AKKA

QCon London

From Concept to Code

Navigating Agentic AI Services

Today’s discussion

AKKA

01 Welcome

Alan Klikic, Principal Solutions Architect, Akka

02 Agentic AI overview

03 A blueprint for agentic services

04 Implementing practical use cases

Incident resolution system (ReAct), Chatbot Assistant (RAG), Live video ASL recognition (RAG)

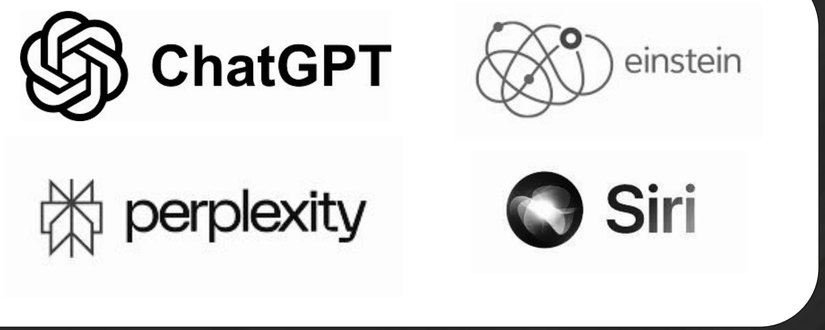
05 Q&A

AI is transforming our lives

AKKA

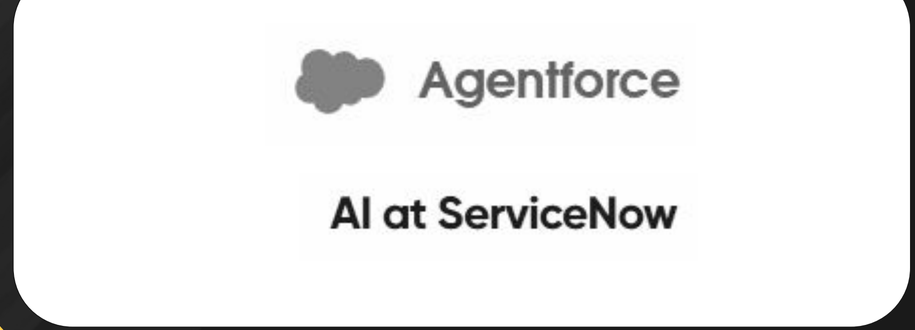
AI Assistant

A *user* app that understands natural language commands and uses a conversational AI interface to complete tasks on-demand.



AI Agent

A *system* that can autonomously fulfill goals by interacting with other systems, agents and humans.



Agentic AI overview

- static general knowledge
  - what they know is all they know
- can NOT perform any side effects
- stateless
- non-deterministic
- low throughput & costly
- conversational interaction

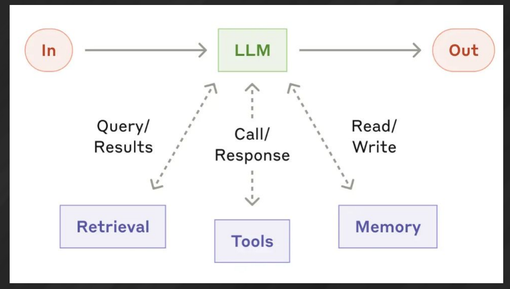
- common patterns to augment, reason, use external tools and human feedback loops
- stateful
- embrace failure

Customise, Automise & Optimise

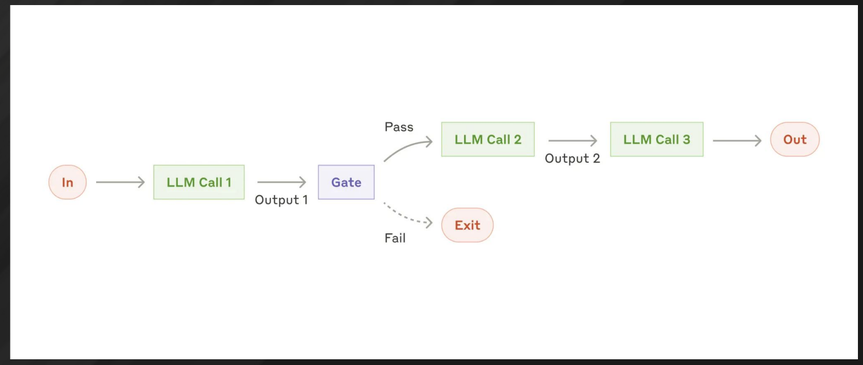
apply software engineering solutions/patterns

Common patterns:

