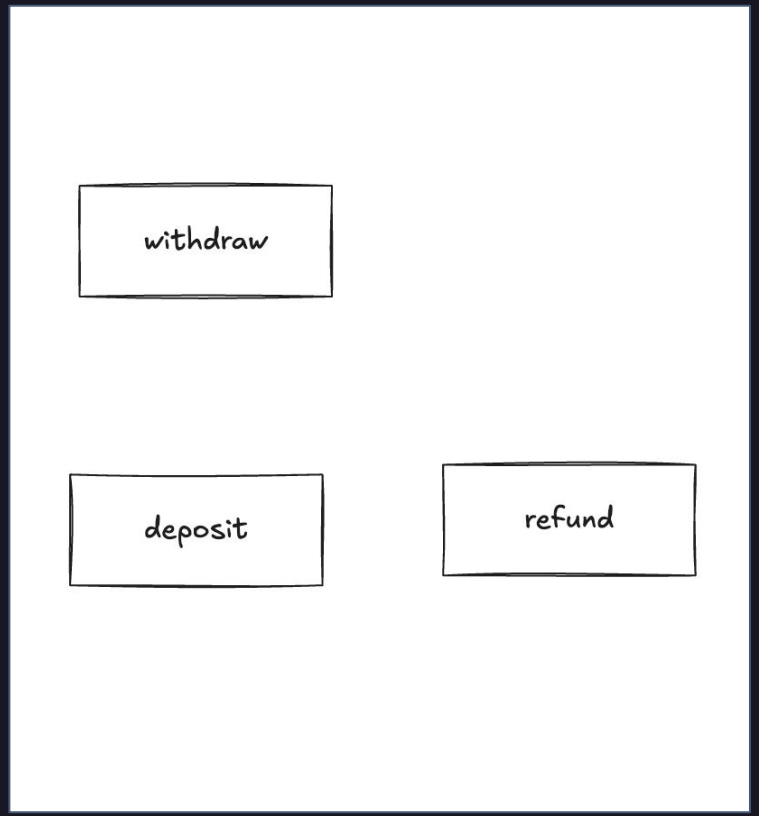
Introducing Temporal

- Technology and open source project that delivers resilience for distributed systems in a novel way.
- Supports a programming model that allows developers to code the **happy path**, while the platform provides services that compensate for a wide range of distributed system failures.
- Platform comes in the form of a service + SDKs

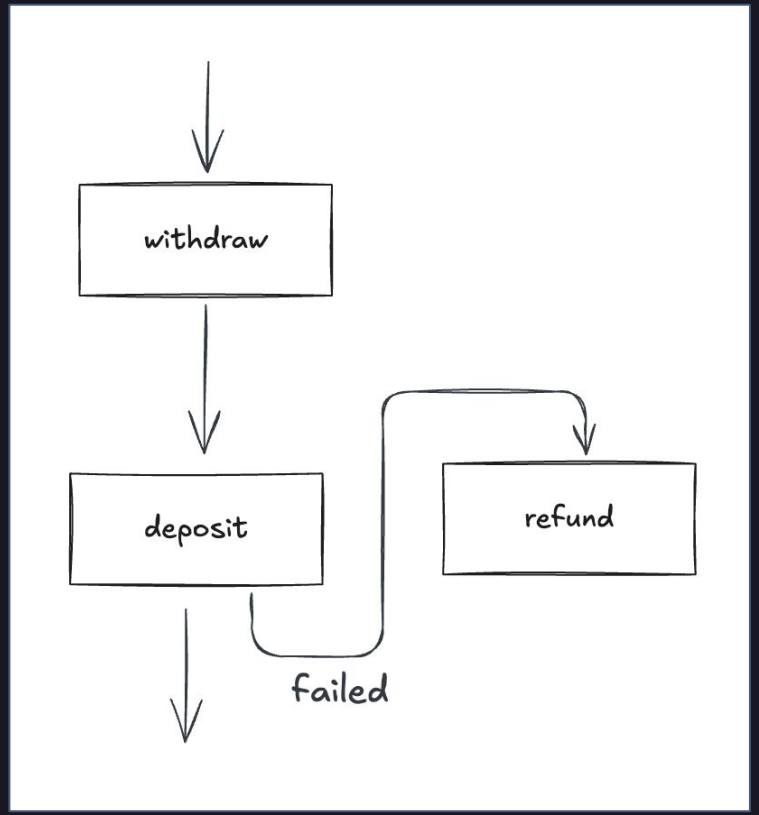
  SDK is available for Go, Java, Python, PHP, Typescript, .Net, Ruby

Temporal

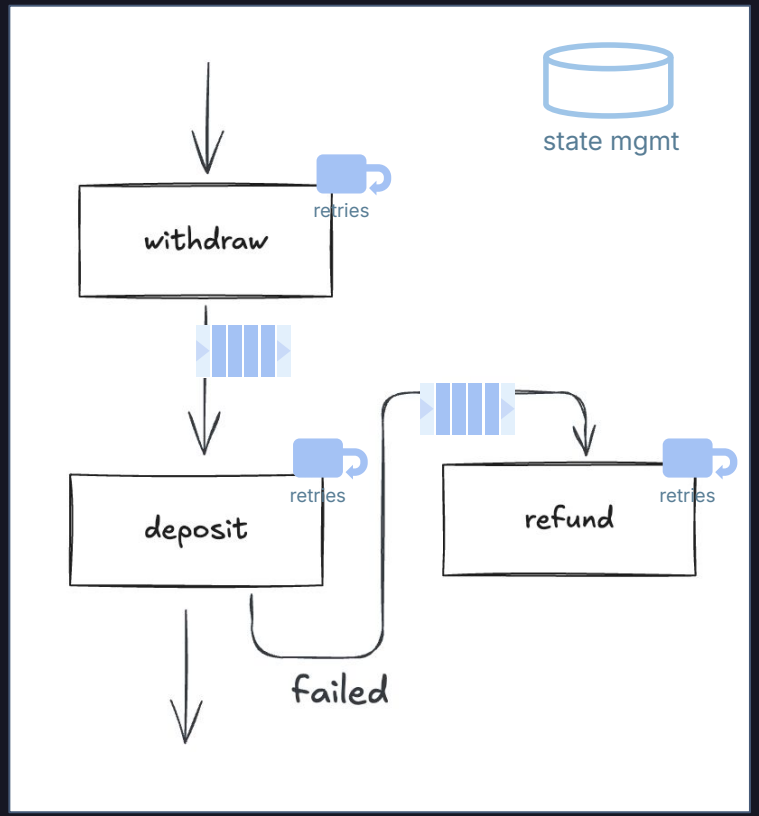
Temporal Activities

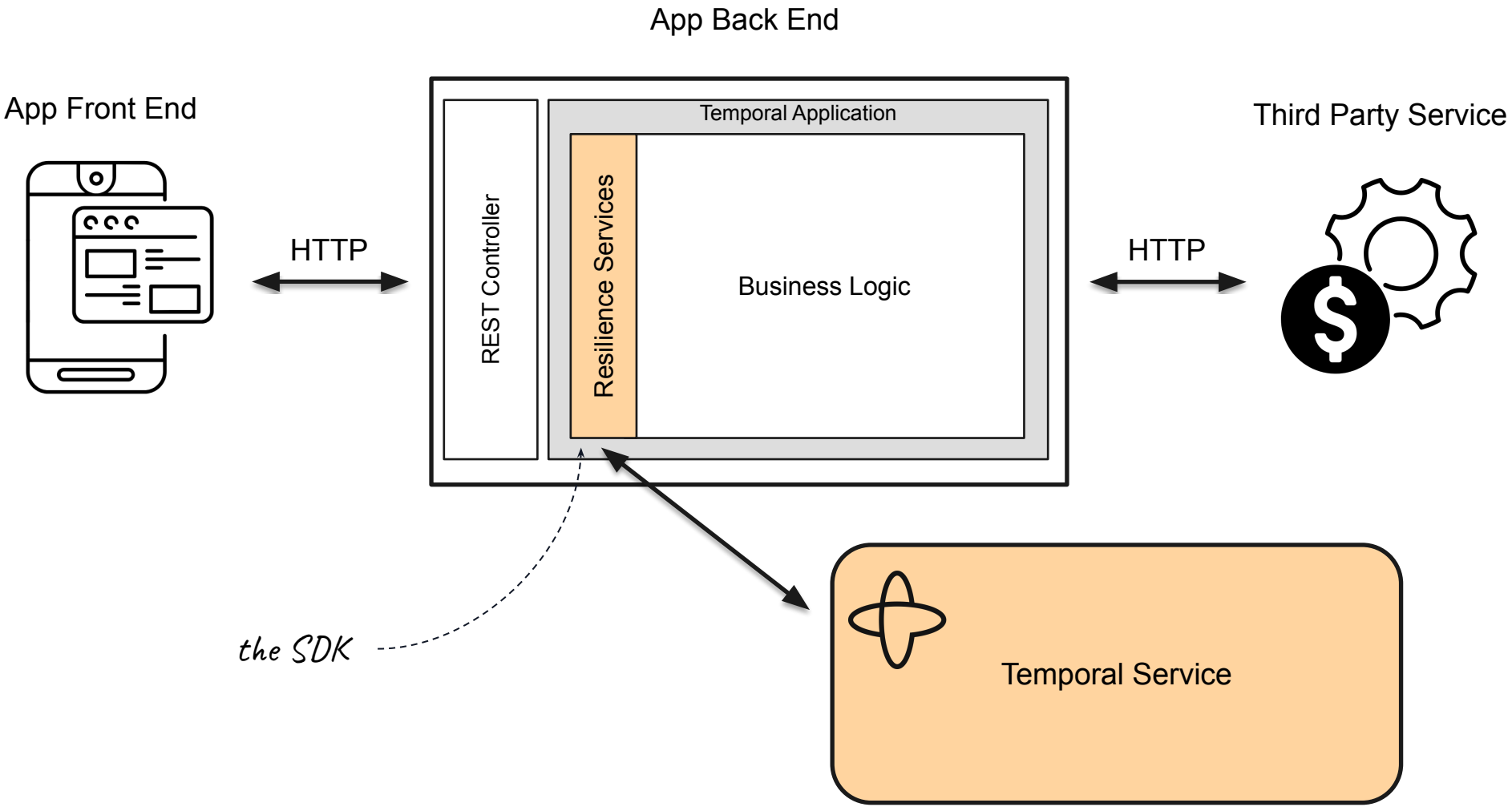


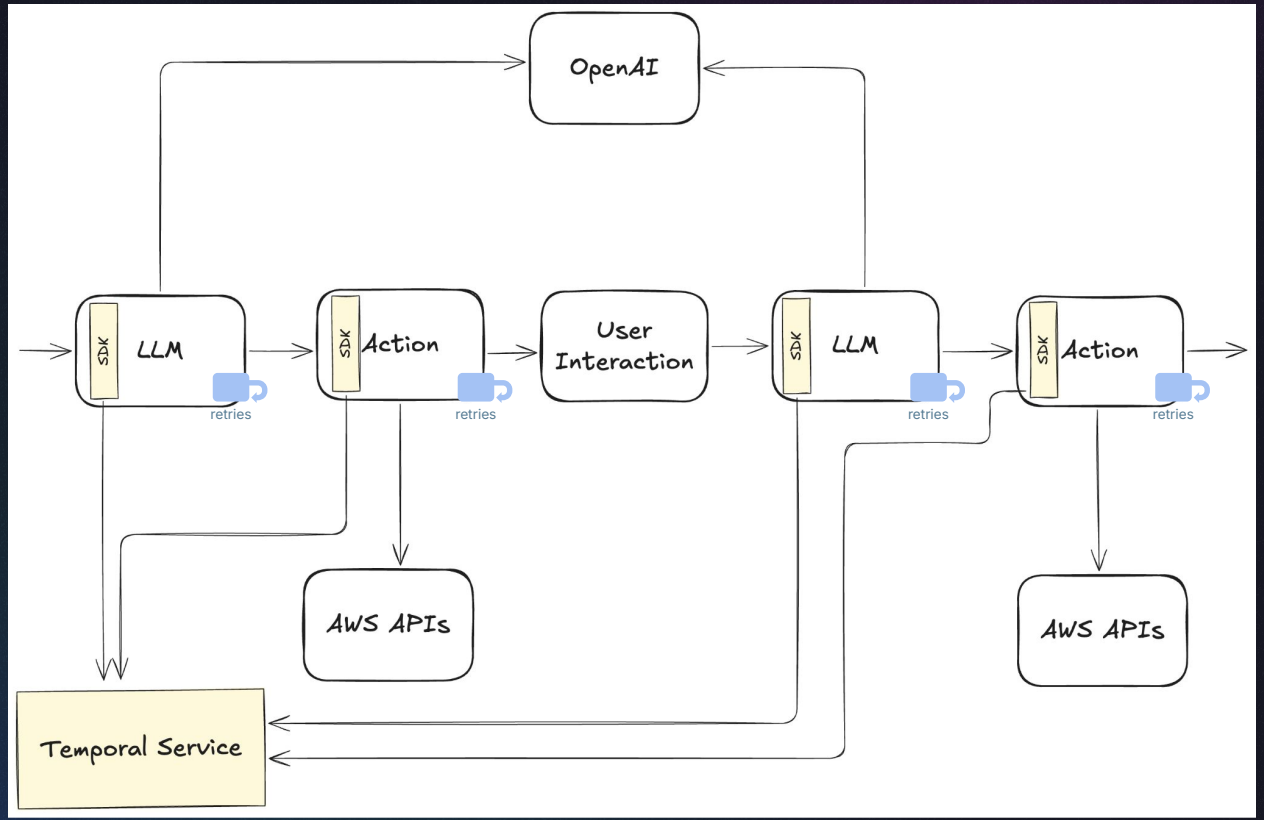
Temporal Workflows



Temporal Workflows and Activities = 🪄




App Back End
App Front End
HTTP
REST Controller
Temporal Application
Resilience Services
Business Logic
HTTP
Third Party Service
the SDK
Temporal Service


