*from* typing *import* Optional, List

*from* src.agents.base\_agent *import* BaseAgent

*from* src.agents.baby\_gear\_agent *import* BabyGearAgent

*from* src.agents.pregnancy\_agent *import* PregnancyAgent

*from* src.services.llm\_service *import* LLMService

*from* src.agents.general\_agent *import* GeneralAgent

*# Import other agents...*

class AgentFactory:

def **\_\_init\_\_**(*self*):

print("\n=== Initializing AgentFactory ===")

*self*.llm\_service = LLMService()

*self*.agents = {}

async def **get\_agent\_for\_query**(*self*, *query*: str) -> BaseAgent:

*# Analyze query to determine agent type*

*if* 'stroller' in *query*.lower() or 'twins' in *query*.lower():

*return* *self*.\_get\_or\_create\_agent('baby\_gear', BabyGearAgent)

*# Default to general agent*

*return* *self*.\_get\_or\_create\_agent('general', GeneralAgent)

def **\_get\_agent\_class**(*self*, *agent\_type*: str):

"""Get the appropriate agent class"""

*return* {

'baby\_gear': BabyGearAgent,

'pregnancy': PregnancyAgent,

'general': GeneralAgent

}.get(*agent\_type*, GeneralAgent)

def **\_get\_or\_create\_agent**(*self*, *agent\_type*: str, *agent\_class*) -> BaseAgent:

"""Get existing agent or create new one"""

*if* *agent\_type* not in *self*.agents:

*self*.agents[*agent\_type*] = *agent\_class*(*self*.llm\_service)

*return* *self*.agents[*agent\_type*]

def **\_format\_agent\_descriptions**(*self*) -> str:

"""Format agent descriptions for the prompt"""

*return* "\n".join([

f"- {key}: {info['description']}"

*for* key, info *in* *self*.agents.items()

])

def **\_parse\_agent\_selection**(*self*, *llm\_response*: str) -> str:

"""Parse LLM response to get agent key"""

*# Default to first agent if parsing fails*

default\_agent = list(*self*.agents.keys())[0]

*# Look for agent keys in response*

*for* agent\_key *in* *self*.agents.keys():

*if* agent\_key in *llm\_response*.lower():

*return* agent\_key

*return* default\_agent

def **\_is\_gear\_query**(*self*, *query*: str) -> bool:

"""Strict check for gear-related queries"""

gear\_terms = [

"stroller", "עגלה",

"car seat", "כיסא בטיחות",

"crib", "מיטת תינוק",

"price", "cost", "מחיר", "עלות"

]

*return* any(term *in* *query*.lower() *for* term *in* gear\_terms)

def **\_extract\_keywords**(*self*, *query*: str) -> List[str]:

"""Extract keywords from query"""

*# Simple keyword extraction for now*

*return* *query*.lower().split()

*from* typing *import* Dict, List, Any, Optional, Union

*from* .base\_agent *import* BaseAgent

*import* json

class BabyGearAgent(BaseAgent):

def **\_\_init\_\_**(*self*, *llm\_service*):

super().\_\_init\_\_(

*name*="Baby Gear Expert",

*expertise*=["stroller", "car seat", "crib"],

*llm\_service*=*llm\_service*

)

*# Define expertise for agent selection*

*self*.expertise = [

"stroller", "car seat", "crib", "baby gear",

"עגלה", "כיסא בטיחות", "מיטת תינוק", "ציוד לתינוק"

]

async def **\_get\_next\_question**(*self*, *query*: str) -> Dict:

"""Dynamically determine next question based on query"""

*# First time - analyze what info we need*

*if* not *self*.conversation\_state.get('required\_info'):

analysis = *await* *self*.llm\_service.analyze\_query\_intent(*query*)

*self*.conversation\_state['required\_info'] = analysis['required\_info']

*# Get next question based on what we still need*

next\_q = *await* *self*.llm\_service.determine\_next\_question(

*query*,

*self*.conversation\_state['gathered\_context']

)

*return* {

'type': 'follow\_up\_question',

'question': next\_q['question'] *if* next\_q['type'] == 'follow\_up' *else* None

}

async def **\_generate\_response**(*self*) -> Dict:

"""Generate final response using all context"""

enriched\_prompt = *self*.enrich\_prompt\_with\_context()

*# Use Perplexity for real-time data*

recommendations = *await* *self*.perplexity.search(enriched\_prompt)

*return* {

'type': 'answer',

'text': recommendations

}

def **\_calculate\_confidence**(*self*, *query*: str, *keywords*: List[str]) -> float:

"""Calculate confidence score for handling a query based on keyword matches."""

query\_lower = *query*.lower()

*# Core gear-related keywords*

gear\_keywords = [

"stroller", "עגלה", "car seat", "כיסא בטיחות",

"crib", "מיטה", "bassinet", "עריסה",

"gear", "equipment", "ציוד"

]

matching\_keywords = sum(

1 *for* keyword *in* gear\_keywords

*if* keyword *in* query\_lower *or* any(word *in* query\_lower *for* word *in* keyword.split())

)

confidence = min(matching\_keywords / 2, 1.0)

print(f"BabyGearAgent: Found {matching\_keywords} gear-related keywords; confidence = {confidence}")

*return* confidence

def **\_prepare\_prompt**(*self*, *query*: str, *lang*: str) -> str:

"""Prepare the final prompt for the LLM based on the query and language."""

prompt = f"""As a baby gear expert, provide detailed advice about: {*query*}

Analysis Framework:

1. Safety Considerations:

- Age/stage appropriateness

- Relevant safety guidelines

2. Product Evaluation:

- Essential features and quality markers

- Durability and ease of use

3. Value Assessment:

- Price ranges and cost-effectiveness

- Long-term usability and maintenance needs

4. Practical Usage:

- Real-world usage tips and common challenges

5. Recommendations:

- Specific suggestions and alternatives with clear reasoning

Please respond in {*lang*}."""

*return* prompt

def **\_identify\_gear\_type**(*self*, *query*: str) -> Optional[str]:

"""Identify which type of gear the user is asking about."""

gear\_keywords = {

"stroller": ["stroller", "pushchair", "buggy", "עגלה"],

"car\_seat": ["car seat", "carseat", "כיסא בטיחות"],

"crib": ["crib", "bed", "bassinet", "מיטה", "עריסה"]

}

query\_lower = *query*.lower()

*for* gear, keywords *in* gear\_keywords.items():

*if* any(keyword *in* query\_lower *for* keyword *in* keywords):

*return* gear

*return* None

async def **can\_handle\_query**(*self*, *query*: str, *keywords*: List[str]) -> bool:

"""Decide if this agent should handle the query."""

print("\n=== BabyGearAgent Checking Query ===")

print(f"Query received: {*query*}")

*# First check if this is explicitly about baby gear*

gear\_related = any(word *in* *query*.lower() *for* word *in* [

"stroller", "עגלה", "car seat", "כיסא בטיחות",

"crib", "מיטה", "bassinet", "עריסה"

])

*if* gear\_related:

print("Detected gear-related query")

*return* True

*# If not explicitly about gear, calculate general confidence*

confidence = *self*.\_calculate\_confidence(*query*, *keywords*)

print(f"General confidence score: {confidence}")

*# Only handle high confidence gear-related queries*

*return* confidence > 0.8 and any(word *in* *query*.lower() *for* word *in* [

"gear", "equipment", "product", "buy", "purchase",

"ציוד", "מוצר", "לקנות", "רכישה"

])

def **\_set\_role\_boundaries**(*self*):

"""Define what this agent can or cannot do."""

*self*.role\_boundaries = {

"can\_do": [

"product recommendations",

"price comparisons",

"feature analysis",

"safety guidelines",

"age appropriateness",

"product specifications"

],

"cannot\_do": [

"medical advice",

"health recommendations",

"pregnancy guidance",

"child development",

"parenting advice"

]

}

def **\_is\_hebrew**(*self*, *text*: str) -> bool:

"""Check if the text contains Hebrew characters."""

*return* any('\u0590' <= c <= '\u05FF' *for* c *in* *text*)

def **\_get\_language**(*self*, *text*: str) -> str:

"""Return 'he' if Hebrew, otherwise 'en'."""

