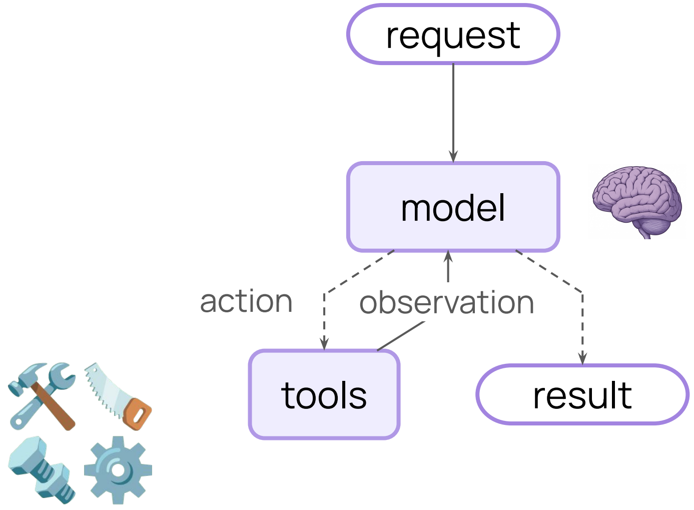
LangChain

LangChain Essentials

Create Agent

Outline

LangChain Agent



ReAct Agent

- ReAct: Synergizing Reasoning and Acting in Language Models.
  - Reason
  - Action
  - Observation

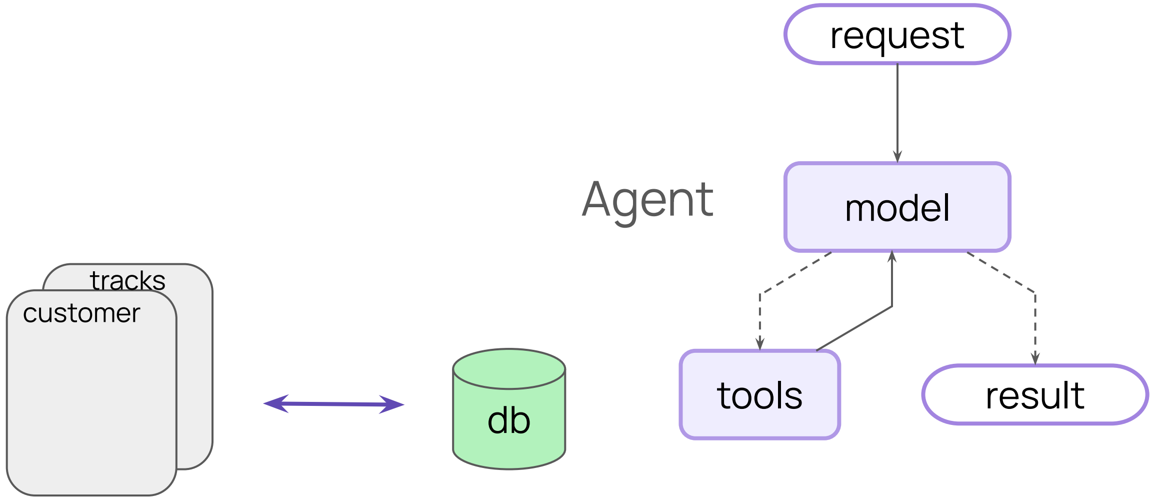
LangChain Agent

```
from langchain.agents import create_agent

agent = create_agent(
    model="openai:gpt-5",
    system_prompt="you are…",
    tools=tools
)
```

ReAct Agent

- ReAct: Synergizing Reasoning and Acting in Language Models.
    - Reason
    - Action
    - Observation
- Built with LangGraph
    - Persistence
    - Streaming
    - Interrupts (Human-In-The-Loop)
    - Tracing (LangSmith Observability & Evals)
    - Deployment (LangSmith Deployment)
- Quick to build

Lab Setup



```python
def execute_sql(query: str) -> str:
    """Execute a SQLite command and return results."""
```

- Chinook (music shop) example database
- The agent is not provided the schema (it will have to figure it out!)
- Note - you would want to add further safeguards in a production setting!
- Run in an editor and then in the agent debugger!

```python
db = SQLDatabase.from_uri("sqlite:///Chinook.db")
@dataclass
class RuntimeContext:
    db: SQLDatabase


@tool
def execute_sql(query: str) -> str:
    """Execute a SQLite command and return results."""
    runtime = get_runtime(RuntimeContext)
    db = runtime.context.db

    try:
        return db.run(query)
    except Exception as e:
        return f"Error: {e}"

SYSTEM = f"""You are a careful SQLite analyst.
Rules:
- Think step-by-step.
- When you need data, call the tool `execute_sql` with ONE SELECT query.
- Read-only only; no INSERT/UPDATE/DELETE/ALTER/DROP/CREATE/REPLACE/TRUNCATE.
- Limit to 5 rows of output unless the user explicitly asks otherwise.
- If the tool returns 'Error:', revise the SQL and try again.
- Prefer explicit column lists; avoid SELECT *.
"""

agent = create_agent(
    model="openai:gpt-5",
    tools=[execute_sql],
    system_prompt=SYSTEM,
    context_schema=RuntimeContext,
)
```

```python
question = "Which table has the largest number of
entries?"

for step in agent.stream(
    {"messages": question},
    context=RuntimeContext(db=db),
    stream_mode="values",
):
    step["messages"][-1].pretty_print()
```