OpenAI
SDK
LLM
retries
SDK
Action
retries
User Interaction
SDK
LLM
retries
SDK
Action
retries
AWS APIs
AWS APIs
Temporal Service

WHAT STAYS THE SAME

- Your core business logic (LLM Call -> PDF generation)
- Inputs and outputs

WHAT WE UPDATE

- Put things into Temporal Activities and Workflows
- Connect to the Temporal Service (via SDKs)

WHAT ACTIVITIES GIVE YOU

- Automatic retries
- Timeout handling
- Detailed visibility
- Automatic checkpoints

WHAT WORKFLOWS GIVE YOU

- Event history
- Pick up exactly where you left off
- Long running
- HITL
- ...

FOR ACTIVITIES

- Add the `@activity.defn` decorator
- Package activity arguments into a data structure

FOR WORKFLOWS

- Add the `@workflow.defn` decorator
- Configure activity durability

LET'S CREATE ACTIVITIES

Functions that are making external calls are "wrapped" as activities

```python
from temporalio import activity
from litellm import completion, ModelResponse
from dataclasses import dataclass

@dataclass
class LLMCallInput:
  prompt: str
  llm_api_key: str
  llm_model: str

@activity.defn
def llm_call(input: LLMCallInput) -> ModelResponse:
    response = completion(
      model=input.llm_model,
      api_key=input.llm_api_key,
      messages=[{ "content": input.prompt,"role": "user"}]
    )
    return response
```

ACTIVITIES ARE CALLED FROM WORKFLOWS

- Activities are orchestrated within a Temporal Workflow
- Workflows contain the decision-making flow but Activities perform the actual work
- Each Activity is recorded in the History with inputs and outputs
- Workflows can wait for Activity completion, handle failures, make decisions based on results

CREATING THE WORKFLOW

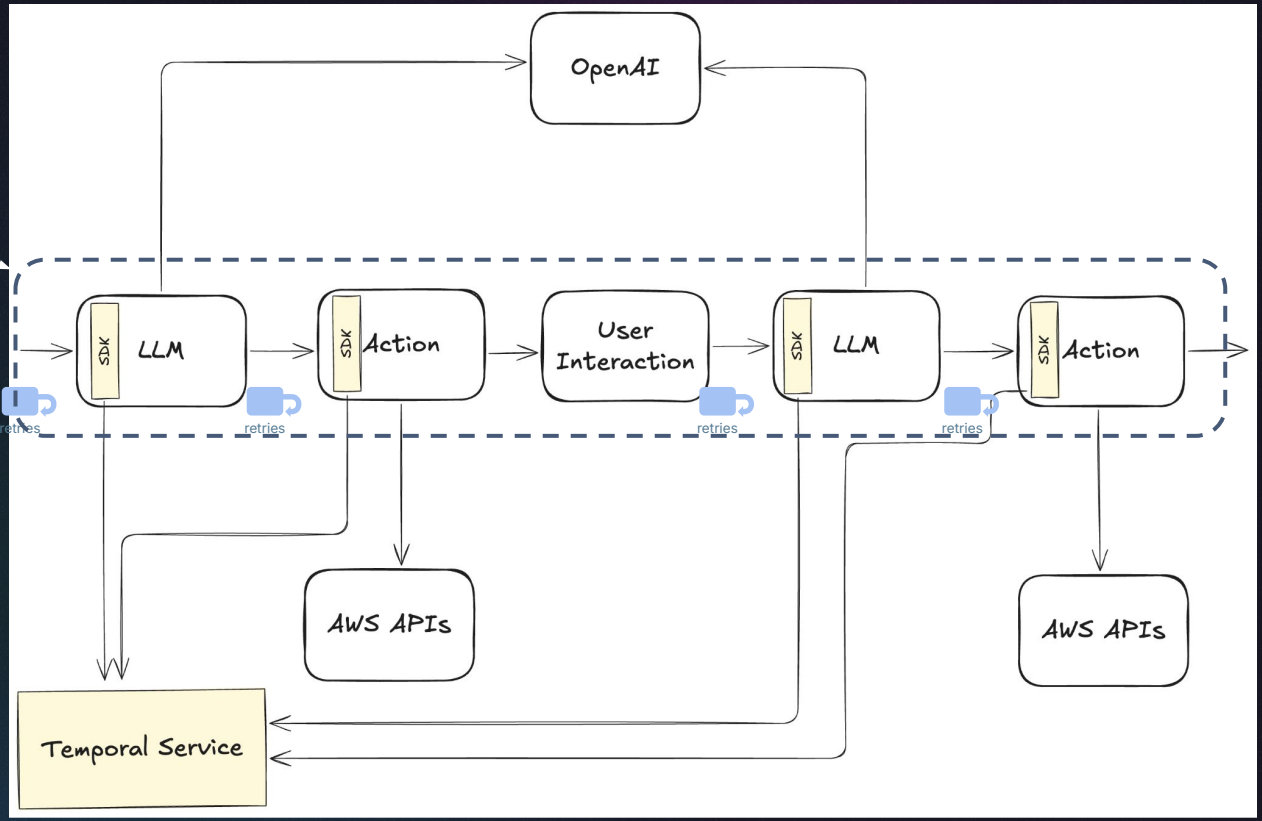
- Workflows must not make API calls, file system calls, or anything non-deterministic.

LET'S ORCHESTRATE WITH A WORKFLOW

```python
@workflow.defn(sandboxed=False)
class GenerateReportWorkflow:

    @workflow.run
    async def run(self, input: GenerateReportInput) -> GenerateReportOutput:

        llm_call_input = LLMCallInput(
            prompt=input.prompt, llm_api_key=LLM_API_KEY, llm_model=LLM_MODEL)

        research_facts = await workflow.execute_activity(
            llm_call,
            llm_call_input,
            start_to_close_timeout=timedelta(seconds=30))

        workflow.logger.info("Research complete!")

        pdf_generation_input = PDFGenerationInput(
            content=research_facts["choices"][0]["message"]["content"])

        pdf_filename = await workflow.execute_activity(
            create_pdf_activity,
            pdf_generation_input,
            start_to_close_timeout=timedelta(seconds=10))

        return GenerateReportOutput(
            result=f"Research report PDF: {pdf_filename}")
```

OKAY, SO THERE'S ONE MORE THING...

We've not talked about how these things run



TEMPORAL WORKERS

- Temporal Workflows and Activities are run in Workers
- Workers wait for tasks to do and execute them

RUNNING A WORKER

- Workers have Workflows and Activities registered to them so the Worker knows what to execute

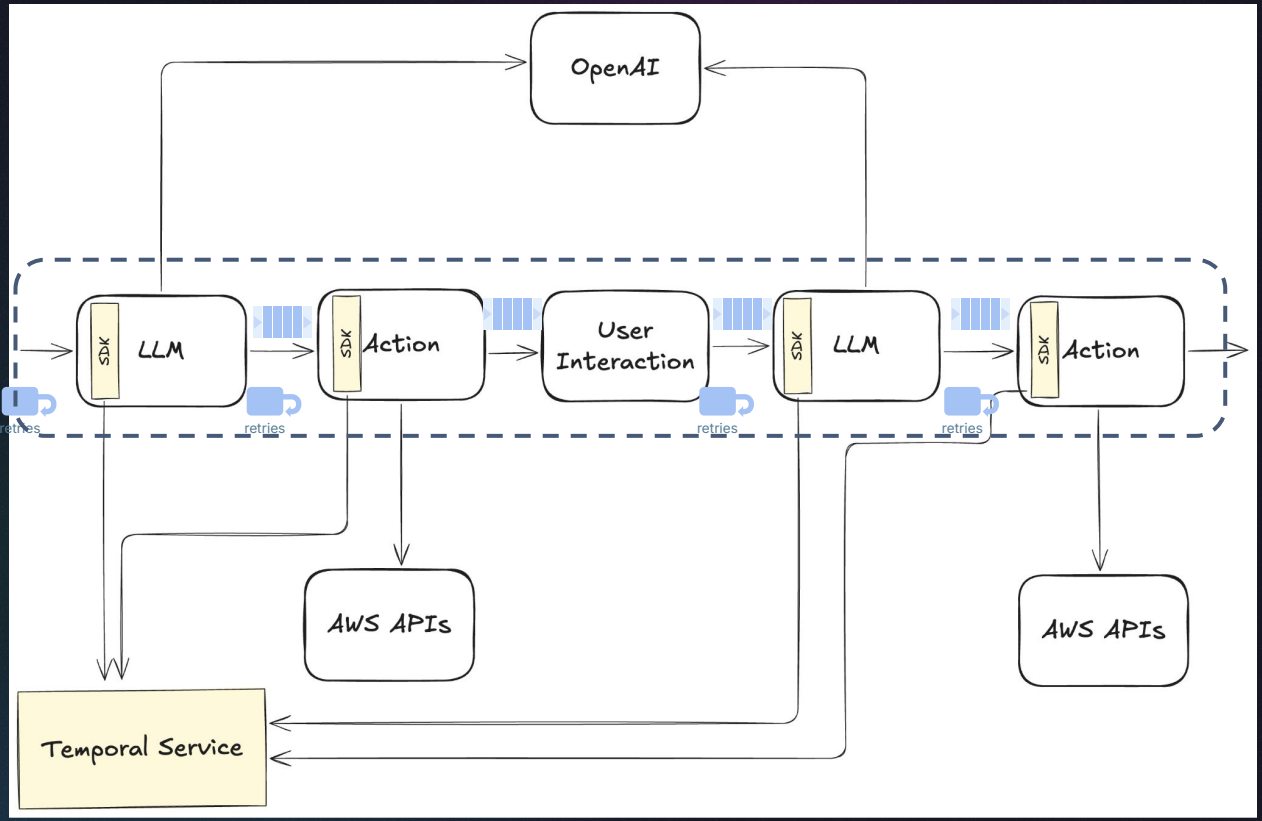
```python
from temporalio.client import Client
from temporalio.worker import Worker
import concurrent.futures

async def run_worker() -> None:
    # Create client connected to server at the given address
    client = await Client.connect("localhost:7233", namespace="default")

    # Run the Worker
    with concurrent.futures.ThreadPoolExecutor(max_workers=100) as activity_executor:
        worker = Worker(
            client,
            task_queue="research", # the task queue the Worker is polling
            workflows=[GenerateReportWorkflow], # register the Workflow
            activities=[llm_call, create_pdf_activity], # register the Activities
            activity_executor=activity_executor
        )

        print(f"Starting the worker....")
        await worker.run()
```

RUNNING A WORKER

- Workers find tasks by listening on a Task Queue
- Any Worker can pick up a registered Workflow or Activity