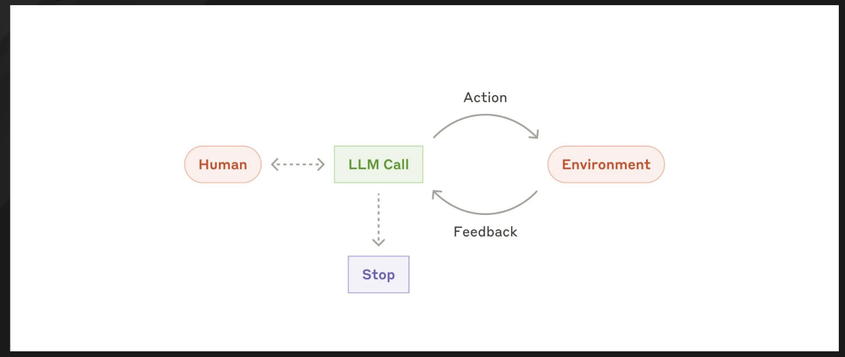
RAG



Workflow

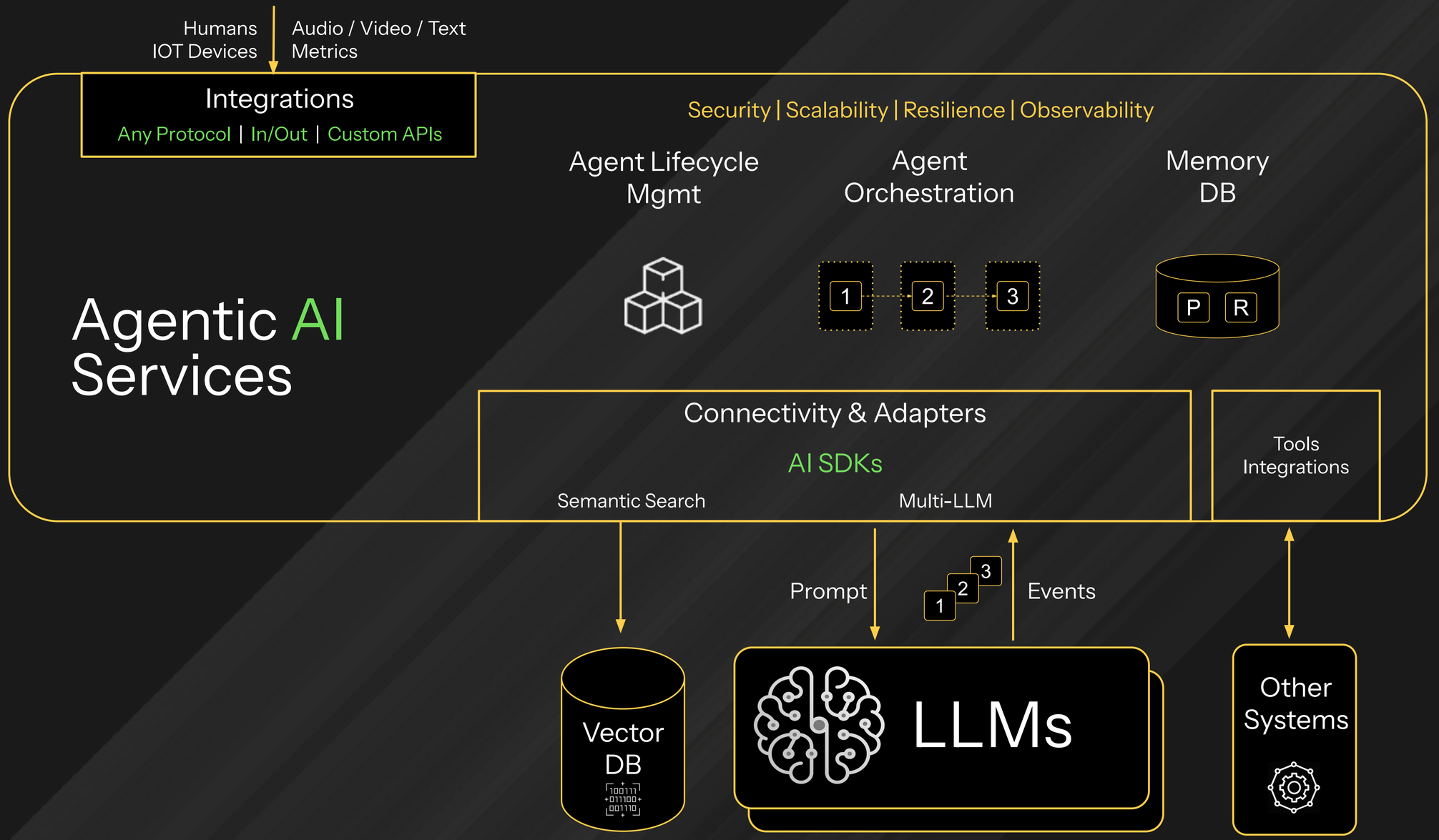


Agent



Source: https://www.anthropic.com/engineering/building-effective-agents

Blueprint for Agentic AI services



Akka

SDK - Components

Entities
Build apps that act as their own in-memory, durable, and replicated database.

Workflows
Execute durable, long-running processes with point-in-time recovery.

Streaming
Streaming producers and consumers enable real-time data integration.

Timers
Execute actions with a reliability guarantee.

Endpoints
Design HTTP and gRPC APIs.

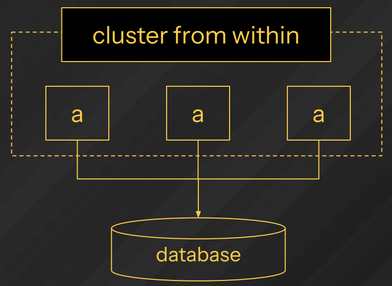
Views
Access multiple entities or retrieve entities by attributes other than entity id.

Ops environments

Offline development

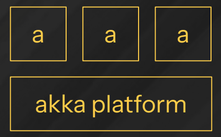
Dev, debugging, functional testing, no infra → low friction

Self-managed nodes



Akka clusters on bare metal, VMs, container PaaS, edge or k8s with do-it-yourself ops.

