Use case 1

7. Building Email Agent: Step-by-Step

By weaving together all three types of long-term memories, our agent becomes truly intelligent and personalized. Imagine an assistant who:

- Notices that emails from a particular client often require follow-up if not answered within 24 hours
- Learns your writing style and tone, adapting formal responses for external communications and casual ones for team members
- Remembers complex project contexts without you having to explain them repeatedly
- Gets better at predicting which emails you'll want to see versus handle automatically

Press enter or click to view image in full size

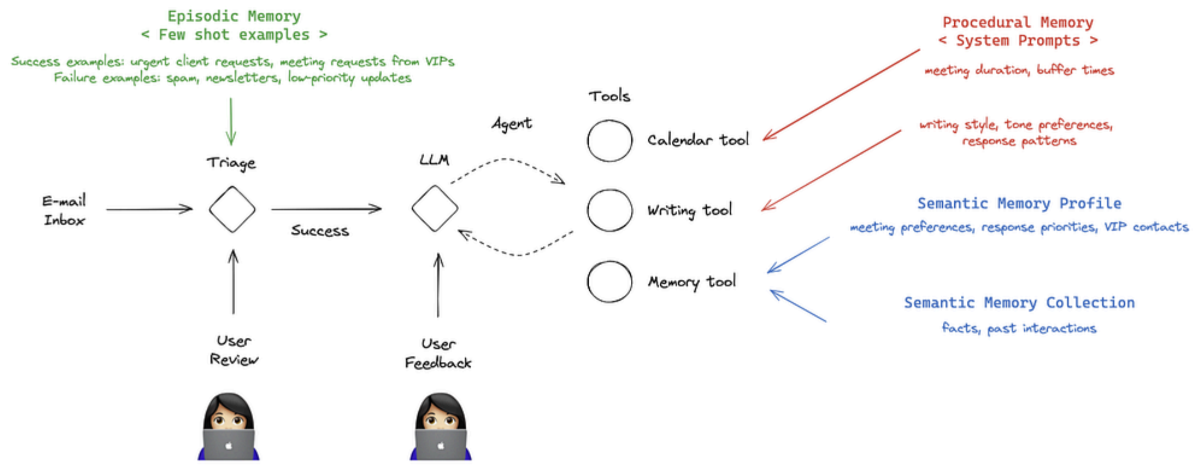

Episodic Memory
< Few shot examples >
Success examples: urgent client requests, meeting requests from VIPs
Failure examples: spam, newsletters, low-priority updates
E-mail Inbox
Triage
Success
LLM
Agent
Tools
Calendar tool
Writing tool
Memory tool
User Review
User Feedback
Procedural Memory
< System Prompts >
meeting duration, buffer times
writing style, tone preferences, response patterns
Semantic Memory Profile
meeting preferences, response priorities, VIP contacts
Semantic Memory Collection
facts, past interactions

Image Credits: DeepLearning.AI

- **Workflow:**

START → Triage (Episodic + Procedural Memory) → Decision → Response Agent (Semantic + Procedural Memory) → END

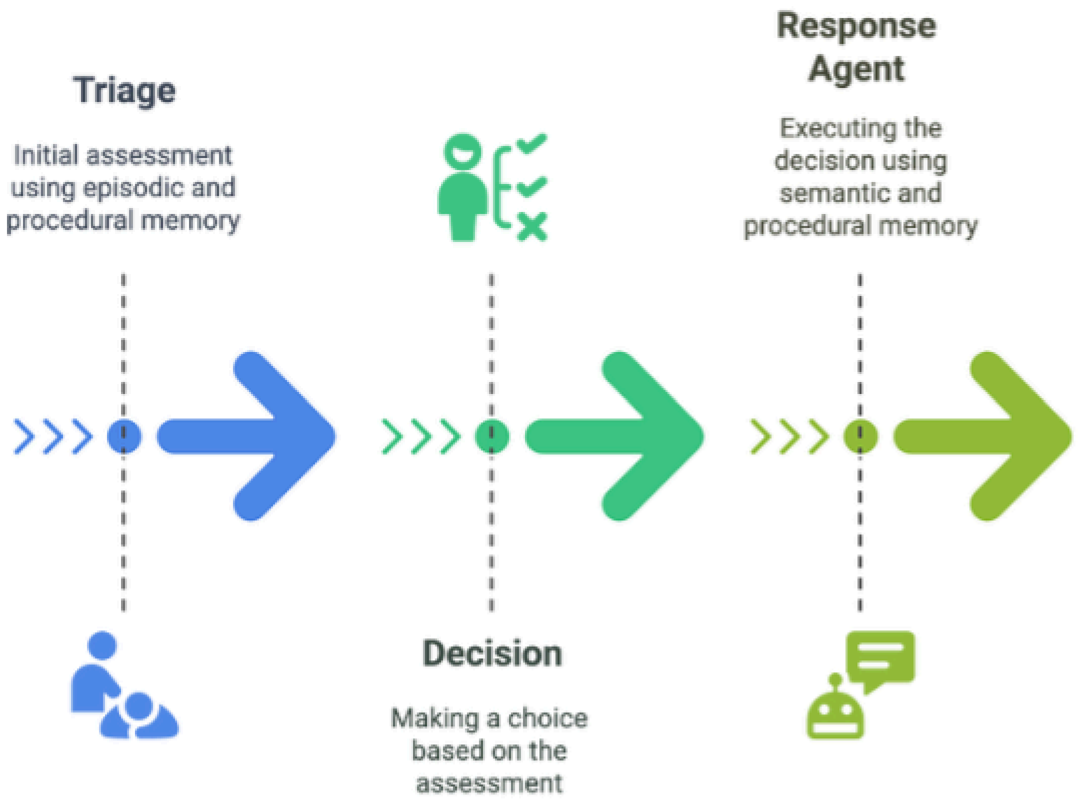

Triage
Initial assessment using episodic and procedural memory
Decision
Making a choice based on the assessment
Response Agent
Executing the decision using semantic and procedural memory

