**Marketing Campaign Assistant - Detailed Technical Workflow**

**System Architecture Overview**

The Marketing Campaign Assistant is structured as a multi-layered application that leverages Azure OpenAI's GPT models to power a conversational marketing campaign planner. The application follows a modular design pattern with clear separation of concerns that allows for maintainability and extensibility.

┌─────────────────────────────────────────────────────────────────┐

│ User Interface │

│ (main.py) │

└───────────────────────────────┬─────────────────────────────────┘

│

▼

┌─────────────────────────────────────────────────────────────────┐

│ Campaign Assistant Layer │

│ (agents/campaign\_assistant.py) │

└───────────────────────────────┬─────────────────────────────────┘

│

▼

┌─────────────────────────────────────────────────────────────────┐

│ Message Generation Layer │

│ (agents/message\_agent.py, workflows/message\_workflow.py)│

└───────────────────────────────┬─────────────────────────────────┘

│

▼

┌─────────────────────────────────────────────────────────────────┐

│ Infrastructure Layer │

│ (config/azure\_config.py, utils/text\_extractor.py) │

└─────────────────────────────────────────────────────────────────┘

**Detailed Core Workflow**

Let's trace through the exact flow of execution from when a user starts the application to when they receive the final marketing messages.

**1. Application Initialization**

# main.py

if \_\_name\_\_ == "\_\_main\_\_":

# Initialize configuration

config = AzureConfig()

# Step 1: Run the campaign assistant to get a campaign plan.

campaign\_data = run\_campaign\_assistant()

# ...

* The execution begins in main.py when the script is run directly
* An AzureConfig instance is created to provide LLM connection details
* The run\_campaign\_assistant() function is called to begin the interactive workflow

**2. Intent Classification**

# Function in main.py

def run\_campaign\_assistant():

# ...

intent\_prompt = ChatPromptTemplate.from\_messages([

("system", INTENT\_CLASSIFICATION\_SYSTEM\_PROMPT),

("human", user\_input)

])

intent\_response = assistant.llm.invoke(intent\_prompt.format\_messages())

intent = intent\_response.content.strip().upper()

# ...

* The system uses LangChain's prompt template system to create a structured prompt
* The LLM classifies the user's intent as either "AUTO-GENERATE" or "ASK-QUESTIONS"
* The assistant also checks for specific campaign keywords in the user's input

**3A. Auto-Generate Campaign Path**

# Conditional branch in run\_campaign\_assistant()

if intent == "AUTO-GENERATE" and mentioned\_campaign:

# ...

campaign\_data = assistant.auto\_generate\_campaign\_plan(mentioned\_campaign)

# ...

If the user wants to auto-generate a campaign:

1. **Inside auto\_generate\_campaign\_plan**:
2. # in agents/campaign\_assistant.py
3. def auto\_generate\_campaign\_plan(self, campaign\_name):
4. # Filter for matching campaigns in sample data
5. filtered\_data = [item for item in SAMPLE\_CAMPAIGNS if item["Campaign name"].lower() == campaign\_name.lower()]
7. if filtered\_data:
8. # Find best performing campaign by response rate
9. # ...
10. else:
11. # Fallback: use LLM to generate a plan
12. # ...
13. **Data analysis path**:
    * The system first tries to find matching campaigns in the sample data
    * It calculates which campaign had the highest response rate
    * It builds a campaign plan based on the best-performing historical data
14. **LLM fallback path**:
    * If no historical data exists, it uses the LLM to analyze trends
    * The LLM generates a plan based on similar campaigns
    * Normalized data is returned with strict type compliance

**3B. Interactive Campaign Building Path**

# Alternative branch in run\_campaign\_assistant()

else:

# ...

assistant.process\_input(user\_input)

# Interactive question flow

while True:

next\_question = assistant.get\_next\_question()

if not next\_question:

break

# ...

assistant.process\_input(user\_input)

# ...

If the user wants to build a campaign interactively:

1. **Initial input processing**:
2. # Inside process\_input in campaign\_assistant.py
3. def process\_input(self, text):
4. self.conversation\_history.append(f"User: {text}")
5. extracted\_data = self.extract\_campaign\_data(text)
6. # ...
   * The user's input is added to the conversation history
   * The extract\_campaign\_data method uses the LLM to extract structured data
   * The campaign data model is updated with extracted information
7. **Question generation loop**:
8. # Inside get\_next\_question in campaign\_assistant.py
9. def get\_next\_question(self):
10. for field, question in CAMPAIGN\_FIELDS.items():
11. if getattr(self.campaign\_data, field) is None and field not in self.asked\_fields:
12. self.asked\_fields.add(field)
13. return self.generate\_question(field, question)
14. return None
    * The system checks which fields in the campaign data are still empty
    * It selects the next field to ask about and generates a conversational question
    * The question is presented to the user, and their response is processed
15. **Loop termination**:
    * The loop continues until all necessary fields have values
    * The final campaign data is formatted and returned

**4. Message Generation**

# Back in main.py after getting campaign\_data

# Map campaign data fields to message generator input

message\_input = {

"customer\_segment": campaign\_data.get("customer\_segment", "None"),

# ...

}

# For message generator, supply keys with names expected by the prompt

customer\_input\_for\_messages = {

"customer\_segment": message\_input["customer\_segment"],

# ...

}

# Generate marketing messages

results = run\_marketing\_message\_generator(config, [customer\_input\_for\_messages])

1. **Data transformation**:
   * The campaign data is mapped to the format expected by the message generator
   * Field names are transformed to match the message prompt template
2. **Inside run\_marketing\_message\_generator**:
3. # In workflows/message\_workflow.py
4. def run\_marketing\_message\_generator(config, customers\_data):
5. client = AzureOpenAI(
6. azure\_endpoint=config.ENDPOINT,
7. # ...
8. )
10. agent = MessageAgent(client)
11. workflow = MarketingMessageWorkflow(agent)
12. # ...
    * An OpenAI client is initialized with Azure configurations
    * A MessageAgent is created to handle the actual message generation
    * A MarketingMessageWorkflow is created to orchestrate the process
13. **Workflow execution**:
14. # Inside workflow.run
15. def run(self, customer\_input):
16. initial\_state: GraphState = {
17. "customer\_input": customer\_input,
18. "message\_options": None
19. }
20. final\_state = self.graph.invoke(initial\_state)
21. return {"message\_options": final\_state["message\_options"]}
    * The workflow initializes a state graph with the customer input
    * It invokes the graph, which delegates to the message agent
    * The agent generates message options based on the campaign data
22. **Message generation**:
23. # Inside MessageAgent.generate\_message
24. def generate\_message(self, state):
25. # ...
26. prompt = MESSAGE\_PROMPT\_TEMPLATE.format(\*\*customer\_input)
28. # Request message options from LLM
29. response = self.client.chat.completions.create(
30. model=customer\_input.get("deployment\_name", "gpt-4"),
31. messages=[{"role": "user", "content": prompt}],
32. max\_tokens=300
33. )
34. # ...
    * The message prompt template is formatted with the campaign data
    * The LLM is called directly using the OpenAI client
    * The response is parsed to extract three message options

**5. Result Presentation**

# Back in main.py

# Display results

print("\nGenerated Marketing Messages:")

for result in results:

print(f"Customer: {result['customer\_name']}")

for idx, msg in enumerate(result["message\_options"], start=1):

print(f"Option {idx}: {msg}")

print("-" \* 50)

* The message options are displayed to the user
* Each option is numbered for easy reference
* The user can select their preferred message for the campaign

**Technical Implementation Details**

**State Management**

The application uses several approaches to manage state:

1. **Object attributes** in the MarketingAssistant class:
2. self.campaign\_data = CampaignData()
3. self.asked\_fields = set()
4. self.conversation\_history = []
   * Campaign data is stored in a Pydantic model for type safety
   * Asked fields are tracked to prevent duplicate questions
   * Conversation history provides context for LLM interactions
5. **LangGraph state management**:
6. class GraphState(TypedDict):
7. customer\_input: Dict
8. message\_options: Optional[List[str]]
   * The workflow uses a typed dictionary to represent state
   * State transitions are managed by the graph framework
   * This provides a structured way to pass data between nodes