Fully-automated regions

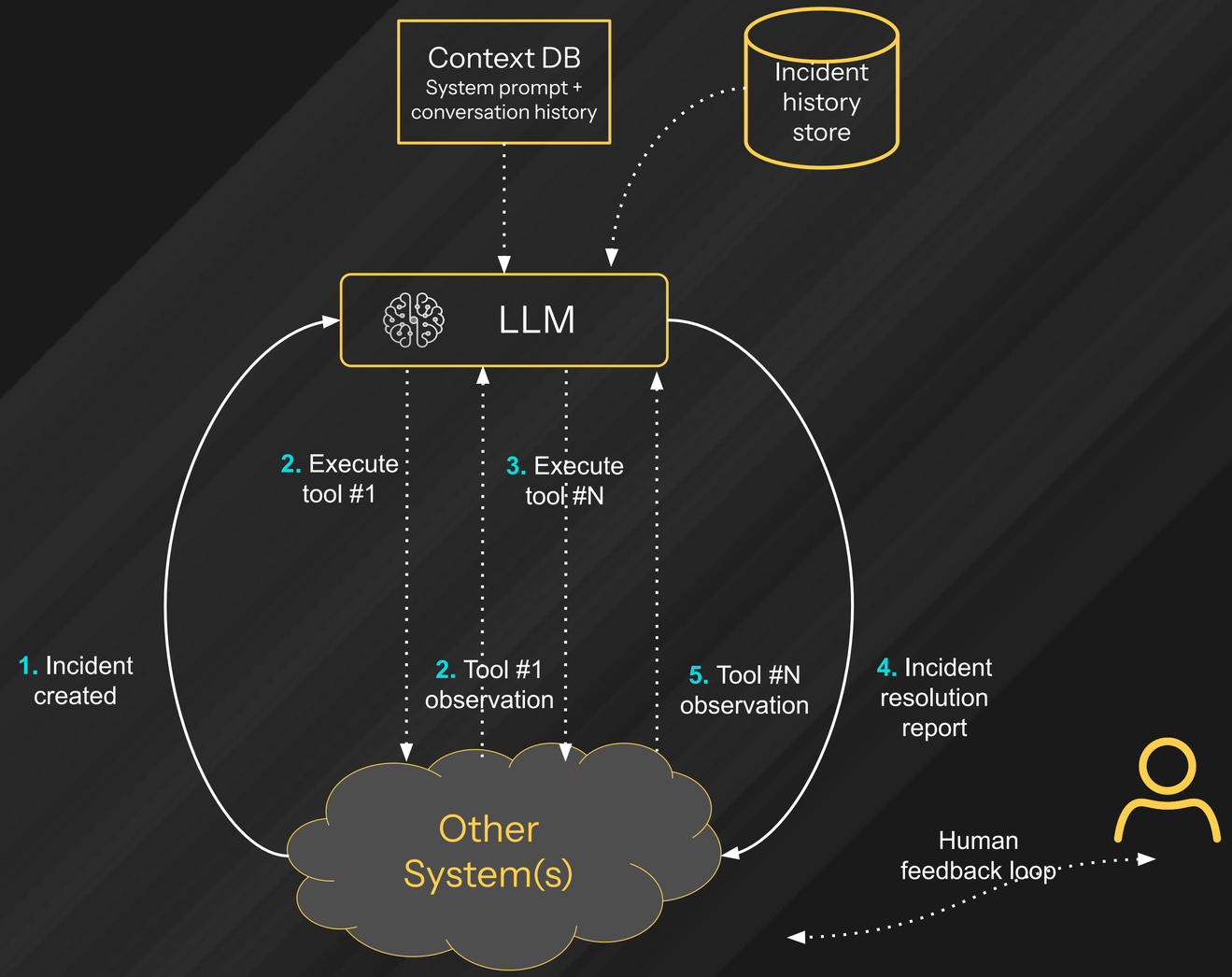


Automated, multi-tenant, multi-region, secure infrastructure

- Akka's Serverless environment
- your VPC with BYOC (managed by Akka)
- Self-Hosted (managed by you).

Use case: Incident resolution system

Pattern: **Agent** - ReAct (Reason - Act) - Chain of Thought (CoT) prompting



Use case: Incident resolution system

AKKA

System prompt:

You run in a loop of Thought, Tool, PAUSE, Observation.

At the end of the loop you output an Answer, when you have the final answer. Make sure you include "Answer:" in your output in this case.
Use Thought to describe your thoughts about the question you have been asked.

Use Tool to run one of the tools available to you - then return PAUSE.

Make sure you include exactly the term "PAUSE" in your output in this case.

Observation will be the result of running those tools.

Your available tools are:

```
[
{
  "get_customer_details" : {
    "description": "takes userId as input and outputs user:{userId, name, address, age, postalcode} as output",
    "example": "
        tool: get_customer_details
        input: 123
        output: {userId:123, name:ABC, address:123 sfd street, age: 44, postalcode: 1234} s
        "
    },
  "get_sales_details" : {
    "description":"takes userId as input and outputs The invoice with {userId, invoiceId, amount, date }"
    "example": "
        tool: get_sales_details
        input: user123
        output: {userId:user123 , invoiceId: inv001, amount: $1000, date:  2025-03-01}
        "
    },
  "get_payment_details": {
    "description":"takes userId as input and outputs get_payment_details {paymentId, userId, invoiceId, amount, date},
    "example": "
        tool: get_payment_details
        input: user123
        output: {paymentId:p001, userId:user123, invoiceId:inv001, amount: $1000, date: 2025-03-01}
        "
    },

  "update_payment_request": {
  "description": "If there is any discrepancy in the reconciliation due to invoice mismatch or missing billing address details, use this tool to get the additional discrepancy payment or billing address.
  The INPUT MUST BE FORMED with values separated by a comma separated String. The tool uses string operation like split. So give in the same format. I TRUST YOU for the input format.Dont pass black values amount. default shoudl be 0 for amoutn. Please."
  "example": "
      tool: retry_payment_request
      input: <payment Id>,<invoice_id>,<discrepancy-amount>,<customer_id>,<payment_date>,<billing_address>,<postal_code>
      output: 201
  "
}
 ]
```

Example session 1:

```
Question: What is the capital of France?
{
"Thought": "I should look up France on Wikipedia.",
"Tool": "wikipedia: ",
"input: "France",
"PAUSE" : true
}
You will be called again with this:
{"Observation": "France is a country. The capital is Paris"}
You then output:
{"Answer": "The capital of France is Paris"}
====================================
```

Example session 2:

```
Question: Who is the current president of the United States?
{
"Thought": "I should look up the current president of the United States on duckduckgo.",
"Tool": "duckduckgo",
"input": "current president of the United States",
"PAUSE": true
}
You will be called again with this:
{"Observation": "Joe biden elected president of the United States in 2024."}
You then output:
{"Answer": "Joe biden is the current president of the United States"}
====================================
```

Example session 3:

```
Question: What is the result of this expression "2 * 3 + 4 / 2 * 10 + 12 / 32"?

{
"Thought": "I should calculate using the calculator tool.",
"Tool": "calculator",
"input":  "2 * 3 + 4 / 2 * 10 + 12 / 32",
"PAUSE": true
}
You will be called again with this:
{"Observation": "26.375"}
You then output:
{"Answer": "The result of "2 * 3 + 4 / 2 * 10 + 12 / 32" is 26.375"}
====================================
```

