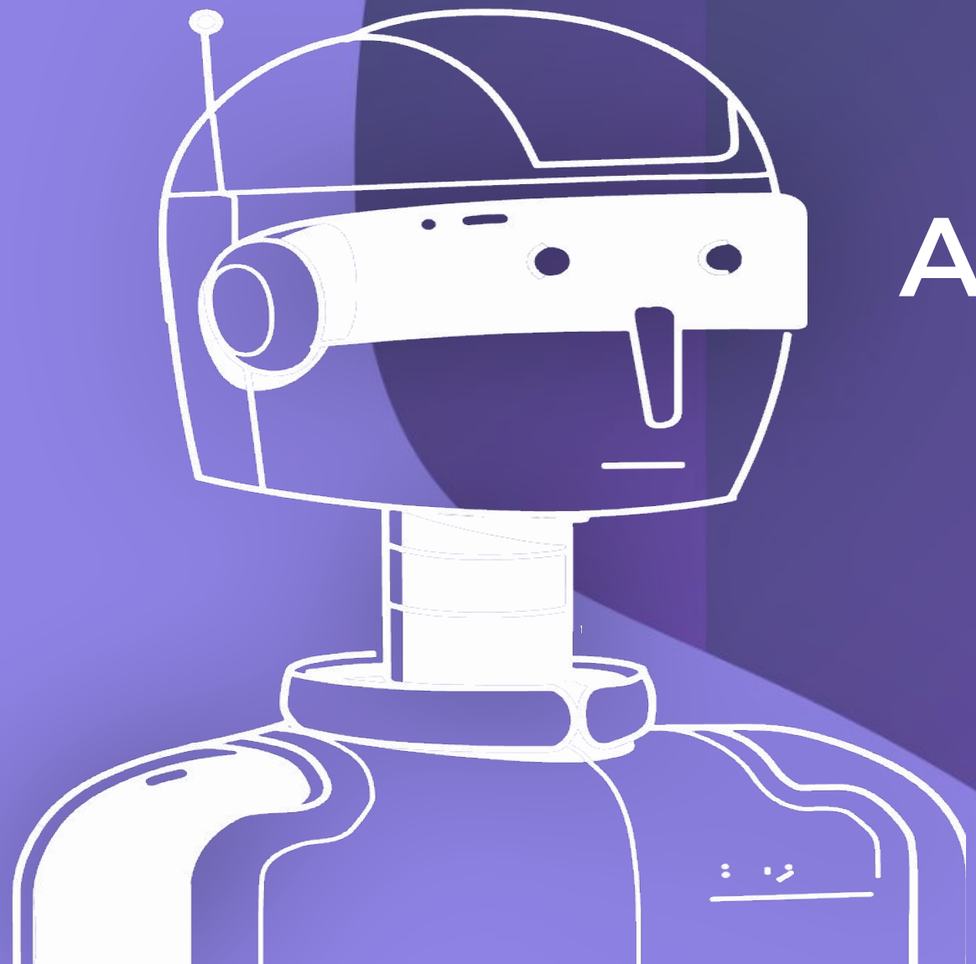
DEVCONF.cz



Llama Stack
AI Development made easy !

Abhishek Kumar
Sr. Principal Engineer,
Red Hat

Pranita Ghole
Associate Manager,
Red Hat

Pravin Pareekh
Manager,
Red Hat

Agenda

- Challenges in AI development
- Introducing Llama stack
- How Llama stack overcomes developer challenges
- Architecture and Core Concepts of Llama stack
- Hands-On Learning
  - Setup
  - Chat completion
  - Chat using Agent
    - Simple agent
    - Agent with tool calling
    - RAG (Retrieval Augmented Generation) with Agent

Challenges in AI development

❖ Infrastructure Complexity

- ➢ Difficulty to transition from prototype to production.
- ➢ The complexity of integrating AI systems into existing IT infrastructure

