LangChain Coding Challenges

Interview Questions and Answers

Core Concepts

This section focuses on fundamental principles and advanced concepts that an experienced developer should master.

1. Implement a routing chain that directs queries to different specialized chains

Solution:

```
from langchain.chains.router import MultiPromptChain

destination_chains = {'math': math_chain, 'history': history_chain}
router_chain = MultiPromptChain(destination_chains=destination_chains,
                                llm_chain=router_chain)
```

2. Create a custom output parser for structured LLM responses

Implementation:

```
from langchain.output_parsers import PydanticOutputParser
from pydantic import BaseModel

class CustomResponse(BaseModel):
    answer: str
    confidence: float
parser = PydanticOutputParser(pydantic_object=CustomResponse)
```

3. Implement a retrieval-augmented generation (RAG) chain using LangChain

Solution:

```
from langchain.chains import RetrievalQA

retriever = vectorstore.as_retriever()
qa_chain = RetrievalQA.from_chain_type(llm=OpenAI(),
                                       chain_type='stuff',
                                       retriever=retriever)
```

4. Create a custom callback handler for logging LangChain operations

Example:

```
from langchain.callbacks import BaseCallbackHandler

class CustomCallback(BaseCallbackHandler):
    def on_llm_start(self, serialized, prompts, **kwargs):
        print(f'Starting LLM with prompts: {prompts}')
```

5. Implement a custom LangChain Agent that uses OpenAI for reasoning and DuckDuckGo for web searches

Solution:

```
from langchain.agents import Tool, AgentExecutor, LLMSingleActionAgent
from langchain.tools import DuckDuckGoSearchRun

search = DuckDuckGoSearchRun()
tools = [Tool(name='Search', func=search.run, description='Search the web')]
llm = OpenAI(temperature=0)
agent = LLMSingleActionAgent(llm_chain=llm, tools=tools)
```

```
AgentExecutor.from_agent_and_tools(agent=agent, tools=tools)
```

6. Create a conversation chain with memory that maintains context across multiple interactions

Implementation:

```
from langchain.memory import ConversationBufferMemory
from langchain.chains import ConversationChain

memory = ConversationBufferMemory()
chain = ConversationChain(llm=OpenAI(), memory=memory)
response = chain.run('What is the capital of France?')
```

7. Implement a custom prompt template that formats system and user messages for ChatGPT

Example:

```
from langchain.prompts import ChatPromptTemplate

template = ChatPromptTemplate.from_messages([
    ('system', 'You are a helpful assistant'),
    ('human', '{input}'),
    ('ai', '{output}')
])
```

8. Create a document loader and vectorstore for semantic search using LangChain

Solution:

```
from langchain.document_loaders import TextLoader
from langchain.vectorstores import Chroma

loader = TextLoader('data.txt')
docs = loader.load()
db = Chroma.from_documents(docs, embedding=OpenAIEmbeddings())
results = db.similarity_search('query')
```

9. Implement a sequential chain that processes text through multiple LLM calls

Implementation:

```
from langchain.chains import SimpleSequentialChain

chain1 = LLMChain(llm=llm, prompt=prompt1)
chain2 = LLMChain(llm=llm, prompt=prompt2)
sequential_chain = SimpleSequentialChain(chains=[chain1, chain2])
result = sequential_chain.run(input_text)
```

10. Create a custom LangChain tool that interacts with an external API

Example:

```
from langchain.tools import BaseTool

class CustomAPITool(BaseTool):
    name = 'custom_api'
    def _run(self, query: str) -> str:
        response = requests.get(f'https://api.example.com/?q={query}')
        return response.json()
```

Data Structures and Algorithms

Questions in this section test your understanding of how to work with and manipulate data efficiently.

1. How would you implement a custom LRU cache using LangChain's memory components?

Implementation Approach:

- Use OrderedDict for O(1) access and maintain order
- Implement size limit and eviction policy

```
from collections import OrderedDict
class LRUCache:
    def __init__(self, capacity):
        self.cache = OrderedDict()
        self.capacity = capacity
    def get(self, key):
        if key in self.cache:
            self.cache.move_to_end(key)
```

2. Explain how you would implement a rate-limiting mechanism for LangChain API calls

Rate Limiting Solution:

- Use Token Bucket algorithm
- Track request timestamps

```
class RateLimiter:
    def __init__(self, max_requests, time_window):
        self.requests = []
        self.max_requests = max_requests
        self.time_window = time_window
    def allow_request(self):
        now = time.time()
```

3. How would you implement a custom memory buffer for conversation history in LangChain?

Memory Buffer Implementation:

- Use circular buffer for fixed-size memory
- Implement FIFO eviction

```
class ConversationBuffer:
    def __init__(self, max_size):
        self.buffer = deque(maxlen=max_size)
    def add_message(self, message):
        self.buffer.append(message)
    def get_history(self):
        return list(self.buffer)
```

4. Design a custom prompt template cache with TTL implementation

Prompt Cache Design:

- Use dictionary with timestamp tracking
- Implement TTL cleanup

```
class PromptCache:
    def __init__(self, ttl_seconds):
        self.cache = {}
        self.ttl = ttl_seconds
    def get_prompt(self, key):
        if key in self.cache and time.time() - self.cache[key]['time'] < self.ttl:
            return self.cache[key]['value']
```

5. Implement a custom vector store for semantic search with LSH (Locality-Sensitive Hashing)

LSH Vector Store:

- Implement multiple hash tables
- Use random projections

```
class LSHVectorStore:
    def __init__(self, num_tables, num_hash_functions):
        self.tables = [{} for _ in range(num_tables)]
        self.hash_functions = self._generate_hash_functions(num_hash_functions)
    def add_vector(self, vector, data):
        for table_idx, table in enumerate(self.tables):
```

6. Design a custom chain that implements the Sliding Window pattern for document processing

Sliding Window Chain:

- Process documents in chunks
- Maintain overlap between windows

```
class SlidingWindowChain:
    def __init__(self, window_size, overlap):
        self.window_size = window_size
        self.overlap = overlap
    def process_document(self, text):
        chunks = self._create_chunks(text)
        return self._process_chunks(chunks)
```

7. Implement a custom agent that uses a priority queue for task scheduling

Priority Queue Agent:

- Use heapq for priority queue
- Implement task scoring

```
class PriorityAgent:
    def __init__(self):
        self.task_queue = []
        heapq.heapify(self.task_queue)
    def add_task(self, task, priority):
        heapq.heappush(self.task_queue, (priority, task))
```

8. Design a custom output parser that implements the Trie data structure

Trie Output Parser:

- Implement prefix tree structure
- Add pattern matching

```
class TrieOutputParser:
    def __init__(self):
        self.root = {}
    def add_pattern(self, pattern):
        node = self.root
        for char in pattern:
            node = node.setdefault(char, {})
```

9. Implement a custom retriever using a Skip List data structure

Skip List Retriever:

- Implement probabilistic data structure
- Multiple level links

```
class SkipListRetriever:
    def __init__(self, max_level):
        self.max_level = max_level
        self.head = self._create_node(float('-inf'), max_level)
    def insert(self, value):
        current = self.head
        path = [None] * (self.max_level + 1)
```

10. Design a custom tool that implements a Bloom Filter for caching responses

Bloom Filter Cache:

- Implement probabilistic set membership
- Multiple hash functions

```
class BloomFilterCache:
    def __init__(self, size, num_hash_functions):
        self.size = size
        self.bit_array = [0] * size
        self.num_hash_functions = num_hash_functions
    def add(self, item):
        for seed in range(self.num_hash_functions):
```

System Design

These questions evaluate your ability to think about the bigger picture, including architecture, scalability, and performance.

1. Design a scalable LangChain-based document processing pipeline that can handle millions of documents per day while maintaining high availability and fault tolerance.

Key Components and Considerations:

- **Document Ingestion Layer**: Distributed message queue (Kafka/RabbitMQ) for document intake
- **Processing Pipeline**: Multiple LangChain instances behind load balancer
- **Storage Layer**: Vector store (Pinecone/Weaviate) for embeddings
- **Caching Layer**: Redis for frequent queries and embeddings

Architecture:

- Use horizontal scaling with multiple LangChain workers
- Implement circuit breakers for LLM API calls
- Add dead letter queues for failed processing
- Use health checks and automatic failover

```
from langchain.retrievers import TimeWeightedVectorStoreRetriever
from langchain.cache import RedisCache

retriever = TimeWeightedVectorStoreRetriever(
    vectorstore=pinecone_store,
    cache=RedisCache(redis_url="redis://localhost:6379")
)
```

2. How would you design a real-time content moderation system using LangChain that can process user-generated content within milliseconds?

System Components:

- **Streaming Pipeline**: WebSocket connections for real-time content
- **Content Classification**: Pre-loaded LangChain classification chains
- **Caching Strategy**: Two-level caching (L1: Memory, L2: Redis)

Implementation:

```
from langchain.chains import create_moderation_chain
from langchain.cache import InMemoryCache

moderation_chain = create_moderation_chain()
moderation_chain.cache = InMemoryCache(max_size=1000)

def moderate_content(text):
    return moderation_chain.run(text)
```

3. Design a distributed LangChain-based question answering system that can handle concurrent requests and maintain consistency across multiple regions.