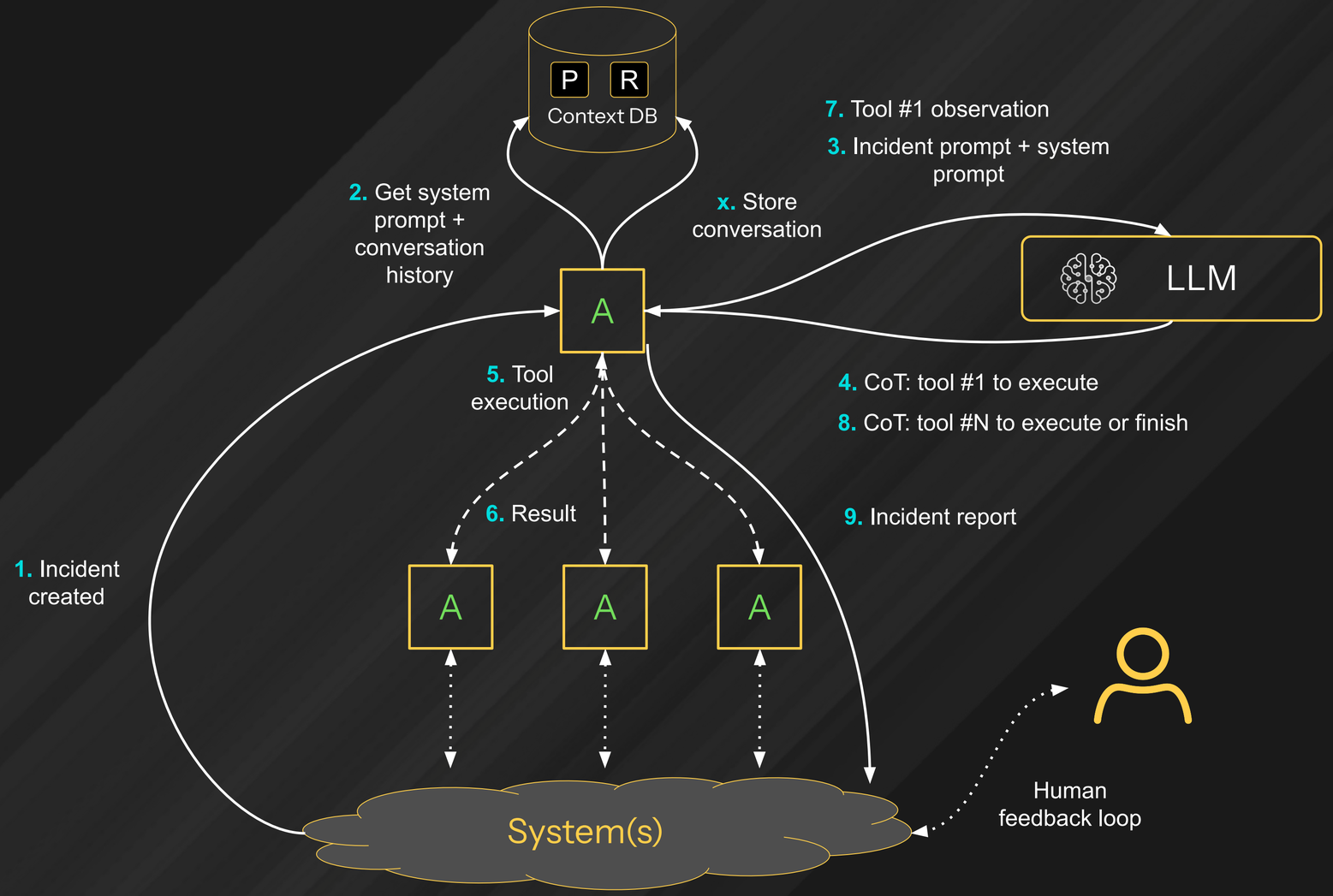
Example session 4:

```
Question: What are the betting sites predicting on the American Presidential Election? Who is going to win according to them?
{
"Thought": "I should look up the current president of the United States on serp_api_tool.",
"Tool": "serp_api_tool",
"input": "American Presidential Election betting odds"
"PAUSE": true
}
You will be called again with this:
{"Observation": "Donald Trump has a 48% per cent chance of winning against Kamala Harris with a 51% chance."}
You then output:
{"Answer": "Joe biden is the current president of the United States"
```