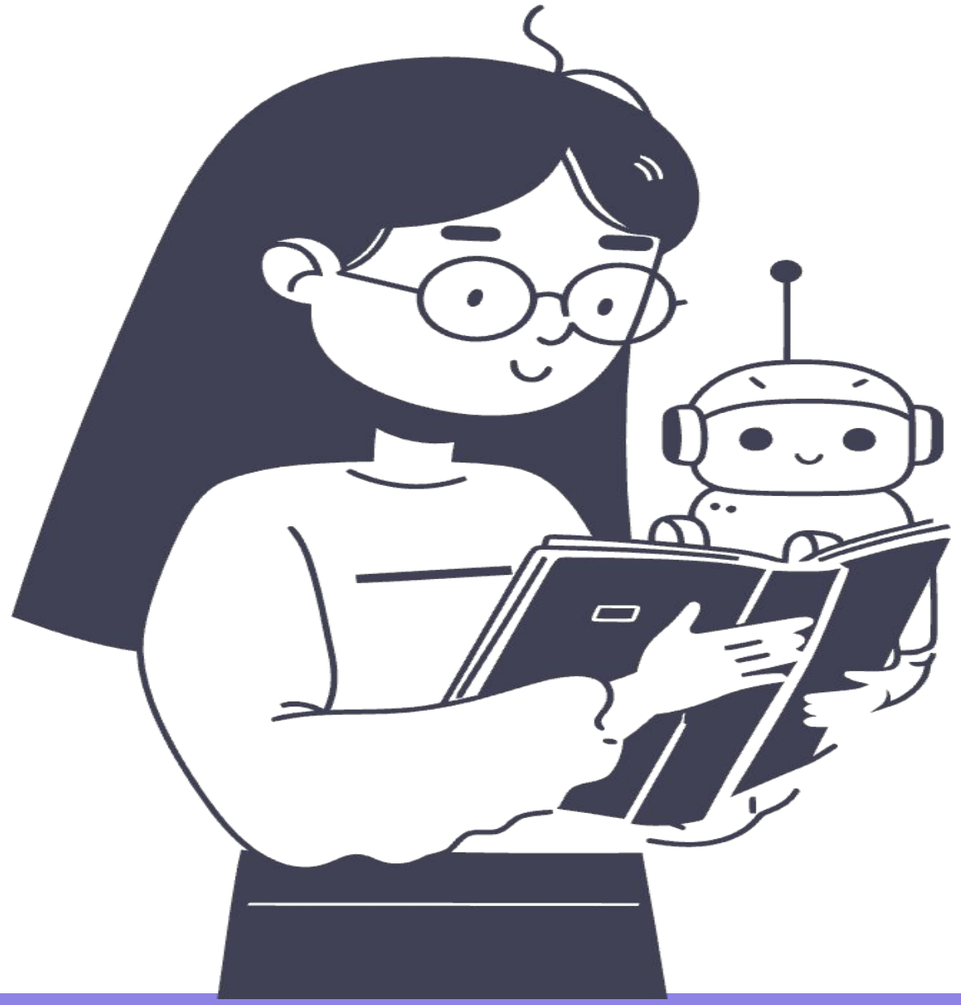
❖ Fragmented tooling for Essential Capabilities

- ➢ Developers juggle with separate tools for RAG, evals, safety guardrails, monitoring etc.

❖ Lack of Flexibility and Standardization

- ➢ Different providers have different APIs and abstractions
- ➢ Changing providers requires significant code changes.

Introducing Llama Stack !



- Open-source framework for building generative AI applications.
- Llama Stack defines and standardizes the core building blocks needed to develop Production ready generative AI applications.
- Llama Stack consists of a server (with multiple pluggable API providers) and Client SDKs meant to be used in your applications.

How Llama Stack Solves Developer Challenges

- **Standardisation through a service oriented, API-first approach**
  - Develop anywhere, deploy everywhere
  - REST APIs ensure clean interfaces and seamless transition across multiple environments.