*return* 'he' *if* *self*.\_is\_hebrew(*text*) *else* 'en'

def **add\_context**(*self*, *field*: str, *value*: str):

"""Maintain compatibility with existing code"""

*self*.conversation\_state['gathered\_info'][*field*] = *value*

def **\_get\_missing\_context**(*self*) -> List[str]:

"""Return a list of required context fields that haven't been answered."""

*return* [field *for* field *in* *self*.question\_flow[0]['extract\_keys'] *if* field not in *self*.conversation\_state['gathered\_info']]

def **\_generate\_relevant\_question**(*self*, *field*: str, *lang*: str) -> str:

"""Generate a follow-up question for the missing context field."""

*return* *self*.question\_flow[0]['question'][*lang*]

def **\_analyze\_query\_needs**(*self*, *query*: str) -> List[str]:

"""Analyze the query to determine which context fields are actually needed."""

needed\_info = []

query\_lower = *query*.lower()

print("\n=== Analyzing Query Needs ===")

print(f"Query: {query\_lower}")

*# For queries about specific price points (cheapest/most expensive), we don't need context*

*if* any(word *in* query\_lower *for* word *in* [

"cheapest", "most expensive", "highest price", "lowest price",

"הכי זול", "הכי יקר", "המחיר הגבוה", "המחיר הנמוך"

]):

print("Price comparison query - no context needed")

*return* []

*# For general price queries, we need budget*

*if* any(word *in* query\_lower *for* word *in* ["price", "cost", "budget", "מחיר", "עלות", "תקציב"]):

print("Price query - need budget")

needed\_info.append("budget\_range")

*# For recommendation queries, we need more context*

*if* any(word *in* query\_lower *for* word *in* ["recommend", "best", "good", "suitable", "מומלץ", "טוב", "מתאים"]):

print("Recommendation query - need relevant context")

needed\_info.append("age")

*if* "car" in query\_lower or "רכב" in query\_lower:

needed\_info.append("car\_type")

*if* "space" in query\_lower or "אחסון" in query\_lower:

needed\_info.append("living\_situation")

print(f"Needed information: {needed\_info}")

*return* needed\_info

async def **process\_query**(*self*, *query*: str, *context*: Dict) -> Dict:

*# First query - set query type*

*if* not *context*.get('query\_type'):

*if* 'twins' in *query*.lower():

*context*['query\_type'] = 'twin\_stroller'

*else*:

*context*['query\_type'] = 'stroller'

*# Use base processing*

*return* *await* super().process\_query(*query*, *context*)

async def **\_generate\_final\_response**(*self*, *query*: str, *context*: Dict) -> Dict:

"""Generate stroller-specific recommendations"""

prompt = f"""Original Stroller Query: {*context*['original\_query']}

Requirements:

{json.dumps(*context*['gathered\_info'], *indent*=2)}

Provide specific stroller recommendations that:

1. Match the budget and requirements

2. Include model names and features

3. Explain why each recommendation fits their needs

"""

response = *await* *self*.llm\_service.generate\_response(prompt)

*return* {

'type': 'answer',

'text': response['text']

}

def **\_format\_gathered\_info**(*self*) -> str:

"""Format gathered information for prompt"""

info = []

*for* question, answer *in* *self*.conversation\_state['gathered\_info'].items():

info.append(f"Q: {question}\nA: {answer}")

*return* "\n\n".join(info)

def **\_extract\_product\_info**(*self*, *text*: str) -> List[Dict]:

"""Extract structured product information from response text"""

products = []

lines = *text*.split('\n')

current\_product = {}

*for* line *in* lines:

*if* line.strip():

*# Look for product names (capitalized words)*

*if* any(word[0].isupper() *for* word *in* line.split()):

*if* current\_product:

products.append(current\_product)

current\_product = {"name": line.strip()}

*# Look for prices*

*elif* "$" in line:

current\_product["price"] = line[line.find("$"):].split()[0]

*# Extract features from the same line*

features = line[:line.find("$")].strip()

*if* features:

current\_product["features"] = features

*if* current\_product:

products.append(current\_product)

*return* products

def **\_calculate\_domain\_relevance**(*self*, *query*: str) -> float:

"""Check if query is relevant to agent's domain"""

query\_lower = *query*.lower()

*# Core gear terms that strongly indicate this is a gear query*

gear\_terms = [

"stroller", "car seat", "crib", "bassinet",

"עגלה", "כיסא בטיחות", "מיטת תינוק", "עריסה",

"gear", "equipment", "product", "buy", "purchase",

"ציוד", "מוצר", "לקנות", "רכישה"

]

*# If query contains gear terms, this is definitely our domain*

*if* any(term *in* query\_lower *for* term *in* gear\_terms):

print("Found gear-specific terms")

*return* 1.0 *# Return 1.0 for gear queries*

*return* 0.0 *# Not our domain*

*from* typing *import* Union, Dict, Optional, List

*from* src.agents.base\_agent *import* BaseAgent

*import* json

class ChatSession:

def **\_\_init\_\_**(*self*, *agent\_factory*):

*self*.agent\_factory = *agent\_factory*

*self*.conversation\_state = {

'original\_query': None,

'query\_type': None, *# e.g., "twin\_stroller"*

'current\_agent': None,

'gathered\_info': {},

'last\_field': None

}

async def **process\_query**(*self*, *query*: str) -> Dict:

*try*:

*# First message - initialize conversation*

*if* not *self*.conversation\_state['original\_query']:

*self*.conversation\_state.update({

'original\_query': *query*,

'query\_type': 'twin\_stroller', *# Set based on analysis*

'current\_agent': *await* *self*.agent\_factory.get\_agent\_for\_query(*query*)

})

*# Store previous answer if applicable*

*if* *self*.conversation\_state['last\_field']:

*self*.conversation\_state['gathered\_info'][

*self*.conversation\_state['last\_field']

] = *query*

*# Process with FULL context*

response = *await* *self*.conversation\_state['current\_agent'].process\_query(

*query*=*query*,

*context*=*self*.conversation\_state *# Pass ENTIRE state*

)

*# Update last field if it's a question*

*if* response.get('type') == 'follow\_up\_question':

*self*.conversation\_state['last\_field'] = response['field']

*return* response

*except* Exception *as* e:

print(f"Error in chat session: {e}")

*raise*

def **\_update\_conversation\_state**(*self*, *query*: str, *response*: Dict):

"""Update conversation state with new interaction"""

*self*.conversation\_state['conversation\_history'].append({

'query': *query*,

'response': *response*,

'context': *self*.conversation\_state['gathered\_context'].copy()

})

def **\_is\_new\_topic**(*self*, *query*: str) -> bool:

"""Determine if this query is about a new topic"""

*if* not *self*.conversation\_state['current\_topic']:

*return* True

*# Use LLM to check if query is related to current topic*

topic\_similarity = *self*.\_calculate\_topic\_similarity(

*query*,

*self*.conversation\_state['current\_topic']

)

*return* topic\_similarity < 0.3 *# Threshold for new topic*

def **\_is\_followup\_question**(*self*, *message*: str) -> bool:

"""Detect if this is a follow-up question"""

print("\n=== Checking for Follow-up ===")

*# Check if we have a previous topic/agent*

*if* not *self*.conversation\_state['current\_topic']:

print("No previous topic found")

*return* False

*# Check for product names from previous response*

*if* *self*.conversation\_state['gathered\_context'].get('text'):

product\_names = *self*.\_extract\_product\_names(*self*.conversation\_state['gathered\_context']['text'])

print(f"Found product names: {product\_names}")

*# Check if any product name is in the query*

*for* name *in* product\_names:

*if* name.lower() in *message*.lower():

print(f"Found product name '{name}' in query")

*return* True

*# Common follow-up indicators*

followup\_indicators = [

"what about", "how about", "and", "can you", "what if",

"tell me more", "more info", "links", "where", "which one",

"מה לגבי", "ומה עם", "ספר לי עוד", "איפה", "איזה", "link"

]

*# Check for follow-up indicators*

is\_followup = any(indicator *in* *message*.lower() *for* indicator *in* followup\_indicators)

print(f"Follow-up indicator found: {is\_followup}")