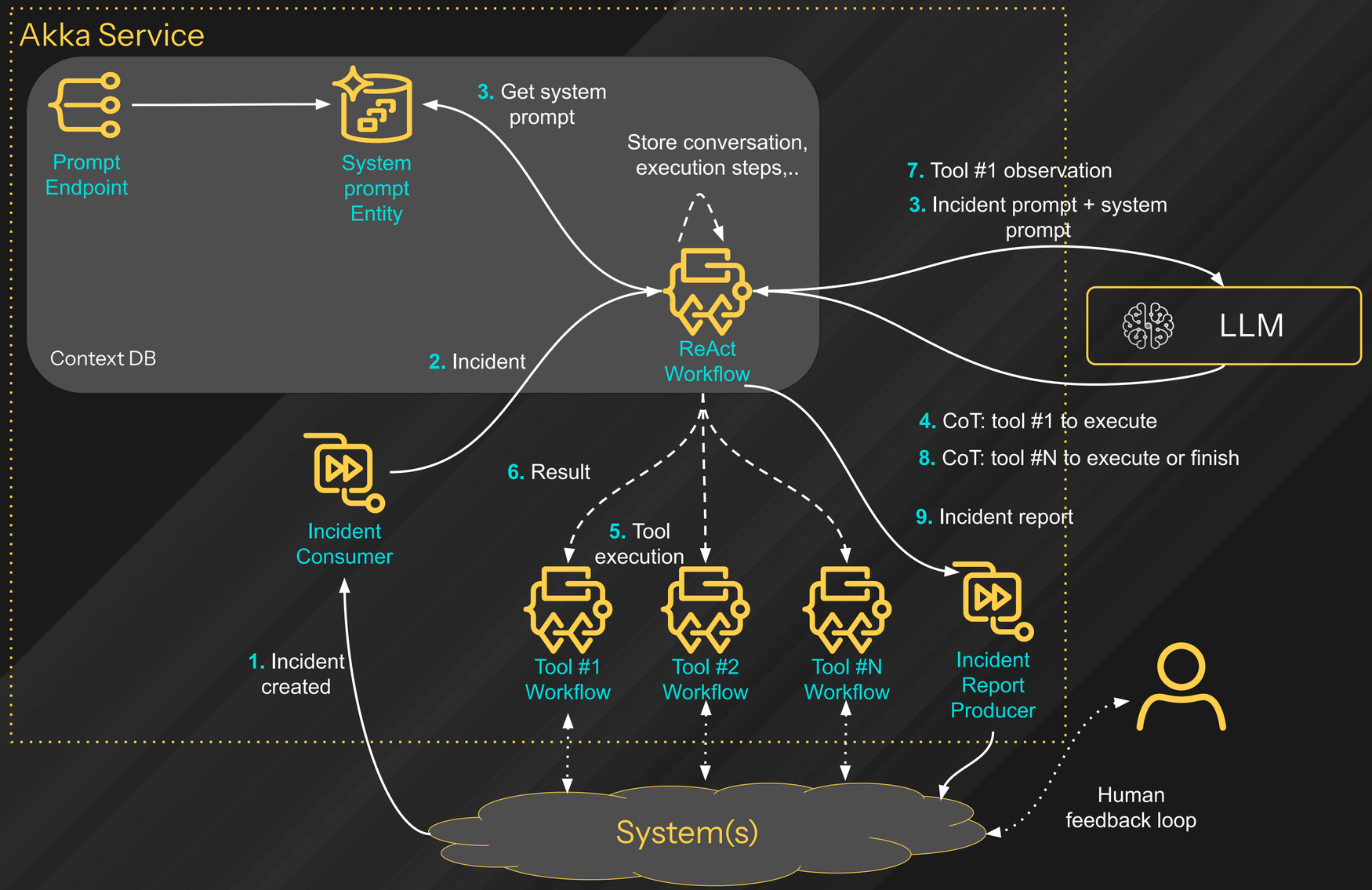
Model Context Protocol

Use case: Incident resolution system

Pattern: **Agent** - ReAct (Reason - Act) - Chain of Thought (CoT) prompting



Use case: Incident resolution system



Demo

Incident Resolution System

Use case: Incident resolution system

ReAct Workflow

```
// Workflow orchestrates long running requests with durable execution
class ReActFlow extends Workflow<ReActFlowState> {

  public Effect<Done> start(SystemPrompt systemPrompt){
     return effect().updateState(currentState().start(systemPrompt)).transitionTo(SEND_TO_LLM_STEP);
  }
  public Effect<Done> definition() {

     // send to LLM using AI SDKs
     var sentToLLM = step(SEND_TO_LLM_STEP)
                     .call(<< AI SDKs >>)
                     .andThen(res -> {
                        if(res.isError())
                            return effect().updateState(currentState().llmFailed(res)).transitionTo(HUMAN_LOOP_STEP);
                        else
                           return effect().updateState(currentState().llmSuccess(res)).transitionTo(getToolStep(res));
                     });
     // delegate tool execution to other workflow
      var tool1Exec = step(TOOL1_EXEC_STEP)
                     .call(componentClient.forWorkflow(currentState().toolId()).method(Tool1Workflow::start).invokeAsync(currentState().toolRequest());
                     .andThen(res ->
                          effect().updateState(currentState().tool1Started()).pause();
                     );

     // delegate tool execution to other workflow
     var toolNExec = step(TOOLN_EXEC_STEP) << tool N step definition >>

     // delegate tool execution to other workflow
     var toolForHumanLoopExec = step(HUMAN_LOOP_STEP) << tool for human loop definition>>

     // Orchestration steps defined; can add failover and exception steps
     return workflow()
         .addStep(sentToLLM, failoverTo(HUMAN_LOOP_STEP))
         .addStep(tool1Exec, maxRetries(3).failoverTo(SEND_TO_LLM_STEP))
         .addStep(toolNExec, maxRetries(3).failoverTo(SEND_TO_LLM_STEP))
         .timeout(Duration(2,SECONDS));
  }
  public Effect<Done> toolObservationReply(ToolObservation res){
     return effect().updateState(currentState().toolObservationReceived(res)).transitionTo(SEND_TO_LLM_STEP);
  }

}
```

Use case: Incident resolution system

AKKA

```
// Workflow orchestrates long running requests with durable execution
class Tool1Flow extends Workflow<Tool1FlowState> {

  public Effect<Done> start(ToolReuqest req){
     return effect().updateState(currentState().start(req)).transitionTo(EXEC);
  }
  public Effect<Done> definition() {

     // send to LLM using AI SDKs
     var externalExec = step(EXEC_STEP)
                        .call(
                            << API call, push to message broker,... >>
                        .andThen(res →
                             effect().updateState(currentState().observation(res)).transitionTo(SEND_OBSERVATION_STEP)
                        );

     // send observation back to ReAct Workflow
      var sendObservation = step(SEND_OBSERVATION_STEP)
                            .call(
                                  componentClient.forWorkflow(currentState().reActWorkflowId()).method(ReActFlow::toolObservationReply)
                                                                                               .invokeAsync(currentState().observation())
                                                .andThen(res →
                                 effect().updateState(currentState().observation(res)).end()
                            );

     // Orchestration steps defined; can add failover and exception steps
     return workflow()
         .addStep(externalExec, maxRetries(3).failoverTo(SEND_OBSERVATION_STEP))
         .addStep(sendObservation)
  }
}
```

Tool1
Workflow

Use case: Incident resolution system

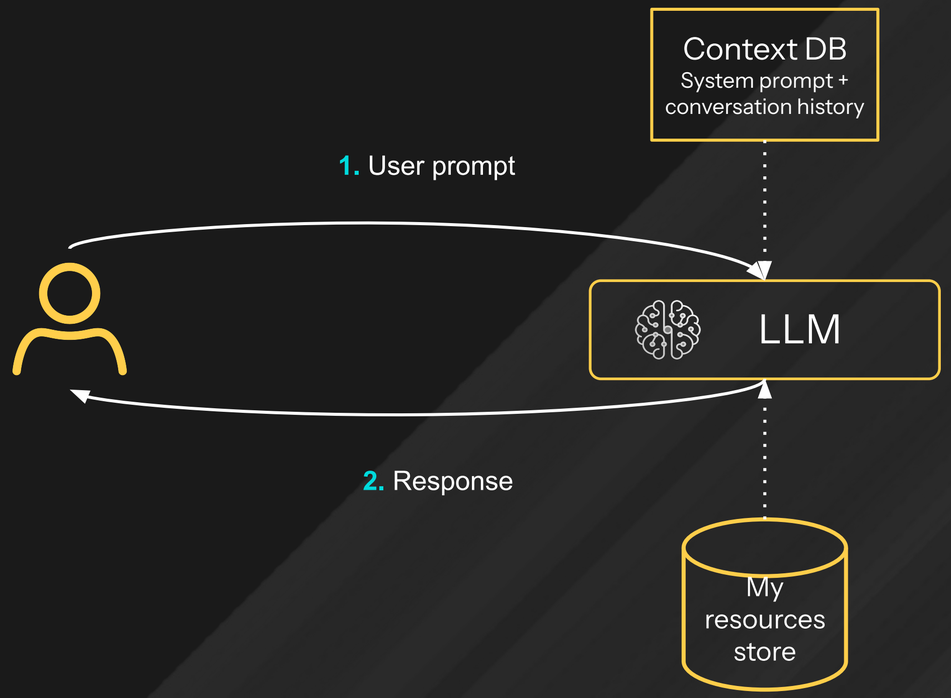
Incident Consumer

```
// Consume from external Topic
@Consume.FromTopic(INCIDENT_TOPIC)
class IncidentConsumer extends Consumer {

  public Effect onMessage(ExternalMessage message) {
        return componentClient.forWorkflow(message.incidentId).method(ReActFlow::start).invokeAsync(message);
  }
}
```

Incident Report Producer

```
// Consume from Workflow
@Consume.FromWorkflow(ReActFlow)
class IncidentReportProducer extends Consumer {

  // Produce to external Topic
  @Produce.ToTopic(REPORT_TOPIC)
  public Effect<ExternalMessage> onWorkflowStateChange(ReActFlowState stateChange) {
        return effect().produce(ExternalMessage.of(state))
  }
}
```