- **Unified API layer for Production Ready Building Blocks**
  - Built-in support for RAG, tools and agent capabilities.
  - Support for safety guardrails, evaluation toolkit, monitoring

- **True Independence**
  - Swap providers without application changes.
  - Multiple developer interfaces like CLI and SDKs for Python, Node, iOS, and Android

Architecture

1. **Client Layer**

   REST API, Client SDKs in Python, Swift, Node, Kotlin

2. **Core Layer**

   APIs for inference, RAG, Safety agents

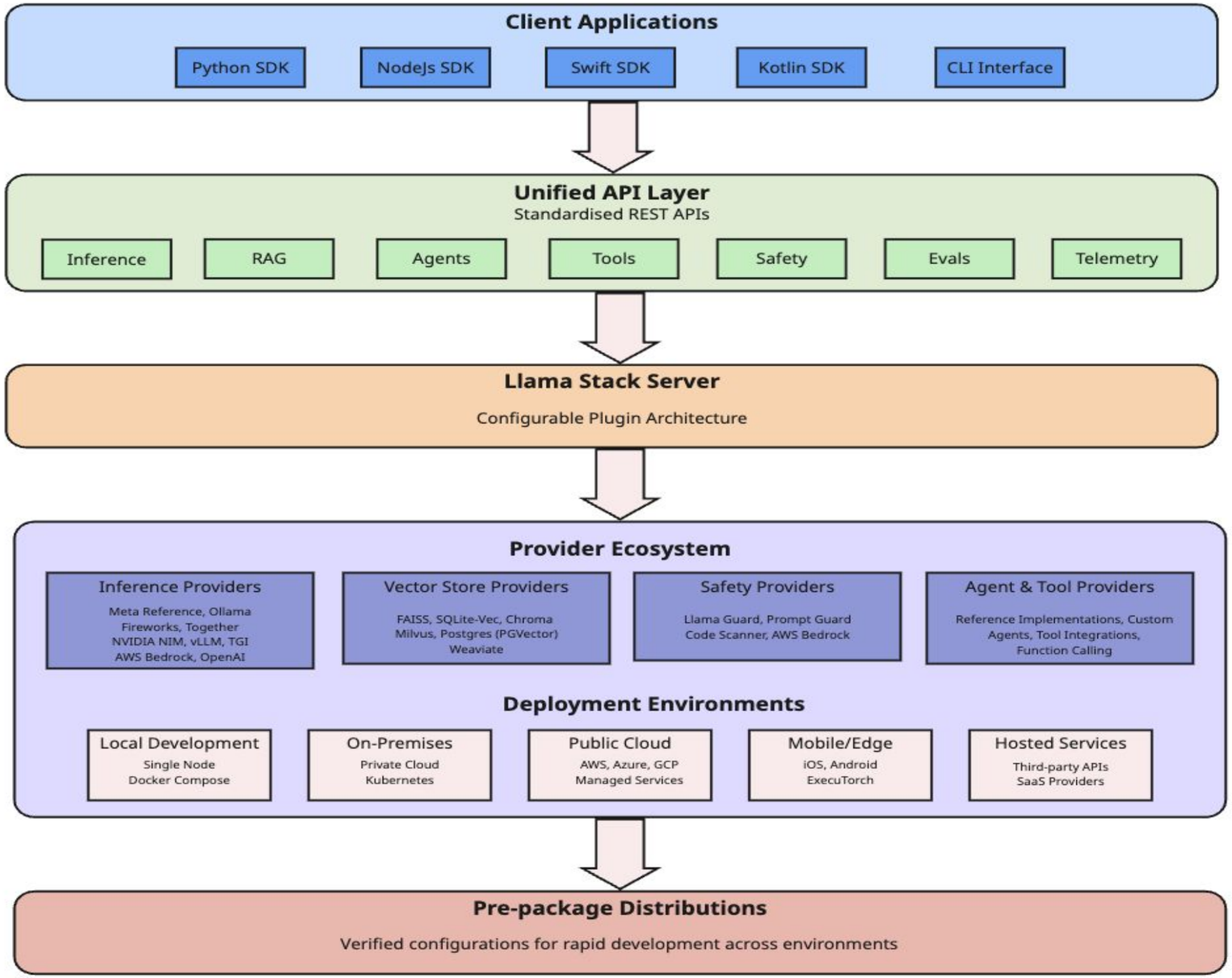
3. **Provider Layer**

   Integration with inference providers

4. **Resources Layer**

   LLM, Vector DB, Shield, Tools

Llama Stack Architecture



Core Concepts - WIP ; this might be time taking to explain. SHould we skip ?

APIs

APIs for Inference, Agents, VectorIO, Eval, Safety etc.