Models

Messages/Prompts

Streaming

Tools

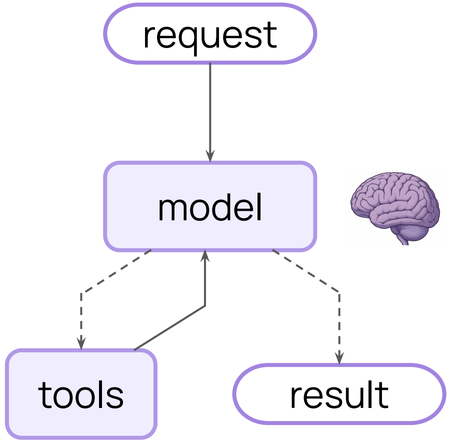
Runtime Context

LangChain

Outline

Models and Messages

Models

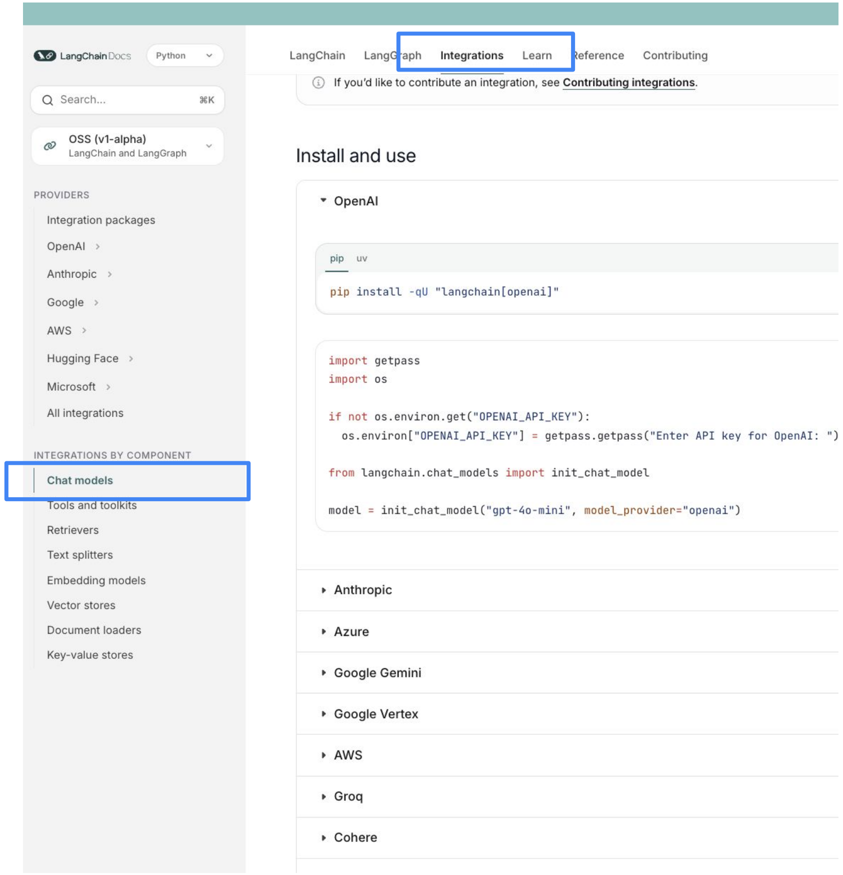


- The 'Reasoning' in ReAct.
- Agents typically use 'chat' models
- LangChain supports over 100 model vendors

```
llm = init_chat_model("openai:gpt-5")
```

```
agent = create_agent(
    model=llm,
    tools=[execute_sql],
    system_prompt=SYSTEM_PROMPT
)
```

```
agent = create_agent(
    model="openai:gpt-5",
    tools=[execute_sql],
    system_prompt=SYSTEM_PROMPT
)
```



If you'd like to contribute an integration, see Contributing integrations.

Install and use

OpenAI

pip uv

```
pip install -qU "langchain[openai]"
```

```
import getpass
import os

if not os.environ.get("OPENAI_API_KEY"):
  os.environ["OPENAI_API_KEY"] = getpass.getpass("Enter API key for OpenAI: ")

from langchain.chat_models import init_chat_model

model = init_chat_model("gpt-4o-mini", model_provider="openai")
```

Anthropic

Azure

Google Gemini

Google Vertex

AWS

Groq

Cohere

Models

To explore models, see

https://docs.langchain.com/oss/python/integrations/chat

LangChain

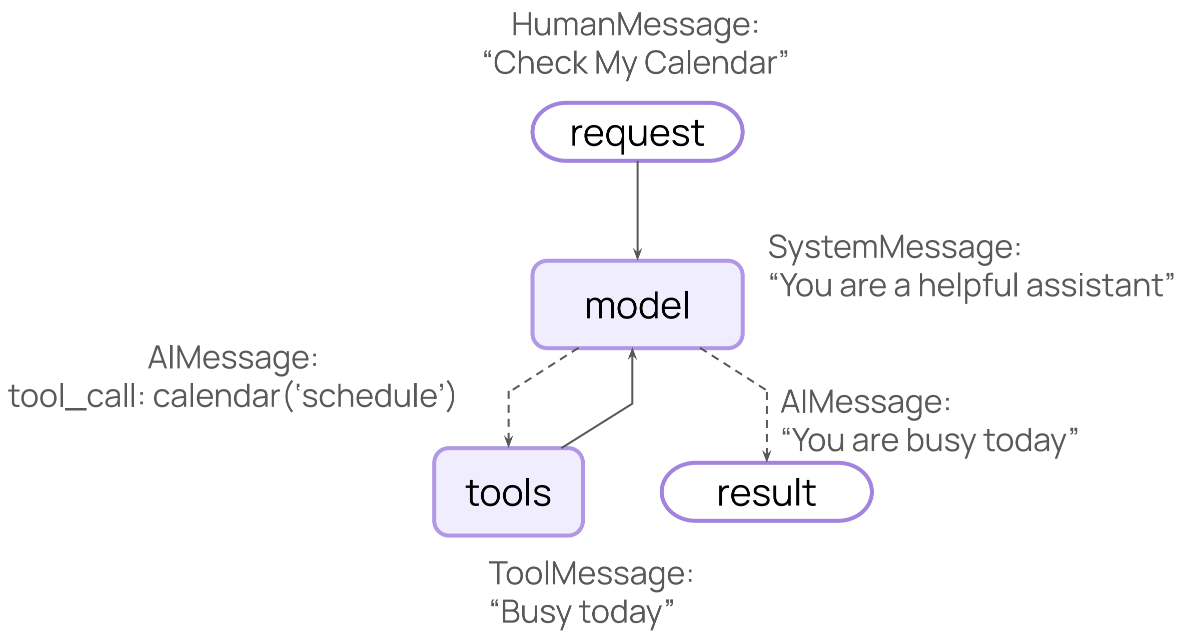
Messages

Messages

Elements of the system communicate by passing messages.