*return* is\_followup

def **\_extract\_product\_names**(*self*, *text*: str) -> List[str]:

"""Extract product names from text"""

*# Simple extraction - look for capitalized words*

words = *text*.split()

product\_names = []

*for* i, word *in* enumerate(words):

*if* word[0].isupper() and len(word) > 2:

*# Get full product name (multiple capitalized words)*

name = [word]

j = i + 1

*while* j < len(words) and words[j][0].isupper():

name.append(words[j])

j += 1

*if* len(name) > 0:

product\_names.append(" ".join(name))

*return* product\_names

def **\_get\_next\_context\_field**(*self*) -> Optional[str]:

"""Get the next field we need to ask about"""

*if* not *self*.conversation\_state['current\_agent']:

*return* None

*for* field *in* *self*.conversation\_state['current\_agent'].required\_context:

*if* field not in *self*.conversation\_state['gathered\_context']:

*return* field

*return* None

async def **\_handle\_context\_requirements**(*self*, *message*: str) -> Union[str, Dict]:

"""Check and handle any required context before processing query"""

print("\n=== Checking Context Requirements ===")

*# Get missing context*

missing\_context = *self*.conversation\_state['current\_agent'].\_get\_missing\_context()

print(f"Missing context: {missing\_context}")

*if* missing\_context:

*# Set up for first context question*

*self*.conversation\_state['awaiting\_context'] = True

*self*.conversation\_state['awaiting\_field'] = missing\_context[0]

*# Generate context question*

question = *self*.conversation\_state['current\_agent'].\_generate\_context\_question(missing\_context[0])

print(f"Asking context question: {question}")

*return* {

'type': 'follow\_up\_question',

'question': question,

'field': missing\_context[0]

}

*# If no context needed, proceed with query*

*return* *await* *self*.\_generate\_final\_response(*message*)

async def **\_handle\_context\_answer**(*self*, *message*: str) -> Union[str, Dict]:

"""Handle answer to a context question"""

print("\n=== Processing Context Answer ===")

field = *self*.conversation\_state['awaiting\_field']

print(f"Field: {field}, Answer: {*message*}")

*# Store the context*

*self*.conversation\_state['current\_agent'].add\_context(field, *message*)

*self*.conversation\_state['gathered\_context'][field] = *message*

*# Check if we need more context*

missing\_context = *self*.conversation\_state['current\_agent'].\_get\_missing\_context()

print(f"Remaining missing context: {missing\_context}")

*if* missing\_context:

*# Ask for next piece of context*

next\_field = missing\_context[0]

*self*.conversation\_state['awaiting\_field'] = next\_field

question = *self*.conversation\_state['current\_agent'].\_generate\_context\_question(next\_field)

print(f"Asking next question: {question}")

*return* {

'type': 'follow\_up\_question',

'question': question,

'field': next\_field

}

*# All context gathered, generate final response*

print("All context gathered, generating response")

*return* *await* *self*.\_generate\_final\_response(

*self*.conversation\_state['current\_topic']

)

async def **\_generate\_final\_response**(*self*, *query*: str) -> Dict:

"""Generate final response using all gathered context"""

response = *await* *self*.conversation\_state['current\_agent'].process\_query(

*query*=*query*,

*context*={

'context\_gathered': *self*.conversation\_state['gathered\_context'],

'original\_query': *query*

}

)

*# Store but don't modify the response*

*self*.\_update\_conversation\_state(*query*, response)

*# Return original response - no wrapping!*

*return* response

def **\_reset\_state**(*self*):

"""Reset conversation state but keep history"""

history = *self*.conversation\_state['conversation\_history']

*self*.conversation\_state = {

'current\_agent': None,

'current\_topic': None,

'gathered\_context': {},

'conversation\_history': history

}

def **get\_conversation\_summary**(*self*) -> Dict:

"""Get summary of current conversation state"""

*return* {

'current\_topic': *self*.conversation\_state['current\_topic'],

'gathered\_context': *self*.conversation\_state['gathered\_context'],

'questions\_asked': len(*self*.conversation\_state['conversation\_history']),

'current\_field': *self*.conversation\_state['awaiting\_field']

}

def **get\_gathered\_context**(*self*) -> Dict:

"""Get all gathered context"""

*return* *self*.conversation\_state['gathered\_context']

def **clear\_context**(*self*):

"""Clear all gathered context"""

*self*.\_reset\_state()

*from* openai *import* AsyncOpenAI

*from* typing *import* Dict, Any, Optional

*from* src.config *import* Config

*import* json

class LLMService:

def **\_\_init\_\_**(*self*, *api\_key*: str = None, *model*: str = None):

print("\n=== Initializing LLMService ===")

*self*.api\_key = *api\_key* or Config.OPENAI\_API\_KEY

*self*.model = *model* or Config.MODEL\_NAME

print(f"Using model: {*self*.model}")

*# Use standard OpenAI client without custom base\_url*

*self*.client = AsyncOpenAI(*api\_key*=*self*.api\_key)

*self*.system\_prompt = """You are a parenting expert assistant.

Keep your responses concise and focused.

Provide specific, actionable recommendations.

Limit response to 2-3 key points.

Use bullet points for clarity.

Avoid general advice and focus on the specific context provided."""

async def **generate\_response**(*self*, *prompt*: str) -> Dict:

"""Generate response with consistent format. Only accepts prompt parameter."""

print("\n=== Generating Response ===")

print(f"Using model: {*self*.model}")

*try*:

response = *await* *self*.client.chat.completions.create(

*model*=*self*.model,

*messages*=[

{"role": "system", "content": *self*.system\_prompt},

{"role": "user", "content": *prompt*}

]

)

*# Should return ONLY:*

*return* {

"type": "answer",

"text": str(response.choices[0].message.content)

}

*except* Exception *as* e:

print(f"Error in LLMService: {str(e)}")

*return* {

"type": "error",

"text": "Error connecting to LLM service. Please check your API key and try again."

}

def **\_prepare\_prompt**(*self*, *prompt*: str, *context*: Dict) -> str:

*if* not *context* or not *context*.get('context\_gathered'):

*return* *prompt*

context\_str = "\n".join([

f"{key}: {value}"

*for* key, value *in* *context*.get('context\_gathered', {}).items()

])

*return* f"""Based on this context:

{context\_str}

Query: {*prompt*}

Provide a concise, specific response that:

1. Addresses the exact question

2. Uses the provided context

3. Gives clear recommendations

Keep it brief and focused."""

async def **analyze\_query\_intent**(*self*, *query*: str, *context*: Dict = None) -> Dict:

*# Build context string*

context\_str = ""

*if* *context* and *context*.get('original\_query'):

context\_str = f"""

IMPORTANT - This is a conversation about: {*context*['query\_type']}

Original Query: {*context*['original\_query']}

Information we already have:

{json.dumps(*context*.get('gathered\_info', {}), *indent*=2)}

"""

prompt = f"""Given this conversation:

{context\_str}

Current User Input: "{*query*}"

IMPORTANT: This is a CONTINUOUS conversation.

The current input is either:

1. An answer to our previous question about {*context*.get('last\_field') *if* *context* *else* 'N/A'}

2. A new query (only if there's no context)

Return JSON:

{{

"query\_type": "{*context*['query\_type'] *if* *context* *else* 'determine\_from\_query'}",

"missing\_info": [

{{

"field": "what we still need to know",

"question": "how to ask for it"

}}

]

}}"""

response = *await* *self*.generate\_response(prompt)

*return* json.loads(response['text'])

async def **generate\_response\_with\_context**(*self*, *query*: str, *context*: Dict) -> Dict:

"""Generate final response using all context"""

prompt = f"""Original Query: {*context*.get('original\_query', *query*)}

Context Information:

{json.dumps(*context*.get('user\_inputs', {}), *indent*=2)}

Current Query: {*query*}

Provide a detailed response that:

1. Uses all available context

2. Addresses the original query

3. Incorporates the latest information

Format: Provide a clear, structured response."""

response = *await* *self*.generate\_response(prompt)

*return* response

async def **determine\_next\_question**(*self*, *query*: str, *context*: Dict) -> Dict:

"""Determine what information to ask for next"""

prompt = f"""Original query: {*query*}

Current context: {json.dumps(*context*)}

What's the most important piece of information we still need?