Reference

API Providers

- LLM Inference Providers(eg- Fireworks, Ollama, Together etc.)
- Vector Databases(eg- ChromaDB, Faiss, PGVector etc.)
- Safety Providers(eg- Meta's Llama Guard etc.)

Resources

Some of these APIs are associated with a set of Resources.

Eg - Inference, Eval and Post Training are associated with Model resources.

Distributions

- Remotely Hosted Distro
- Locally Hosted Distro
- On-device Distro

Hands-On Learning !

https://tinyurl.com/llama-stack-workshop

https://tinyurl.com/ollama-model

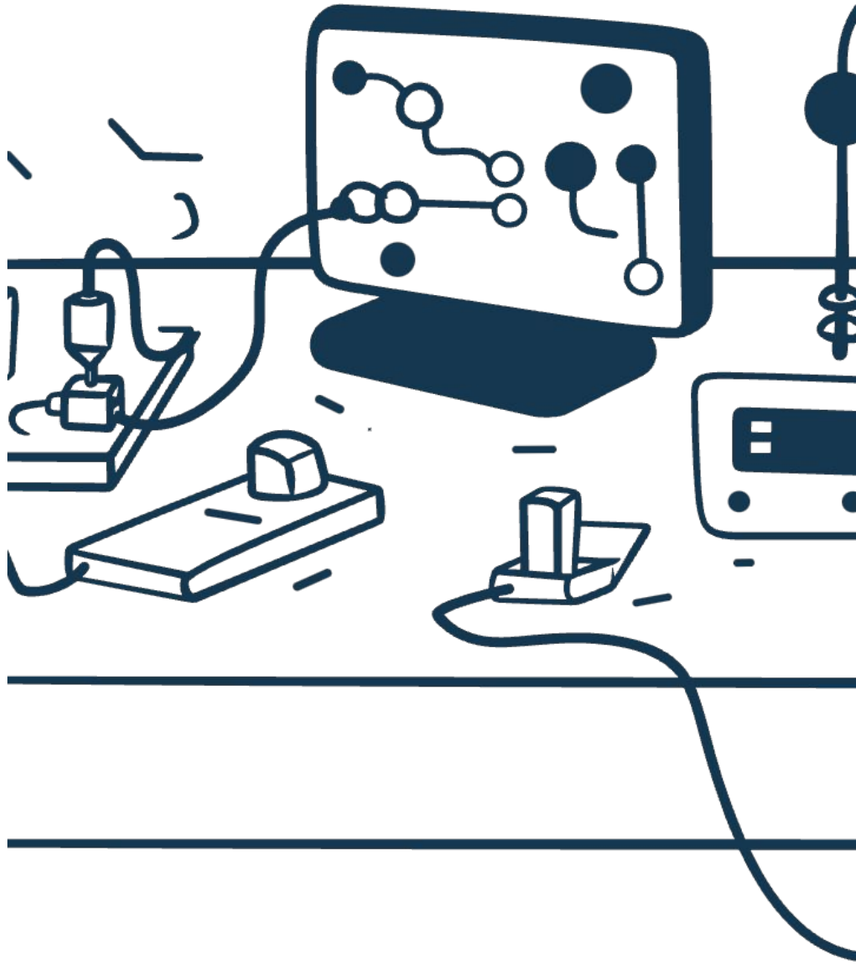
Local Setup

1. The prerequisites have been installed from the installer.sh OR steps from llama-stack-workshop repo
2. In a terminal, start the ollama server as follows OR `export OLLAMA_URL="https://massive-vertically-muskrat.ngrok-free.app"`
    - `ollama list` #check if the model llama3.2:3b-instruct-fp16 is listed there.
    - `ollama serve`
    - `Open http://localhost:11434/ in the browser`
3. Go to the "llama-stack-server" directory in a separate terminal
    - Build Llama stack
        - `source .venv/bin/activate; export INFERENCE_MODEL="llama3.2:3b-instruct-fp16"`
        - `llama stack build --template ollama --image-type venv`
    - Run Llama Stack server
        - `llama stack run .venv/lib/python3.11/site-packages/llama_stack/templates/ollama/run.yaml --image-type venv`
4. Ensure Llama Stack server is up
    - `Open http://localhost:8321/v1/models in the browser and it should list models`

Setup - Contd.