**Error Handling**

The application implements several error-handling strategies:

1. **Default values** to handle missing data:
2. message\_input = {
3. "customer\_segment": campaign\_data.get("customer\_segment", "None"),
4. # ...
5. }
6. **Type normalization** for consistent data formats:
7. # In TextExtractor.normalize\_customer\_segment
8. if "customer\_segment" in data:
9. if data["customer\_segment"] not in VALID\_SEGMENTS:
10. # Normalize to valid value
11. **Fallback mechanisms** when primary strategies fail:
12. # In auto\_generate\_campaign\_plan
13. # Absolute fallback if no data is available
14. if not data:
15. return {
16. "customer\_segment": "VIP",
17. # Default values...
18. }

**LLM Prompt Strategies**

The application uses several prompt engineering techniques:

1. **Role specification**:
2. """You are an AI assistant that extracts marketing campaign information from conversations."""
3. **Task decomposition**:
4. """
5. Extract the following fields if present in the ENTIRE CONVERSATION:
6. 1. customer\_segment: ...
7. 2. campaign\_theme: ...
8. """
9. **Output format specification**:
10. """Output only JSON format with no explanation or additional text."""
11. **Few-shot examples**:
12. """
13. Examples of AUTO-GENERATE intent:
14. - "Create a White Wednesday campaign for me"
15. - "I need a complete Ramadan marketing plan"
16. """
17. **Constraint specification**:
18. """
19. IMPORTANT EXTRACTION RULES:
20. - For customer\_segment, ONLY output one of these values: ...
21. - For discount, ALWAYS output a numerical percentage with % symbol
22. """

**Integration Points**

The system has several key integration points:

1. **LangChain Integration**:
   * Uses AzureChatOpenAI for higher-level LLM interactions
   * Leverages ChatPromptTemplate for structured prompt management
2. **OpenAI Client Integration**:
   * Uses AzureOpenAI for direct control over completion parameters
   * Enables fine-tuning of parameters like max\_tokens
3. **LangGraph Integration**:
   * Uses StateGraph to model message generation workflow
   * Provides a structured way to handle state transitions
4. **Pydantic Integration**:
   * Uses BaseModel for data validation and serialization
   * Ensures type safety and consistent data formats

**Performance Considerations**

The system includes several optimizations:

1. **Conversation history management**:
2. # Use only recent conversation history to keep context manageable
3. history = "\n".join(self.conversation\_history[-6:]) if self.conversation\_history else ""
   * Only recent messages are included in the context
   * This helps prevent token limits and reduces costs
4. **Sample data prioritization**:
5. # Filter for matching campaigns in sample data
6. filtered\_data = [item for item in SAMPLE\_CAMPAIGNS if item["Campaign name"].lower() == campaign\_name.lower()]
7. if filtered\_data:
8. # Use sample data...
9. else:
10. # Fall back to LLM...
    * Historical data is preferred over LLM generation
    * This reduces API calls and improves response time
11. **Token optimization**:
12. response = self.client.chat.completions.create(
13. # ...
14. max\_tokens=300
15. )
    * Token limits are specified to control costs
    * The prompt is designed to generate concise, focused responses

**Extensibility Points**

The system can be extended in several ways:

1. **New campaign types**:
   * Add new campaign templates to SAMPLE\_CAMPAIGNS
   * The system will automatically use them for recommendations
2. **Additional channels**:
   * Support for new communication channels can be added
   * Message templates can be extended with channel-specific constraints
3. **Enhanced analytics**:
   * The system could be extended to track generated campaign performance
   * This would create a feedback loop to improve recommendations
4. **Multi-modal support**:
   * The message generation could be extended to include image suggestions
   * This would require integration with image generation APIs

**Conclusion**

The Marketing Campaign Assistant demonstrates a sophisticated integration of LLMs into a business process. It combines structured data analysis with natural language generation to create a powerful tool for marketing professionals.

The modular architecture allows for easy maintenance and extension, while the prompt engineering techniques ensure high-quality outputs tailored to marketing needs. The workflow exemplifies how AI can be applied to semi-automate complex creative tasks while still giving users control over the final output.