Use case: Chatbot Assistant

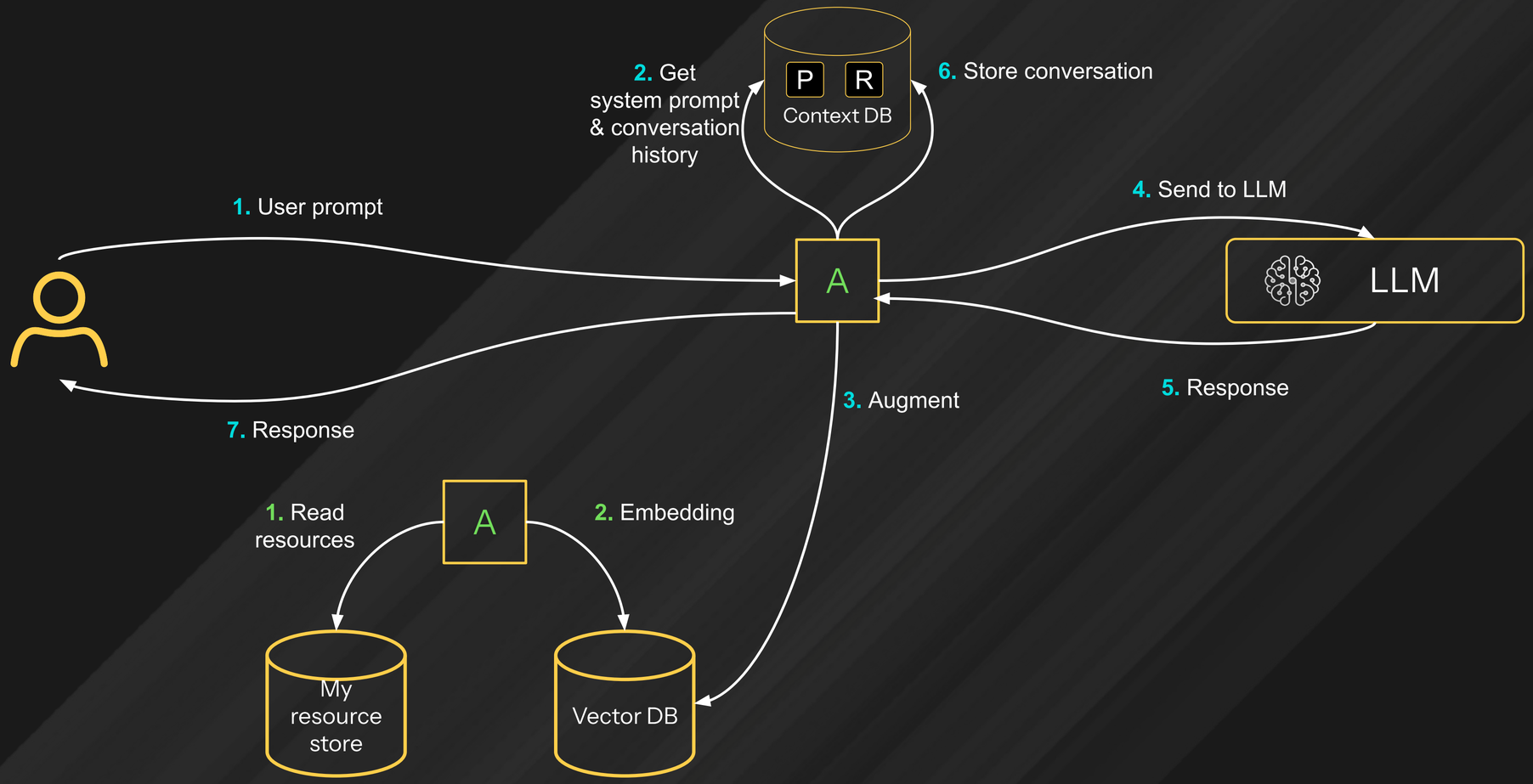


System prompt:

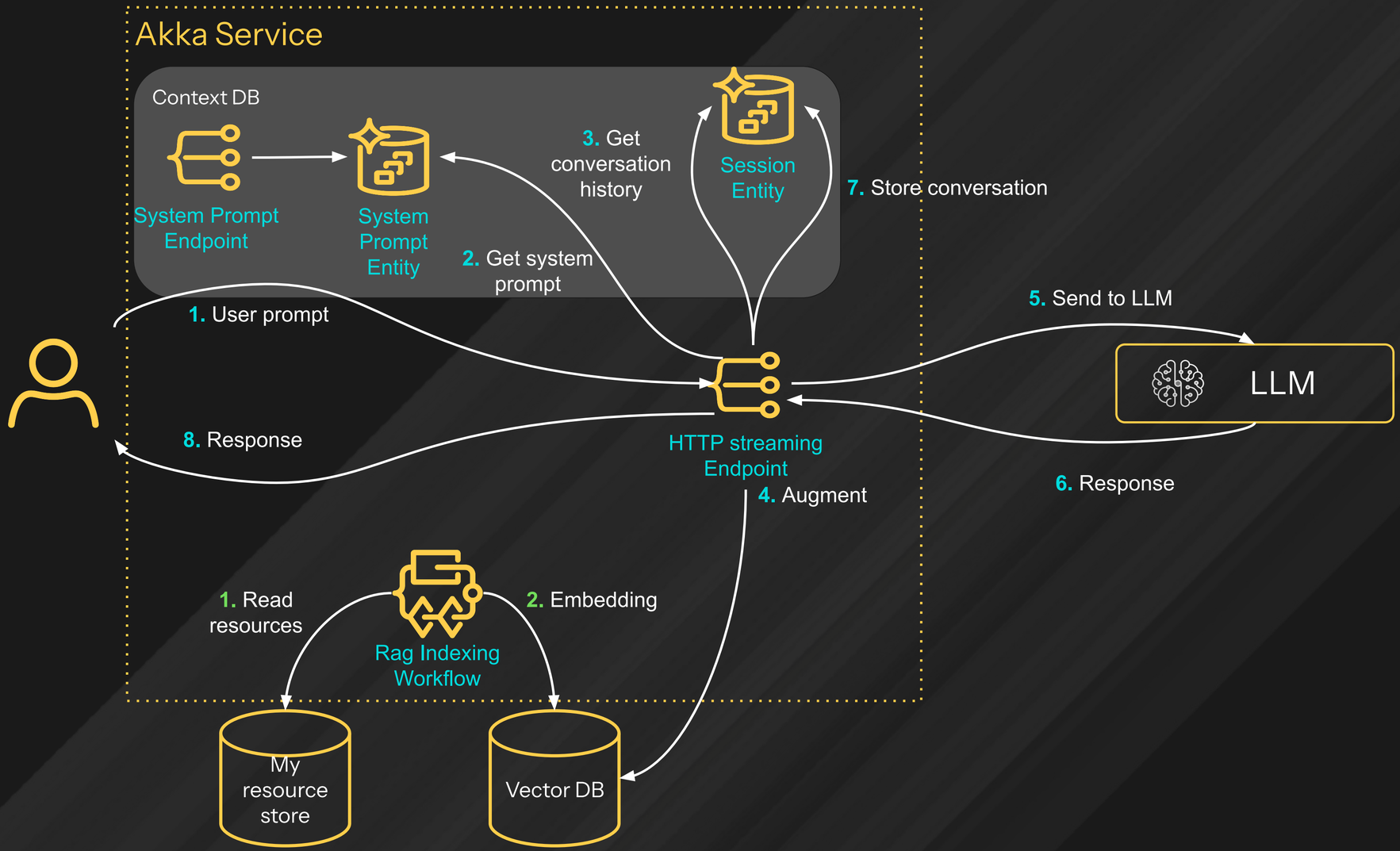
```
You are a very enthusiastic Akka representative who loves to help people!
Given the following sections from the Akka SDK documentation, answer the question using only that information, outputted in markdown format.
If you are unsure and the text is not explicitly written in the documentation, say:
Sorry, I don't know how to help with that.
```