Image by Author

Let's translate this vision into reality by implementing it! :)

Imports and Setup

```
!pip install langmem langchain_community python-dotenv --quiet
# ==============================================================================
# IMPORTS
# ==============================================================================
import os
import warnings
import numpy as np
from dotenv import load_dotenv
from typing import TypedDict, Literal, Annotated, List

# LangGraph and LangChain imports
from langgraph.graph import StateGraph, START, END, add_messages
from langgraph.prebuilt import create_react_agent
from langgraph.store.memory import InMemoryStore
from langchain.chat_models import init_chat_model
from langchain_core.tools import tool
from langchain.prompts import PromptTemplate
from langchain.schema import HumanMessage
from pydantic import BaseModel, Field

# LangMem imports for memory tools
from langmem import create_manage_memory_tool, create_search_memory_tool, create_multi_prompt_optimizer

# IPython for visualization
from IPython.display import Image, display

# Suppress numpy warnings from embedding calculations
warnings.filterwarnings("ignore", category=RuntimeWarning, module="numpy")
warnings.filterwarnings("ignore", category=RuntimeWarning, module="langgraph")
```

Now, let's initialize our environment and tools:

```
%env OPENAI_API_KEY=<your_openai_api_key>
# ==============================================================================
# CONFIGURATION
# ==============================================================================

# Load environment variables
load_dotenv()
```

```
# User Configuration
USER_ID = "test_user"
CONFIG = {"configurable": {"langgraph_user_id": USER_ID}}

# Initialize core components
llm = init_chat_model("openai:gpt-4o-mini")
store = InMemoryStore(index={"embed": "openai:text-embedding-3-small"})
```

The memory store is particularly important here — it's like giving our agent a brain where it can store and retrieve information. We're using an in-memory store for simplicity, but in a production environment, you might want to use a persistent database.

7.1 Defining Our Agent's "Brain": The State

Now we need to design our agent's working memory — the mental scratchpad where it keeps track of what it's currently processing. This is different from the long-term memory we'll implement later — it's more like the things you actively hold in mind while working on a task.

```
class State(TypedDict):
  """
  State management for the email agent workflow.

  This state is passed between nodes in the LangGraph workflow and contains:
  - email_input: The incoming email to be processed
  - messages: Conversation history for the response agent
  - triage_result: Classification result from triage ('ignore', 'notify', 'respond')

  The state evolves as it moves through the workflow, accumulating information
  that helps the agent make better decisions and provide contextual responses.
  """
  email_input: dict  # The incoming email
  messages: Annotated[list, add_messages]  # The conversation history
  triage_result: str # The result of the triage (ignore, notify, respond)
```

This State object holds three crucial pieces of information:

- The current email being processed
- The ongoing conversation (if any)
- The decision about how to handle the email

Think of it like a doctor's clipboard during patient rounds — it contains the immediate information needed to make decisions, while the patient's full medical history remains in the chart room (our memory store).

7.2 The Triage Center: Deciding What to Do (with Episodic Memory)

First, we'll create a structure for our agent to explain its reasoning and classification:

```
class Router(BaseModel):
  """
  Structured output model for email triage classification.

  This model ensures that the LLM provides both reasoning and classification
  in a consistent format, making the triage process transparent and debuggable.

  The reasoning field helps us understand how the agent made its decision,
  which is crucial for improving prompts and training examples.
  """
  reasoning: str = Field(description="Step-by-step reasoning behind the classification.")
  classification: Literal["ignore", "respond", "notify"] = Field(
     description="The classification of an email: 'ignore', 'notify', or 'respond'."
  )

# Initialize LLM router with structured output
llm_router = llm.with_structured_output(Router)
```

To leverage episodic memory, we need a way to format examples from past interactions:

```
def format_few_shot_examples(examples):
  """
  Format episodic memory examples for few-shot learning.

  This function converts stored email examples from the episodic memory
  into a formatted string that can be included in the triage prompt.

  Args:
     examples: List of stored email examples from memory.search()

  Returns:
     str: Formatted examples string for few-shot learning
  """
  formatted_examples = []
  for eg in examples:
     email = eg.value['email']
     label = eg.value['label']
     formatted_examples.append(
         f"From: {email['author']}\nSubject: {email['subject']}\nBody: {email['email_thread'][:300]}...\n\nClassification: {label}"
     )
  return "\n\n".join(formatted_examples)
```

This function transforms our stored examples into a format that helps the model learn from them, like showing a new employee a training manual with annotated examples of how to handle different situations.

