Transforming Agent-based Chatbots with Declarative Programming

Sirui Zeng, Xifeng Yan
Computer Science
University of California at Santa Barbara
https://github.com/Mica-labs/MICA

Write a customer service chatbot

swarm

```python
seat_booking_agent = Agent[AirlineAgentContext](
    name="Seat Booking Agent",
    handoff_description="A helpful agent that can update a seat on a flight.",
    instructions=f"""{RECOMMENDED_PROMPT_PREFIX}
    You are a seat booking agent. If you are speaking to a customer, you probably were transferred to from the triage agent
    Use the following routine to support the customer.
    # Routine
    1. Ask for their confirmation number.
    2. Ask the customer what their desired seat number is.
    3. Use the update seat tool to update the seat on the flight.
    If the customer asks a question that is not related to the routine, transfer back to the triage agent. """,
    tools=[update_seat],
)

triage_agent = Agent[AirlineAgentContext](
    name="Triage Agent",
    handoff_description="A triage agent that can delegate a customer's request to the appropriate agent.",
    instructions=(
        f"{RECOMMENDED_PROMPT_PREFIX} "
        "You are a helpful triaging agent. You can use your tools to delegate questions to other appropriate agents."
    ),
    handoffs=[
        faq_agent,
        handoff(agent=seat_booking_agent, on_handoff=on_seat_booking_handoff),
    ],
)

faq_agent.handoffs.append(triage_agent)
```

Write a customer service chatbot

swarm

```python
seat_booking_agent = Agent
    name="Seat Booki
    handoff_descript
    instructions=f""
    You are a seat b
    Use the followin
    # Routine
    1. Ask for their
    2. Ask the custo
    3. Use the updat
    If the customer
    tools=[update_se
)

triage_agent = Agent
    name="Triage Age
    handoff_descript
    instructions=(
        f"{RECOMMEND
        "You are a h
    ),
    handoffs=[
        faq_agent,
        handoff(agen
    ],
)

faq_agent.handoffs.append(triage_agent)
```

rasa

```yaml
flows:
  block_card:
    description: "Block or freeze a user's debit or credit card to prevent unauthorized
    ↪  use, stop transactions, or report it lost, stolen, damaged, or misplaced
    for added security"
    name: block a card
    steps:
      - action: utter_block_card_understand
      - call: select_card
      - collect: reason_for_blocking
        description: |
          The reason for freezing or blocking the card, described as lost, damaged,
          ↪  stolen, suspected of fraud, malfunctioning, or expired. The user may say
          ↪  they are traveling or moving, or they may say they want to temporarily
          ↪  freeze their card. For all other responses, set reason_for_blocking slot to
          ↪  'unknown'.
        next:
          - if: "slots.reason_for_blocking == 'damaged' or slots.reason_for_blocking ==
          ↪  'expired'"
            then: "acknowledge_reason_damaged_expired"
```