5. Navigate to the “llama-stack-workshop” directory and open the .env file

- Ensure that the TAVILY_SEARCH_API_KEY is set properly. If not create a free account to get the API key.
- `source .venv/bin/activate`



Setup - Step by Step !

For those who have not run the installer script. To be updated

1. Install Python3.11, venv and llama stack from script shared <SCRIPT>
2.
3. For the inference provider, we will use Together.ai - https://llama-stack.together.ai
   (Confirm if this is present in the .env file)

   WILL RUN ONCE AND UPDATE THE REST OF THE STEPS

Exercise 1 : Chat Completion

1. Create Llama Stack Client

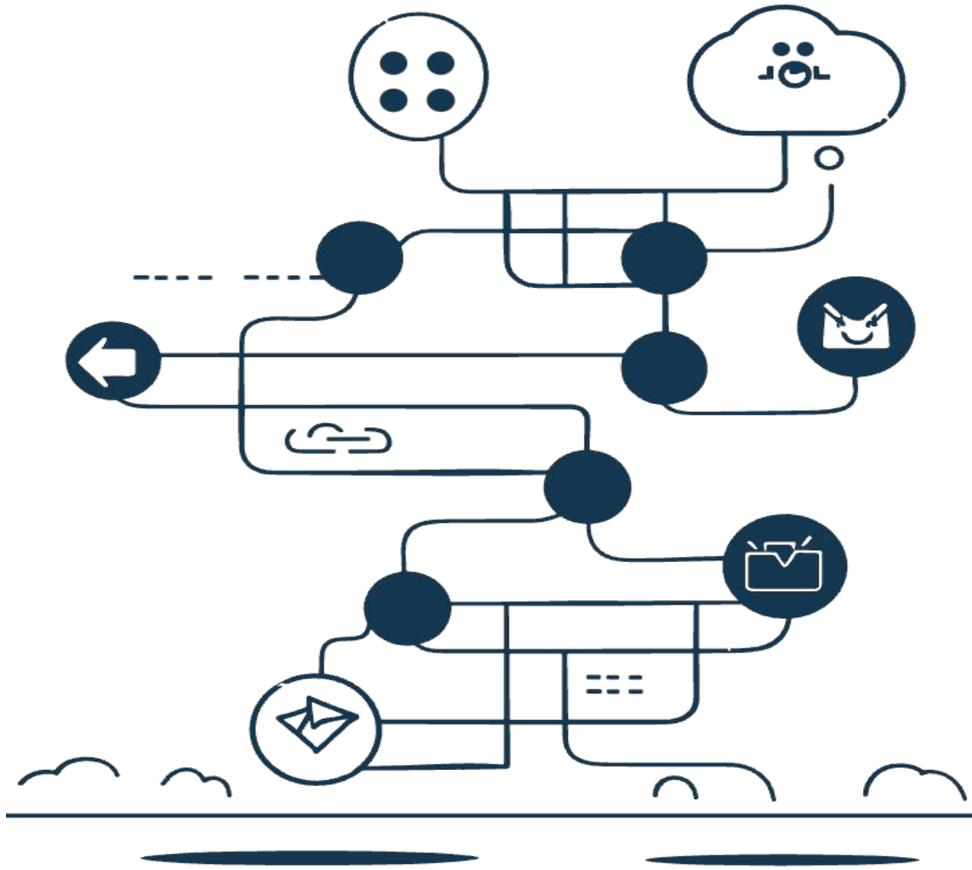
```
client = LlamaStackClient(base_url=os.getenv(“LLAMA_STACK_SERVER”)
```