Now, let’s create our email triage function that uses episodic memory:

```
def triage_email(state: State, config: dict, store: InMemoryStore) -> dict:
    """
    Basic email triage function using episodic memory.

    This function classifies incoming emails using:
    1. Static prompt template
    2. Few-shot examples from episodic memory

    Args:
        state: Current workflow state containing email_input
        config: Configuration containing user_id for memory namespacing
        store: InMemoryStore for accessing episodic memory

    Returns:
        dict: Updated state with triage_result
    """
    email = state["email_input"]
    user_id = config["configurable"]["langgraph_user_id"]
    namespace = ("email_assistant", user_id, "examples")  # Namespace for episodic memory

    # Retrieve relevant examples from memory
    examples = store.search(namespace, query=str(email))
    formatted_examples = format_few_shot_examples(examples)

    prompt_template = PromptTemplate.from_template("""You are an email triage assistant.  Classify the following email:
    From: {author}
    To: {to}
    Subject: {subject}
    Body: {email_thread}

    Classify as 'ignore', 'notify', or 'respond'.

    Here are some examples of previous classifications:
    {examples}
    """)

    prompt = prompt_template.format(examples=formatted_examples, **email)
    messages = [HumanMessage(content=prompt)]
    result = llm_router.invoke(messages)
    return {"triage_result": result.classification}
```

This function is the core of episodic memory at work. When a new email arrives, it doesn’t analyze it in isolation — it searches for similar emails from the past and sees how those were handled. It’s like a doctor who remembers, “The last three patients with these symptoms responded well to this treatment.”

7.3 Defining Tools with Semantic Memory

Now let’s give our agent some tools to work with. First, basic abilities to write emails and check calendars:

```
@tool
def write_email(to: str, subject: str, content: str) -> str:
    """
    Tool for composing and sending email responses.

    This tool is available to the Response Agent and allows it to:
    - Draft professional email responses
    - Send replies to email inquiries
    - Handle email communication tasks

    Args:
        to: Recipient email address
        subject: Email subject line
        content: Email body content

    Returns:
        str: Confirmation message of email sent
    """
    print(f"Sending email to {to} with subject '{subject}'\nContent:\n{content}\n")
    return f"Email sent to {to} with subject '{subject}'"

@tool
def check_calendar_availability(day: str) -> str:
    """
    Tool for checking calendar availability.

    This tool allows the Response Agent to:
    - Check available meeting times
    - Schedule appointments
    - Provide calendar information

    Args:
        day: Day to check availability for

    Returns:
        str: Available time slots for the specified day
    """
    return f"Available times on {day}: 9:00 AM, 2:00 PM, 4:00 PM"
```

Note that these are simplified implementations for demonstration purposes. In a production environment, you would connect these functions to actual email and calendar APIs (like Gmail API or Microsoft Graph API). For example, the `write_email` function would interact with a real email service to send messages, and `check_calendar_availability` would query your actual calendar data.

Now, let's add the semantic memory tools — our agent's ability to store and retrieve facts about the world:

```
# ==============================================================================
# TOOLS CONFIGURATION
# ==============================================================================

# Create LangMem memory tools
manage_memory_tool = create_manage_memory_tool(namespace=("email_assistant", "{langgraph_user_id}", "collection"))
search_memory_tool = create_search_memory_tool(namespace=("email_assistant", "{langgraph_user_id}", "collection"))

# All available tools for the response agent
tools = [write_email, check_calendar_availability, manage_memory_tool, search_memory_tool]
```

7.4 The Response Agent: Creating Our Core Assistant (with Semantic Memory)

We'll now create the core agent that handles responses using all its memory systems:

```
def create_agent_prompt(state, config, store):
    """
    Create dynamic prompts for the Response Agent using procedural memory.

    This function demonstrates how PROCEDURAL MEMORY enables adaptive behavior:
    1. Retrieves the current response prompt from memory storage
    2. Combines it with conversation history
    3. Returns properly formatted prompt for the LLM

    Args:
        state: Current workflow state with message history
        config: Configuration containing user_id for memory access
        store: InMemoryStore for accessing procedural memory

    Returns:
        list: Formatted messages for the LLM including system prompt + history
    """
    messages = state['messages']
    user_id = config["configurable"]["langgraph_user_id"]

    print("RESPONSE AGENT: Initializing with adaptive system prompt")

    # Get the current response prompt from procedural memory
    system_prompt = store.get(("email_assistant", user_id, "prompts"), "response_prompt")
    print("PROCEDURAL MEMORY: Retrieved current response prompt")

    # Ensure the system prompt is a string
    if not isinstance(system_prompt, str):
        system_prompt = str(system_prompt.value) if hasattr(system_prompt, 'value') else str(system_prompt)

    return [{"role": "system", "content": system_prompt}] + messages
```

This function creates a prompt that pulls instructions from procedural memory and passes the current conversation along with it. It's like a manager who checks the company handbook before responding to a complex situation, ensuring they follow the latest protocols.

Note that we've now set up two key memory systems:

- Episodic memory (in the triage function)
- Semantic memory (in these agent tools)

But we still need to add procedural memory to complete our agent's cognitive abilities. This will come in the following sections, where we'll enable our agent to refine its own behavior over time based on feedback.