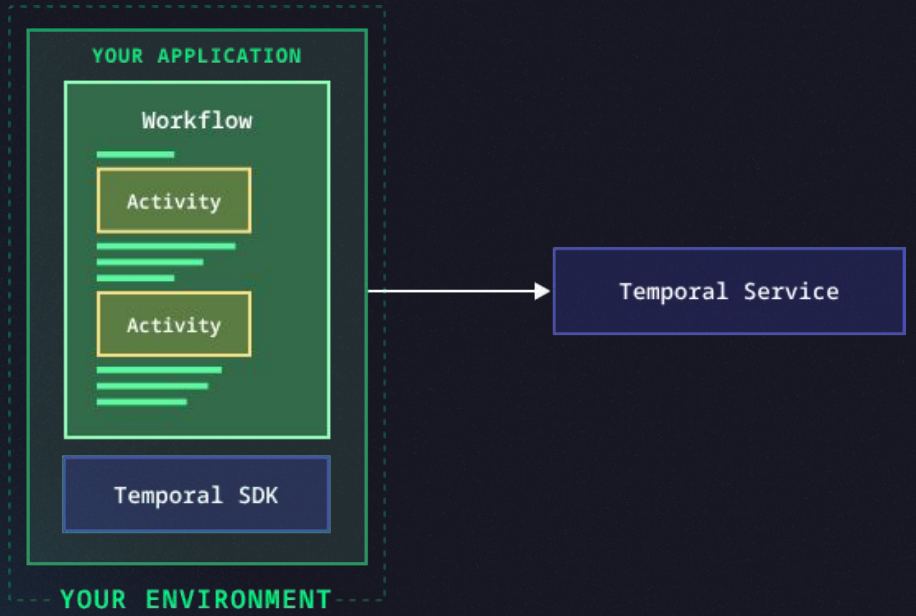
```python
from temporalio.client import Client
from temporalio.worker import Worker
import concurrent.futures

async def run_worker() -> None:
    # Create client connected to server at the given address
    client = await Client.connect("localhost:7233", namespace="default")

    # Run the Worker
    with concurrent.futures.ThreadPoolExecutor(max_workers=100) as activity_executor:
        worker = Worker(
            client,
            task_queue="research", # the task queue the Worker is polling
            workflows=[GenerateReportWorkflow], # register the Workflow
            activities=[llm_call, create_pdf_activity], # register the Activities
            activity_executor=activity_executor
        )

        print(f"Starting the worker....")
        await worker.run()
```

The worker architecture turns your monolith into a modular, event driven application!



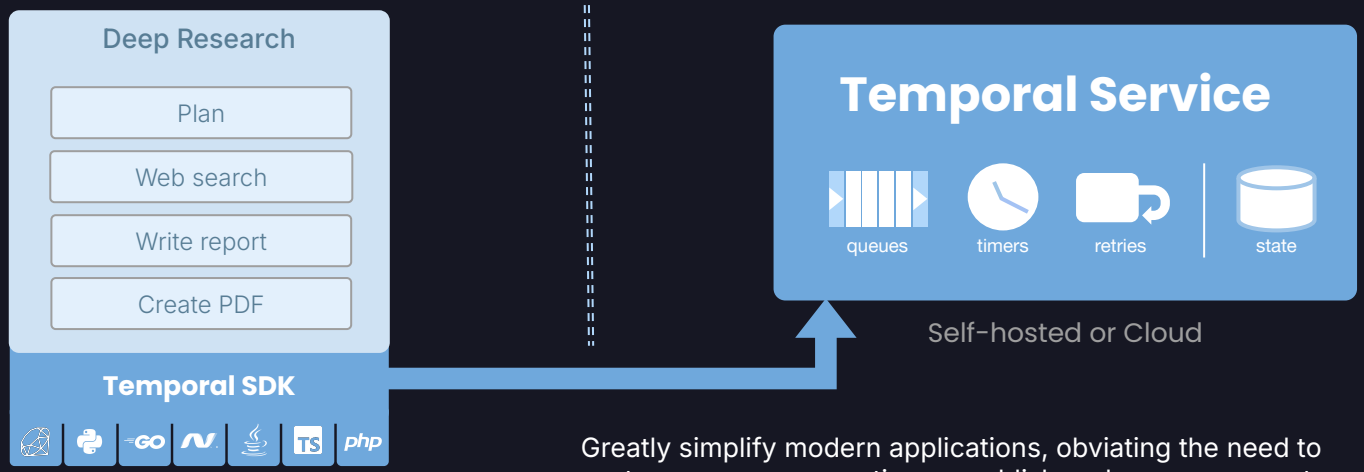
RUNNING A TEMPORAL SERVICE

- The Temporal Service brings it all together
- Can be run locally, self-hosted, or you can use Temporal Cloud
- Acts as the supervisor of your Workflows, Activities, and everything else



DURABLE EXECUTION

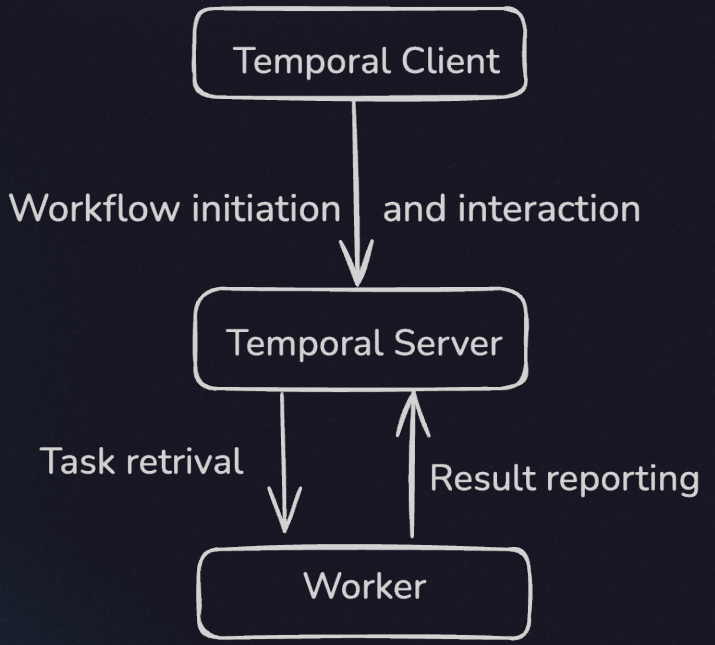
Implemented across the Temporal Service and SDKs



Greatly simplify modern applications, obviating the need to create queues, manage timers, publish and consume events, implement retries and rollbacks, checkpoint state, and more.

EXECUTING THE WORKFLOW

- Temporal Workflows are executed indirectly
- Request execution from the Temporal Service
- You do this with the **Temporal Client**



EXECUTING THE WORKFLOW

```python
import asyncio

from temporalio.client import Client

# Create client connected to server at the given address
client = await Client.connect("localhost:7233", namespace="default")

print("Welcome to the Research Report Generator!")
prompt = input("Enter your research topic or question: ").strip()

if not prompt:
    prompt = "Give me 5 fun and fascinating facts about tardigrades. Make them interesting and
educational!"
    print(f"No prompt entered. Using default: {prompt}")

# Asynchronous start of a Workflow
handle = await client.start_workflow(
    GenerateReportWorkflow.run,
    GenerateReportInput(prompt=prompt),
    id="generate-research-report-workflow", # user-defined Workflow identifier, which typically has
some business meaning
    task_queue="research", # the task-queue that your Worker is polling
)

print(f"Started workflow. Workflow ID: {handle.id}, RunID {handle.result_run_id}")
```

EXECUTING THE WORKFLOW

- Task queue must match exactly Task Queue specified in the Worker

```python
import asyncio

from temporalio.client import Client

# Create client connected to server at the given address
client = await Client.connect("localhost:7233", namespace="default")

print("Welcome to the Research Report Generator!")
prompt = input("Enter your research topic or question: ").strip()

if not prompt:
    prompt = "Give me 5 fun and fascinating facts about tardigrades. Make them interesting and
educational!"
    print(f"No prompt entered. Using default: {prompt}")

# Asynchronous start of a Workflow
handle = await client.start_workflow(
    GenerateReportWorkflow.run,
    GenerateReportInput(prompt=prompt),
    id="generate-research-report-workflow", # user-defined Workflow identifier, which typically has
some business meaning
    task_queue="research", # the task-queue that your Worker is polling
)

print(f"Started workflow. Workflow ID: {handle.id}, RunID {handle.result_run_id}")
```

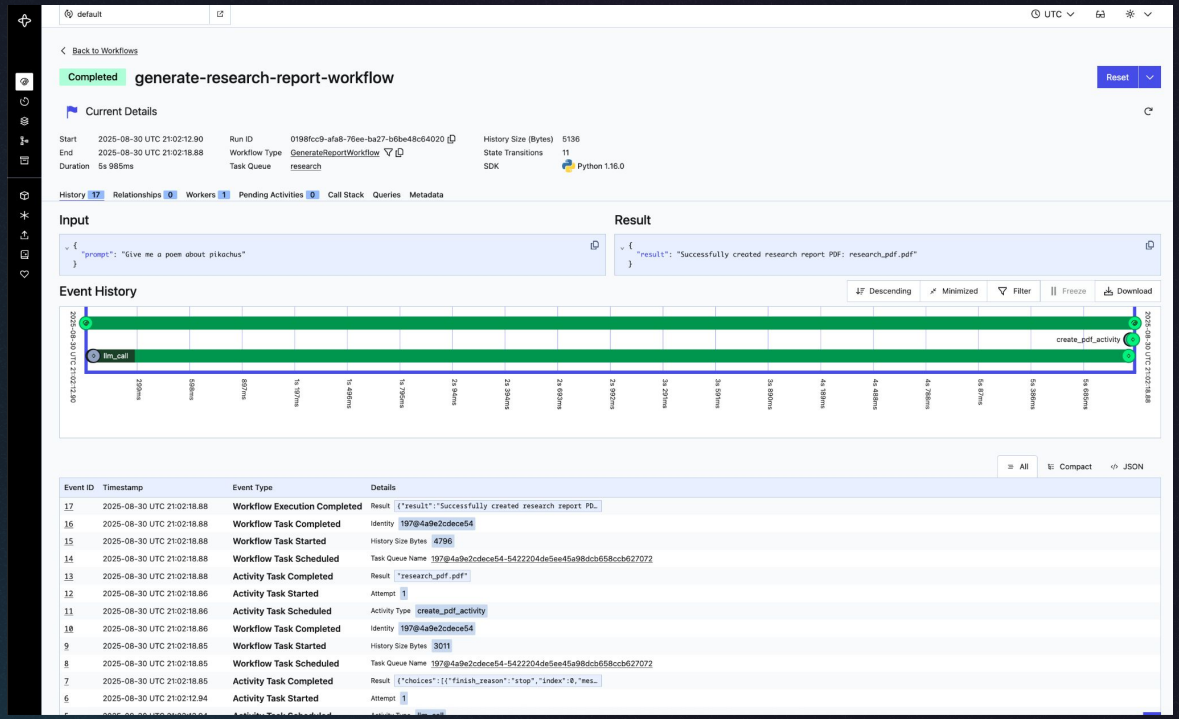
EXECUTING THE WORKFLOW

- Workflows can be started asynchronously or synchronously

```python
import asyncio

from temporalio.client import Client

# Create client connected to server at the given address
client = await Client.connect("localhost:7233", namespace="default")

print("Welcome to the Research Report Generator!")
prompt = input("Enter your research topic or question: ").strip()

if not prompt:
    prompt = "Give me 5 fun and fascinating facts about tardigrades. Make them interesting and
educational!"
    print(f"No prompt entered. Using default: {prompt}")

# Asynchronous start of a Workflow
handle = await client.start_workflow(
    GenerateReportWorkflow.run,
    GenerateReportInput(prompt=prompt),
    id="generate-research-report-workflow", # user-defined Workflow identifier, which typically has
some business meaning
    task_queue="research", # the task-queue that your Worker is polling
)

print(f"Started workflow. Workflow ID: {handle.id}, RunID {handle.result_run_id}")
```

TEMPORAL WEB UI

- Temporal provides a robust Web UI for managing Workflow Executions
- Can gain insights like responses from Activities, execution times, failures
- Great for debugging



DEMO TIME!

THIS IS DURABLE EXECUTION.

WHAT IS DURABLE EXECUTION?