Write a customer service chatbot

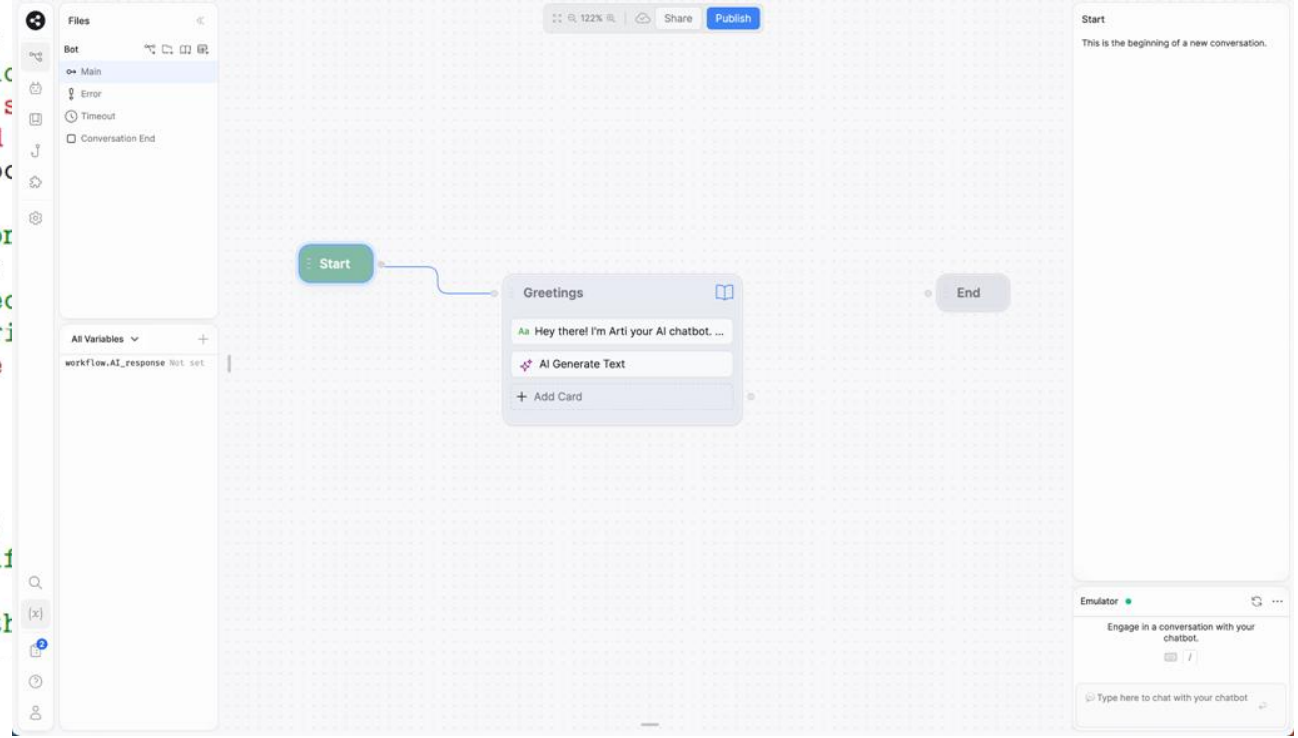
swarm

```
seat_booking_agent =
    name="Seat Booki
    handoff_descript
    instructions=f""
    You are a seat b
    Use the followin
    # Routine
    1. Ask for their
    2. Ask the custo
    3. Use the updat
    If the customer
    tools=[update_se
)

triage_agent = Agent
    name="Triage Age
    handoff_descript
    instructions=(
        f"{RECOMMEND
        "You are a h
    ),
    handoffs=[
        faq_agent,
        handoff(agen
    ],
)

faq_agent.handoffs.append(triage_agent)
```

rasa

```
flows:
  block_card:
    descriptic
    ↪  use, s
    for added
    name: bloc
    steps:
      - actior
      - call:
      - collec
        descri
          The
          ↪
          ↪
          ↪
          ↪
        next:
          - if
          ↪
            th
```

botpress



Write a customer service chatbot

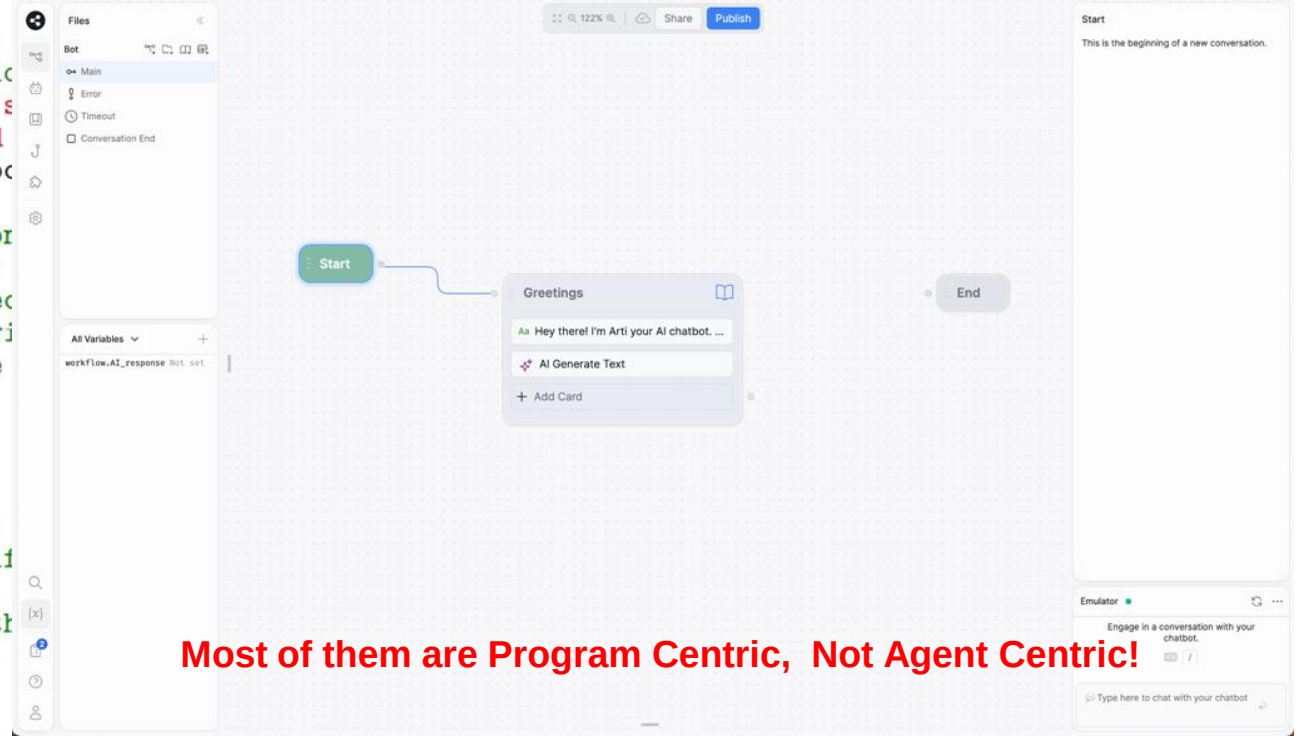
swarm

```
101  seat_booking_agent =
102      name="Seat Booki
103      handoff_descript
104      instructions=f""
105      You are a seat b
106      Use the followin
107      # Routine
108      1. Ask for their
109      2. Ask the custo
110      3. Use the updat
111      If the customer
112      tools=[update_se
113  )
114
115  triage_agent = Agent
116      name="Triage Age
117      handoff_descript
118      instructions=(
119          f"{RECOMMEND
120          "You are a h
121      ),
122      handoffs=[
123          faq_agent,
124          handoff(agen
125      ],
126  )
127
128  faq_agent.handoffs.append(triage_agent)
```