Architecture Components:

- **Load Distribution**: Geographic DNS routing with CloudFront/Cloudflare
- **Data Consistency**: Multi-region vector store replication
- **Caching Strategy**: Regional caching with eventual consistency

Implementation Example:

```
from langchain.chains import RetrievalQAWithSourcesChain
from langchain.cache import RedisSemanticCache

qa_chain = RetrievalQAWithSourcesChain.from_chain_type(
    llm=llm,
    retriever=distributed_retriever,
    return_source_documents=True
)
```

4. How would you implement a LangChain-based semantic search engine that can handle 10,000 queries per second with sub-second latency?

System Design:

- **Query Processing**: Parallel embedding generation
- **Vector Search**: Distributed HNSW index
- **Result Ranking**: Multi-stage ranking pipeline

Performance Optimizations:

- Pre-computed embeddings
- Approximate nearest neighbor search
- Query result caching

```
from langchain.embeddings import CacheBackedEmbeddings
from langchain.vectorstores import FAISS

embeddings = CacheBackedEmbeddings(
    underlying_embeddings=base_embeddings,
    document_embedding_cache=document_cache
)
```

5. Design a fault-tolerant LangChain agent system that can handle LLM API outages and maintain service availability.

Resilience Strategies:

- **Circuit Breaker Pattern**: Prevent cascading failures
- **Fallback Mechanisms**: Multiple LLM providers
- **Retry Logic**: Exponential backoff

Implementation:

```
from langchain.llms import OpenAI, Anthropic
from langchain.agents import initialize_agent

fallback_llm = Anthropic()
def get_response(prompt):
    try: return OpenAI().generate(prompt)
    except: return fallback_llm.generate(prompt)
```

6. How would you implement a LangChain-based document summarization system that can process large documents while maintaining memory efficiency?

Design Approach:

- **Chunking Strategy**: Sliding window with overlap
- **Memory Management**: Streaming processing
- **Summary Generation**: Hierarchical summarization

Implementation:

```
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.chains.summarize import map_reduce_chain
```

```
splitter = RecursiveCharacterTextSplitter(chunk_size=1000)
chunks = splitter.split_documents(document)
summary = map_reduce_chain.run(chunks)
```

7. Design a LangChain-based recommendation system that can provide personalized suggestions based on user interaction history.

System Components:

- **User Profiling**: Embedding-based user vectors
- **Recommendation Engine**: Hybrid filtering approach
- **Feedback Loop**: Real-time preference updates

Implementation:

```
from langchain.embeddings import OpenAIEmbeddings
from langchain.vectorstores import Chroma

def get_recommendations(user_id, history):
    user_embedding = create_user_embedding(history)
    return find_similar_items(user_embedding)
```

8. How would you implement a LangChain-based chatbot system that can maintain context across multiple conversations while scaling to millions of users?

Architecture Components:

- **Session Management**: Redis-based conversation storage
- **Context Window**: Sliding window with relevance scoring
- **Memory Management**: TTL-based cleanup

Implementation:

```
from langchain.memory import RedisChatMessageHistory
from langchain.chains import ConversationChain

history = RedisChatMessageHistory(session_id)
chain = ConversationChain(memory=history)
response = chain.predict(input=user_message)
```

9. Design a LangChain-based content generation system that can handle multiple content types while ensuring output quality and consistency.

System Design:

- **Content Templates**: Parameterized prompts
- **Quality Assurance**: Multi-stage validation
- **Output Formatting**: Standardized post-processing

Implementation:

```
from langchain.prompts import PromptTemplate
from langchain.chains import LLMChain

template = PromptTemplate(input_variables=["topic"])
chain = LLMChain(llm=llm, prompt=template)
output = chain.run(topic="example")
```

10. How would you implement a LangChain-based knowledge base that automatically updates and maintains consistency across multiple data sources?

System Components:

- **Data Ingestion**: Source-specific connectors
- **Consistency Checking**: Cross-reference validation
- **Update Mechanism**: Incremental updates

Implementation:

```
from langchain.docstore import InMemoryDocstore
from langchain.embeddings import OpenAIEmbeddings

def update_knowledge_base(new_data):
    embeddings = OpenAIEmbeddings()
    docstore.add_documents(process_and_embed(new_data))
```

Coding and Debugging

This section presents practical coding challenges and questions about debugging techniques.