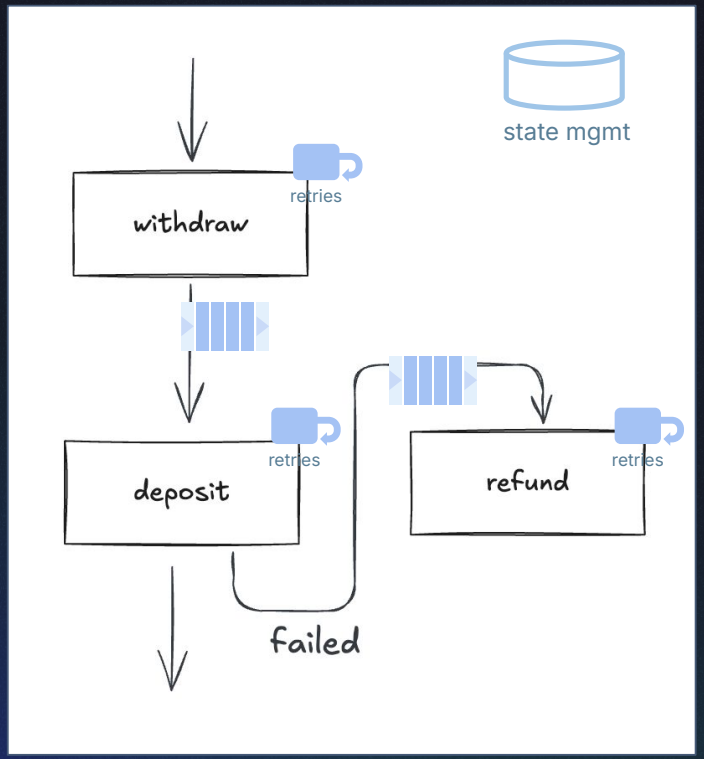
- Crash-proof execution
- Retries upon failure
- Maintains application state, resuming after a crash at the point of failure
- Can run across a multitude of processes, even on different machines

TEMPORAL PROVIDES DURABLE EXECUTION

- Handles state, retries, timeouts, state preservation right out the box
- Open-source MIT licensed
- Code base approach to Workflow design
- Use your own tools, processes, and libraries
- Support for 7 languages

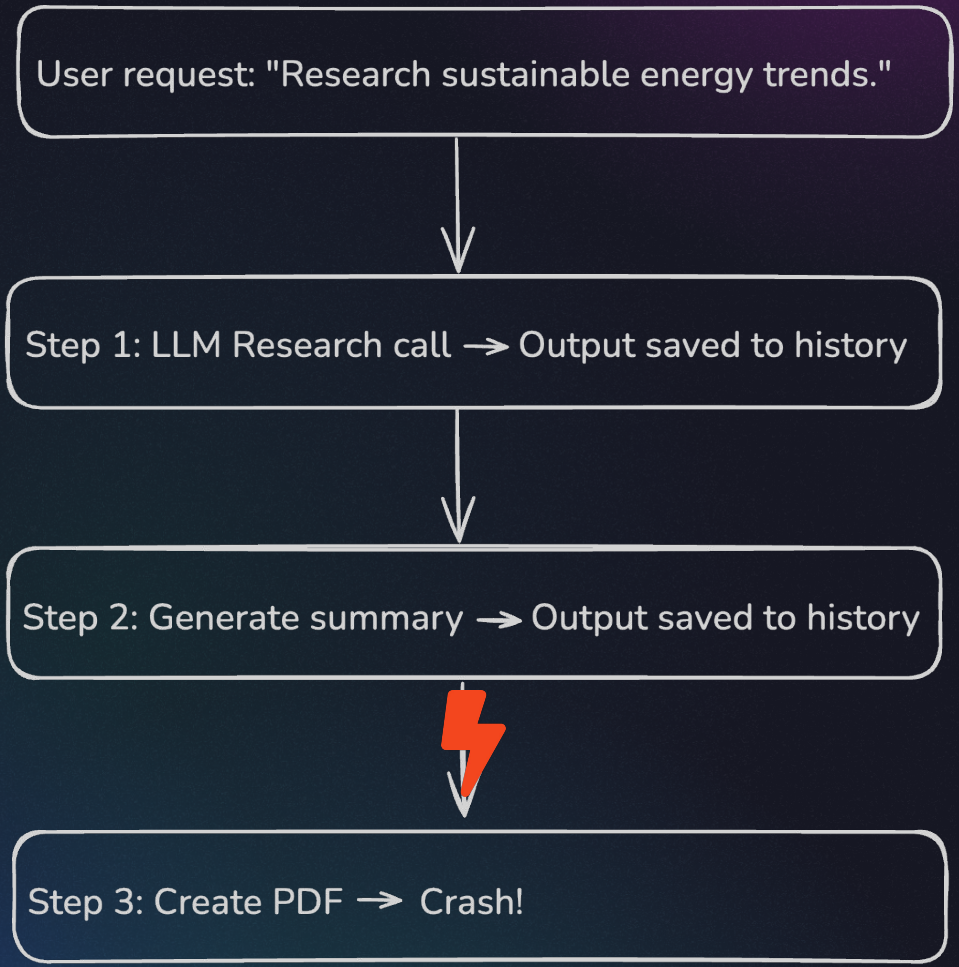
LET'S LOOK AT THAT WHOLE "PICK UP WHERE YOU LEFT OFF" THING, IN MORE DETAIL

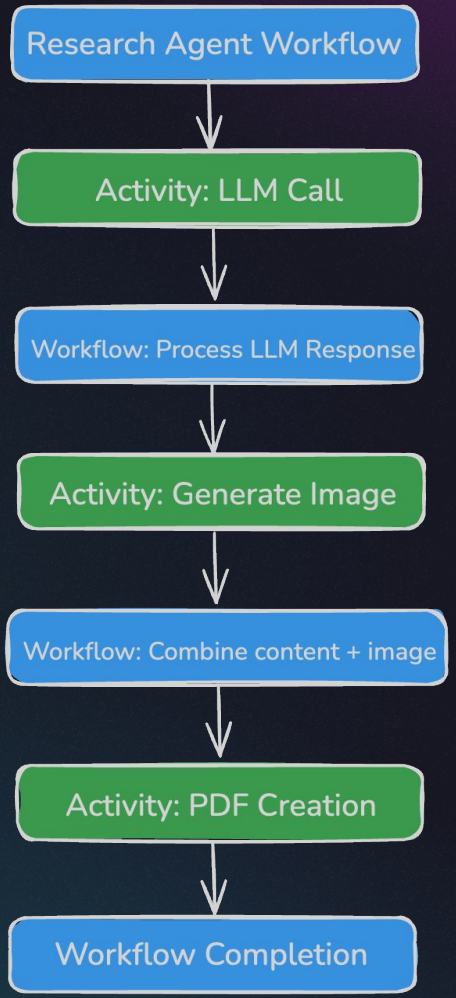
DURABLE EXECUTION - STATE PRESERVATION



- Temporal records:
  - Event History
  - Activity Results

- This allows a failed program to restart where it left off


User request: "Research sustainable energy trends."
Step 1: LLM Research call → Output saved to history
Step 2: Generate summary → Output saved to history
Step 3: Create PDF → Crash!


Research Agent Workflow
Activity: LLM Call
Workflow: Process LLM Response
Activity: Generate Image
Workflow: Combine content + image
Activity: PDF Creation
Workflow Completion

EXERCISE 2: ADDING DURABILITY

- In this exercise, you'll:
  - Transform your LLM calls and your execution of tools to Activities
  - Use a Temporal Workflow to orchestrate your Activities
  - Observe how Temporal handles your errors
  - Debug your error and observe your Workflow Execution successfully complete

Temporal

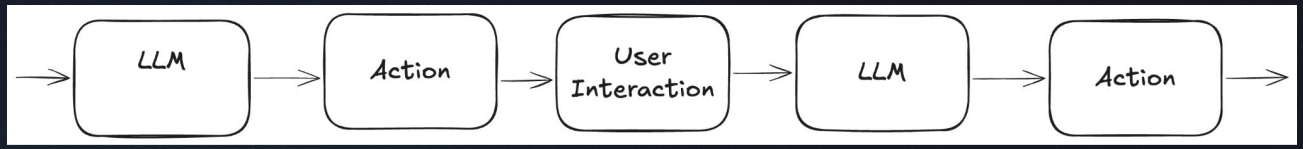
Human in the Loop

IN THIS SECTION

✅ Allow users to supply information to an application

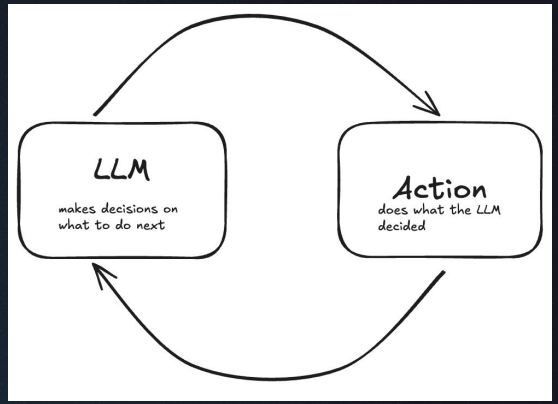
✅ Allow an application to make information available users

MOST GENAI APPS STILL ENGAGE HUMANS

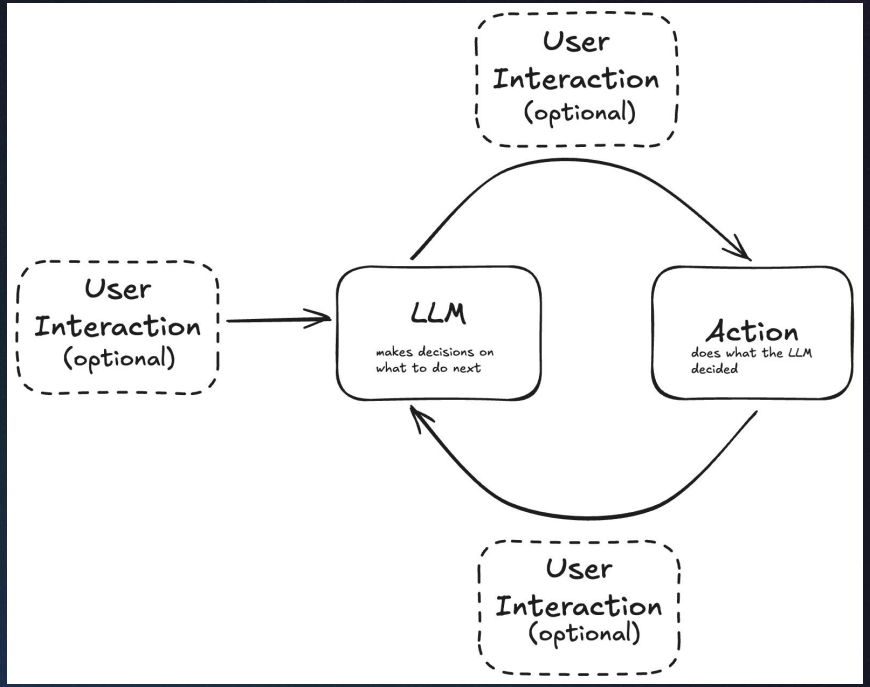


- Examples
  - Validation at critical decision points
  - Final review before implementation
  - Feedback loops

MOST GENAI APPS STILL ENGAGE HUMANS

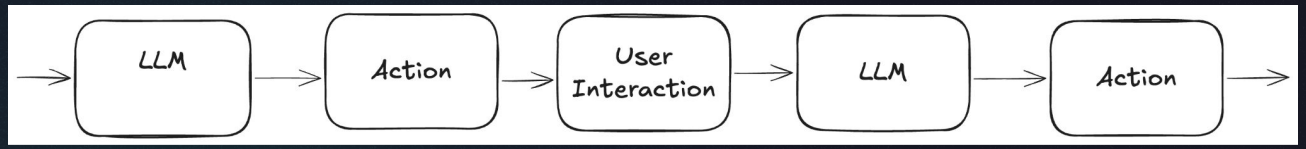


MOST GENAI APPS STILL ENGAGE HUMANS

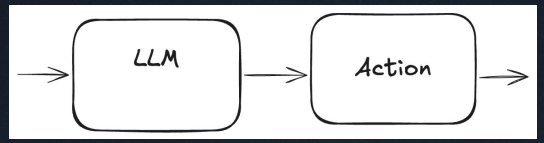


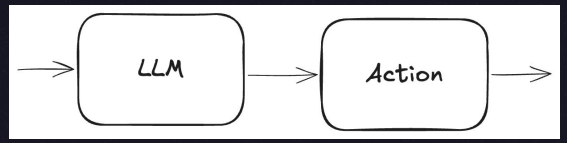
Challenges in Non-Durable Human-in-the-Loop Processes

