Building Context-Aware Conversations in AI Systems

The Problem: Broken Reference Resolution

Our AI Coordinator system needed to support natural conversations where users could ask follow-up questions using pronouns and references:

```
User: What is the capital of Spain?
User: What is its population?
User: What is the currency there?
User: Tell me about the previous answer
User: What language is spoken in this country?
```

Initially, the system failed to resolve these references, producing errors:

```
ERROR: Error executing answer_general_question: 'entities'
```

Our Solution: A 4-Part Fix

We implemented a comprehensive solution consisting of four key improvements:

1. Fixed structural code issues
2. Enhanced conversation context tracking
3. Improved follow-up question detection
4. Added defensive programming for context retrieval

1. Structural Fix: Method Indentation

The Problem

The `Nova` class had methods incorrectly indented, causing Python to interpret them as nested functions:

```
# Before: Incorrectly indented methods
@log_function_call
async def identify_multiple_intents_async(self, prompt: str):
    # Method code...

    @log_function_call  # WRONG: Nested inside another method
    async def create_task_for_category(self, intent_text: str, category: str):
        # Method code...
```

The Fix

We corrected the indentation to make all methods direct class members:

```
# After: Properly indented class methods
@log_function_call
async def identify_multiple_intents_async(self, prompt: str):
    # Method code...

@log_function_call  # CORRECT: At class level
async def create_task_for_category(self, intent_text: str, category: str):
    # Method code...
```

2. Enhanced Conversation Context Management

The Problem

The system wasn't maintaining conversation context across turns, preventing it from resolving references.

The Fix

We enhanced the `answer_general_question` function to properly store and retrieve conversation state:

```
@log_function_call
def answer_general_question(kb: KnowledgeBase, prompt: str, input2="-"):
    # Get conversation state from knowledge base, or initialize if not exists
    conversation = kb.get_item("current_conversation") or {
        "questions": [],
        "answers": [],
        "current_country": None,
        "current_city": None,
        "entities": {}
    }

    # Store question
    conversation["questions"].append(prompt)

    # Track entities (e.g., countries) that might be referenced
    for location in LOCATIONS:
        if location.lower() in prompt.lower():
            conversation["current_country"] = location
            conversation["entities"]["country"] = location
            print(f"DETECTED: Country mention: {location}")

    # Keep a maximum of 5 turns
    if len(conversation["questions"]) > 5:
        conversation["questions"] = conversation["questions"][-5:]
        conversation["answers"] = conversation["answers"][-5:]

    # Update conversation state in knowledge base
    kb.set_item("current_conversation", conversation)
```

3. Improved Follow-up Detection

We added sophisticated detection for different types of follow-up questions:

```
 # Check if this is a follow-up question with pronouns or references
follow_up_indicators = [
    " it ", " its ", " it's ", " their ", " there ", " this country ", " this city ",
    " this place ", " that country ", " that city ", " that place "
]

is_follow_up = False
for indicator in follow_up_indicators:
    if indicator in " " + prompt.lower() + " ":  # Ensure we match whole words
        is_follow_up = True
        print(f"DETECTED: Follow-up indicator: {indicator}")
        break

# Questions about "previous answer" are follow-ups
if "previous answer" in prompt.lower() or "last answer" in prompt.lower():
    is_follow_up = True
    print(f"DETECTED: Reference to previous answer")

# Short questions are often follow-ups
if len(prompt.split()) <= 5 and len(conversation["answers"]) > 0:
    is_follow_up = True
    print(f"DETECTED: Short question (likely follow-up): {prompt}")
```

This code identifies several follow-up patterns:

- Explicit references like "it", "its", "there"
- Phrases like "this country" or "that city"
- Questions about "previous answer"
- Short questions (potential implicit follow-ups)

4. Defensive Programming for Error Prevention

Adding safeguards to prevent KeyError exceptions:

```
 # Get conversation state from knowledge base, or initialize if not exists
conversation = kb.get_item("current_conversation") or {
    "questions": [],
    "answers": [],
    "current_country": None,
    "current_city": None,
    "entities": {}
}

# CRITICAL FIX: Ensure the entities key exists
if "entities" not in conversation:
    conversation["entities"] = {}
```