2. Complete the function chat_completion_with_inference()

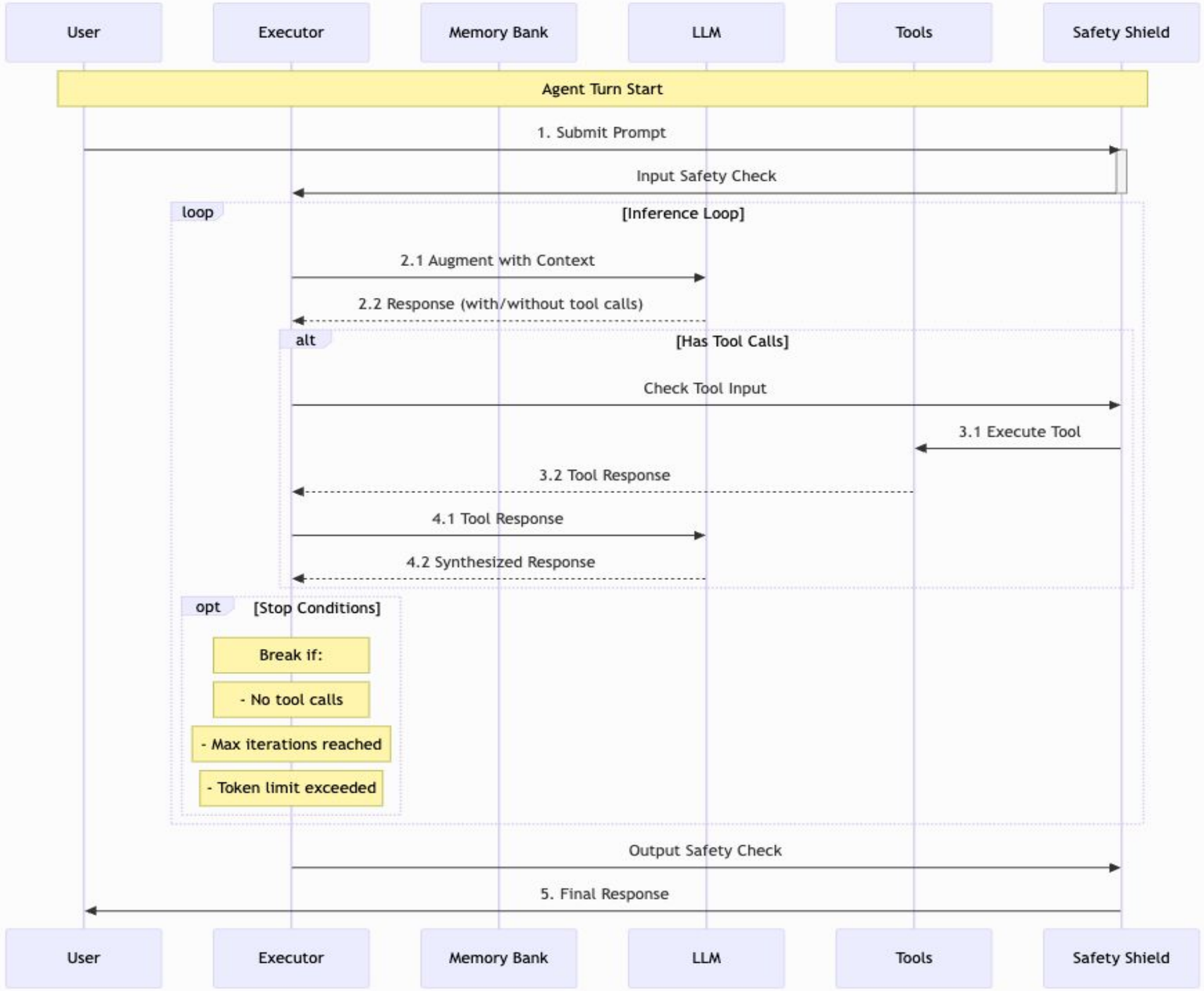
```
def chat_completion_with_inference(content: str):
    response = client.inference.chat_completion(
        model_id=os.getenv("INFERENCE_MODEL_ID"),
        messages=[
            SystemMessage(role="system", content="You're a helpful assistant."),
            UserMessage(role="user", content=content),
        ],
        stream=False,
    )
    return response.completion_message.content
```

Exercise 2 : Chat using Agents

1. Agent in Llama-Stack
2. Interaction with Agents
3. Examples of Agents



Agent Execution Flow



1. Create Agent

```
agent = Agent(
        client=client,
        model=os.getenv("INFERENCE_MODEL_ID"),
        instructions="You are a helpful assistant.",
)
```