Use case: Chatbot Assistant

Pattern: **RAG** (Retrieval Augmented Generation)



Use case: Chatbot Assistant



Demo

Chatbot Assistant

Use case: Chatbot Assistant

Document Indexing Workflow

```
record RagIndexingFlowState(List<Path> toProcess, List<Path> processed){
   public State listFailed(ListingResult res)<<impl>>
   public State listDone(ListingResult res)<<impl>>
   public State embeddingDone(EmbeddingResult res)<<impl>>
}


// Workflow orchestrates long running requests with durable execution
class RagIndexingFlow extends Workflow<RagIndexingFlowState> {

  public Effect<Done> definition() {
     // integration with the resource source(s)
     var list = step(LIST_STEP)
                    .call(<< GIT/S3/DB/NSF list and load >>)
                    .andThen(res -> {
                       if(res.isError())
                           return effect().updateState(currentState().listFailed(res)).transitionTo(LIST_STEP);
                       else
                           return effect().updateState(currentState().listDone(res)).transitionTo(EMBEDDING_STEP);
                    });
     // use AI SDKs for embeddings and connection to vector DB
     var embedding = step(EMBEDDING_STEP)
                     .call(<< embeddings and storing to Vector DB -> AI SDKs >>)
                     .andThen(resOrErr ->
                          effect().updateState(currentState().embeddingDone(res)).transitionTo(LIST_STEP)
                     );

     // Orchestration steps defined; can add failover and exception steps
     return workflow()
            .addStep(list)
            .addStep(embedding, maxRetries(3).failoverTo(LIST_STEP))
            .timeout(Duration(2,SECONDS));

  }
  public Effect<Done> start(){
     return effect().updateState(currentState().start()).transitionTo(LIST_STEP);
  }
}
```

Use case: Chatbot Assistant

Session Entity

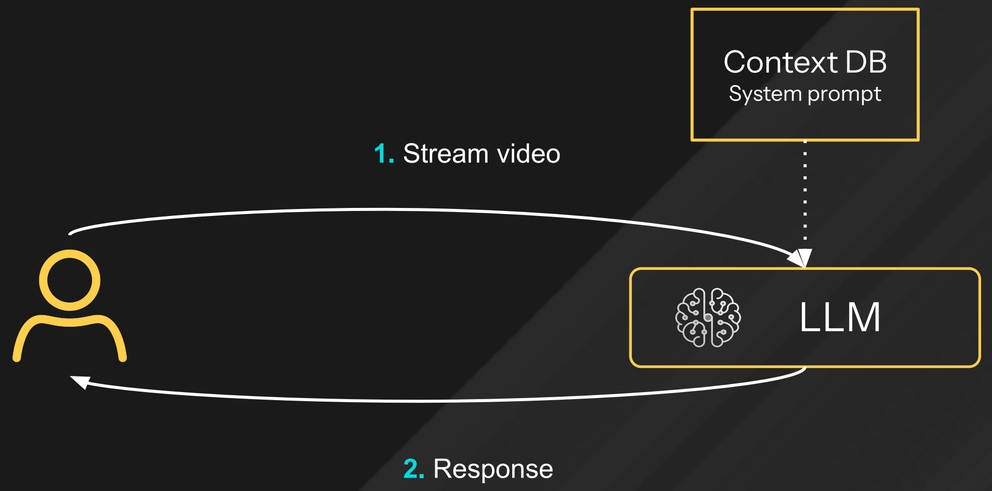
```
// state domain model
record SessionState(List<Conversation> conversationList){}

// Entities capture state changes, are persisted into journal
class SessionEntity extends EventSourcedEntity<SessionState, SessionEvent> {

  // Command handler to capture prompts
   public Effect<ConversationList> storeConversation(Conversation cmd) {
      return effects()
             .persist(SessionEvent.ConversationAdded.of(cmd.origin(), cmd.content()))
             .thenReply(state -> state.conversationList());
   }
   // Command handler to return conversation history from the state
   public Effect<ConversationList> get() {
      return effects().reply(currentState().conversationList());
   }
}
```

HTTP Endpoint

```
// HTTP endpoint accepts exception process requests, returns AI suggestions
class HttpStreamingEndpoint {
  // Akka HTTP endpoint implementation
  @Post("/{sessionId}")
  public HttpResponse ask(String sessionId, String question) {

     var conv  = componentClient.forEventSourceEntity(sessionId).method(SessionEntity::storeConversation)
                                                                .invokeAsync(Conversation.of("user",llmResponse.text()));

     var systemPrompt = componentClient.forEventSourceEntity(SYSTEM_PROMPT).method(SystemPromptEntity::get).invokeAsync();

     var res = allOf(convHistory, systemPrompt)
               .thenCompose(systemPromptAndConvHistory ->
                    << rag and send to LLM -> AI SDKs >>
               ).thenCompose(llmResponse ->
                    componentClient.forEventSourceEntity(sessionId)
                                   .method(SessionEntity::storeConversation).invokeAsync(Conversation.of("llm",llmResponse.text()))
                                   .thenAppy(d -> llmResponse)
               );

     return HttpResponse.serverSentEvents(res);
  }
```