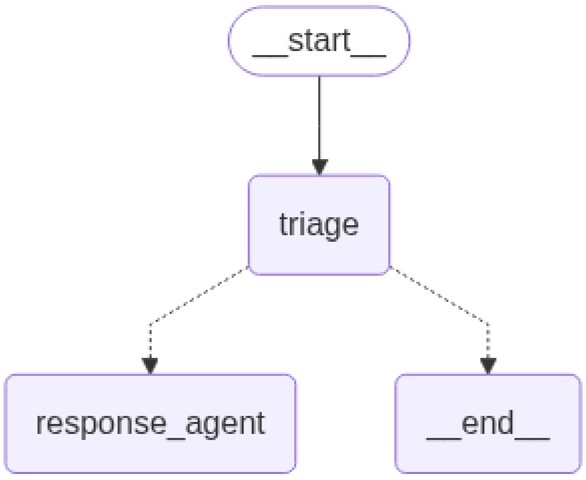
7.5 Building the Graph: Connecting the Pieces

Now, let's bring everything together into a cohesive workflow:

```
def create_basic_email_agent(store):
  """
  Factory function to create a basic email processing agent using only episodic memory.

  This function creates a simpler workflow for comparison purposes that only uses:
  1. EPISODIC MEMORY: Past email examples for few-shot learning

  Args:
     store: InMemoryStore containing episodic memory

  Returns:
     CompiledGraph: Executable workflow with basic memory capabilities
  """
  # Define the workflow
  workflow = StateGraph(State)

  # Use the basic triage function (episodic memory only)
  workflow.add_node("triage", lambda state, config: triage_email(state, config, store))

  # Create a basic response agent with static prompts
  response_agent = create_react_agent(
     tools=tools,
     prompt=create_agent_prompt,
     store=store,
     model=llm
  )

  workflow.add_node("response_agent", response_agent)

  def route_based_on_triage(state):
    if state["triage_result"] == "respond":
     return "response_agent"
    else:
     return END

  # The routing logic remains the same
  workflow.add_edge(START, "triage")
  workflow.add_conditional_edges("triage", route_based_on_triage,
                        {
                          "response_agent": "response_agent",
                          END: END
                        })

  # Compile and return the graph
  return workflow.compile(store=store)
```

This is how our agent looks:

```
from langchain_core.runnables.graph import MermaidDrawMethod
from IPython.display import display, Image

display(
  Image(
     create_basic_email_agent(store).get_graph().draw_mermaid_png(
        draw_method=MermaidDrawMethod.API,
     )
  )
)
```



This workflow defines the logical path for our agent:

1. First, triage the incoming email using episodic memory
2. If it needs a response, activate the response agent with semantic memory
3. Otherwise, end the process (for “ignore” or “notify” emails)

It’s like setting up assembly line stations in a factory — each piece has its own job, but they work together to create the final product.

7.6 Let’s Run It! (and Store Some Memories)

Time to put our agent to the test:

```
# SAMPLE DATA AND INITIALIZATION
# ==============================================================================

# Sample email for testing
email_input = {
  "author": "Alice Smith <alice.smith@company.com>",
  "to": "John Doe <john.doe@company.com>",
  "subject": "Quick question about API documentation",
  "email_thread": """Hi John,

I was reviewing the API documentation and noticed a few endpoints are missing. Could you help?

Thanks,
Alice""",
}

# Initial prompts
initial_triage_prompt = """You are an email triage assistant. Classify the following email:
From: {author}
To: {to}
Subject: {subject}
Body: {email_thread}

Classify as 'ignore', 'notify', or 'respond'.

Here are some examples of previous classifications:
{examples}
"""

initial_response_prompt = """You are a helpful assistant. Use the tools available, including memory tools, to assist the user."""
```

Let’s also add a training example to our episodic memory to help the agent recognize spam in the future:

```
def initialize_memory():
  """Initialize memory with default examples and prompts"""
  # Add few shot examples to episodic memory
  example1 = {
     "email": {
        "author": "Spammy Marketer <spam@example.com>",
        "to": "John Doe <john.doe@company.com>",
        "subject": "BIG SALE!!!",
        "email_thread": "Buy our product now and get 50% off!",
     },
     "label": "ignore",
  }
  store.put(("email_assistant", USER_ID, "examples"), "spam_example", example1)

  # Initialize procedural memory with default prompts
  store.put(("email_assistant", USER_ID, "prompts"), "triage_prompt", initial_triage_prompt)
  store.put(("email_assistant", USER_ID, "prompts"), "response_prompt", initial_response_prompt)
```

This is like training a new assistant: “See this kind of email? You can safely ignore these.” The more examples we provide, the more nuanced the agent’s understanding becomes.

7.7 Adding Procedural Memory (Updating Instructions) — The Final Touch!

Now, for the most sophisticated memory system, procedural memory allows our agent to improve its own instructions based on feedback.

Let’s create a version of our triage function that pulls its instructions from memory:

```
def triage_email_with_procedural_memory(state: State, config: dict, store: InMemoryStore) -> dict:
  """
  Advanced email triage using BOTH episodic and procedural memory.

  This is the heart of the adaptive learning system, combining:
  1. PROCEDURAL MEMORY: Dynamic prompts that improve over time
  2. EPISODIC MEMORY: Few-shot examples from past classifications
```

```
    Args:
        state: Current workflow state containing email_input
        config: Configuration with user_id for memory namespacing
        store: InMemoryStore with both memory types

    Returns:
        dict: Updated state with triage_result classification
    """
    email = state["email_input"]
    user_id = config["configurable"]["langgraph_user_id"]

    print(f"TRIAGE: Analyzing email from {email['author']} with subject: '{email['subject']}'")

    # Retrieve the current triage prompt (procedural memory)
    current_prompt_template = store.get(("email_assistant", user_id, "prompts"), "triage_prompt")
    print("PROCEDURAL MEMORY: Retrieved current triage prompt")

    # Ensure the prompt template is a string
    if not isinstance(current_prompt_template, str):
        current_prompt_template = str(current_prompt_template)  # Convert to string if it's an Item object

    # Retrieve relevant examples from memory (episodic memory)
    namespace = ("email_assistant", user_id, "examples")
    examples = store.search(namespace, query=str(email))
    formatted_examples = format_few_shot_examples(examples)
    print(f"EPISODIC MEMORY: Found {len(examples)} relevant examples from past classifications")

    # Format the prompt
    prompt = PromptTemplate.from_template(current_prompt_template).format(examples=formatted_examples, **email)
    messages = [HumanMessage(content=prompt)]
    result = llm_router.invoke(messages)

    print(f"TRIAGE RESULT: {result.classification}")
    print(f"REASONING: {result.reasoning}")

    return {"triage_result": result.classification}
```

This function integrates procedural memory (the current prompt template) with episodic memory (relevant examples) to make triage decisions.