1. How would you implement a custom LangChain tool that performs text summarization with rate limiting?

Solution:

Here's how to create a rate-limited custom tool:

```
from langchain.tools import BaseTool
from time import sleep

class RateLimitedSummarizer(BaseTool):
    name = 'text_summarizer'
    description = 'Summarizes text with rate limiting'

    def _run(self, text: str) -> str:
        sleep(1)  # Rate limit: 1 request per second
        return self.llm.summarize(text)
```

Key points:

- Inherit from BaseTool for proper integration
- Implement rate limiting using sleep()
- Handle errors gracefully

2. Write a function to chain multiple LLM calls while handling potential API failures gracefully

Solution:

```
from langchain.callbacks import CallbackManager
from typing import List

async def chain_llm_calls(prompts: List[str], max_retries=3):
    results = []
    for prompt in prompts:
        for attempt in range(max_retries):
            try:
                response = await llm.agenerate([prompt])
                results.append(response.generations[0])
                break
            except Exception as e:
                if attempt == max_retries - 1:
                    results.append(f'Failed: {str(e)}')
```

Key features:

- Async implementation for better performance
- Retry mechanism with backoff
- Error handling and logging

3. Implement a custom memory system for LangChain that stores conversation history in Redis

Solution:

```
from langchain.memory import BaseMemory
import redis
```

```
class RedisMemory(BaseMemory):
    def __init__(self, redis_url):
        self.redis = redis.from_url(redis_url)

    def save_context(self, inputs, outputs):
        key = f'chat:{inputs['conversation_id']}'
        self.redis.rpush(key, str({'input': inputs, 'output': outputs}))

    def load_memory_variables(self, inputs):
        return self.redis.lrange(f'chat:{inputs['id']}', 0, -1)
```

4. Create a custom LangChain agent that can handle both synchronous and asynchronous tools

Solution:

```
from langchain.agents import Agent
from typing import Union, Any

class HybridAgent(Agent):
    async def _execute_tool(self, tool_name: str, tool_input: Any):
        tool = self._get_tool(tool_name)
        if hasattr(tool, 'acall'):
            return await tool.acall(tool_input)
        return tool.run(tool_input)
```

Important aspects:

- Handle both sync and async execution
- Proper tool resolution
- Error handling for both modes

5. Implement a custom output parser for structured data from LLM responses

Solution:

```
from langchain.output_parsers import BaseOutputParser
import json

class StructuredDataParser(BaseOutputParser):
    def parse(self, text: str) -> dict:
        try:
            return json.loads(text)
        except json.JSONDecodeError:
            return self._extract_structured_data(text)
```

Features:

- Handles both JSON and non-JSON responses
- Fallback parsing mechanism
- Type validation

6. Write a function to implement parallel LLM calls with token usage tracking

Solution:

```
async def parallel_llm_calls(prompts, max_tokens=1000):
    tasks = []
    token_count = 0
    async with asyncio.TaskGroup() as group:
        for prompt in prompts:
            if token_count < max_tokens:
                task = group.create_task(llm.apredict(prompt))
                tasks.append(task)
                token_count += len(prompt.split())
```

Key aspects:

- Async implementation for parallelization
- Token tracking
- Resource management

7. Create a custom prompt template that includes dynamic few-shot examples

Solution:

```
from langchain.prompts import BaseChatPromptTemplate

class DynamicFewShotPrompt(BaseChatPromptTemplate):
    def format_messages(self, **kwargs):
        examples = self._get_similar_examples(kwargs['input'])
        template = self._format_few_shot_template(examples)
        return [HumanMessage(content=template.format(**kwargs))]
```

Features:

- Dynamic example selection
- Similarity-based matching
- Template composition

8. Implement a custom callback handler for monitoring and logging LangChain operations

Solution:

```
from langchain.callbacks import BaseCallbackHandler

class MonitoringCallback(BaseCallbackHandler):
    def on_llm_start(self, serialized, prompts, **kwargs):
        self.log_event('llm_start', {'prompts': prompts})

    def on_llm_end(self, response, **kwargs):
        self.log_metrics({'latency': response.latency})
```

Capabilities:

- Event logging
- Metric collection
- Performance monitoring

9. Write a function to implement streaming responses with custom processing

Solution:

```
async def stream_with_processing(prompt: str):
    async for chunk in llm.astream(prompt):
        processed_chunk = await process_chunk(chunk)
        if should_filter(processed_chunk):
            continue
        yield processed_chunk
```

Key features:

- Async streaming
- Custom chunk processing
- Filtering capability

10. Implement a custom LangChain chain for document comparison with similarity scoring

Solution:

```
from langchain.chains import Chain
from sklearn.metrics.pairwise import cosine_similarity

class DocumentComparisonChain(Chain):
    def _call(self, inputs):
        doc1_embedding = self.embed_document(inputs['doc1'])
```

```
doc2_embedding = self.embed_document(inputs['doc2'])
similarity = cosine_similarity([doc1_embedding], [doc2_embedding])[0][0]
return {'similarity_score': similarity}
```