What we want



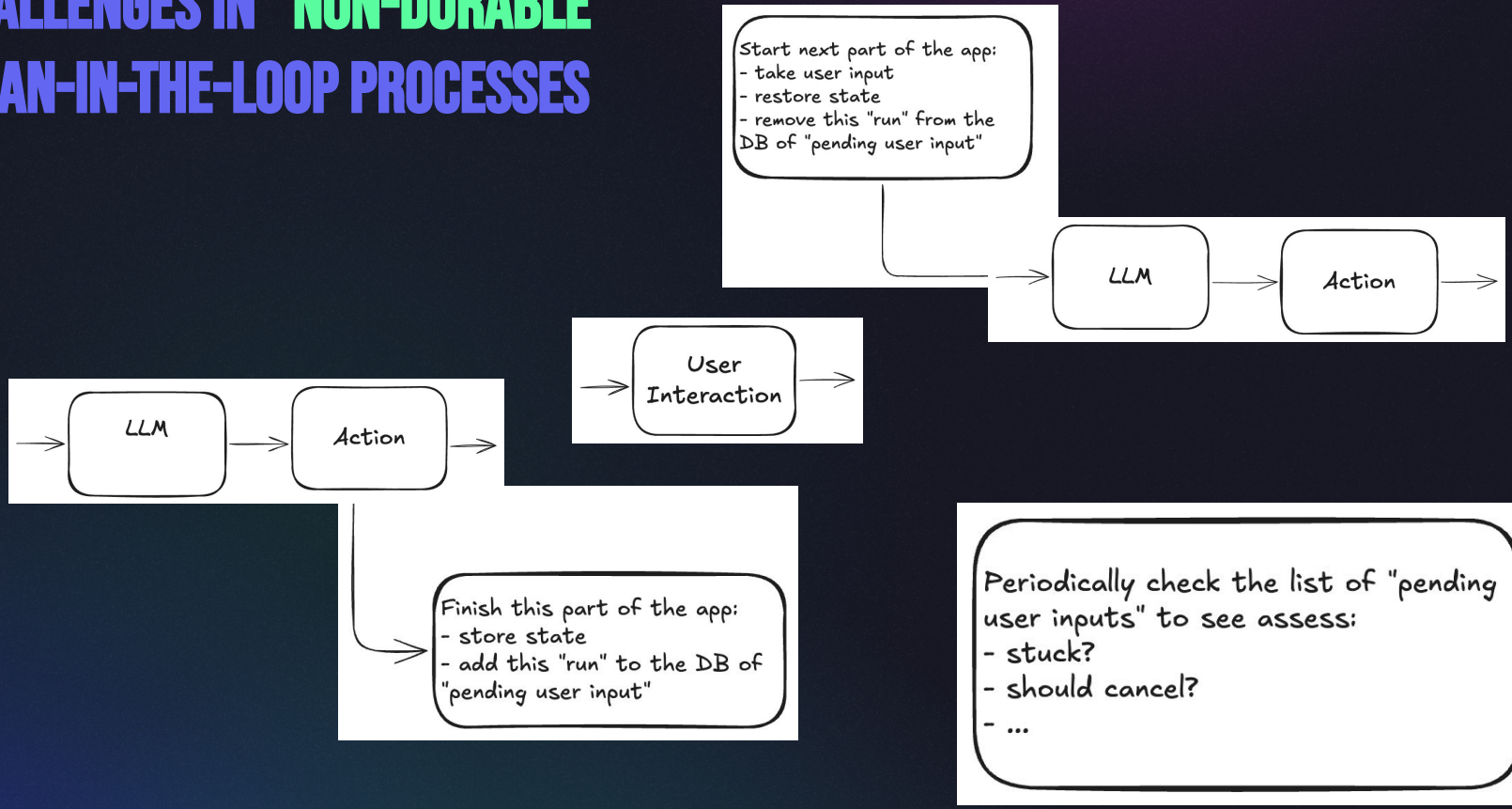
CHALLENGES IN NON-DURABLE HUMAN-IN-THE-LOOP PROCESSES





What we get
(or have to deal with)

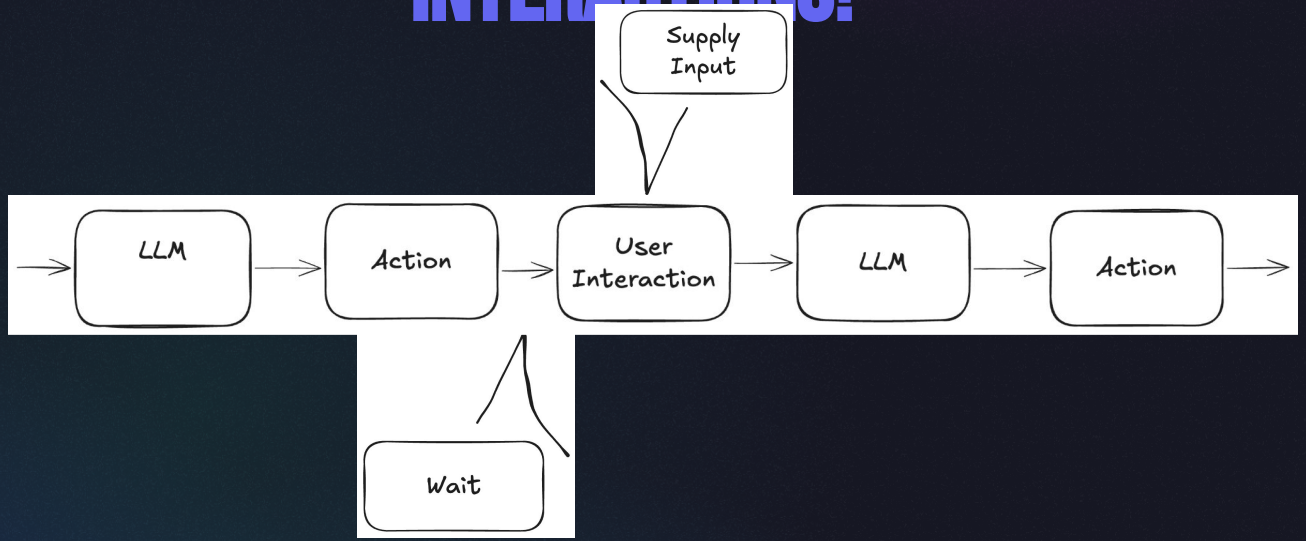
CHALLENGES IN NON-DURABLE HUMAN-IN-THE-LOOP PROCESSES



IT'S DISTRIBUTED SYSTEMS CHALLENGES ALL OVER AGAIN!

BUT WITH DURABLE EXECUTION IT BECOMES SIMPLE!

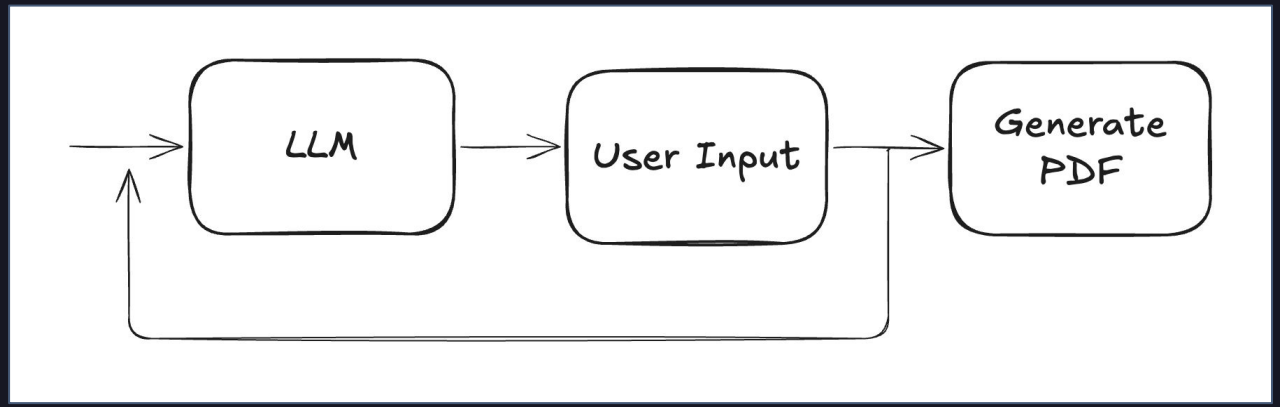
THE WORKFLOW LIVES THROUGH THESE INTERACTIONS!



So then, we need to address a few things:

- The UX is a process separate from the agent
- User inputs may be needed in many different parts of the application (i.e. not always right before or after an LLM invocation)
- The agentic task in its entirety will generally outlive any client connection
- The agentic task in its entirety will often outlive the actual process it is running in
  - (and it probably won't run in a single process anyway)

A really simple research application



So then, we need to address a few things:

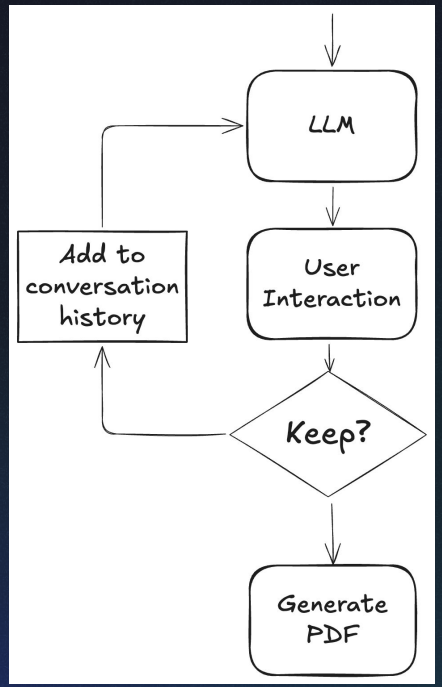
- The UX is a process separate from the agent
- User inputs may be needed in many different parts of the application (i.e. not always right before or after an LLM invocation)
- The agentic task in its entirety will generally outlive any client connection
- The agentic task in its entirety will often outlive the actual process it is running in
  - (and it probably won't run in a single process anyway)

DEMO TIME!

SIGNALS IN TEMPORAL

- A Signal is a:
  - Message sent asynchronously to a running Workflow Execution
  - Used to change the state and control the flow of a Workflow Execution

LET'S IMPLEMENT A SIGNAL!

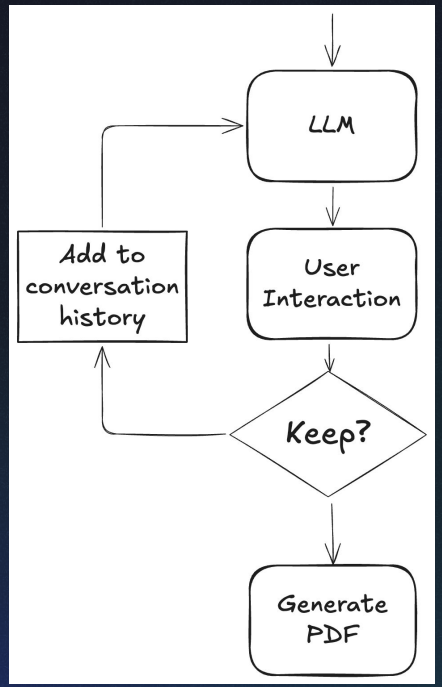


Updating our app

1. We will now give the user the opportunity to either
    - Accept the research results from the LLM, or
    - Refine with an additional prompt
2. And they can do this until they are satisfied with the research...