Now, let’s create a function that can improve our prompts based on feedback:

```
def optimize_prompts(feedback: str, config: dict, store: InMemoryStore):
    """
    PROCEDURAL MEMORY LEARNING: Optimize prompts based on performance feedback.

    This function implements the core learning mechanism that enables gradual improvement:
    1. Analyzes current prompt performance via feedback
    2. Uses AI-powered optimization to improve prompts
    3. Updates procedural memory with better prompts
    4. Future agent instances automatically use improved prompts

    Args:
        feedback: Human feedback describing performance issues
        config: Configuration with user_id for memory access
        store: InMemoryStore for updating procedural memory

    Returns:
        str: Confirmation message about improvements made

    Example Evolution:
    Initial: "How can I assist you today?"
    After feedback: "How can I assist you with API documentation?"
    """
    print("\nOPTIMIZATION: Starting prompt improvement process...")
    user_id = config["configurable"]["langgraph_user_id"]

    # Get current prompts
    print("RETRIEVING: Current prompts from procedural memory")
    triage_prompt = store.get(("email_assistant", user_id, "prompts"), "triage_prompt").value
    response_prompt = store.get(("email_assistant", user_id, "prompts"), "response_prompt").value

    # Create a more relevant test example based on our actual email
    sample_email = {
        "author": "Alice Smith <alice.smith@company.com>",
        "to": "John Doe <john.doe@company.com>",
        "subject": "Quick question about API documentation",
        "email_thread": "Hi John, I was reviewing the API documentation and noticed a few endpoints are missing. Could you help? Thanks, Alice",
    }

    print("ANALYZING: Creating conversation trajectory with feedback")

    # Create the optimizer
    optimizer = create_multi_prompt_optimizer(llm)

    # Create a more relevant conversation trajectory with feedback
    conversation = [
        {"role": "system", "content": response_prompt},
        {"role": "user", "content": f"I received this email: {sample_email}"},
        {"role": "assistant", "content": "How can I assist you today?"}
    ]

    # Format prompts
    prompts = [
        {"name": "triage", "prompt": triage_prompt},
        {"name": "response", "prompt": response_prompt}
    ]

    # More relevant trajectories
    trajectories = [(conversation, {"feedback": feedback})]
    print("OPTIMIZING: Using AI to improve prompts based on feedback...")
    result = optimizer.invoke({"trajectories": trajectories, "prompts": prompts})

    # Extract the improved prompts
    improved_triage_prompt = next(p["prompt"] for p in result if p["name"] == "triage")
    improved_response_prompt = next(p["prompt"] for p in result if p["name"] == "response")

    # Append specific instruction for API documentation issues
    improved_triage_prompt = improved_triage_prompt + "\n\nPay special attention to emails about API documentation or missing endpoints - these are high priority and should ALWAYS be classified as 'respond'."
    improved_response_prompt = improved_response_prompt + "\n\nWhen responding to emails about documentation or API issues, acknowledge the specific issue mentioned and offer specific assistance rather than generic responses."

    print("STORING: Updated prompts in procedural memory")

    # Store the improved prompts
    store.put(("email_assistant", user_id, "prompts"), "triage_prompt", improved_triage_prompt)
    store.put(("email_assistant", user_id, "prompts"), "response_prompt", improved_response_prompt)

    print("IMPROVEMENT COMPLETE: Prompts have been enhanced!")
    print(f"Triage prompt preview: {improved_triage_prompt[:100]}...")
    print(f"Response prompt preview: {improved_response_prompt[:100]}...")

    return "Prompts improved based on feedback!"
```

This function is the essence of procedural memory. It takes feedback like 'You're not prioritizing API documentation emails correctly' and uses it to rewrite the agent's core instructions. The optimizer works like a coach watching game footage, studying what went wrong and updating the playbook accordingly. It analyzes conversation examples alongside feedback, then refines the prompts that guide the agent's behavior. Instead of just memorizing specific corrections, it absorbs the underlying lessons into its overall approach, similar to how a chef improves recipes based on customer feedback rather than simply following different instructions each time.

7.8 Let's Run Our Complete Memory-Enhanced Agent!

Now let's bring everything together into a complete system that can evolve over time:

```
def create_email_agent(store):
    """
    Factory function to create a memory-enabled email processing agent.

    This function builds a LangGraph workflow that combines all three memory types:
    1. EPISODIC MEMORY: Past email examples for few-shot learning
    2. SEMANTIC MEMORY: Contextual information via memory tools
    3. PROCEDURAL MEMORY: Adaptive prompts that improve over time

    Args:
        store: InMemoryStore containing all memory systems

    Returns:
        CompiledGraph: Executable workflow with memory capabilities
    """
    # Define the workflow
    workflow = StateGraph(State)
    workflow.add_node("triage", lambda state, config: triage_email_with_procedural_memory(state, config, store))

    # Create a fresh response agent that will use the latest prompts
    response_agent = create_react_agent(
        tools=tools,
        prompt=create_agent_prompt,
        store=store,
        model=llm
    )

    workflow.add_node("response_agent", response_agent)

    def route_based_on_triage(state):
      if state["triage_result"] == "respond":
        return "response_agent"
      else:
        return END

    # The routing logic remains the same
    workflow.add_edge(START, "triage")
    workflow.add_conditional_edges("triage", route_based_on_triage,
                {
                    "response_agent": "response_agent",
                    END: END
                })

    # Compile and return the graph
    return workflow.compile(store=store)
```

This function creates a fresh agent that uses the most current version of our prompts — ensuring it always reflects our latest learnings and feedback.