2. Create a Session

```
def create_simple_agent_session(prefix):
    session_id = agent.create_session(f"{prefix}-{uuid.uuid4()}")
    print(f"Created session id: {session_id}")
    return session_id
```

3. Chat with Agent

```
def chat_with_simple_agent(session_id, query):
     response = agent.create_turn(
                        session_id=session_id,
                        messages=[
                        UserMessage(role="user", content=query)
                        ],
                        stream=False,
                )
    print(f"Query: {query}, response: {response}")
    return response.output_message.content
```

Exercise 3 - Agent with Tool Calling

1. What are tools ?

2. How the tool calling can be done in the Agents ?

3. ***Example - WebSearch using TAVILY***
    a. Navigate to the “llama-stack-workshop” directory and open the .env file
        - Ensure that the TAVILY_SEARCH_API_KEY is set properly. If not create a free account to get the API key.

1. Create TavilyClient

```
@client_tool
def travily_web_search(query: str):
    client = TavilyClient(os.getenv("TAVILY_SEARCH_API_KEY"))
    response = client.search(
        query=query
    )
    print(f"query: {query}, response: {response}")
    return response
```

2. Create Agent

```
agent = Agent(client=client,
              model=os.getenv("INFERENCE_MODEL_ID"),
              instructions=(
                  "You are a web search assistant,
                     must use websearch tool to look up the most current and precise information available. "
              ),
              tools=[travily_web_search],
              tool_config=ToolConfig(tool_choice="required")
    )
```

3. Create Agent session

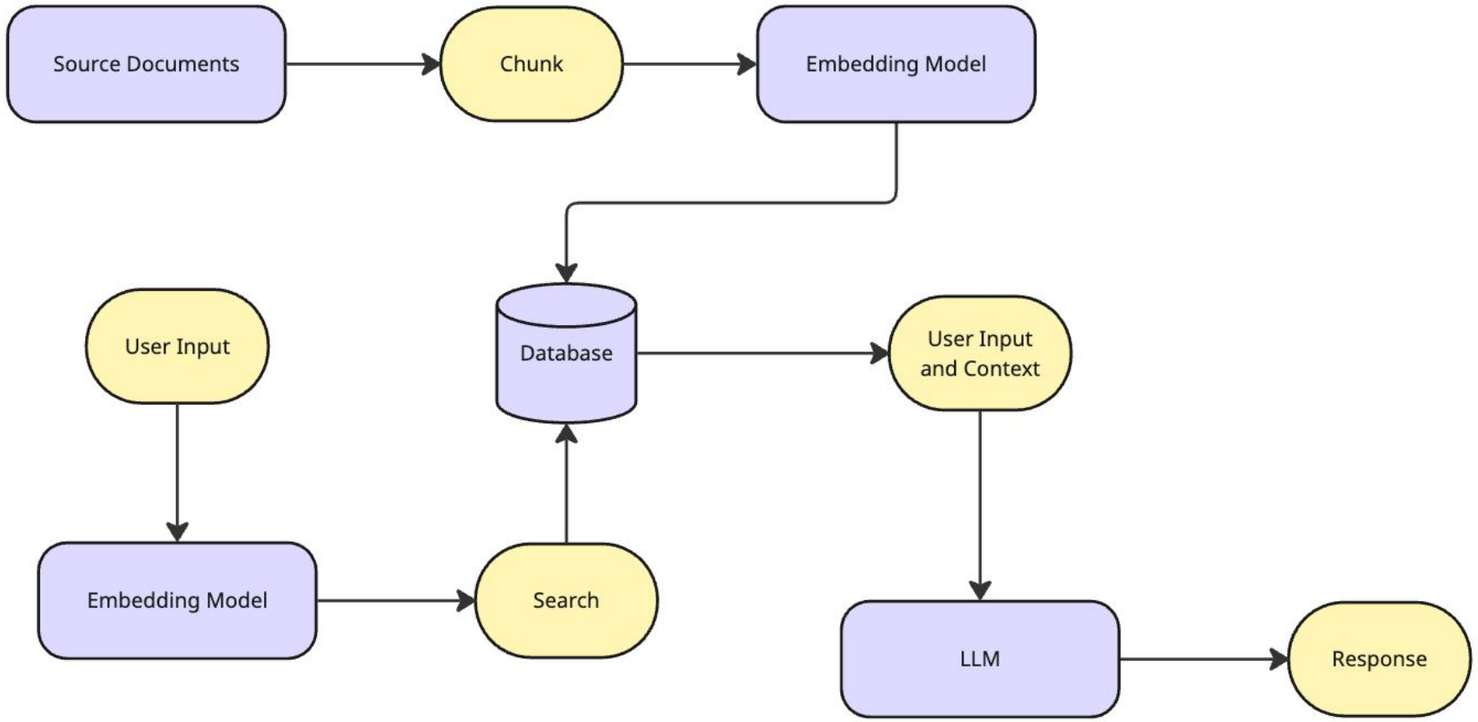
```
def create_websearch_tool_agent_session(prefix: str):
    return agent.create_session(f"{prefix}-{uuid.uuid4()}")
```