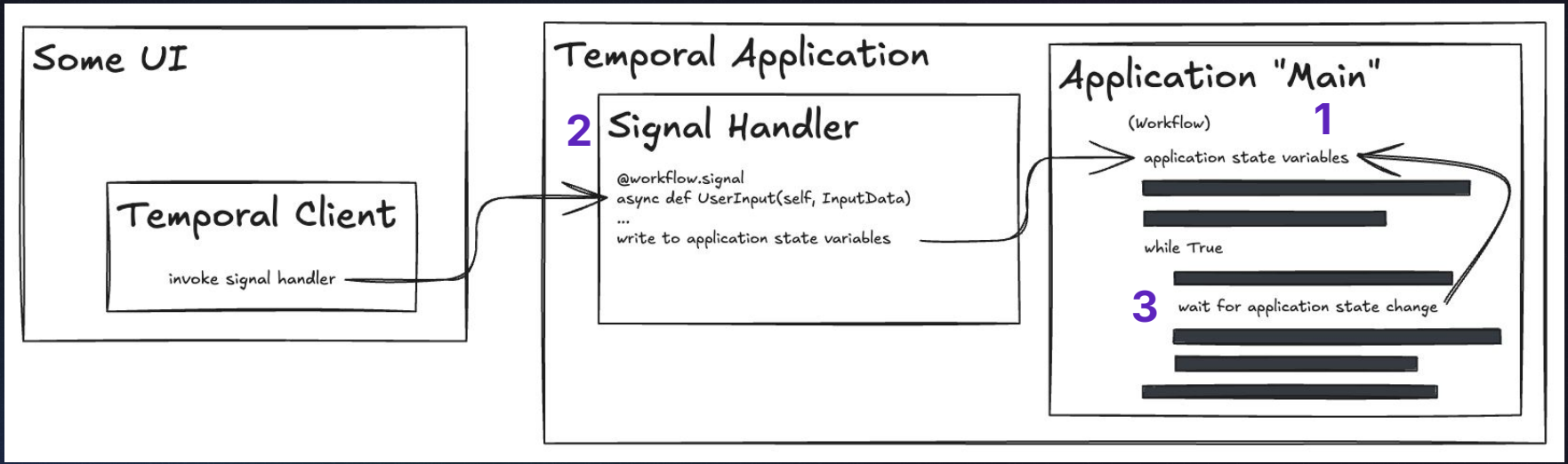
So we'll loop

LET'S IMPLEMENT A SIGNAL!



So, we need to do three things

1. Define the structure for the application state being impacted via the signal
2. Implement a signal handler that is called when the user supplies the information.
    - This signal handler will write into the above defined data structure
3. Implement the point(s) in the flow where user input is sought


Some UI
Temporal Client
invoke signal handler
Temporal Application
2
Signal Handler
@workflow.signal
async def UserInput(self, InputData)
...
write to application state variables
Application "Main"
(Workflow)
1
application state variables
while True
3
wait for application state change

CREATE A MODEL

- Create a model for the Signal to be stored in
- Similar to Activities and Workflows, **`dataclasses`** are recommended here

STORING THE SIGNAL STATE

- Use instance variables to persist signal data across Workflow Execution
- Can be a simple variable or a Queue for handling many Signals
- Initialize with default values that indicate “no Signal received yet

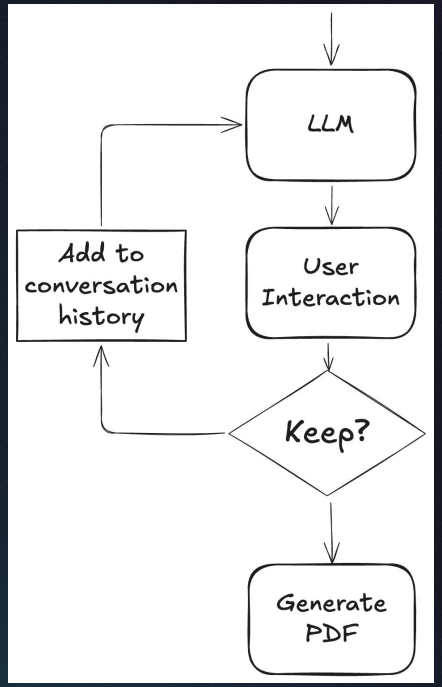
DEFINING A SIGNAL HANDLER

- A Signal is defined in your code and handled in your Workflow Definition
- A given Workflow Definition can support multiple Signals
- To define a Signal, set the Signal decorator `@workflow.signal` on the Signal function inside your Workflow class
- Signal methods define what happens when the Signal is received

DEFINING A SIGNAL HANDLER

```python
from temporalio import workflow

@workflow.defn
class GenerateReportWorkflow:
    def __init__(self) -> None:
        # Instance variable to store Signal data
        self._user_decision: UserDecisionSignal =
UserDecisionSignal(decision=UserDecision.WAIT) # UserDecision
Signal starts with WAIT as the default state

    # Define the Signal handler
    @workflow.signal
    async def user_decision_signal(self, decision_data:
UserDecisionSignal) -> None:
        # Update instance variable when Signal is received
        self._user_decision = decision_data

    @workflow.run
    async def run(self, input: GenerateReportInput) ->
GenerateReportOutput:
      self._current_prompt = input.prompt

      llm_call_input = LLMCallInput(
          prompt=self._current_prompt,
          llm_api_key=input.llm_api_key,
          llm_model=input.llm_research_model,
      )
      # rest of code here
```

RECALL OUR FLOW



WAITING FOR A SIGNAL

- Use **`workflow.wait_condition()`** to pause until Signal is received (user decides the next step)
- Creates a blocking checkpoint where the Workflow stops and waits
- Resumes execution only when specified condition becomes true

```
await workflow.wait_condition(lambda: self._user_decision.decision != UserDecision.WAIT)
```

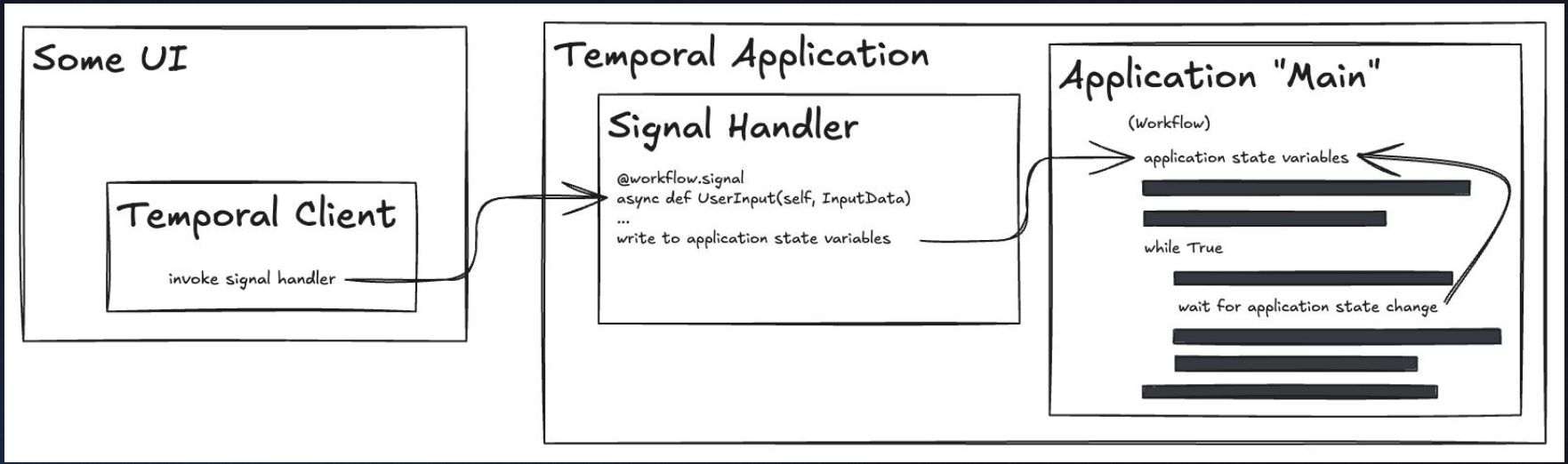
WAITING FOR A SIGNAL

- You can also pass in a timeout in `wait_condition`:
  - Wait until a Signal is received or
  - A set amount of time has elapsed: whichever happens first

```
await workflow.wait_condition(
                lambda: self._user_decision.decision != UserDecision.WAIT),
                timeout=timedelta(seconds=20)
            )
```

SENDING A SIGNAL

- We need a handle to a running workflow
  - `handle = client.get_workflow_handle(workflow_id)`
- Package the signal payload
  - `signal_data = UserDecisionSignal(decision=UserDecision.KEEP)`
- Invoke signal method on the handle
  - `await handle.signal`


Some UI
Temporal Client
invoke signal handler
Temporal Application
Signal Handler
@workflow.signal
async def UserInput(self, InputData)
...
write to application state variables
Application "Main"
(Workflow)
application state variables
while True
wait for application state change

QUERIES IN TEMPORAL

- Extract state to show the user
- Can be done during or even after the Workflow Execution has completed
- Examples:
  - Monitor progress of long-running Workflows
  - Retrieve results

HANDLING A QUERY

- You can handle Queries by annotating a function within your Workflow with `@workflow.query`

```python
@workflow.query
def get_research_result(self) -> str | None:
    return self._research_result
```

SENDING A QUERY

- We are going to send a Query through the Temporal Client
  - Get a handle of the Workflow Execution we will query
  - Send a Query with the **`query`** method

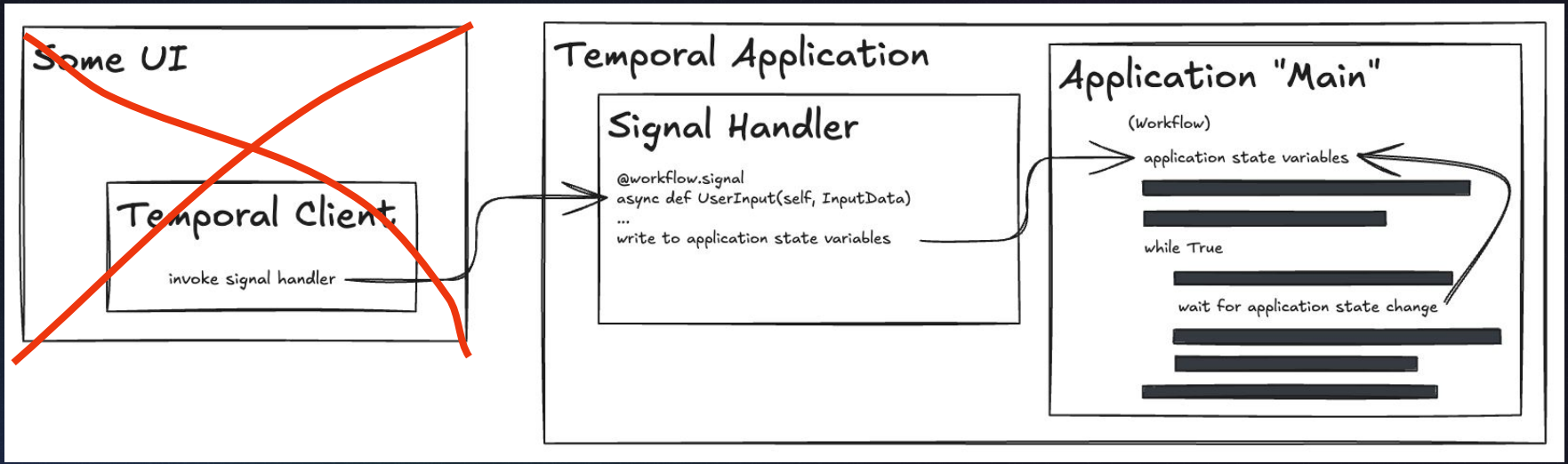
```python
async def query_research_result(client: Client, workflow_id: str) -> None:
    handle = client.get_workflow_handle(workflow_id)

    try:
        research_result = await handle.query(GenerateReportWorkflow.get_research_result)
        if research_result:
            print(f"Research Result: {research_result}")
        else:
            print("Research Result: Not yet available")

    except Exception as e:
        print(f"Query failed: {e}")
```

CONNECTING UI TO WORKFLOW QUERIES (& SIGNALS)

- We will create an application that will
  a. **query** the workflow and, when it is in a wait state it will:
  b. Prompt the user for input
  c. Provide the user's input to the workflow via **signal**

So then, we need to address a few things:

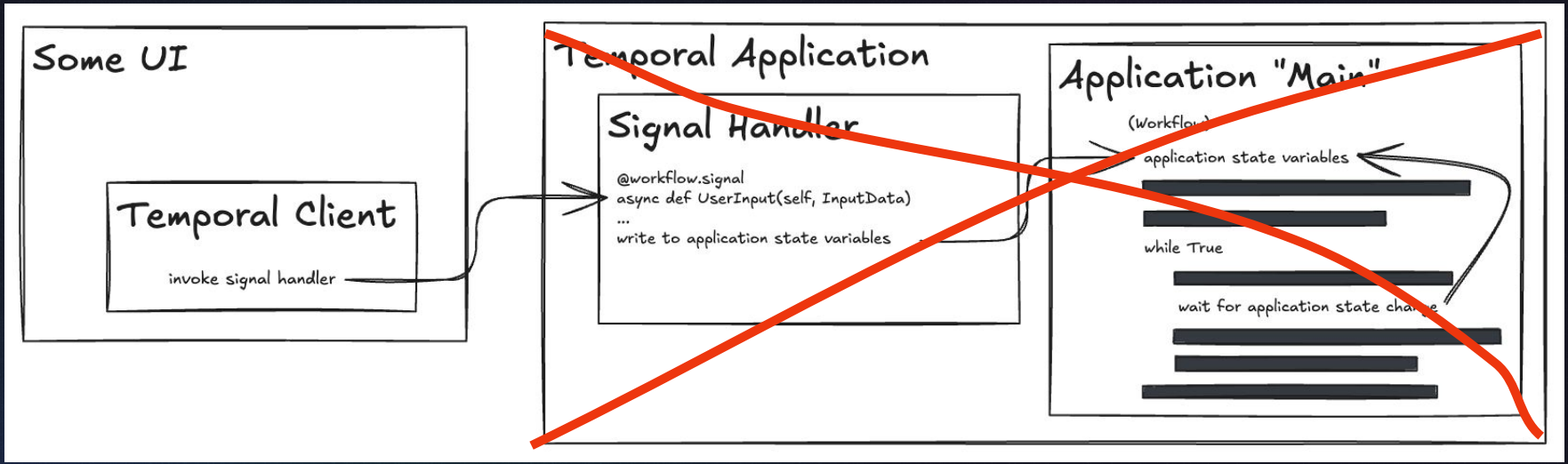
- The UX is a process separate from the agent
- User inputs may be needed in many different parts of the application (i.e. not always right before or after an LLM invocation)
- The agentic task in its entirety will generally outlive any client connection
- The agentic task in its entirety will often outlive the actual process it is running in
  - (and it probably won't run in a single process anyway)


Some UI
Temporal Client
invoke signal handler
Temporal Application
Signal Handler
@workflow.signal
async def UserInput(self, InputData)
...
write to application state variables
Application "Main"
(Workflow)
application state variables
while True
wait for application state change

DEMO TIME!

So then, we need to address a few things:

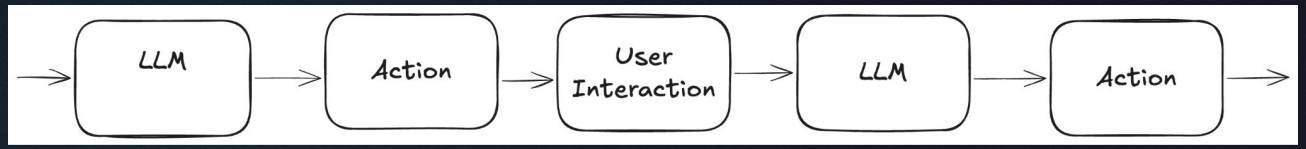
- The UX is a process separate from the agent
- User inputs may be needed in many different parts of the application (i.e. not always right before or after an LLM invocation)
- The agentic task in its entirety will generally outlive any client connection
- The agentic task in its entirety will often outlive the actual process it is running in
  - (and it probably won't run in a single process anyway)


Some UI
Temporal Client
invoke signal handler
Temporal Application
Signal Handler
@workflow.signal
async def UserInput(self, InputData)
...
write to application state variables
Application "Main"
application state variables
while True
wait for application state change

DEMO TIME!

CHALLENGES IN NON-DURABLE HUMAN-IN-THE-LOOP PROCESSES

What we want



And with Temporal, what we can have!

Temporal

AI Agents

IN THIS SECTION:

- ✅ Understand LLM agency
- ✅ Cover a bit of context engineering
- ✅ Create some tools
- ✅ Build a couple of durable AI agents (non-looping, and looping)

WE'VE BUILT SOME GENAI APPLICATIONS

- Up until now we have built some **durable(!!)** GenAI applications that
  - Called LLMs and APIs
  - Allowed for human interaction
  - Looped
- But we - the developer - encoded flow decisions into those apps. That is, it was a **fixed flow**!

Somewhat 🙃

IN THAT LAST EXAMPLE - WHERE WE WE LOOPED - WHAT (WHO?) DECIDED THE FLOW?

Ah ha!!!

WHAT MAKES A GENAI APP AGENTIC?

When we give

agency

to the the

LLM

to drive the flow of the application!!

DEMO TIME!

LET'S TALK A BIT ABOUT MODELS (AND APIS)

- The first models were designed to complete the prompt that was passed in.
  - The original OpenAI API was the *completions* API
- Since then, models have been created/optimize to respond in different ways.
  - And in March 2025, OpenAI released the *responses* API

Note: We use OpenAI as an example here - other model and API creators are doing similar things

THE RESPONSES API - DESIGNED FOR AGENTIC THINGS

```
1    resp = await client.responses.cre
2        model=...,
3        instructions=...,
4        input=...,
5        tools=...,
6        timeout=30,
7    )
```

Choose a model that is designed for agentic tasks

For agents this describes the role of the agent

Gathers the context as the agent progresses (through iterations)

A list of the APIs available for use by the agent

TOOLS

```
{
  "tools": [
    {
      "name": "get_weather_alerts",
      "description": "Retrieves current weather alerts for a given U.S. state.",
      "parameters": {
        "type": "object",
        "properties": {
          "state": {
            "type": "string",
            "description": "The U.S. state to get weather alerts for, e.g. 'California' or 'TX'."
          }
        },
        "required": ["state"]
      }
    },
```

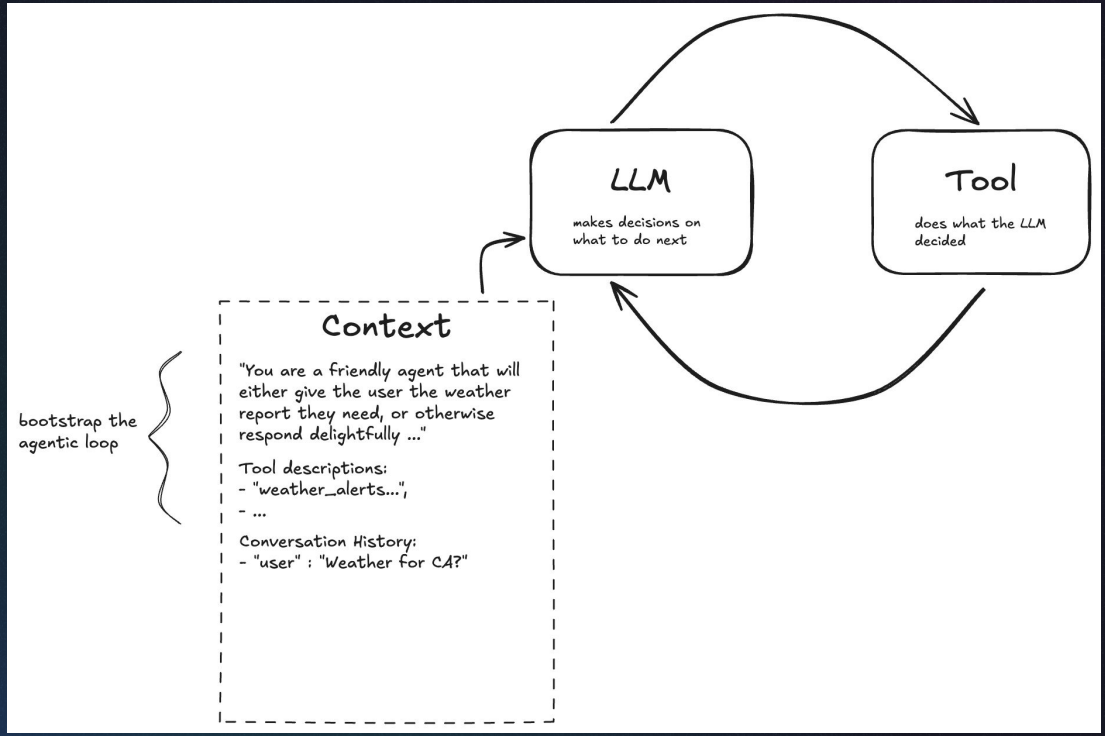
TOOLS (CONT)

I sometimes call tools:
“LLM enabled APIs”

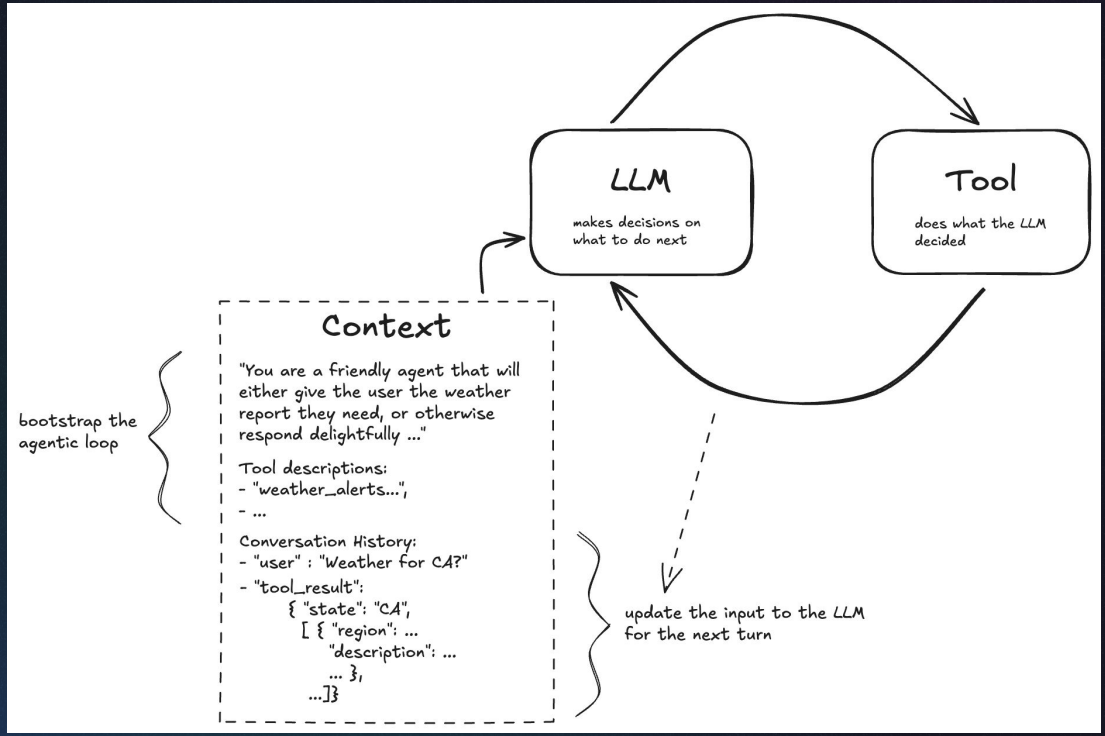
```
    {
      "name": "generate_random_number",
      "description": "Generates a random number with no input parameters.",
      "parameters": {
        "type": "object",
        "properties": {}
      }
    }
  ]
}
```

Note: Ultimately, your agents need to manage a registry of tools, mapping these descriptions to the actual API calls. We will cover this in a dedicated agent workshop