Now this is the final version of our agent (including everything):

Press enter or click to view image in full size

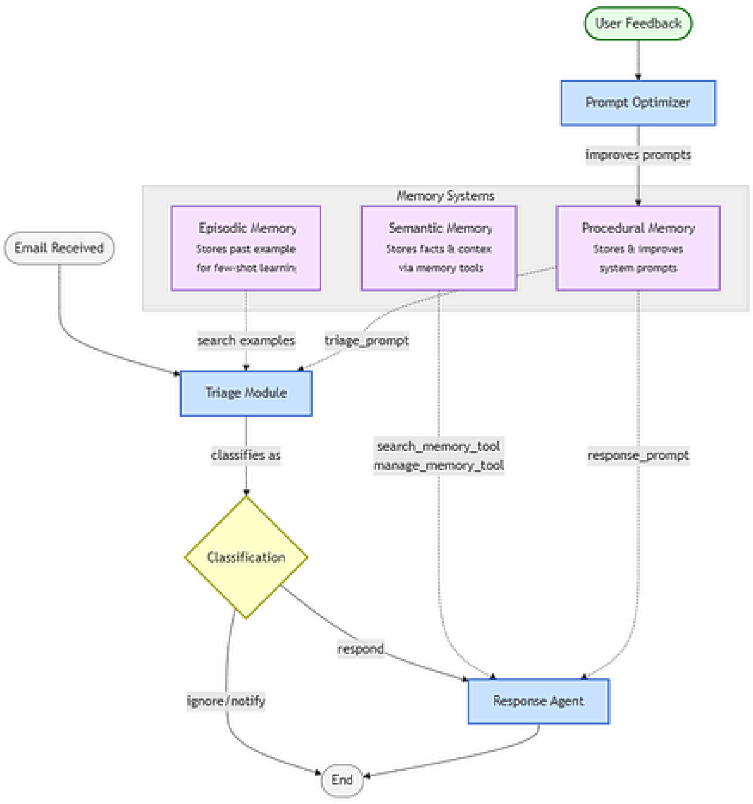


Image Credits: DeepLearning.AI

Let's run it twice — once with the original settings, and once after we've provided feedback to improve it:

```
# =================================================================================
# MAIN EXECUTION: DEMONSTRATION OF GRADUAL IMPROVEMENT
# =================================================================================
"""
This demonstration shows how the agent learns and improves over time:

1. BEFORE OPTIMIZATION: Agent uses initial prompts and minimal examples
2. MEMORY ACCUMULATION: Add new examples to episodic memory
3. FEEDBACK PROCESSING: Human feedback triggers prompt optimization
4. AFTER OPTIMIZATION: Agent uses improved prompts with more examples

The key insight: Each run benefits from ALL previous learning!
"""

def run_demonstration():
    """Main demonstration of the memory-enabled learning system"""

    print("🚀 STARTING MEMORY-ENABLED EMAIL AGENT DEMONSTRATION")
    print("=" * 60)

    # Initialize memory with default examples and prompts
    print("📚 INITIALIZATION: Setting up memory systems...")
    initialize_memory()
    print("✅ MEMORY INITIALIZED: Episodic and procedural memory ready")

    # Setup for demonstration
    inputs = {"email_input": email_input, "messages": []}

    # COMPARISON: Basic vs Advanced Agent
```

```python
    print("\n" + "=" * 60)
    print("📧 PHASE 1: BASIC AGENT (Episodic Memory Only)")
    print("=" * 60)

    basic_agent = create_basic_email_agent(store)
    print("🏗️  BASIC AGENT CREATED: Using only episodic memory")

    print("\n🔄 BASIC WORKFLOW EXECUTION:")
    for output in basic_agent.stream(inputs, config=CONFIG):
        for key, value in output.items():
            if key not in ['triage', 'response_agent']:  # Skip internal node outputs
                print(f"-----\n{key}:")
                print(value)
        print("-----")

    # Advanced Agent - Before Optimization
    print("\n" + "=" * 60)
    print("📧 PHASE 2: ADVANCED AGENT (Before Optimization)")
    print("=" * 60)

    agent = create_email_agent(store)
    print("🏗️  ADVANCED AGENT CREATED: Using episodic + procedural memory")

    print("\n🔄 ADVANCED WORKFLOW EXECUTION (Before Learning):")
    for output in agent.stream(inputs, config=CONFIG):
        for key, value in output.items():
            if key not in ['triage', 'response_agent']:  # Skip internal node outputs
                print(f"-----\n{key}:")
                print(value)
        print("-----")

    print("\n" + "=" * 60)
    print("🧠 PHASE 3: MEMORY ENHANCEMENT & FEEDBACK")
    print("=" * 60)

    # Add a specific example to episodic memory
    api_doc_example = {
        "email": {
            "author": "Developer <dev@company.com>",
            "to": "John Doe <john.doe@company.com>",
            "subject": "API Documentation Issue",
            "email_thread": "Found missing endpoints in the API docs. Need urgent update.",
        },
        "label": "respond",
    }
    store.put(("email_assistant", USER_ID, "examples"), "api_doc_example", api_doc_example)
    print("📚 EPISODIC MEMORY: Added API documentation example")

    # Provide feedback for optimization
    feedback = """The agent didn't properly recognize that emails about API documentation issues
are high priority and require immediate attention. When an email mentions
'API documentation', it should always be classified as 'respond' with a helpful tone.
Also, instead of just responding with 'How can I assist you today?', the agent should
acknowledge the specific documentation issue mentioned and offer assistance."""

    print("💬 FEEDBACK RECEIVED: Performance improvement suggestions")

    # Optimize prompts based on feedback
    optimize_prompts(feedback, CONFIG, store)

    # Process the SAME email after optimization with a FRESH agent
    print("\n" + "=" * 60)
    print("📧 PHASE 4: ADVANCED AGENT (After Optimization)")
    print("=" * 60)

    new_agent = create_email_agent(store)
    print("🏗️  OPTIMIZED AGENT CREATED: Fresh agent with optimized memory state")

    print("\n🔄 ADVANCED WORKFLOW EXECUTION (After Learning):")
    for output in new_agent.stream(inputs, config=CONFIG):
        for key, value in output.items():
            if key not in ['triage', 'response_agent']:  # Skip internal node outputs
                print(f"-----\n{key}:")
                print(value)
        print("-----")

    print("\n" + "=" * 60)
    print("🎉 DEMONSTRATION COMPLETE: Comparison shows memory-enabled learning!")
    print("📊 RESULTS SUMMARY:")
    print("  1️⃣ Basic Agent: Static prompts, episodic memory only")
    print("  2️⃣ Advanced Agent (Before): Initial procedural memory + episodic memory")
    print("  3️⃣ Advanced Agent (After): Optimized procedural memory + enhanced episodic memory")
    print("=" * 60)

# Run the demonstration
if __name__ == "__main__":
    run_demonstration()
```