There are several types”

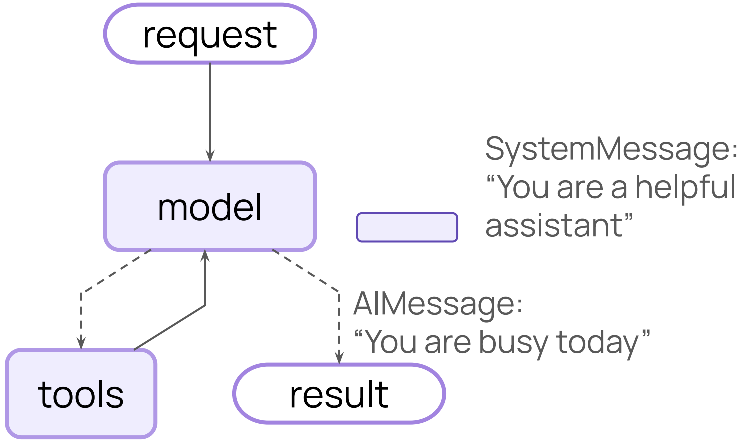
- SystemMessage
- HumanMessage
- AIMessage
- ToolMessage



Messages

State:
“Messages” : [list of messages]

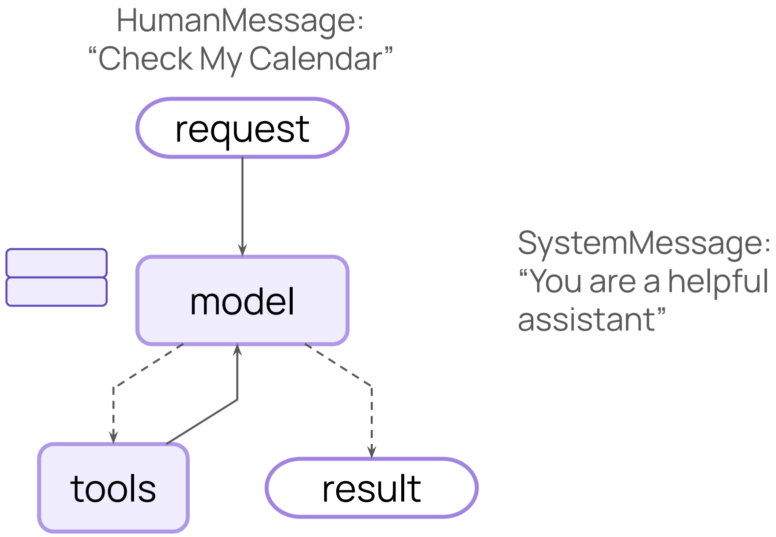
SystemMessage:
“You are a helpful assistant”



Messages

State:
"Messages" : [list of messages]

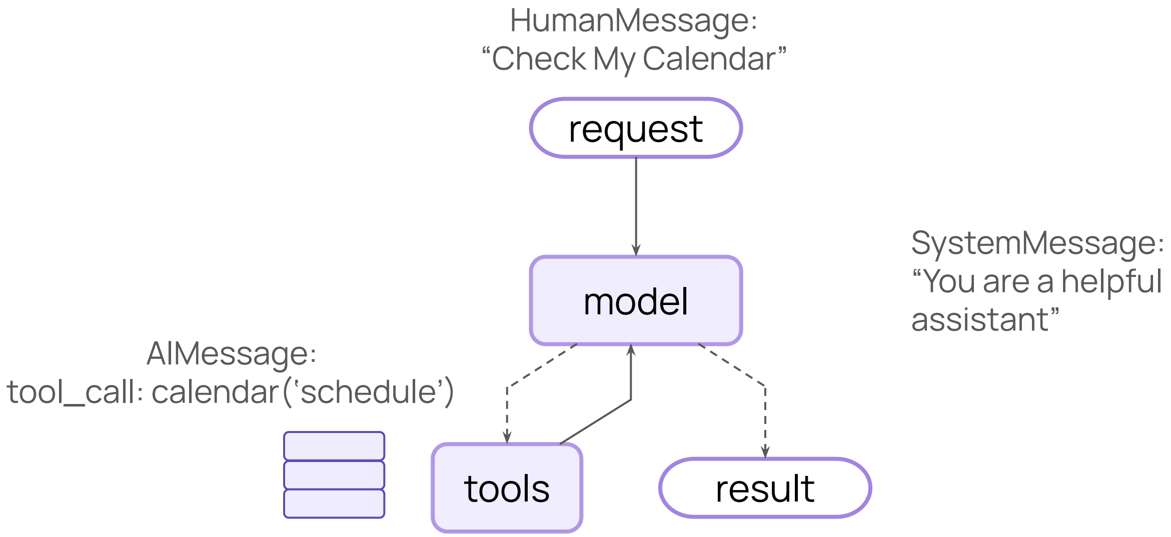
SystemMessage: "You are a helpful assistant"
HumanMessage: "Check My Calendar"



Messages

State:
"Messages" : [list of messages]

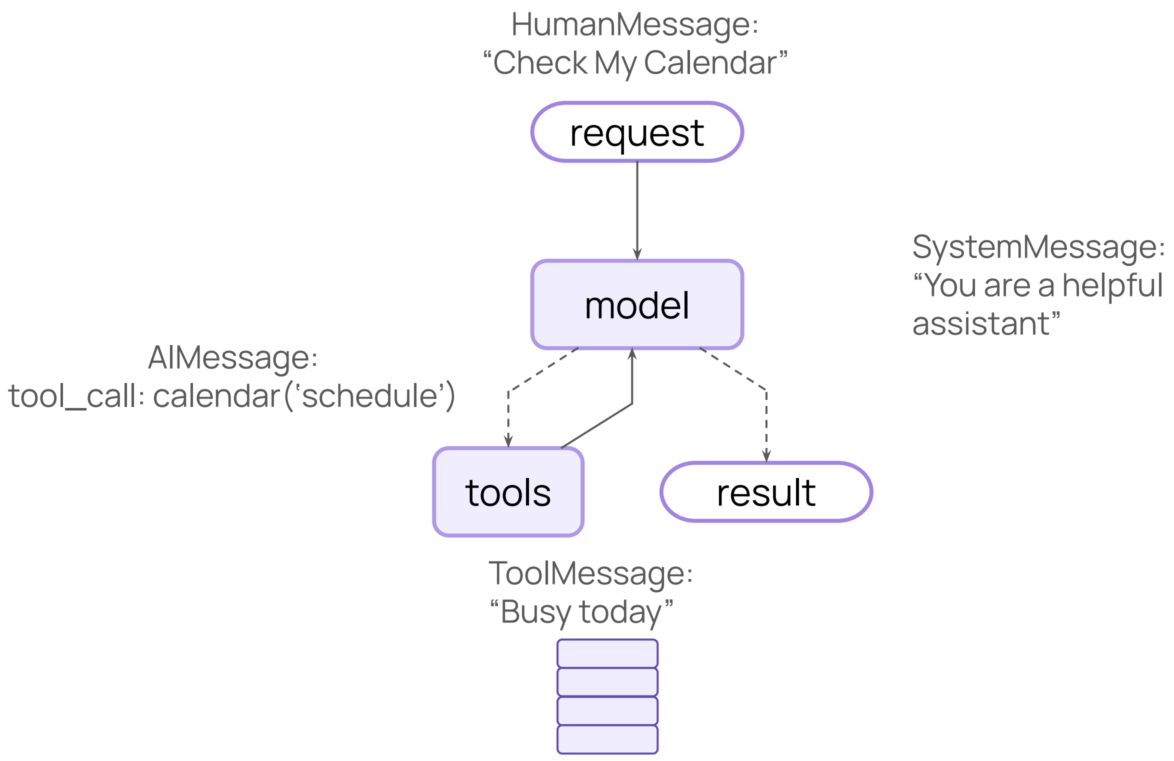
SystemMessage: "You are a helpful assistant"
HumanMessage: "Check My Calendar"
AIMessage: tool_call: calendar('schedule')