4. Create Websearch Tool Agent

```
def chat_with_websearch_tool_agent(session_id: str, user_message: str):
    response = agent.create_turn(
        messages=[UserMessage(role="user", content=user_message)],
        session_id=session_id,
        stream=False
    )
    print(f"user message: {user_message}, response: {response}")
    return response.output_message.content
```

Exercise 4 : Retrieval-Augmented Generation (RAG) Pipeline

1. What is RAG ?
2. How Llama Stack handles it ?
3. RAG on Devconf CZ talks data
    a. Pre-processing of the data.



1. Create Llama Stack Client and vector DB id.

```
vector_db_id = "dev_conf_cz_info"
client = LlamaStackClient(base_url=os.getenv("LLAMA_STACK_SERVER")
```

2. List all vector DBs

```
vector_dbs = client.vector_dbs.list()
```

3. Register Vector DB

```
client.vector_dbs.register(
    vector_db_id=vector_db_id,
    provider_id="faiss",
    embedding_model=os.getenv("EMBEDDING_MODEL_ID"),
    embedding_dimension=384
)
```

4. Insert Content

```
client.tool_runtime.rag_tool.insert(documents=fetch_dev_conf_talks_md(),
                                    vector_db_id=vector_db_id,
                                    chunk_size_in_tokens=512)
```

5. Query Rag

```
rag_results = client.tool_runtime.rag_tool.query(
    content="Find all talks on llama stack",
    vector_db_ids=[vector_db_id],
)
```

RAG with Inference

1. Query RAG

```
rag_results = client.tool_runtime.rag_tool.query(
    vector_db_ids=[vector_db_id],
    content=query,
)
print(rag_results)
context = [chunk.text for chunk in rag_results.content]
```

2. Set the RAG context

```
messages = [
    UserMessage(role="user", content=query),
    SystemMessage(role="system",
    content="You are a helpful assistant. Use the provided context to answer accurately. \nContext: \n"
+ "\n".join(context)),]
```

3. Pass the context + Query to the Chat completion

```
response = client.inference.chat_completion(
            model_id=os.getenv("INFERENCE_MODEL_ID"),
            messages=messages,)
return response.completion_message.content
```

Exercise 5 : RAG with Agent - Devconf CZ !

1. Create Agent with the Instructions

```
agent = Agent(
    client=client,
    model=os.getenv("INFERENCE_MODEL_ID"),
    instructions="You are a helpful Dev Conf CZ assistant. Use RAG tool to fetch context and provide
response to user query. If context is not available then use RAG tool.",
    tools=[ToolgroupAgentToolGroupWithArgs(
        name="builtin::rag/knowledge_search",
        args={
            "vector_db_ids": [vector_db_id]
        }
    )],
    )
```

2. Create Agent Session

```
def create_dev_conf_cz_agent_session(prefix: str):
    session_id = agent.create_session(f"{prefix}-{uuid.uuid4()}")
    print(f"session_id: {session_id}")
    return session_id
```

3. Chat with Agent

```
def chat_with_dev_conf_cz_agent(session_id: str, query: str):
    response = agent.create_turn(
        session_id=session_id,
        messages=[UserMessage(role="user", content=query)],
        stream=False
    )
    print(f"query: {query}, response: {response}")
    return response.output_message.content
```

Questions ?

Thank You