Use case: Live video ASL recognition

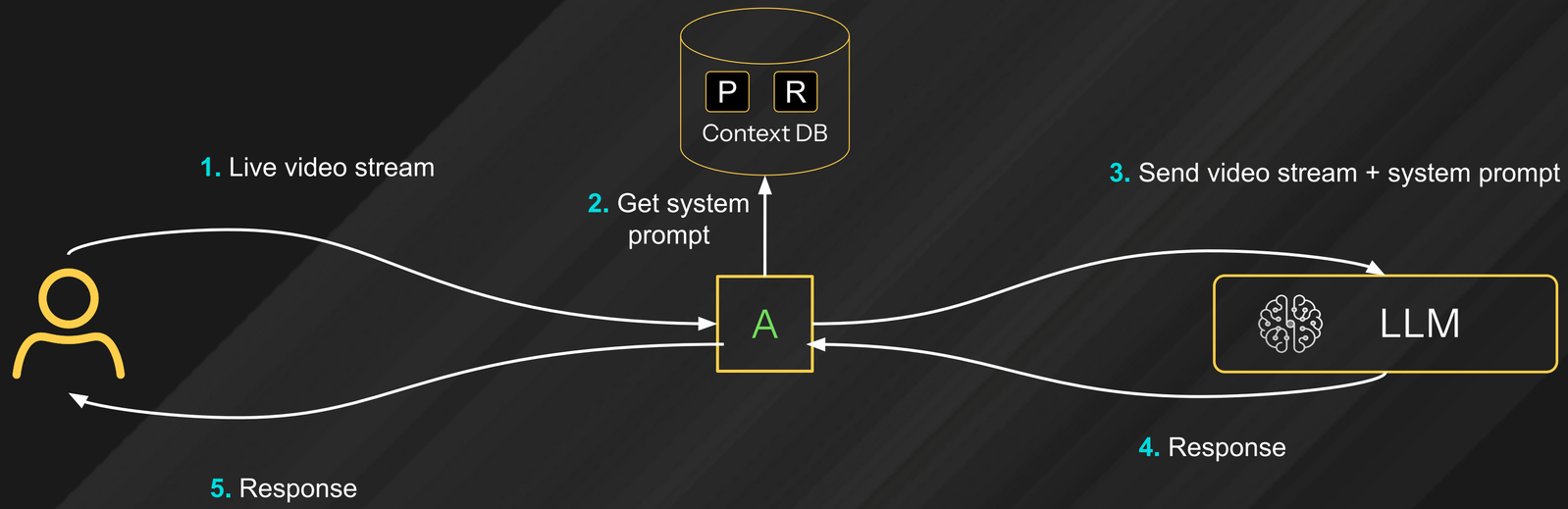
Pattern: **RAG** (Retrieval Augmented Generation)



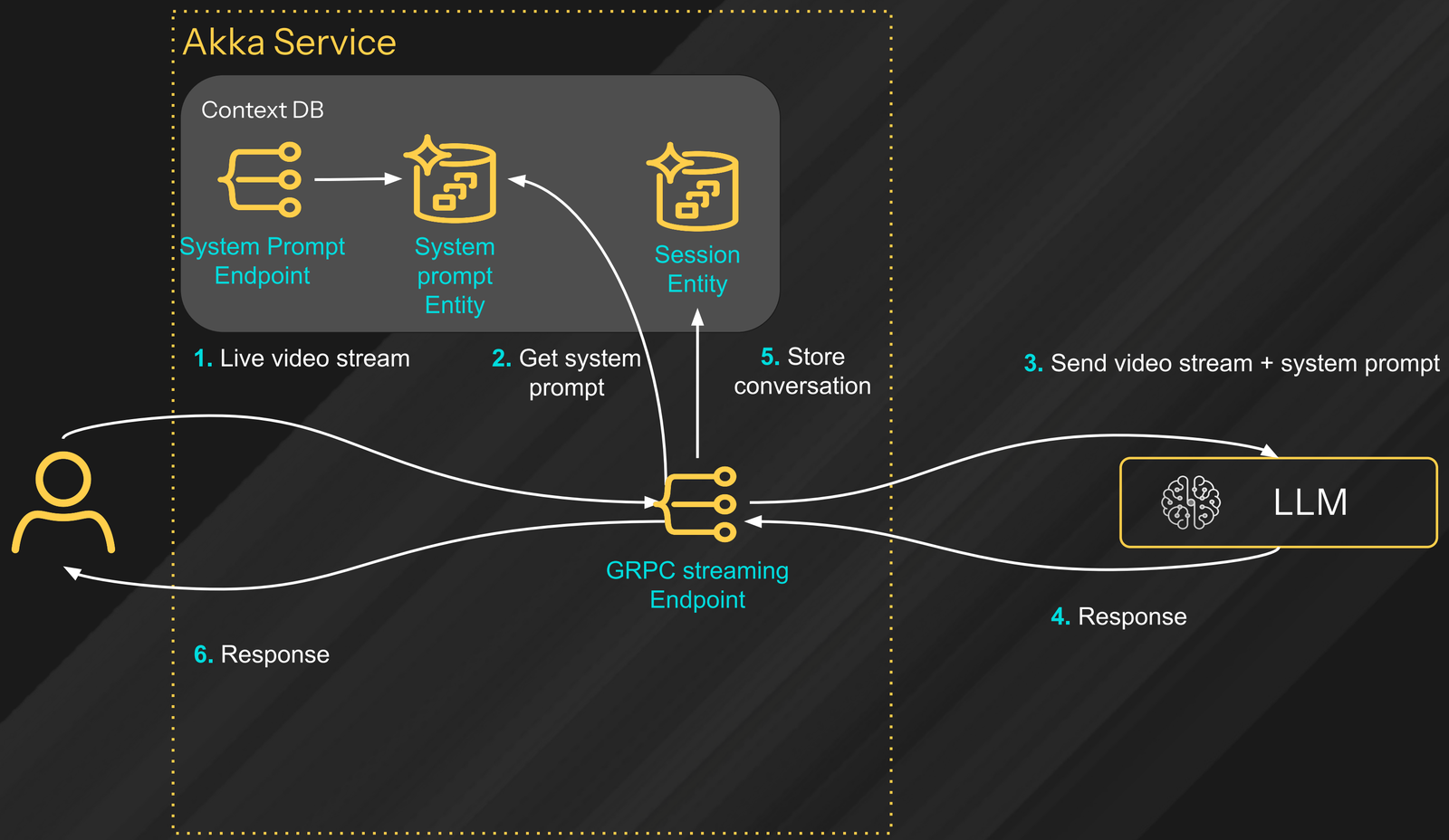
System prompt:

```
Alan is trying to learn ASL alphabet.
He will show you a sign and guess what that he is showing.
Please tell him correct or tell him the actual sign he is holding up.
If he is not holding up a valid ASL alphabet sign then please tell him it is not a valid sign.
```

Use case: Live video ASL recognition



Use case: Live video hand signal recognition



Demo

Live video ASL recognition

Use case: Live video hand signal recognition

GRPC Streaming Endpoint

```
// gRPC endpoint accepts streaming input, sends streaming response
// Protobuf definition
service VideoStreamingEndpoint {
   rpc StreamVideo (stream Chunk) return (stream Ack)
}

// Akka gRPC endpoint implementation
public Effect<Ack, NotUsed> streamVideo(Source<Chunk, NotUsed> in, String sessionId) {

  var client = << Gemini streaming client >>

  // Concatenate liveContentStream with system prompt and conversation history

  var systemPrompt = componentClient.forEventSourceEntity(SYSTEM_PROMPT).method(SystemPromptEntity::getSystemPrompt).invokeAsync();

  var contentStream = in.concat(Source.from(systemPrompt));

  // Send contentStream and handle the many Gemini responses continuously
  return client.connect(contentStream)
               .map(liveServerMessage →
                  componentClient.forEventSourceEntity(sessionId).method(SessionEntity::storeConversation).invokeAsync(Response.of(liveServerMessage))
                );
}
```

Q&A

concept

proof

48 hours