Messages

State:
"Messages" : [list of messages]

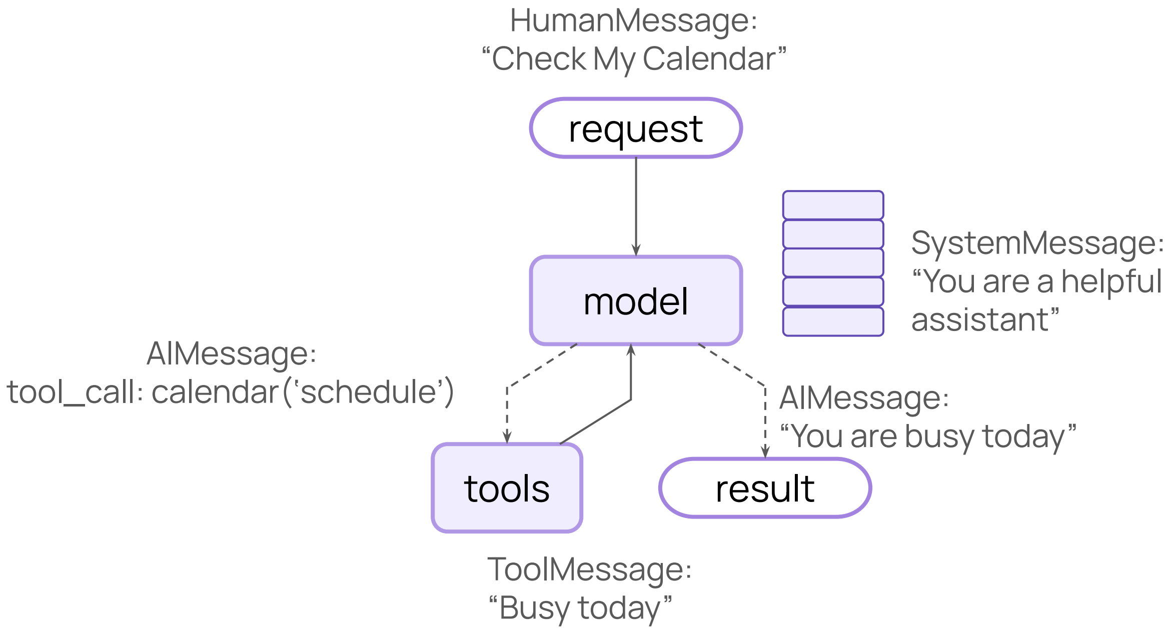
SystemMessage: "You are a helpful assistant"
HumanMessage: "Check My Calendar"
AIMessage: tool_call: calendar('schedule')
ToolMessage: "Busy today"



Messages

State:
"Messages" : [list of messages]

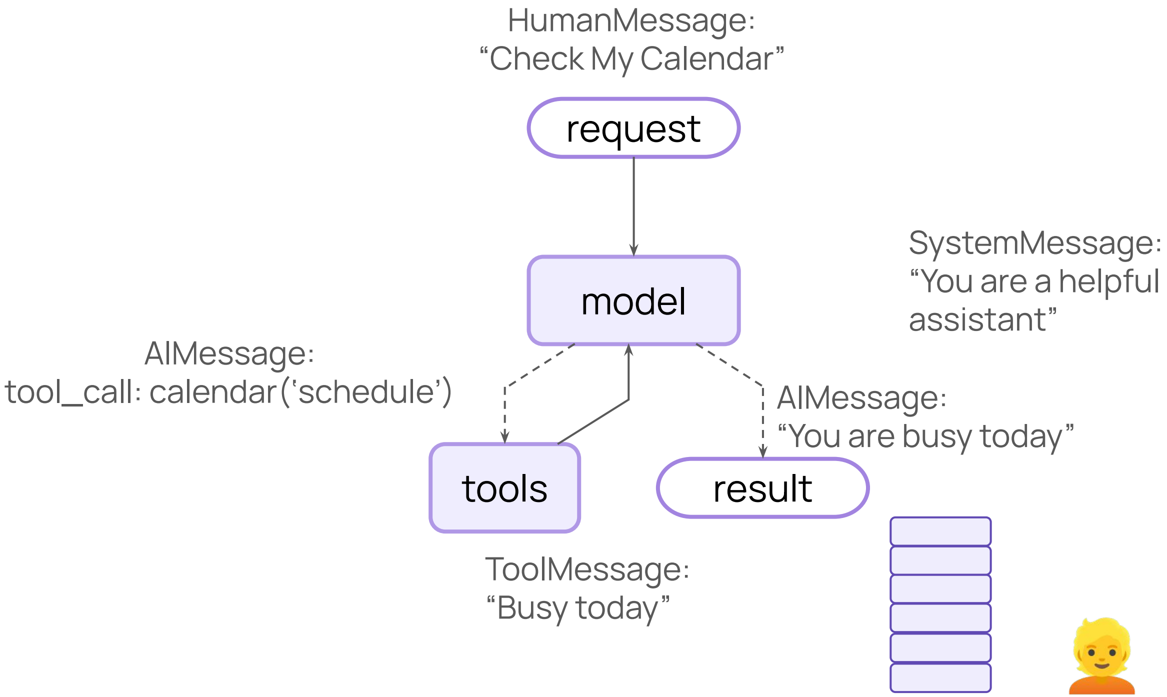
SystemMessage: "You are a helpful assistant"
HumanMessage: "Check My Calendar"
AIMessage: tool_call: calendar('schedule')
ToolMessage: "Busy today"
AIMessage: "You are busy today"



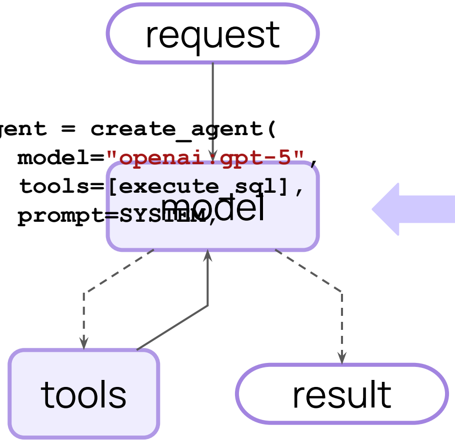
Messages

State:
"Messages" : [list of messages]