rasa

```
flows:
  block_card:
    descriptic
     ↪  use, s
    for added
    name: bloc
    steps:
      - actior
      - call:
      - collec
        descri
          The
          ↪
          ↪
          ↪
          ↪
        next:
          - if
          ↪
            th
```

botpress



Most of them are Program Centric, Not Agent Centric!

Make It Agent Centric!

Agent Declarative Language (ADL) puts all the domain specific knowledge and business logic in one agent centric file: Describe what agents can do and their relationship

```
store_policy_kb:
  type: kb agent
  description: I can answer questions related to the
store's policy.
  sources:
    - https://www.bookstores.com/policies/shipping
    - https://www.bookstores.com/about
  faq:
    - q: Thanks, bye
      a: Looking forward to serving you next time.
```

```
book_recommendation:
  type: llm agent
  description: I can recommend books to customers.
  args:
    - genre
    - book_name
    - book_info
    - book_wanted
  prompt: |
    1. Ask the user if they have a preferred book
genre.
    2. If the user has a favorite book, call the
"query_book_genre" function based on their favorite
book to obtain the "genre".
    3. Using the genre, call the "find_bestsellers"
function to recommend relevant books to the user.
Then ask user if they need to order this one.
    4. If the user agree, append this book to
argument "book_wanted", which should be an array,
and then complete this agent.
  uses:
    - query_book_genre
    - find_bestsellers
```

```
order:
  type: flow agent
  description: I can place an order.
  args:
    - books
    - order_status
  fallback: Sorry, I didn't understand that. Could
you rephrase it?
  steps:
    - bot: "I'll place the order for you."
    - label: confirm_books
    - bot: "You have selected these books so far:
${books}. Would you like to add anything else to
your order?"
    - user
    - if: the user claims "Yeah", "Do you have other
types of books?"
      then:
        - call: book_recommendation
        - next: confirm_books
    - else if: the user claims "No", "I don't have
anything else I want to buy."
      then:
        - next: start_ordering_operation
      else:
        - next: confirm_books

  start_ordering_operation:
    - call: place_order
      args:
        ordered_book: books
        date: triage.date
    - if: place_order.status == True
      then:
        - return: success, Order placed
successfully.
      else:
        - return: error, Order failed.
```

```
guardrail:
  type: llm agent
  description: I can check if the user's input is
relevant to the bookstore.
  args: ["is_relevant"]
  prompt: |
    Determine whether the user's message is highly
unrelated to a typical bookstore customer. It is
acceptable for the customer to send messages such as
"Hi," "OK," or other conversational responses.
However, if the message is non-conversational, it
must still be at least somewhat related to books.
Return is_relevant = True if it is, otherwise return
False.
```

```
triage:
  type: ensemble agent
  description: Select an agent to respond to users.
  args: ["book"]
  contains:
    - guardrail
    - store_policy_kb
    - book_recommendation:
        args:
          book_wanted: ref book
    - order:
        args:
          books: ref book
  steps:
    - bot: "Hi, I'm your bookstore assistant. How
can I help you?"
  policy: Call the guardrail first each time, and
decide whether to proceed with the conversation
based on its output.
  exit: default
```