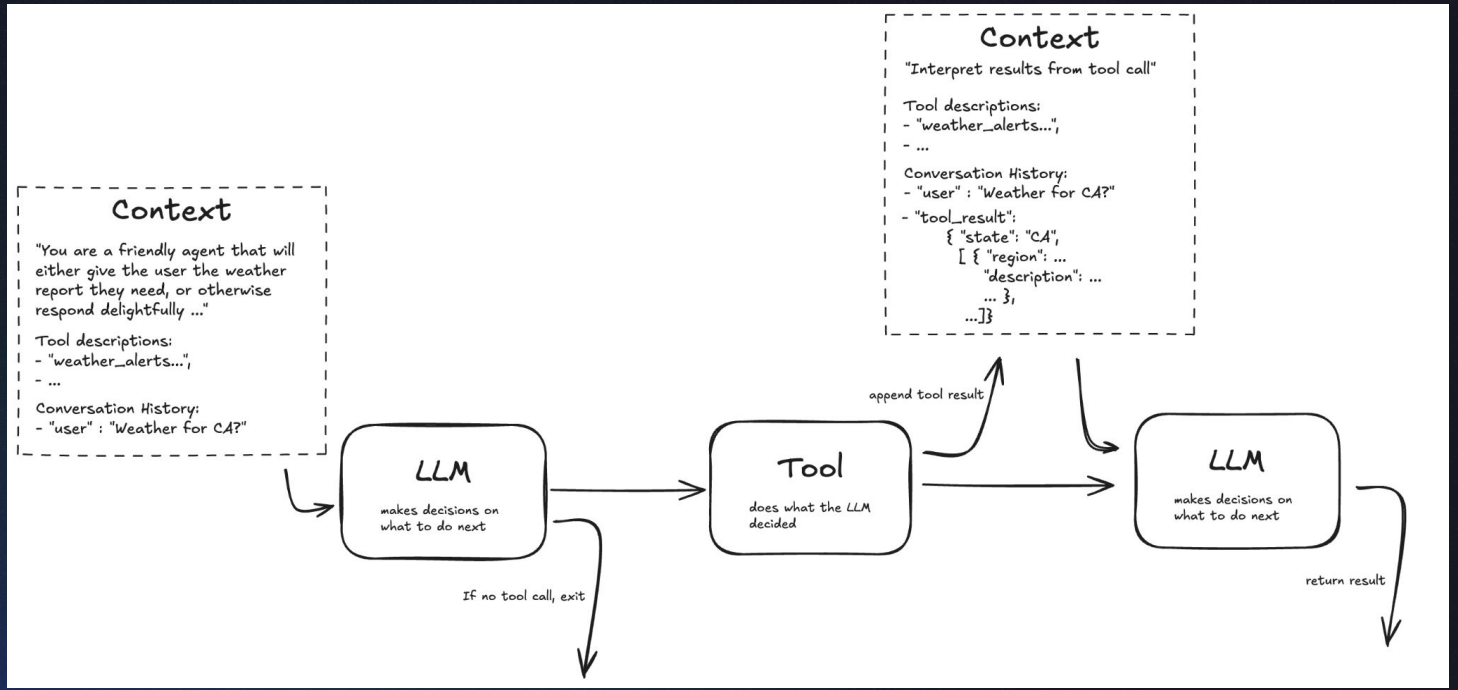
CONTEXT ENGINEERING



CONTEXT ENGINEERING (CONT)



LET'S BUILD AN AGENT!



https://docs.temporal.io/ai-cookbook/tool-calling-python

NOW, LOOP

What happens when the user asks
“What is **MY** weather?”

Instead of “what’s CA weather?”

OBSERVATIONS

There are APIs to look up the user's IP address

There are APIs that can find the location of an IP address

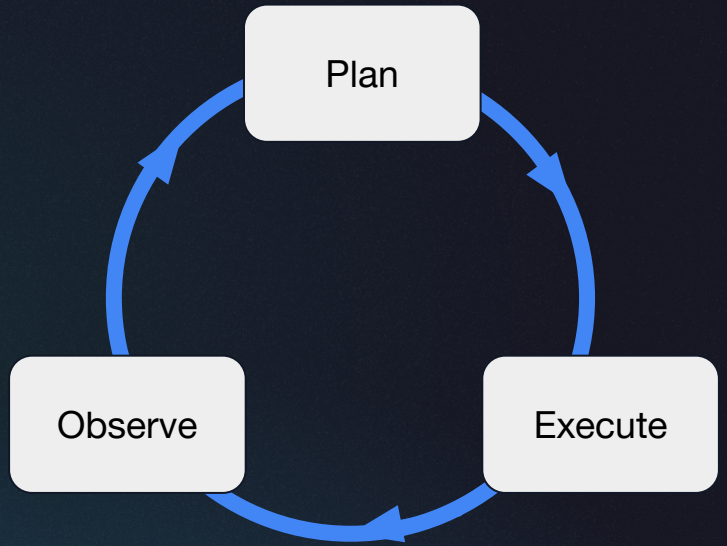
(and we already have a tool that gets weather for a location)

OBSERVATIONS

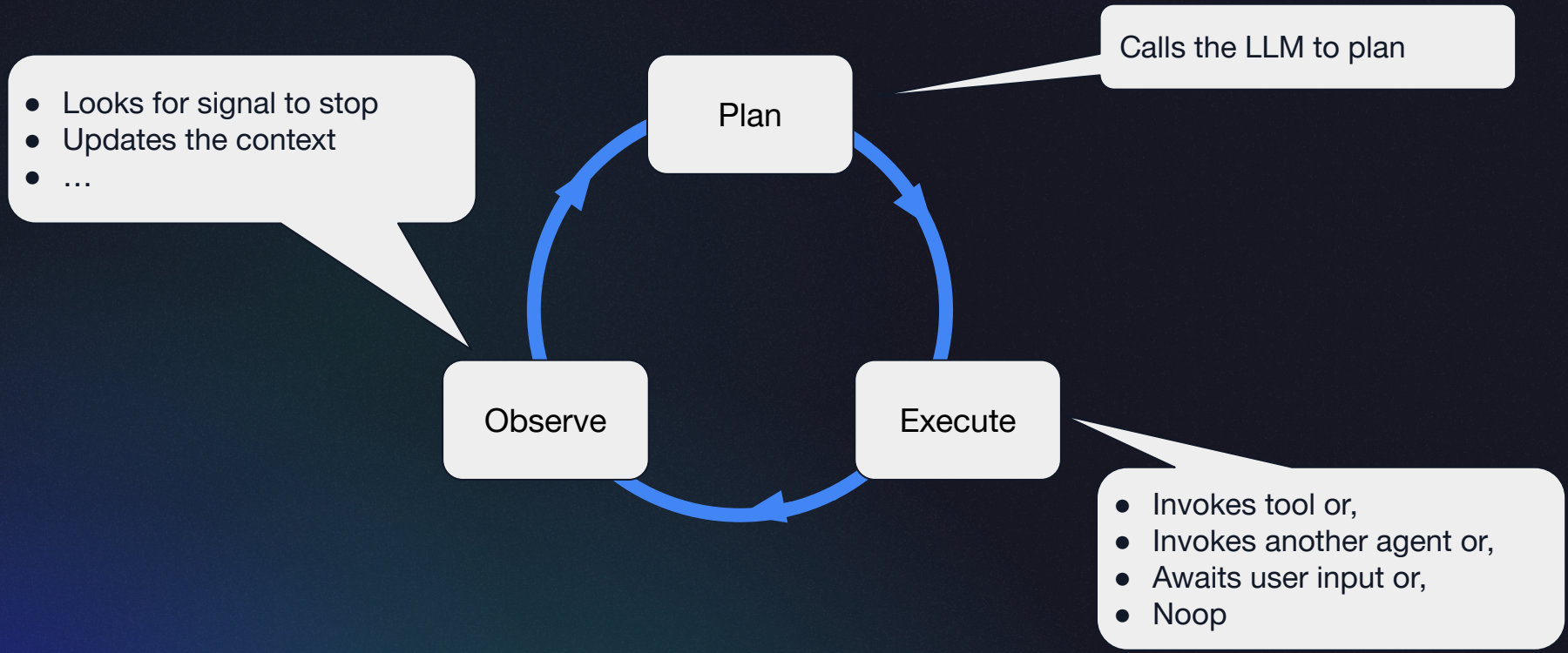
We could build the sequence

But the LLM can do that too!

WHAT IS AN AGENTIC LOOP?



WHAT IS AN AGENTIC LOOP?



WHY IS THIS POWERFUL?

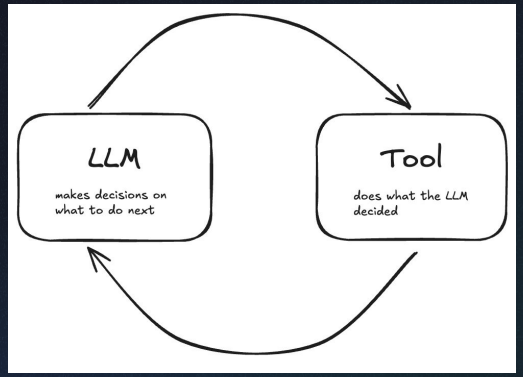
- The AI makes its own decisions
- Can adapt to different situations dynamically
- Can use different tools in different orders based on context

THE AGENTIC LOOP IN TEMPORAL

- Workflow = Agentic Loop
- Activities for LLM and Tool calls
- Complete Step Tracking
- Durable Execution

LET'S BUILD ANOTHER AGENT!

Agentic Loop:



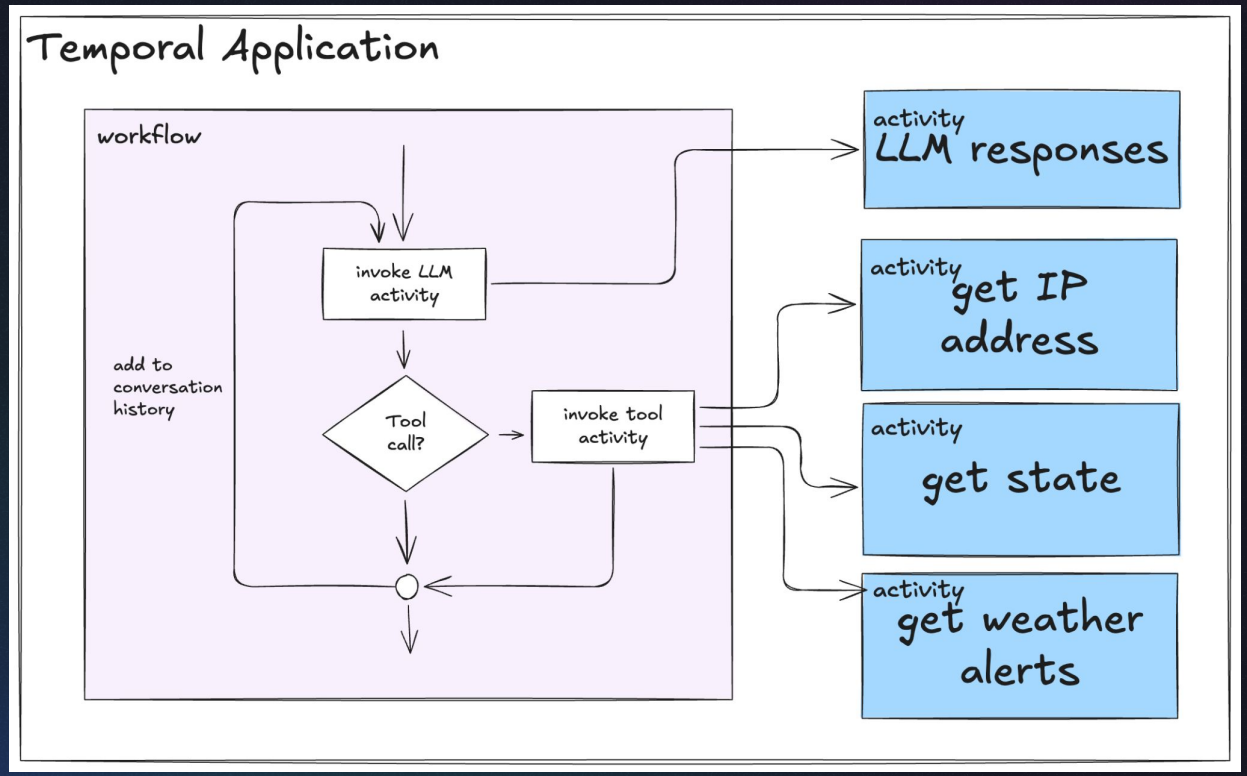
Instructions:

<Use tools or don't>

Tools:

get_IP
get_State(IP)
get_weather_alerts(state)

LET'S BUILD ANOTHER AGENT!



DEMO TIME!

WHAT WE'VE JUST DONE

- ✅ Foundations of GenAI applications
- ✅ Making GenAI applications durable
- ✅ Humans are still needed
- ✅ Durable AI Agents - putting it all together

WE WELCOME YOUR FEEDBACK!

T.MP/AI-WORKSHOP-FEEDBACK

GET STARTED WITH TEMPORAL!

- Join the community!
- Read the documentation
- Follow a tutorial
- Take a free course online and more

Replay 2026

Moscone Center, SF

May 5-7, 2026

1500+ attendees

Use Code **LAUNCHANDLEARN75** for 75% off

// JUST ANNOUNCED

01 TEMPORAL

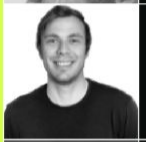
SAMAR ABBAS

02 TEMPORAL

MAXIM FATEEV

03 REPLIT

AMJAD MASAD



04 NVIDIA

JEREMY COOK

05 NVIDIA

MATT TESCHER

06 FIVETRAN

ZAINAB MERCHANT

07 PYDANTIC

SAMUEL COLVIN

REPLAY

MAY 05-07, 2026

MOSCONE CENTER, SAN FRANCISCO

Temporal