Providing Rich Context to the LLM

The key innovation was providing rich context to the language model, including:

```
 # Create context for the LLM
if len(conversation["answers"]) > 0 and (is_follow_up or True):
    context_prompt = """
INSTRUCTION: You are continuing a conversation. The user's new question may refer to things mentioned earlier.
IMPORTANT: Do NOT ask for clarification about pronouns or references - instead, use the conversation history to figure out what they

CONVERSATION HISTORY:
"""

    # Add conversation history (last 3 exchanges)
    for i in range(min(3, len(conversation["answers"]))):
        q_idx = len(conversation["questions"]) - 2 - i
        a_idx = len(conversation["answers"]) - 1 - i
        if q_idx >= 0 and a_idx >= 0:
            context_prompt += f"User: {conversation['questions'][q_idx]}\n"
            context_prompt += f"You: {conversation['answers'][a_idx]}\n\n"

    # Add current entities explicitly
    if conversation["current_country"]:
        context_prompt += f"CURRENT COUNTRY BEING DISCUSSED: {conversation['current_country']}\n"

    if conversation["current_city"]:
        context_prompt += f"CURRENT CITY BEING DISCUSSED: {conversation['current_city']}\n"

    # Add the current question with explicit resolution instructions
    context_prompt += f"\nCURRENT QUESTION: {prompt}\n\n"
    context_prompt += """
IMPORTANT INSTRUCTIONS:
1. If the question uses pronouns like "it", "its", "there", or phrases like "this country", resolve them based on the conversation h
2. DO NOT ask for clarification about what these pronouns refer to - use the conversation history to determine the referent.
3. Answer the question directly, providing the requested information based on the conversation context.
4. Do not mention that you are using conversation history to understand the question.
"""
```

Entity Extraction from Responses

The system also extracts entities from LLM responses to keep conversation context up-to-date:

```
 # Extract entities from the answer
for location in LOCATIONS:
    if location.lower() in result.lower():
        conversation["current_country"] = location
        conversation["entities"]["country"] = location
        print(f"EXTRACTED: Country from answer: {location}")

# Look for capital cities
cities = ["Paris", "London", "Berlin", "Madrid", "Rome", "Brussels", "Amsterdam", "Vienna", "Athens"]
for city in cities:
    if city.lower() in result.lower():
        conversation["current_city"] = city
        conversation["entities"]["city"] = city
        print(f"EXTRACTED: City from answer: {city}")
```

Results: Successful Context-Aware Conversations

Our fixes transformed the system into a truly context-aware conversational AI:

```
 ===== RESULTS FOR PROMPT 1: What is the capital of spain? =====
Madrid is the capital of Spain.

===== RESULTS FOR PROMPT 2: What is its population? =====
Madrid, the capital city of Spain, has an administrative population of approximately 3.3 million residents. However, if you consider

===== RESULTS FOR PROMPT 3: What is the currency there? =====
The currency used in Spain is the euro.

===== RESULTS FOR PROMPT 4: Tell me about the previous answer =====
The previous answer explained that Madrid is the capital of Spain. It highlighted that Madrid is not only the political center but a

===== RESULTS FOR PROMPT 5: What language is spoken in this country? =====
The official language spoken in Spain is Spanish, also known as Castilian. While Spanish is the predominant language, you might also
```

Key Patterns for Context-Aware AI Systems

1. **Explicit Context Tracking**: Store entities, references, and previous exchanges
2. **Follow-up Detection**: Use linguistic patterns to identify references
3. **Rich Context Provision**: Give the LLM both conversation history and explicit entity information
4. **Error Prevention**: Use defensive programming to handle edge cases
5. **Entity Extraction**: Update context by extracting entities from both user inputs and AI responses

Implementation Checklist

- [x] Store conversation history in a persistent knowledge base
- [x] Track named entities like countries, cities, etc.
- [x] Detect various forms of references in follow-up questions
- [x] Provide rich context to the language model
- [x] Extract and update entities from model responses
- [x] Use defensive programming to handle edge cases
- [x] Support mixed conversation types (questions + math + other tasks)

Conclusion

By combining proper code structure, sophisticated context tracking, and rich context provision to the LLM, we built a system that can maintain coherent conversations across multiple turns, even when handling different types of requests in between.