SystemMessage: "You are a helpful assistant"
HumanMessage: "Check My Calendar"
AIMessage: tool_call: calendar('schedule')
ToolMessage: "Busy today"
AIMessage: "You are busy today"



System Prompt



```
agent = create_agent(
    model="openai:gpt-5",
    tools=[execute_sql],
    prompt=SYSTEM,
)
```

```
SYSTEM = f"""You are a careful SQLite analyst.

Rules:
- Think step-by-step.
- When you need data, call the tool `execute_sql` with ONE SELECT query.
- Read-only only; no INSERT/UPDATE/DELETE/ALTER/DROP/CREATE/REPLACE/TRUNCATE.
- Limit to 5 rows of output unless the user explicitly asks otherwise.
- If the tool returns 'Error:', revise the SQL and try again.
- Prefer explicit column lists; avoid SELECT *.
"""
```

Streaming

Streaming

Streaming supports interactive applications:

- Messages: token by token as the LLM produces tokens
- Values : return state after every step
- Custom: User defined data when invoked
- Multiple sources

```
for chunk in agent.stream(
    {"messages": [{"role": "user",
                   "content": "Tell me a joke"}]},
    stream_mode="messages",
):
```

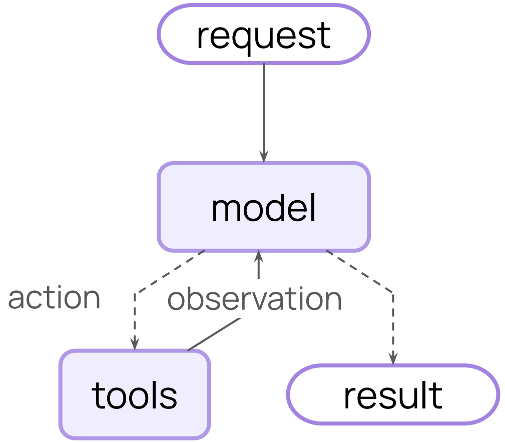
```
for chunk in agent.stream(
    {"messages": [{"role": "user",
                   "content": "Tell me a joke"}]},
    stream_mode="values",
):
```

```
for chunk in agent.stream(
    {"messages": [{"role": "user",
                   "content": "Tell me a joke"}]},
    stream_mode=["custom", "messages"]
):
```

Tools

Tools

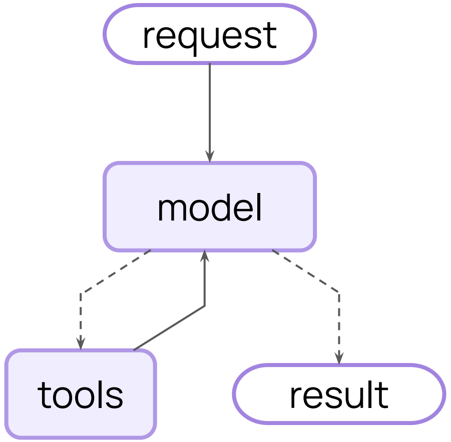
- Tools provide the "Action" part of ReAct
- Their results are the "Observations"



Tools

- You can define tools yourself
- Or use existing libraries of tools

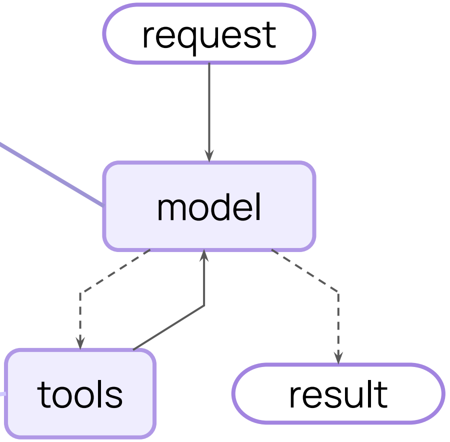
```
@tool
def multiply(a: int, b: int) -> int:
    """Multiply two numbers.

    Args:
        a (int): The first number to multiply.
        b (int): The second number to multiply.

    Returns:
        int: The product of the two input numbers.
    """
    # Perform the multiplication and return the result
    return a * b
```



Tools

```
@tool
def multiply(a: int, b: int) -> int:
   """Multiply two numbers.

   Args:
       a (int): The first number to multiply.
       b (int): The second number to multiply.

   Returns:
       int: The product of the two input numbers.
   """
```

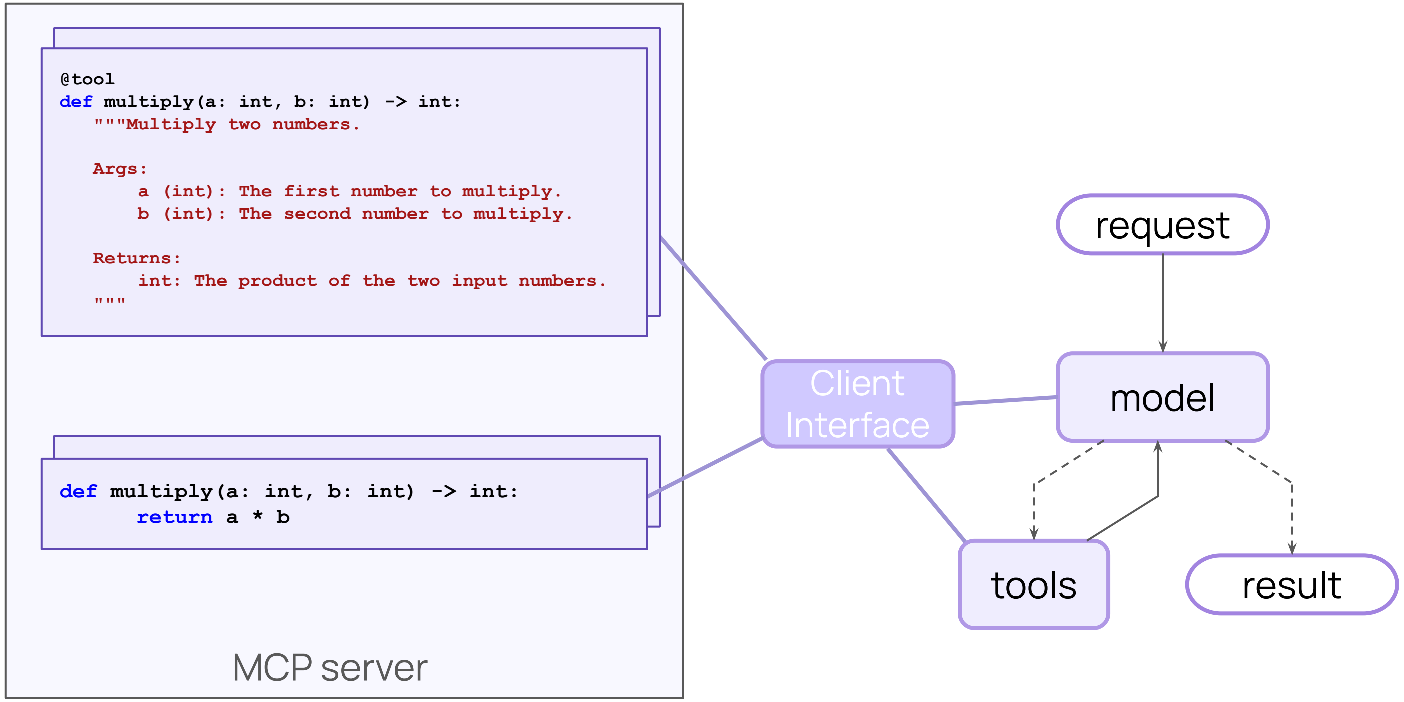
```
def multiply(a: int, b: int) -> int:
      return a * b
```