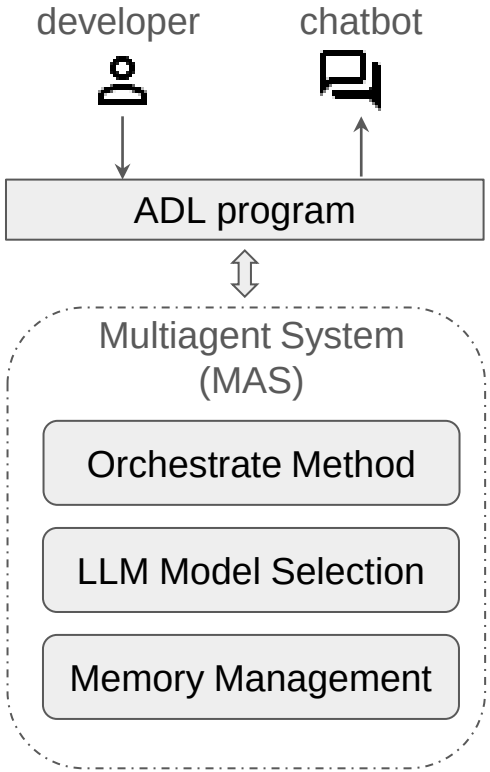
```
main:
  type: flow agent
  steps:
    - call: triage
```

Do we really need it?

```
guardrail:
  type: llm agent
  description: I can check if the user's input is
relevant to the bookstore.
  args: ["is_relevant"]
  prompt: |
    Determine whether the user's message is highly
unrelated to a typical bookstore customer. It is
acceptable for the customer to send messages such as
"Hi," "OK," or other conversational responses.
However, if the message is non-conversational, it
must still be at least somewhat related to books.
Return is_relevant = True if it is, otherwise return
False.
```

```
triage:
  type: ensemble agent
  description: Select an agent to respond to users.
  args: ["book"]
  contains:
    - guardrail
    - store_policy_kb
    - book_recommendation:
        args:
          book_wanted: ref book
    - order:
        args:
          books: ref book
  steps:
    - bot: "Hi, I'm your bookstore assistant. How
can I help you?"
  policy: Call the guardrail first each time, and
decide whether to proceed with the conversation
based on its output.
  exit: default
```

Make it declarative (no vendor lock-in)



Separate logic and optimization

AutoGen

```
while True:
    user = input("User: ").strip()
    msg = TextMessage(content=user,
                      source="user")
    rsp = await agent.on_messages([msg])
    intent = rsp.chat_message.content
    if intent == "...":
        print("Bot: Continue...")
    else:
        print("Bot: Sorry...")
```

ADL

```
- user
- if: the user claims "..."
  then:
    - bot: "Continue..."
  else:
    - bot: "Sorry..."
```

Natural language programming

ADL simplifies chatbot maintenance and updates

For example, revise the ordering process so that when users ask about a discount, they are informed that a special discount is available.

```
order:
  type: flow agent
  ...
  steps:
    - ...
    - bot: "You have selected these books
so far: ${books}. Would you like to add
anything else to your order?"
    - if: the user claims "No"
      then:
        - next: start_ordering_operation
    - else if: the user asks for any
discount
      then:
        - bot: "Here's a special discount
for you..."
```

ADL

```
print(f"You have selected these books so
far: {books}. Would you like to add anything
else to your order?")
while True:
    user_msg = input().strip()
    intent = call_agent(intent_classifier,
                        user_msg)
    if intent == "deny":
        call_agent(start_ordering_operation)
    elif intent == "ask_for_discount":
        while True: # discount subflow
            print("Here's a special discount
for you")
            ... # offer a discount
            if ...: # termination condition
                break
```

```
intent_classifier = Agent(
instruction="...When the user's
intent is related to asking about
a discount, output
'ask_for_discount'...",
...)    # update intent
classification agent's prompt
```

Python

ADL facilitates debugging

Infinite loop discovery

ADL

```
add_payee:
  prompt: |
    ... Ask the user if they need to make a
transfer; if so, enter the transfer_money agent ...

transfer_money:
  prompt: |
    ...enter the add_payee agent ...
```

Swarm

```
TRANSFER_MONEY_POLICY = """... enter the add_payee
agent ..."""
ADD_PAYEE = """... Ask the user if they need to
make a transfer; if so, enter the transfer_money
agent..."""
add_payee = Agent(
  instructions=ADD_PAYEE_PROMPT,
  functions=transfer_to_triage,
  ...)
transfer_money = Agent(
  instructions=TRANSFER_MONEY_POLICY,
  functions=transfer_to_triage,
  ...)
```

Is there any infinite loop in this chatbot?