Let's look at the output:

```
🚀 STARTING MEMORY-ENABLED EMAIL AGENT DEMONSTRATION
============================================================
📚 INITIALIZATION: Setting up memory systems...
✅ MEMORY INITIALIZED: Episodic and procedural memory ready

============================================================
📧 PHASE 1: BASIC AGENT (Episodic Memory Only)
============================================================
🏗️ BASIC AGENT CREATED: Using only episodic memory

🔄 BASIC WORKFLOW EXECUTION:
-----
RESPONSE AGENT: Initializing with adaptive system prompt
PROCEDURAL MEMORY: Retrieved current response prompt
-----

============================================================
📧 PHASE 2: ADVANCED AGENT (Before Optimization)
============================================================
🏗️ ADVANCED AGENT CREATED: Using episodic + procedural memory

🔄 ADVANCED WORKFLOW EXECUTION (Before Learning):
TRIAGE: Analyzing email from Alice Smith <alice.smith@company.com> with subject: 'Quick question about API documentation'
PROCEDURAL MEMORY: Retrieved current triage prompt
EPISODIC MEMORY: Found 2 relevant examples from past classifications
TRIAGE RESULT: respond
REASONING: The email from Alice Smith is a request for help regarding missing endpoints in the API documentation. This indicates a requirement for a response to
assist her with the issue, as it relates to important documentation that is likely affecting her work or the work of her team. Unlike spam or non-urgent inquiries, this is a
valid work-related request that warrants a response.
-----
RESPONSE AGENT: Initializing with adaptive system prompt
PROCEDURAL MEMORY: Retrieved current response prompt
-----

============================================================
🧠 PHASE 3: MEMORY ENHANCEMENT & FEEDBACK
============================================================
📚 EPISODIC MEMORY: Added API documentation example
💬 FEEDBACK RECEIVED: Performance improvement suggestions

OPTIMIZATION: Starting prompt improvement process...
RETRIEVING: Current prompts from procedural memory
ANALYZING: Creating conversation trajectory with feedback
OPTIMIZING: Using AI to improve prompts based on feedback...
STORING: Updated prompts in procedural memory
IMPROVEMENT COMPLETE: Prompts have been enhanced!
Triage prompt preview: You are an email triage assistant. Classify the following email based on its content and determine i...
Response prompt preview: You are a helpful assistant. Use the tools available, including memory tools, to assist the user. Wh...

============================================================
📧 PHASE 4: ADVANCED AGENT (After Optimization)
============================================================
🏗️ OPTIMIZED AGENT CREATED: Fresh agent with optimized memory state

🔄 ADVANCED WORKFLOW EXECUTION (After Learning):
TRIAGE: Analyzing email from Alice Smith <alice.smith@company.com> with subject: 'Quick question about API documentation'
PROCEDURAL MEMORY: Retrieved current triage prompt
EPISODIC MEMORY: Found 2 relevant examples from past classifications
TRIAGE RESULT: respond
REASONING: The email from Alice Smith discusses missing endpoints in the API documentation, which aligns perfectly with the high-priority key phrases outlined in the
triage prompt. The prompt explicitly states that such emails should ALWAYS be classified as 'respond'. Given the importance of API documentation for the workflow, it is
necessary to provide assistance.
-----
RESPONSE AGENT: Initializing with adaptive system prompt
PROCEDURAL MEMORY: Retrieved current response prompt
-----

============================================================
🎉 DEMONSTRATION COMPLETE: Comparison shows memory-enabled learning!
📊 RESULTS SUMMARY:
1️⃣ Basic Agent: Static prompts, episodic memory only
2️⃣ Advanced Agent (Before): Initial procedural memory + episodic memory
3️⃣ Advanced Agent (After): Optimized procedural memory + enhanced episodic memory
============================================================
```