- The Reasoning Node uses the **description** to decide **when** and **how** to call the tool.
- The function itself is executed by the **tool node**

Tools with MCP

Tools with MCP



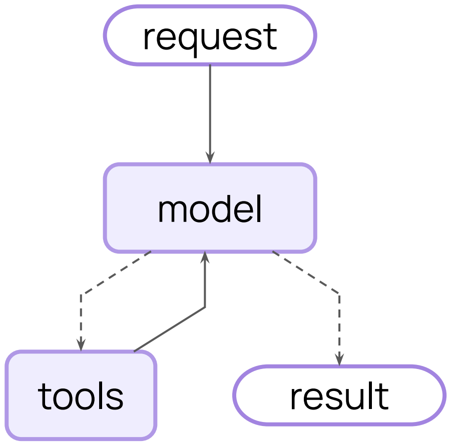
```
@tool
def multiply(a: int, b: int) -> int:
   """Multiply two numbers.

   Args:
       a (int): The first number to multiply.
       b (int): The second number to multiply.

   Returns:
       int: The product of the two input numbers.
   """
```

```
def multiply(a: int, b: int) -> int:
       return a * b
```

Memory

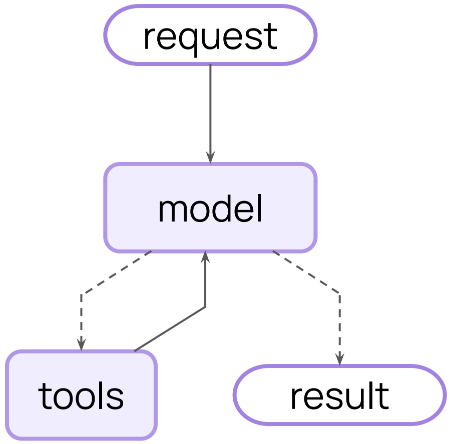
Memory

This is Frank Harris, What was my last invoice?



Hi Frank, it was $10.95

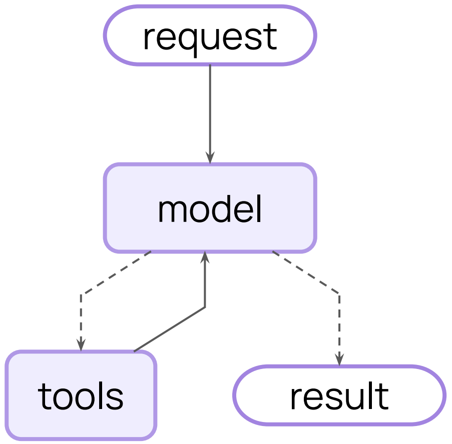
What did I buy?

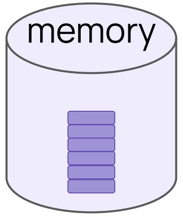


Who is this?

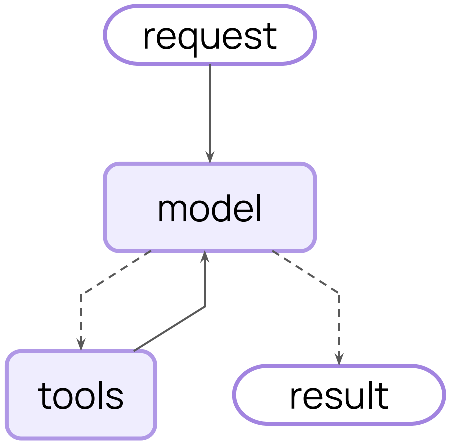
Memory

This is Frank Harris, What was my last invoice?





What did I buy?



Hi Frank, it was $10.95

Taylor Swift

Runtime Context

LangChain's `create_agent` runs on LangGraph's runtime under the hood.

LangGraph exposes a Runtime object with the following information:

1. Context: static information like user id, db connections, or other dependencies for an agent invocation
2. Store: a `BaseStore` instance used for long-term memory
3. Stream writer: an object used for streaming information via the `"custom"` stream mode

You can access runtime information in tools, as well as via custom agent middleware.

Access Runtime Context

```python
db = SQLDatabase.from_uri("sqlite:///Chinook.db")

class RuntimeContext(TypedDict):
    db: SQLDatabase


@tool
def execute_sql(query: str) -> str:
    """Execute a SQLite command and return results."""
    runtime = get_runtime(RuntimeContext)
    db = runtime.context.db

    try:
        return db.run(query)
    except Exception as e:
        return f"Error: {e}"

SYSTEM = f"""You are a careful SQLite analyst.
Rules:
- Think step-by-step.
- When you need data, call the tool `execute_sql` with ONE SELECT query.
- Read-only only; no INSERT/UPDATE/DELETE/ALTER/DROP/CREATE/REPLACE/TRUNCATE.
- Limit to 5 rows of output unless the user explicitly asks otherwise.
- If the tool returns 'Error:', revise the SQL and try again.
- Prefer explicit column lists; avoid SELECT *.
"""

agent = create_agent(
    model="openai:gpt-5",
    tools=[execute_sql],
    prompt=SYSTEM,
    context_schema=RuntimeContext,
    checkpointer=InMemorySaver(),
)
```

```python
for step in agent.stream(
    {"messages": question},
    context=RuntimeContext(db=db),
    stream_mode="values",
):
    step["messages"][-1].pretty_print()
```