If we have enough information, respond with "COMPLETE".

Format: Either "COMPLETE" or a specific question."""

response = *await* *self*.generate\_response(prompt)

*return* {

'type': 'follow\_up' *if* 'COMPLETE' not in response['text'].upper() *else* 'complete',

'question': response['text'] *if* 'COMPLETE' not in response['text'].upper() *else* None

}

async def **determine\_needed\_fields**(*self*, *query*: str, *context*: Dict = None) -> Dict:

*# If we have context, include it in the prompt*

context\_str = ""

*if* *context* and *context*.get('original\_query'):

context\_str = f"""

Original Query: {*context*['original\_query']}

Already Known:

{json.dumps(*context*.get('user\_inputs', {}), *indent*=2)}

"""

prompt = f"""Given this conversation:

{context\_str}

Current Query: "{*query*}"

1. Maintain context of the original query

2. Consider information already provided

3. Determine what ADDITIONAL information is still needed

Return as JSON:

{{

"context\_type": "what we're discussing (e.g., twin stroller)",

"fields": [

{{

"name": "specific\_field\_name",

"question": "what to ask next",

"importance": "why needed"

}}

],

"already\_known": {{

"field": "value extracted from previous answers"

}}

}}"""

*try*:

response = *await* *self*.generate\_response(prompt)

result = json.loads(response['text'])

print(f"Context analysis: {result}")

*return* result

*except* Exception *as* e:

print(f"Error analyzing context: {e}")

*return* {"fields": []}

*from* typing *import* Union, Dict, Optional, List

*from* src.agents.base\_agent *import* BaseAgent

*import* json

class ChatSession:

def **\_\_init\_\_**(*self*, *agent\_factory*):

*self*.agent\_factory = *agent\_factory*

*self*.conversation\_state = {

'original\_query': None,

'query\_type': None, *# e.g., "twin\_stroller"*

'current\_agent': None,

'gathered\_info': {},

'last\_field': None

}

async def **process\_query**(*self*, *query*: str) -> Dict:

*try*:

*# First message - initialize conversation*

*if* not *self*.conversation\_state['original\_query']:

*self*.conversation\_state.update({

'original\_query': *query*,

'query\_type': 'twin\_stroller', *# Set based on analysis*

'current\_agent': *await* *self*.agent\_factory.get\_agent\_for\_query(*query*)

})

*# Store previous answer if applicable*

*if* *self*.conversation\_state['last\_field']:

*self*.conversation\_state['gathered\_info'][

*self*.conversation\_state['last\_field']

] = *query*

*# Process with FULL context*

response = *await* *self*.conversation\_state['current\_agent'].process\_query(

*query*=*query*,

*context*=*self*.conversation\_state *# Pass ENTIRE state*

)

*# Update last field if it's a question*

*if* response.get('type') == 'follow\_up\_question':

*self*.conversation\_state['last\_field'] = response['field']

*return* response

*except* Exception *as* e:

print(f"Error in chat session: {e}")

*raise*

def **\_update\_conversation\_state**(*self*, *query*: str, *response*: Dict):

"""Update conversation state with new interaction"""

*self*.conversation\_state['conversation\_history'].append({

'query': *query*,

'response': *response*,

'context': *self*.conversation\_state['gathered\_context'].copy()

})

def **\_is\_new\_topic**(*self*, *query*: str) -> bool:

"""Determine if this query is about a new topic"""

*if* not *self*.conversation\_state['current\_topic']:

*return* True

*# Use LLM to check if query is related to current topic*

topic\_similarity = *self*.\_calculate\_topic\_similarity(

*query*,

*self*.conversation\_state['current\_topic']

)

*return* topic\_similarity < 0.3 *# Threshold for new topic*

def **\_is\_followup\_question**(*self*, *message*: str) -> bool:

"""Detect if this is a follow-up question"""

print("\n=== Checking for Follow-up ===")

*# Check if we have a previous topic/agent*

*if* not *self*.conversation\_state['current\_topic']:

print("No previous topic found")

*return* False

*# Check for product names from previous response*

*if* *self*.conversation\_state['gathered\_context'].get('text'):

product\_names = *self*.\_extract\_product\_names(*self*.conversation\_state['gathered\_context']['text'])

print(f"Found product names: {product\_names}")

*# Check if any product name is in the query*

*for* name *in* product\_names:

*if* name.lower() in *message*.lower():

print(f"Found product name '{name}' in query")

*return* True

*# Common follow-up indicators*

followup\_indicators = [

"what about", "how about", "and", "can you", "what if",

"tell me more", "more info", "links", "where", "which one",

"מה לגבי", "ומה עם", "ספר לי עוד", "איפה", "איזה", "link"

]

*# Check for follow-up indicators*

is\_followup = any(indicator *in* *message*.lower() *for* indicator *in* followup\_indicators)

print(f"Follow-up indicator found: {is\_followup}")

*return* is\_followup

def **\_extract\_product\_names**(*self*, *text*: str) -> List[str]:

"""Extract product names from text"""

*# Simple extraction - look for capitalized words*

words = *text*.split()

product\_names = []

*for* i, word *in* enumerate(words):

*if* word[0].isupper() and len(word) > 2:

*# Get full product name (multiple capitalized words)*

name = [word]

j = i + 1

*while* j < len(words) and words[j][0].isupper():

name.append(words[j])

j += 1

*if* len(name) > 0:

product\_names.append(" ".join(name))

*return* product\_names

def **\_get\_next\_context\_field**(*self*) -> Optional[str]:

"""Get the next field we need to ask about"""

*if* not *self*.conversation\_state['current\_agent']:

*return* None

*for* field *in* *self*.conversation\_state['current\_agent'].required\_context:

*if* field not in *self*.conversation\_state['gathered\_context']:

*return* field

*return* None

async def **\_handle\_context\_requirements**(*self*, *message*: str) -> Union[str, Dict]:

"""Check and handle any required context before processing query"""

print("\n=== Checking Context Requirements ===")

*# Get missing context*

missing\_context = *self*.conversation\_state['current\_agent'].\_get\_missing\_context()

print(f"Missing context: {missing\_context}")

*if* missing\_context:

*# Set up for first context question*

*self*.conversation\_state['awaiting\_context'] = True

*self*.conversation\_state['awaiting\_field'] = missing\_context[0]

*# Generate context question*

question = *self*.conversation\_state['current\_agent'].\_generate\_context\_question(missing\_context[0])

print(f"Asking context question: {question}")

*return* {

'type': 'follow\_up\_question',

'question': question,

'field': missing\_context[0]

}

*# If no context needed, proceed with query*

*return* *await* *self*.\_generate\_final\_response(*message*)

async def **\_handle\_context\_answer**(*self*, *message*: str) -> Union[str, Dict]:

"""Handle answer to a context question"""

print("\n=== Processing Context Answer ===")

field = *self*.conversation\_state['awaiting\_field']

print(f"Field: {field}, Answer: {*message*}")

*# Store the context*

*self*.conversation\_state['current\_agent'].add\_context(field, *message*)

*self*.conversation\_state['gathered\_context'][field] = *message*

*# Check if we need more context*

missing\_context = *self*.conversation\_state['current\_agent'].\_get\_missing\_context()

print(f"Remaining missing context: {missing\_context}")

*if* missing\_context:

*# Ask for next piece of context*

next\_field = missing\_context[0]

*self*.conversation\_state['awaiting\_field'] = next\_field

question = *self*.conversation\_state['current\_agent'].\_generate\_context\_question(next\_field)

print(f"Asking next question: {question}")

*return* {

'type': 'follow\_up\_question',

'question': question,

'field': next\_field

}

*# All context gathered, generate final response*

print("All context gathered, generating response")

*return* *await* *self*.\_generate\_final\_response(

*self*.conversation\_state['current\_topic']

)

async def **\_generate\_final\_response**(*self*, *query*: str) -> Dict:

"""Generate final response using all gathered context"""

response = *await* *self*.conversation\_state['current\_agent'].process\_query(

*query*=*query*,

*context*={

'context\_gathered': *self*.conversation\_state['gathered\_context'],

'original\_query': *query*

}

)

*# Store but don't modify the response*

*self*.\_update\_conversation\_state(*query*, response)

*# Return original response - no wrapping!*

*return* response

def **\_reset\_state**(*self*):

"""Reset conversation state but keep history"""

history = *self*.conversation\_state['conversation\_history']

*self*.conversation\_state = {

'current\_agent': None,

'current\_topic': None,

'gathered\_context': {},

'conversation\_history': history

}

def **get\_conversation\_summary**(*self*) -> Dict:

"""Get summary of current conversation state"""

*return* {

'current\_topic': *self*.conversation\_state['current\_topic'],

'gathered\_context': *self*.conversation\_state['gathered\_context'],

'questions\_asked': len(*self*.conversation\_state['conversation\_history']),

'current\_field': *self*.conversation\_state['awaiting\_field']

}

def **get\_gathered\_context**(*self*) -> Dict:

"""Get all gathered context"""

*return* *self*.conversation\_state['gathered\_context']

def **clear\_context**(*self*):

"""Clear all gathered context"""

*self*.\_reset\_state()

*from* abc *import* ABC, abstractmethod

*from* typing *import* Dict, List, Any, Optional, Tuple, Union

*from* datetime *import* datetime

*import* sys

*from* pathlib *import* Path

*from* difflib *import* SequenceMatcher

*import* re

*import* json

*# Add the project root to Python path*

project\_root = Path(\_\_file\_\_).parent.parent.parent

sys.path.append(str(project\_root))

*from* src.database.db\_manager *import* DatabaseManager, Conversation, Message

*from* src.services.llm\_service *import* LLMService

*from* src.services.perplexity\_service *import* PerplexityService

*from* src.config *import* Config

*from* src.utils.search\_utils *import* search\_knowledge\_base

class BaseAgent(ABC):

def **\_\_init\_\_**(*self*, *name*: str, *expertise*: List[str], *llm\_service*: LLMService):

print(f"\n=== Initializing {*name*} ===")

*self*.name = *name*

*self*.expertise = *expertise*

*self*.db = DatabaseManager()

*self*.current\_conversation = None

*self*.llm\_service = *llm\_service*

*# Initialize Perplexity service*

*try*:

*if* not Config.PERPLEXITY\_API\_KEY:

*raise* ValueError("PERPLEXITY\_API\_KEY not found in configuration")

*self*.perplexity = PerplexityService(Config.PERPLEXITY\_API\_KEY)

print("Perplexity service initialized successfully")

*except* Exception *as* e:

print(f"Error initializing Perplexity service: {e}")

*self*.perplexity = None

*# Keywords that indicate need for real-time information*

*self*.realtime\_keywords = [

*# English*

"latest", "newest", "recent", "current", "today", "modern",

"price", "cost", "compare", "versus", "vs", "review",

"best", "top", "recommended", "market", "available",

*# Hebrew*

"חדש", "עדכני", "נוכחי", "היום", "מודרני",

"מחיר", "עלות", "השוואה", "לעומת", "סקירה",

"הכי טוב", "מומלץ", "בשוק", "זמין"

]

*self*.context\_questions = {}

*self*.required\_context = [] *# List of required context fields*

*self*.ongoing\_query = None *# Store the original query*

*# Define role boundaries and limitations*

*self*.role\_boundaries = {

"can\_do": [], *# List of tasks the agent can perform*

"cannot\_do": [], *# List of tasks the agent should not attempt*

"refer\_to": {} *# Map of topics to refer to other agents*

}

*# Define confidence thresholds*

*self*.confidence\_thresholds = {

"high": 0.8, *# Confident in expertise area*

"medium": 0.5, *# Can provide general guidance*

"low": 0.2, *# Should refer to other agents*

"minimum": 0.1 *# Reject query if below this*

}

*# Add validation rules*

*self*.validation\_rules = {

"factual\_requirements": [

"must cite sources for medical claims",

"must specify age ranges for recommendations",

"must include safety disclaimers",

"must acknowledge limitations of advice"

],

"prohibited\_content": [

"specific medical diagnoses",

"medication recommendations",

"absolute guarantees",

"precise numerical predictions",

"specific product endorsements"

],

"required\_disclaimers": {

"medical": "This information is for educational purposes only and not a substitute for professional medical advice.",

"safety": "Always follow manufacturer guidelines and consult relevant professionals for safety-critical decisions.",

"financial": "This is general guidance only, not professional financial advice.",

"developmental": "Every child develops differently; consult healthcare providers for concerns."

}

}

*# Define required context fields*

*self*.required\_context = [] *# Will be overridden by specific agents*

*# Define context questions*

*self*.context\_questions\_map = {} *# Will be overridden by specific agents*

*# Define expertise categories and weights*

*self*.expertise\_weights = {

"primary": 1.0, *# Direct expertise matches*

"secondary": 0.5, *# Related topics*

"general": 0.2 *# General knowledge*

}

*# Base conversation state*

*self*.conversation\_state = {

'original\_query': None,

'gathered\_context': {

'user\_inputs': {}

},

'last\_question': None,

'needed\_fields': None,

'current\_field\_index': None

}

*# Field parsing patterns*

*self*.field\_patterns = {

'budget': [

r'budget.\*?(\$?\d+)',

r'under.\*?(\$?\d+)',

r'up to.\*?(\$?\d+)',

r'cost.\*?(\$?\d+)'

],

'features': [

r'lightweight',

r'storage',

r'foldable',

r'compact',

r'travel system'

],

'usage': [

r'daily',

r'travel',

r'walking',

r'jogging',

r'public transport'

]

}

def **start\_conversation**(*self*, *user\_id*: str = "anonymous"):

"""Start a new conversation"""

*self*.current\_conversation = *self*.db.create\_conversation(

*user\_id*=*user\_id*,

*agent\_name*=*self*.name

)

def **add\_to\_history**(*self*, *query*: str, *response*: Union[str, Dict], *confidence\_score*: float):

"""Add a message to the conversation history"""

*try*:

*if* not *self*.current\_conversation:

*self*.start\_conversation()

*# Convert response dict to string if needed*

response\_text = *response*["text"] *if* isinstance(*response*, dict) *else* str(*response*)

*self*.db.add\_message(

*conversation\_id*=*self*.current\_conversation.id,

*query*=*query*,

*response*=response\_text, *# Store only the text part*

*confidence\_score*=*confidence\_score*

)

print(f"Added to history: {*query*} -> {response\_text[:100]}...")

*except* Exception *as* e:

print(f"Error adding to history: {e}")

*raise*

async def **can\_handle\_query**(*self*, *query*: str, *keywords*: List[str]) -> float:

"""Calculate confidence for handling this query"""

print(f"\n=== {*self*.name} Confidence Calculation ===")

query\_lower = *query*.lower()

*# 1. Check domain relevance first*

domain\_score = *self*.\_calculate\_domain\_relevance(query\_lower)

*if* domain\_score == 0:

print(f"Query not in {*self*.name}'s domain")

*return* 0.0

*# 2. Check role conflicts*

conflicts = sum(1 *for* task *in* *self*.role\_boundaries["cannot\_do"]

*if* task.lower() *in* query\_lower)

*if* conflicts > 0:

print(f"❌ Found {conflicts} role conflicts")

*return* 0.0

*# 3. Calculate expertise match*

primary\_matches = [keyword *for* keyword *in* *self*.expertise

*if* keyword.lower() in query\_lower]

primary\_score = len(primary\_matches) \* *self*.expertise\_weights["primary"]

*# 4. Check capabilities match*

capability\_matches = [task *for* task *in* *self*.role\_boundaries["can\_do"]

*if* any(word *in* query\_lower *for* word *in* task.split())]

capability\_score = len(capability\_matches) \* *self*.expertise\_weights["secondary"]

*# Calculate final confidence*

confidence = (domain\_score \* primary\_score + capability\_score) / (1 + len(query\_lower.split()))

print(f"Domain score: {domain\_score}")

print(f"Primary matches: {primary\_matches}")

print(f"Capability matches: {capability\_matches}")

print(f"Final confidence: {confidence}")