Yes. transfer_money → add_payee → transfer_money → ... ✓

Loops can happen with any agent that calls transfer_to_triage and is itself callable from triage_agent. ✗

Infinite loops are just one common type of development error; other types of errors also warrant further investigation.

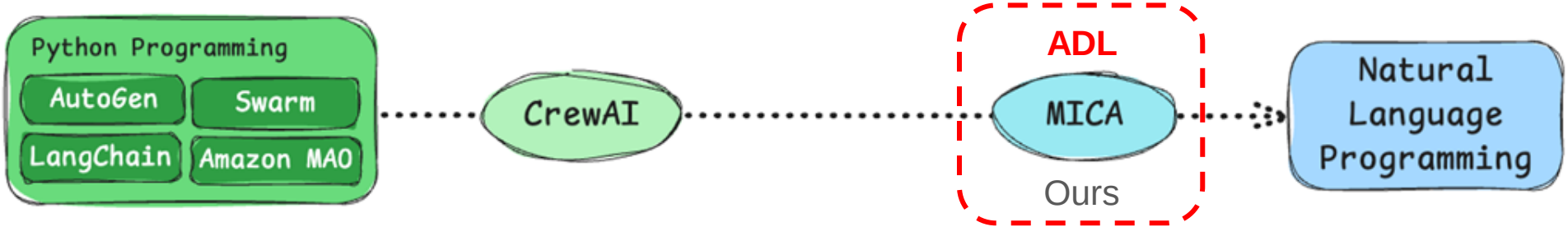
Where does ADL sit?

```python
import json

from swarm import Agent


def get_weather(location, time="now"):
    """Get the current weather in a given location. Location MUST be a city."""
    return json.dumps({"location": location, "temperature": "65", "time": time})


def send_email(recipient, subject, body):
    print("Sending email...")
    print(f"To: {recipient}")
    print(f"Subject: {subject}")
    print(f"Body: {body}")
    return "Sent!"


weather_agent = Agent(
    name="Weather Agent",
    instructions="You are a helpful agent.",
    functions=[get_weather, send_email],
)
```

Python Hybrid

```yaml
tools:
  - tool_impl.py

weather_agent:
  type: llm agent
  description: You are an agent for weather query
  prompt: >
    1. Get the current weather in a given location. Location MUST be a city.
    2. send an email to user. collect recipient, subject and body.
    3. call "send_email" function
  args:
    - location
    - recipient
    - subject
    - body
  uses:
    - send_email

main:
  type: flow agent
  steps:
    - call: weather_agent
```

ADL



The existing hybrid programming frameworks

Thank you

https://github.com/Mica-labs/MICA

Four types of agents collectively form an ADL chatbot

KB agent

- **Retrieves information and answers questions.**

  Powers tasks like retrieval-augmented generation (RAG) and FAQ response.

```
<kb agent name>:
  sources(optional):
    - <sting>
  faq(optional):
    - q: <string>
      a: <string>
```

LLM agent

- **Encodes domain knowledge through prompt programming.**

  Enables task-specific reasoning and constraint handling.

```
<llm agent name>:
  prompt: <string>
  uses(optional):
    - <sting>
  steps(optional):
    - <step>
```

Common Attributes

```
base:
  type: <string>
  description: <string>
  args(optional): <array>
  fallback(optional): <string | agent>
  exit(optional): <string | agent>
```

Flow agent

- **Provides fine-grained and precise flow control.**

  Supports complex dialogue flows similar to traditional programs.

```
<flow agent name>:
  steps:
    - <step>
  <subflow name>(optional):
    - <step>
```

Ensemble agent

- **Coordinates agent responses based on context.**

  Orchestrates multi-agent interaction using user input and dialogue state.

```
<ensemble agent name>:
  contains:
    - <string>
  prompt(optional): <string>
  steps(optional):
    - <step>
```