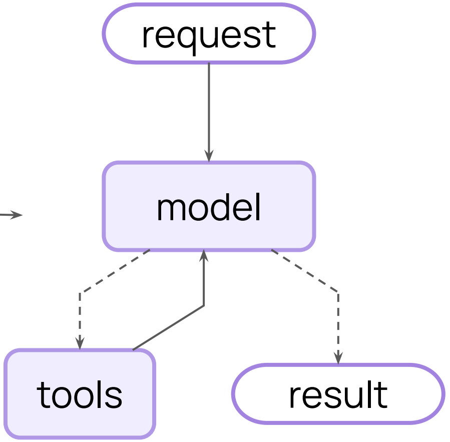
- specify a `context_schema` to define the structure of the `context` stored in the agent Runtime.
- Use the get_runtime function to access the Runtime object inside a tool.
- When invoking the agent, pass the `context` argument with the relevant configuration for the run

Structured Output

Structured Output

Please Provide the contact information for this customer:
Conversation: “We talked with John Doe. He works over at Example. His number is, let's see, five, five, five, one two three, four, five, six seven. Did you get that? And, his email was john at example.com. He wanted to order 50 boxes of Captain Crunch”

```python
class ContactInfo(TypedDict):
    name: str
    email: str
    phone: str
```

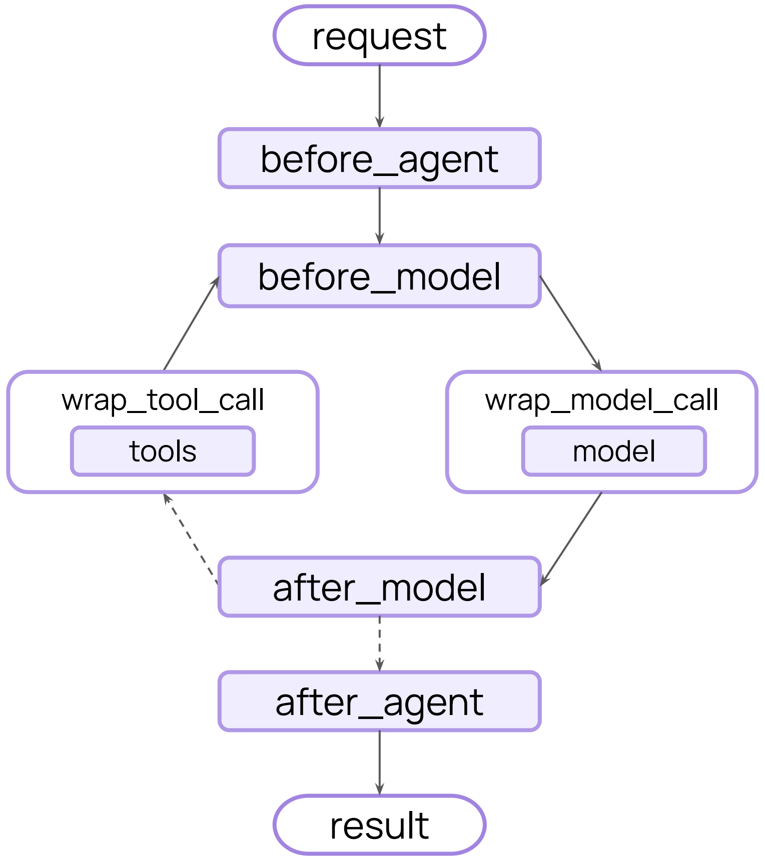


```
ContactInfo:
   name: John Doe
   email: john@example.com
   phone: (555)-123-4567
```

Middleware: Dynamic Prompt

Outline

Middleware

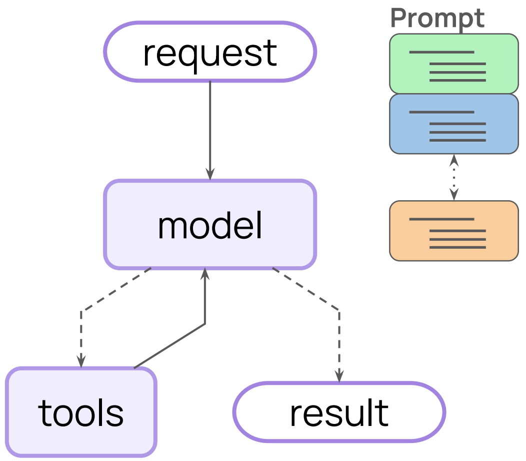


Middleware lets you insert code specific to your agent at key points in the ReAct loop.

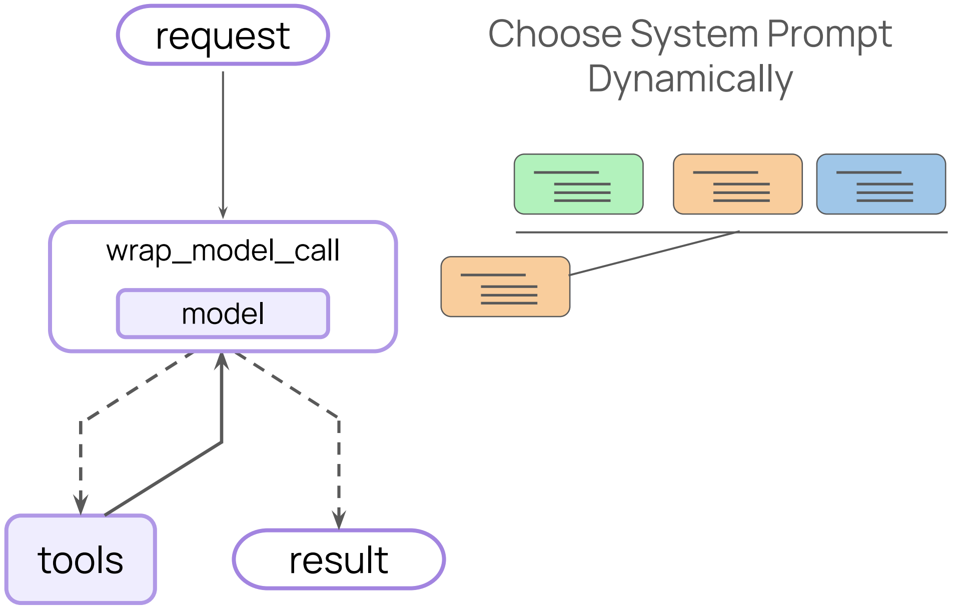
- **before_agent:** setup (files, connections)
- **before_model:** summarization, guardrails
- **wrap_model_call:** dynamic prompt, model
- **wrap_tool_call:** retries, caching
- **after_model:** guardrails
- **after_agent:** teardown