*return* confidence

def **\_calculate\_keyword\_match**(*self*, *query*: str, *keywords*: List[str]) -> float:

"""Calculate how well the query matches agent's expertise keywords"""

query\_lower = *query*.lower()

matching\_keywords = sum(

1 *for* keyword *in* *self*.expertise

*if* keyword.lower() *in* query\_lower

)

*return* min(matching\_keywords / 2, 1.0)

def **\_calculate\_domain\_relevance**(*self*, *query*: str) -> float:

"""Check if query is relevant to agent's domain"""

*return* 0.0 *# Default to 0.0 to prevent wrong agent selection*

def **\_analyze\_query\_intent**(*self*, *query*: str) -> float:

"""Analyze the intent of the query to match agent's capabilities"""

query\_lower = *query*.lower()

*# Check if query intent matches agent's capabilities*

can\_do\_matches = sum(

1 *for* task *in* *self*.role\_boundaries["can\_do"]

*if* any(word *in* query\_lower *for* word *in* task.split())

)

*# Check if query involves things agent cannot do*

cannot\_do\_matches = sum(

1 *for* task *in* *self*.role\_boundaries["cannot\_do"]

*if* any(word *in* query\_lower *for* word *in* task.split())

)

*# Calculate score based on matches and mismatches*

score = can\_do\_matches / (len(*self*.role\_boundaries["can\_do"]) + cannot\_do\_matches)

*return* min(score, 1.0)

def **\_check\_context\_compatibility**(*self*, *query*: str) -> float:

"""Check if the current conversation context is compatible with this agent"""

*# If no ongoing conversation, return neutral score*

*if* not hasattr(*self*, 'current\_conversation') or not *self*.current\_conversation:

*return* 0.5

*# Check if query relates to current conversation topic*

context\_relevance = *self*.\_calculate\_context\_relevance(

*query*,

*self*.current\_conversation.get('topic', '')

)

*return* context\_relevance

def **\_get\_domain\_patterns**(*self*) -> List[str]:

"""Get domain-specific patterns for this agent"""

*# Should be overridden by specific agents*

*return* []

def **\_calculate\_context\_relevance**(*self*, *query*: str, *context*: str) -> float:

"""Calculate how relevant a query is to given context"""

*if* not *context*:

*return* 0.5

*# Use text similarity to compare query and context*

similarity = SequenceMatcher(None, *query*.lower(), *context*.lower()).ratio()

*return* similarity

async def **process\_query**(*self*, *query*: str, *context*: Dict) -> Dict:

*try*:

*# Get analysis with full context*

analysis = *await* *self*.llm\_service.analyze\_query\_intent(

*query*=*query*,

*context*=*context*

)

*# If we have all needed info, generate final response*

*if* not analysis.get('missing\_info'):

*return* *await* *self*.\_generate\_final\_response(*query*, *context*)

*# Otherwise, ask next question*

next\_question = analysis['missing\_info'][0]

*return* {

'type': 'follow\_up\_question',

'field': next\_question['field'],

'question': next\_question['question']

}

*except* Exception *as* e:

print(f"Error in BaseAgent: {e}")

*raise*

async def **\_generate\_final\_response**(*self*, *query*: str, *context*: Dict) -> Dict:

"""Generate final response using all context"""

prompt = f"""Original Query: {*context*['original\_query']}

Gathered Information:

{json.dumps(*context*['gathered\_context']['user\_inputs'], *indent*=2)}

Current Query: {*query*}

Provide a detailed response that:

1. Addresses the original query about {*context*['query\_type']}

2. Uses all gathered information

3. Gives specific recommendations

"""

response = *await* *self*.llm\_service.generate\_response(prompt)

*return* {

'type': 'answer',

'text': response['text']

}

def **get\_recent\_context**(*self*, *limit*: int = 5) -> List[Dict[str, Any]]:

"""Get recent conversation history"""

*if* not *self*.current\_conversation:

*return* []

messages = *self*.db.get\_conversation\_history(

*conversation\_id*=*self*.current\_conversation.id,

*limit*=*limit*

)

*return* [

{

'timestamp': msg.timestamp,

'query': msg.query,

'response': msg.response

}

*for* msg *in* messages

]

async def **generate\_response**(*self*, *query*: str, *context*: Optional[str] = None) -> Dict:

"""Generate response with proper context handling"""

*try*:

*# Prepare the base prompt*

prompt = *self*.\_prepare\_prompt(*query*)

*# If additional context is provided, append it to the prompt*

*if* *context*:

prompt += f"\n\nContext:\n{*context*}"

*# Call LLM service with only the prompt argument*

*return* *await* *self*.llm\_service.generate\_response(prompt)

*except* Exception *as* e:

print(f"Error generating response: {e}")

*return* {

"type": "answer",

"text": "I apologize, but I'm having trouble generating a response right now. Please try again later."

}

@abstractmethod

def **\_prepare\_prompt**(*self*, *query*: str) -> str:

"""Prepare the prompt for the specific agent type"""

*pass*

def **\_format\_context**(*self*) -> str:

"""Format gathered context"""

context = []

*for* field, value *in* *self*.conversation\_state['gathered\_context']['user\_inputs'].items():

context.append(f"- {field}: {value}")

*return* "\n".join(context)

def **\_needs\_realtime\_info**(*self*, *query*: str) -> bool:

"""Check if query needs real-time information"""

query\_lower = *query*.lower()

*# Check for real-time keywords*

*for* keyword *in* *self*.realtime\_keywords:

*if* keyword.lower() in query\_lower:

print(f"Real-time info needed due to keyword: {keyword}")

*return* True

*# Check for comparison requests*

*if* any(word *in* query\_lower *for* word *in* ["compare", "versus", "vs", "השוואה", "לעומת"]):

print("Real-time info needed for comparison")

*return* True

*# Check for price/availability queries*

*if* any(word *in* query\_lower *for* word *in* ["price", "cost", "available", "מחיר", "עלות", "זמין"]):

print("Real-time info needed for price/availability")

*return* True

*return* False

async def **search\_existing\_response**(*self*, *query*: str) -> Tuple[Optional[str], float]:

"""Search for existing responses in knowledge base"""

*return* search\_knowledge\_base(*self*.db, *query*, *self*.name)

def **\_calculate\_confidence**(*self*, *query*: str, *keywords*: List[str]) -> float:

"""Calculate confidence score for handling a query"""

*# Convert query to lowercase for case-insensitive matching*

query\_lower = *query*.lower()

*# Count matching keywords, including partial matches for Hebrew*

matching\_keywords = sum(1 *for* keyword *in* *self*.expertise

*if* keyword.lower() *in* query\_lower *or*

any(word *in* query\_lower *for* word *in* keyword.lower().split()))

*# Normalize confidence score*

confidence = min(matching\_keywords / 2, 1.0)

print(f"{*self*.name} matching keywords: {matching\_keywords}, confidence: {confidence}")

*return* confidence

def **\_get\_missing\_context**(*self*) -> List[str]:

"""Get list of required context fields that are missing"""

print("\n=== Checking Missing Context ===")

print(f"Required fields: {*self*.required\_context}")

print(f"Current context: {*self*.context\_questions}")

missing = [

field *for* field *in* *self*.required\_context

*if* field not in *self*.context\_questions

]

print(f"Missing fields: {missing}")

*return* missing

def **\_generate\_context\_question**(*self*, *context\_field*: str) -> str:

"""Generate specific follow-up questions based on missing context"""

questions = {

"age": "How old is your baby?",

"location": "Where are you located (urban/rural)?",

"car\_type": "What type of car do you have?",

"living\_situation": "Do you live in a house or apartment?",

"budget\_range": "What's your budget range?",

"usage\_frequency": "How often will you use this?",

"lifestyle": "Do you often use public transport or mostly drive?",

"storage\_space": "How much storage space do you have available?"