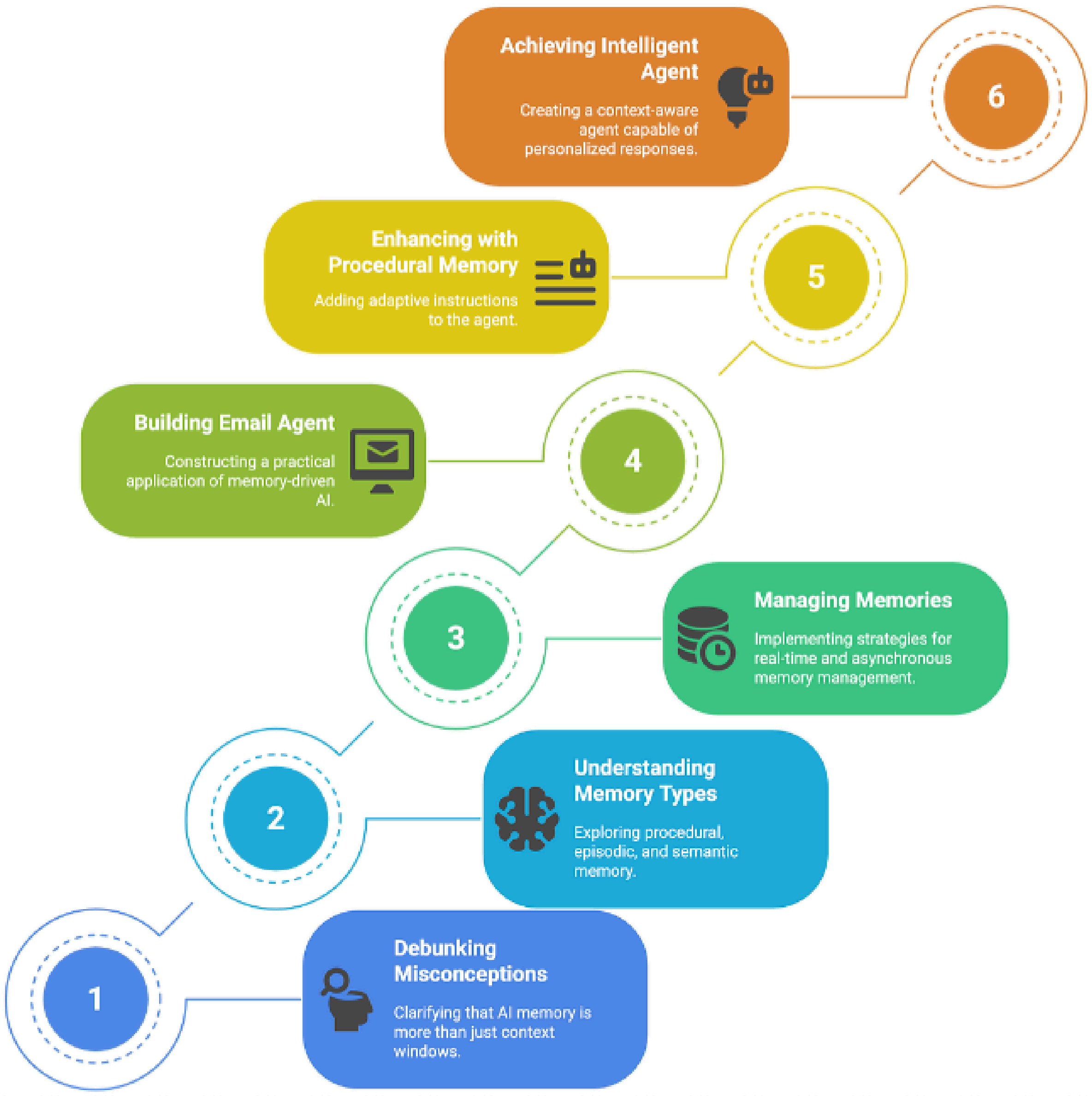
Look at the difference in responses! After our feedback, the agent should:

1. More consistently recognize API documentation issues as high-priority
2. Provide more specific, helpful responses that acknowledge the actual issue
3. Offer concrete assistance rather than generic platitudes

8. Conclusion

Memory is the cornerstone of intelligent AI agents, enabling them to retain, recall, and adapt to information. We've explored how short-term and long-term memory — procedural, episodic, and semantic — integrate into the agent stack, moving beyond mere context windows or RAG. By building a memory-enhanced email agent, we demonstrated practical applications of these concepts, from state management to semantic search and adaptive instructions. Mastering memory empowers AI to deliver personalized, context-aware solutions, paving the way for smarter, more capable systems in production environments.

Press enter or click to view image in full size


6
Achieving Intelligent Agent
Creating a context-aware agent capable of personalized responses.
5
Enhancing with Procedural Memory
Adding adaptive instructions to the agent.
4
Building Email Agent
Constructing a practical application of memory-driven AI.
3
Managing Memories
Implementing strategies for real-time and asynchronous memory management.
2
Understanding Memory Types
Exploring procedural, episodic, and semantic memory.
1
Debunking Misconceptions
Clarifying that AI memory is more than just context windows.