Dynamic Prompting

Prompts must cover all phases and conditions

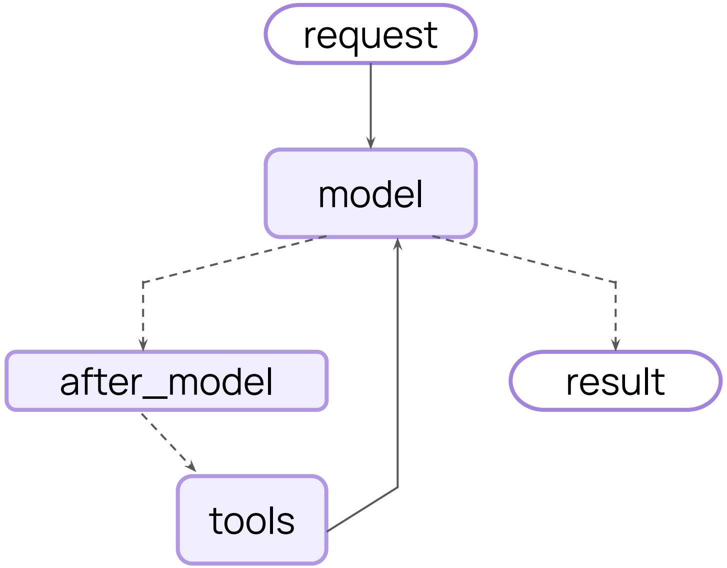


Choose System Prompt Dynamically



MiddleWare: Human in the Loop

Human in the Loop

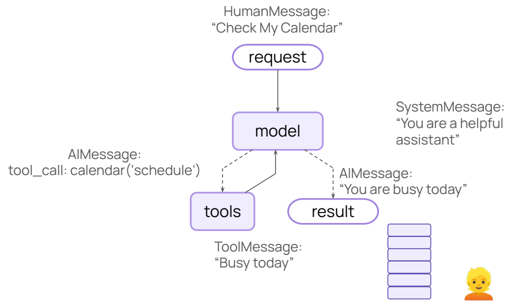


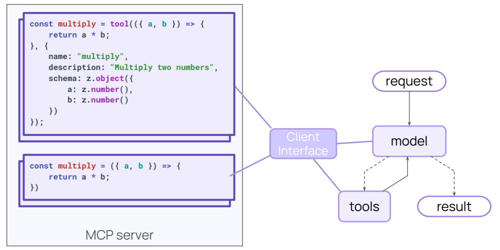
- Add an interrupt in `after_model`
- triggered when `execute_sql` is called.

```
agent = create_agent(
    model="openai:gpt-5",
    tools=[execute_sql],
    system_prompt=SYSTEM_PROMPT,
    checkpointer=InMemorySaver(),
    context_schema=RuntimeContext,
    middleware=[
        HumanInTheLoopMiddleware(
            interrupt_on={
                "execute_sql": True,
            },
        ),
    ],
)
```

Conclusion

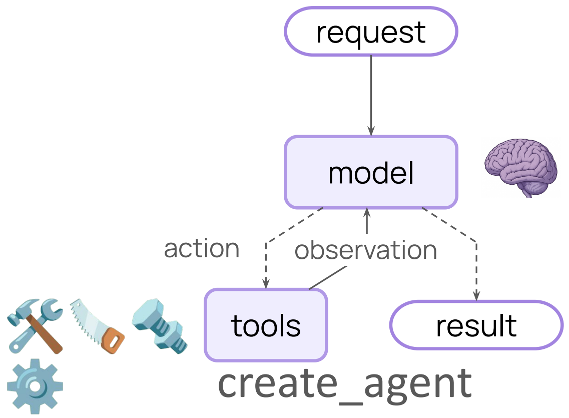
Congratulations!

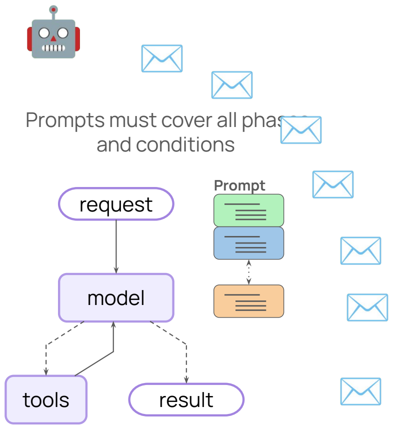


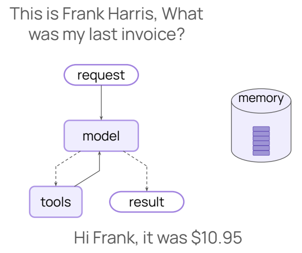


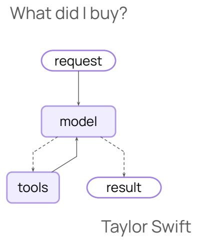
```
const multiply = tool(({ a, b }) => {
    return a * b;
}, {
    name: "multiply",
    description: "Multiply two numbers",
    schema: z.object({
        a: z.number(),
        b: z.number()
    })
});
```

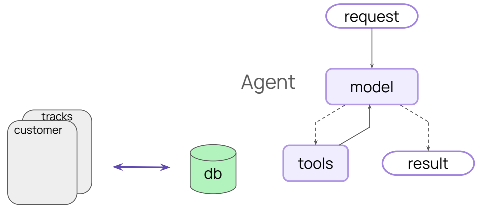
```
const multiply = ({ a, b }) => {
    return a * b;
})
```



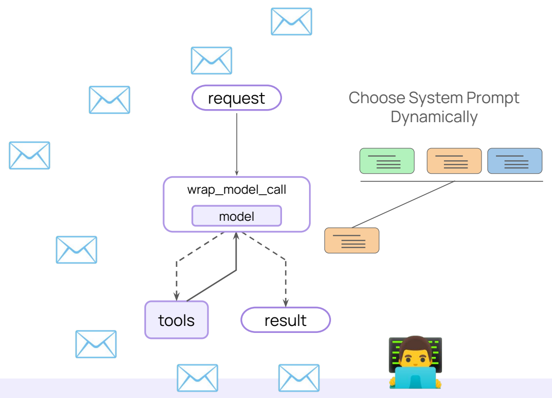








```
def execute_sql(query: str) -> str:
    """Execute a SQLite command and return results."""
```



request
Choose System Prompt Dynamically
wrap_model_call
model
tools
result