}

*return* questions.get(*context\_field*, f"Please provide information about {*context\_field*}")

def **\_enrich\_query\_with\_context**(*self*, *query*: str) -> str:

"""Combine original query with gathered context"""

context\_info = []

*for* key, value *in* *self*.context\_questions.items():

context\_info.append(f"{key}: {value}")

context\_str = "\n".join(context\_info)

*return* f"""Query: {*query*}

Context:

{context\_str}

Please provide a detailed response that specifically takes into account:

1. The baby's age: {*self*.context\_questions.get('age', 'unknown')}

2. Living situation: {*self*.context\_questions.get('living\_situation', 'unknown')}

3. Car type: {*self*.context\_questions.get('car\_type', 'unknown')}

4. Budget range: {*self*.context\_questions.get('budget\_range', 'unknown')}

Make sure the response is tailored to these specific circumstances."""

def **add\_context**(*self*, *field*: str, *value*: str):

"""Store user's answer to a context question"""

*self*.context\_questions[*field*] = *value*

def **\_validate\_role\_boundaries**(*self*, *query*: str) -> Tuple[bool, Optional[str]]:

"""Check if query falls within agent's role boundaries"""

query\_lower = *query*.lower()

*# Check for tasks agent cannot do*

*for* forbidden *in* *self*.role\_boundaries["cannot\_do"]:

*if* any(word *in* query\_lower *for* word *in* forbidden):

*return* False, f"I apologize, but I cannot provide advice about {forbidden}. Please consult {*self*.role\_boundaries['refer\_to'].get(forbidden, 'appropriate expert')}."

*# Check confidence level*

confidence = *self*.\_calculate\_confidence(*query*, [])

*if* confidence < *self*.confidence\_thresholds["minimum"]:

*return* False, "I apologize, but this query is outside my area of expertise."

*return* True, None

@abstractmethod

def **\_set\_role\_boundaries**(*self*):

"""Each agent must define its specific role boundaries"""

*pass*

def **\_validate\_response**(*self*, *response*: str, *query\_type*: str) -> Tuple[bool, str]:

"""Validate response against rules and modify if needed"""

*# Check for prohibited content*

*for* prohibited *in* *self*.validation\_rules["prohibited\_content"]:

*if* *self*.\_contains\_prohibited\_content(*response*, prohibited):

*return* False, f"Response contained prohibited content: {prohibited}"

*# Add required disclaimers*

*if* *query\_type* in *self*.validation\_rules["required\_disclaimers"]:

*response* += f"\n\n{*self*.validation\_rules['required\_disclaimers'][*query\_type*]}"

*# Validate against factual requirements*

missing\_requirements = []

*for* req *in* *self*.validation\_rules["factual\_requirements"]:

*if* not *self*.\_meets\_requirement(*response*, req):

missing\_requirements.append(req)

*if* missing\_requirements:

*response* = *self*.\_enhance\_response(*response*, missing\_requirements)

*return* True, *response*

def **\_contains\_prohibited\_content**(*self*, *response*: str, *prohibited*: str) -> bool:

"""Check if response contains prohibited content patterns"""

prohibited\_patterns = {

"specific medical diagnoses": r"you have|you are suffering from|you might have",

"medication recommendations": r"take|use|try (this|these) medication|drug|medicine",

"absolute guarantees": r"guarantee|always works|100%|definitely|certainly",

"precise numerical predictions": r"will take \d+ (days|weeks|months)|exactly \d+",

"specific product endorsements": r"buy this|the best product is|we recommend (brand|product)"

}

pattern = prohibited\_patterns.get(*prohibited*, *prohibited*)

*return* bool(re.search(pattern, *response*, re.IGNORECASE))

def **\_meets\_requirement**(*self*, *response*: str, *requirement*: str) -> bool:

"""Check if response meets factual requirements"""

requirement\_patterns = {

"must cite sources": r"(according to|based on|research shows|studies indicate)",

"must specify age ranges": r"(age|month|year)s? (\d+[-–]\d+|and up|or older|or younger)",

"must include safety disclaimers": r"(safety note|caution|warning|important|please note)",

"must acknowledge limitations": r"(may not|might not|cannot|individual results vary)"

}

pattern = requirement\_patterns.get(*requirement*, *requirement*)

*return* bool(re.search(pattern, *response*, re.IGNORECASE))

def **\_determine\_query\_type**(*self*, *query*: str) -> str:

"""Determine the type of query for appropriate validation"""

query\_lower = *query*.lower()

type\_patterns = {

"medical": r"(health|sick|symptoms|disease|condition|treatment)",

"safety": r"(safe|danger|hazard|risk|protect|secure)",

"financial": r"(cost|money|budget|expense|price|afford)",

"developmental": r"(milestone|develop|grow|learn|skill)"

}

*for* qtype, pattern *in* type\_patterns.items():

*if* re.search(pattern, query\_lower):

*return* qtype

*return* "general"

async def **\_generate\_constrained\_response**(*self*, *query*: str, *constraints*: List[str]) -> Dict:

"""Generate response with constraints"""

constrained\_prompt = f"""

Query: {*query*}

Please provide a response that follows these constraints:

{' '.join(f'- {c}' *for* c *in* *constraints*)}

"""

*# Call generate\_response without context parameter*

*return* *await* *self*.llm\_service.generate\_response(constrained\_prompt)

async def **\_generate\_enriched\_response**(*self*, *query*: str, *recent\_context*: Optional[str] = None) -> Dict:

"""Generate response with enriched context"""

enriched\_query = *self*.\_enrich\_query\_with\_context(*query*)

*# Use generate\_response which now handles context properly*

*return* *await* *self*.generate\_response(enriched\_query, *context*=*recent\_context*)

def **\_customize\_response\_with\_context**(*self*, *response*: str) -> str:

"""Add context-specific customization to response"""

context\_prefix = []

*if* 'age' in *self*.context\_questions:

context\_prefix.append(f"עבור תינוק בן {*self*.context\_questions['age']}")

*if* 'living\_situation' in *self*.context\_questions:

context\_prefix.append(f"בהתחשב במגורים ב{*self*.context\_questions['living\_situation']}")

*if* 'car\_type' in *self*.context\_questions:

context\_prefix.append(f"עבור רכב מסוג {*self*.context\_questions['car\_type']}")

*if* 'budget\_range' in *self*.context\_questions:

context\_prefix.append(f"בתקציב של {*self*.context\_questions['budget\_range']}")

*if* 'usage\_frequency' in *self*.context\_questions:

context\_prefix.append(f"לשימוש {*self*.context\_questions['usage\_frequency']}")

*if* context\_prefix:

context\_summary = " ".join(context\_prefix)

*response* = f"המלצות מותאמות אישית {context\_summary}:\n\n{*response*}"

*return* *response*

async def **\_find\_similar\_response**(*self*, *query*: str, *context*: Dict[str, Any]) -> Optional[str]:

"""Find similar response from database with matching context"""

*try*:

*# First, find similar queries*

similar\_responses, similarity\_scores = *await* *self*.search\_existing\_response(*query*)

*if* not similar\_responses:

*return* None

*# Check context similarity for each response*

*for* response, score *in* zip(similar\_responses, similarity\_scores):

stored\_context = *self*.\_get\_stored\_context(response)

*if* *self*.\_is\_context\_similar(stored\_context, *context*) and score > 0.8:

print(f"Found response with similarity score: {score}")

*return* response

*return* None

*except* Exception *as* e:

print(f"Error finding similar response: {str(e)}")

*return* None

def **\_is\_context\_similar**(*self*, *stored\_context*: Dict[str, Any], *current\_context*: Dict[str, Any]) -> bool:

"""Compare stored context with current context"""

required\_fields = ['age', 'location', 'car\_type', 'budget\_range']

*for* field *in* required\_fields:

*if* field not in *stored\_context* or field not in *current\_context*:

*return* False

*# Compare values with some flexibility*

stored\_value = str(*stored\_context*[field]).lower()

current\_value = str(*current\_context*[field]).lower()

*# Age comparison with range tolerance*

*if* field == 'age':

*if* not *self*.\_is\_age\_similar(stored\_value, current\_value):

*return* False

*# Budget comparison with range tolerance*

*elif* field == 'budget\_range':

*if* not *self*.\_is\_budget\_similar(stored\_value, current\_value):

*return* False

*# Direct comparison for other fields*

*elif* stored\_value != current\_value:

*return* False

*return* True

def **\_is\_age\_similar**(*self*, *stored\_age*: str, *current\_age*: str) -> bool:

"""Compare ages with some tolerance"""

*try*:

*# Extract numbers from age strings*

stored\_num = int(''.join(filter(str.isdigit, *stored\_age*)))

current\_num = int(''.join(filter(str.isdigit, *current\_age*)))

*# Allow 2 months difference for babies under 1 year*

*if* 'month' in *stored\_age* and 'month' in *current\_age*:

*return* abs(stored\_num - current\_num) <= 2

*# Allow 6 months difference for older babies*

*return* abs(stored\_num - current\_num) <= 6

*except* ValueError:

*return* False

def **\_is\_budget\_similar**(*self*, *stored\_budget*: str, *current\_budget*: str) -> bool:

"""Compare budget ranges with tolerance"""

*try*:

*# Extract numbers from budget strings*

stored\_nums = [int(n) *for* n *in* re.findall(r'\d+', *stored\_budget*)]

current\_nums = [int(n) *for* n *in* re.findall(r'\d+', *current\_budget*)]

*if* len(stored\_nums) >= 2 and len(current\_nums) >= 2:

stored\_avg = sum(stored\_nums) / len(stored\_nums)

current\_avg = sum(current\_nums) / len(current\_nums)

*# Allow 20% difference in budget*

*return* abs(stored\_avg - current\_avg) / stored\_avg <= 0.2

*except* (ValueError, ZeroDivisionError):

*return* False

*return* False

def **\_store\_response**(*self*, *query*: str, *response*: str, *context*: Dict[str, Any]):

"""Store the response and context in database"""

*try*:

*self*.db.add\_response(

*query*=*query*,

*response*=*response*,

*context*=*context*,

*agent\_name*=*self*.name,

*timestamp*=datetime.now()

)

*except* Exception *as* e:

print(f"Error storing response: {str(e)}")

def **\_identify\_query\_type**(*self*, *query*: str) -> str:

"""Identify the type of query to better format the response"""

query\_lower = *query*.lower()

*if* any(word *in* query\_lower *for* word *in* ["עגלה", "stroller"]):

*return* "stroller"

*elif* any(word *in* query\_lower *for* word *in* ["כיסא בטיחות", "car seat"]):

*return* "car\_seat"

*elif* any(word *in* query\_lower *for* word *in* ["מיטה", "crib"]):

*return* "crib"

*return* "general"

async def **\_analyze\_query\_context\_needs**(*self*, *query*: str) -> List[str]:

"""Use LLM to determine what context is needed"""

prompt = f"""Given this query: "{*query*}"

What essential information do we need from the user to provide a meaningful response?

Return only the list of required fields, separated by commas.

Keep it minimal - only ask for what's truly needed for this specific query."""

*# Get LLM response*

response = *await* *self*.llm\_service.generate\_response(prompt)

*# Extract text from response dict*

*if* isinstance(response, dict) and "text" in response:

response\_text = response["text"]

*else*:

*return* [] *# Return empty list if response format is unexpected*

*# Split the text into fields*

needed\_fields = [

field.strip()

*for* field *in* response\_text.split(',')

*if* field.strip()

]

print(f"LLM determined needed fields: {needed\_fields}")

*return* needed\_fields

def **\_is\_safety\_related**(*self*, *query*: str) -> bool:

"""Check if query is related to safety concerns"""

safety\_keywords = [

"safe", "safety", "danger", "dangerous", "warning",

"hazard", "risk", "protect", "secure", "careful",

*# Hebrew*

"בטיחות", "סכנה", "מסוכן", "אזהרה", "להיזהר"

]

*return* any(word *in* *query*.lower() *for* word *in* safety\_keywords)

def **\_is\_medical\_advice**(*self*, *query*: str) -> bool:

"""Check if query is asking for medical advice"""

medical\_keywords = [

"health", "medical", "doctor", "symptom", "treatment",

"medicine", "prescription", "diagnosis", "condition",

*# Hebrew*

"בריאות", "רפואי", "רופא", "תסמין", "טיפול", "תרופה"

]

*return* any(word *in* *query*.lower() *for* word *in* medical\_keywords)

@abstractmethod

async def **\_process\_agent\_specific**(*self*, *query*: str, *context*: Dict) -> Dict:

"""Each agent must implement their specific query processing"""

*pass*

def **get\_conversation\_state**(*self*) -> Dict:

"""Get current conversation state"""

*return* *self*.conversation\_state

def **update\_conversation\_state**(*self*, *updates*: Dict):

"""Update conversation state"""

*self*.conversation\_state.update(*updates*)

def **\_parse\_field\_value**(*self*, *field*: str, *answer*: str) -> str:

"""Parse user's answer for specific field"""

answer\_lower = *answer*.lower()

*if* *field* == 'budget':

*# Extract numeric value with $ sign if present*

*for* pattern *in* *self*.field\_patterns['budget']:

*if* match := re.search(pattern, answer\_lower):

value = match.group(1)

*return* value *if* '$' in value *else* f'${value}'

*return* *answer* *# Return original if no pattern matches*

*elif* *field* == 'features':

*# Extract mentioned features*

features = []

*for* feature *in* *self*.field\_patterns['features']:

*if* feature in answer\_lower:

features.append(feature)

*return* ', '.join(features) *if* features *else* *answer*

*elif* *field* == 'usage':

*# Extract usage patterns*

usages = []

*for* usage *in* *self*.field\_patterns['usage']:

*if* usage in answer\_lower:

usages.append(usage)

*return* ', '.join(usages) *if* usages *else* *answer*

*return* *answer* *# Default to original answer if no specific parsing*

def **\_parse\_llm\_field**(*self*, *field*: str) -> str:

"""Parse field name from LLM response"""

*# Clean up field name*

*field* = *field*.lower().strip('- ').split('\n')[0]

*# Map to standard fields*

field\_mapping = {

'budget range': 'budget',

'price range': 'budget',

'cost range': 'budget',

'desired features': 'features',

'stroller features': 'features',

'important features': 'features',

'intended use': 'usage',

'main use': 'usage',

'primary use': 'usage'

}

*return* field\_mapping.get(*field*, *field*)

def **\_store\_answer**(*self*, *field*: str, *answer*: str):

"""Store parsed answer in conversation state"""

parsed\_field = *self*.\_parse\_llm\_field(*field*)

parsed\_value = *self*.\_parse\_field\_value(parsed\_field, *answer*)

*self*.conversation\_state['gathered\_context']['user\_inputs'][parsed\_field] = parsed\_value

print(f"Stored {parsed\_field}: {parsed\_value}")

async def **analyze\_query\_needs**(*self*, *query*: str, *context*: Dict) -> Dict:

"""Analyze what information is needed for this query"""

prompt = f"""Given this query: "{*query*}"

And this context: {*context*}

What essential information do we still need to provide a helpful response?

If we have enough information, respond with "COMPLETE".

Otherwise, ask ONE specific question to gather the most important missing information.

Format: Either "COMPLETE" or a specific question."""

*return* *await* *self*.llm\_service.generate\_response(prompt)

def **enrich\_prompt\_with\_context**(*self*) -> str:

"""Create enriched prompt with context"""

prompt = f"""Original Query: {*self*.conversation\_state['original\_query']}

Context:

{*self*.\_format\_context()}

Please provide a detailed response based on all the above information."""

*return* prompt

async def **\_get\_next\_question**(*self*, *query*: str) -> Dict:

*if* not *self*.conversation\_state.get('needed\_fields'):

*# First time - determine what we need to ask*

fields = *await* *self*.llm\_service.determine\_needed\_fields(*query*)

*self*.conversation\_state['needed\_fields'] = fields['fields']

*self*.conversation\_state['current\_field\_index'] = 0

*# Get next unanswered field*

current\_index = *self*.conversation\_state['current\_field\_index']

*if* current\_index < len(*self*.conversation\_state['needed\_fields']):

field = *self*.conversation\_state['needed\_fields'][current\_index]

*return* {

'type': 'follow\_up\_question',

'question': field['question'],

'field': field['name']

}

*return* None

*# ... rest of the BaseAgent code ...*