LLM Coding Challenges

Interview Questions and Answers

Core Concepts

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

1. How would you implement efficient token streaming for real-time generation?

Token Streaming Implementation:

- Use async generators
- Implement buffering
- Handle backpressure

```
async def stream_tokens(model, prompt):
    buffer = []
    async for token in model.generate(prompt):
        buffer.append(token)
        if len(buffer) >= CHUNK_SIZE:
```

2. Explain the implementation of efficient context window management

Context Window Management:

- Implement circular buffer
- Handle context truncation
- Manage memory efficiently

```
class ContextWindow:
    def __init__(self, max_size):
        self.buffer = deque(maxlen=max_size)
    def add_tokens(self, tokens):
        self.buffer.extend(tokens)
```

3. How would you implement efficient token-wise caching for an LLM?

Key Implementation Aspects:

- Use a sliding window cache to store recent token predictions
- Implement LRU eviction policy for cache management
- Handle cache invalidation for context changes

```
class TokenCache:
    def __init__(self, capacity):
        self.cache = OrderedDict()
        self.capacity = capacity
    def get_prediction(self, tokens):
        key = tuple(tokens)
        if key in self.cache:
            self.cache.move_to_end(key)
```

4. Explain how you would implement temperature sampling in an LLM decoder?

Temperature Sampling Implementation:

- Scale logits by temperature parameter
- Apply softmax transformation
- Sample from resulting probability distribution

```
def sample_with_temperature(logits, temperature=0.8):
    scaled_logits = logits / temperature
    probs = softmax(scaled_logits)
```

```
    return np.random.choice(len(probs), p=probs)
```

5. How would you implement top-k filtering for LLM output?

Top-k Implementation Strategy:

- Sort probability distribution
- Select top k tokens
- Renormalize probabilities

```
def top_k_filtering(logits, k=40):
    top_k = np.argsort(logits)[-k:]
    filtered_logits = np.where(np.isin(range(len(logits)), top_k), logits, -float('inf'))
    return softmax(filtered_logits)
```

6. Describe how to implement nucleus (top-p) sampling for text generation

Nucleus Sampling Implementation:

- Sort probabilities in descending order
- Calculate cumulative sum
- Filter tokens below threshold

```
def nucleus_sampling(probs, p=0.9):
    sorted_probs = np.sort(probs)[::-1]
    cumsum = np.cumsum(sorted_probs)
    return np.where(cumsum <= p, probs, 0.0)
```

7. How would you implement efficient attention caching?

Attention Caching Strategy:

- Cache key/value pairs
- Implement prefix caching
- Handle cache updates efficiently

```
class AttentionCache:
    def __init__(self):
        self.key_cache = {}
        self.value_cache = {}
    def update(self, position, key, value):
        self.key_cache[position] = key
```

8. Explain how to implement efficient prompt templating with parameter validation

Prompt Template Implementation:

- Define template schema
- Validate input parameters
- Handle escape sequences

```
class PromptTemplate:
    def __init__(self, template: str, params: List[str]):
        self.template = template
        self.required_params = set(params)
    def format(self, **kwargs):
```

9. How would you implement a sliding window attention mechanism?

Sliding Window Implementation:

- Maintain fixed context window
- Implement efficient sliding
- Handle boundary conditions

```
def sliding_attention(query, key, value, window_size):
    batch_size, seq_len = query.shape[:2]
    attention = torch.zeros(batch_size, seq_len, seq_len)
```

```
    mask = create_sliding_mask(seq_len, window_size)
```

10. Describe the implementation of beam search decoding

Beam Search Implementation:

- Track top-k sequences
- Maintain cumulative scores
- Handle sequence termination

```
def beam_search(model, start_token, beam_width=5):
    sequences = [([], 0.0)]
    for _ in range(max_len):
        candidates = []
        for seq, score in sequences:
```

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 an LRU Cache with a capacity limit in Python?

Key Implementation Points:

- Use OrderedDict or combination of HashMap + Doubly Linked List
- Time complexity: O(1) for both get and put operations

```
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 sliding window maximum algorithm for a given array

Solution Approach:

- Use a deque to maintain indices of potential maximum values
- Time complexity: O(n)

```
from collections import deque
def maxSlidingWindow(nums, k):
    result = []
    window = deque()
    for i, n in enumerate(nums):
        while window and window[0] <= i - k:
            window.popleft()
```

3. How would you design an efficient algorithm to find all pairs of integers in an array that sum to a given target?

Optimal Solution:

- Use HashMap to store complements
- Single pass approach
- Time complexity: O(n)

```
def findPairs(nums, target):
    seen = {}
    pairs = []
    for num in nums:
        if target - num in seen:
            pairs.append((num, target - num))
        seen[num] = True
```

4. Implement a thread-safe producer-consumer queue with a maximum size limit

Implementation Details:

- Use threading.Lock or Queue module
- Handle synchronization

```
from threading import Lock, Condition
class BoundedQueue:
    def __init__(self, capacity):
        self.queue = []
        self.lock = Lock()
        self.not_full = Condition(self.lock)
        self.not_empty = Condition(self.lock)
```

5. Design an efficient algorithm to detect a cycle in a linked list

Floyd's Cycle Detection:

- Use fast and slow pointers
- Space complexity: O(1)

```
def hasCycle(head):
    if not head: return False
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
        if slow == fast: return True
```

6. How would you implement a consistent hashing mechanism for load balancing?

Implementation Approach:

- Use a sorted array or binary search tree
- Virtual nodes for better distribution

```
class ConsistentHashing:
    def __init__(self, nodes=None, replicas=3):
        self.ring = {}
        self.sorted_keys = []
        for node in nodes:
            self.add_node(node, replicas)
```

7. Implement a trie (prefix tree) data structure for efficient string operations

Trie Implementation:

- Node structure with children map
- Support for insert and search

```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end = False
    def insert(self, word):
        node = self
        for char in word:
            node = node.children.setdefault(char, TrieNode())
```

8. Design an algorithm to find the k most frequent elements in a stream of numbers

Solution Using Heap:

- Use min-heap of size k
- Maintain frequency map

```
from heapq import heappush, heappop
def topKFrequent(nums, k):
    freq = {}
    heap = []
    for num in nums:
        freq[num] = freq.get(num, 0) + 1
        heappush(heap, (-freq[num], num))
```

9. Implement a rate limiter using the token bucket algorithm

Token Bucket Implementation:

- Track tokens and last refill time
- Thread-safe implementation

```
class TokenBucket:
    def __init__(self, capacity, refill_rate):
        self.capacity = capacity
        self.tokens = capacity
        self.refill_rate = refill_rate
        self.last_refill = time.time()
```

10. Design an efficient algorithm for finding the longest increasing subsequence

Dynamic Programming Approach:

- Use binary search
- Time complexity: O(n log n)

```
def lengthOfLIS(nums):
    if not nums: return 0
    tails = [0] * len(nums)
    size = 0
    for num in nums:
        i = bisect_left(tails, num, 0, size)
```

System Design

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

1. Design a scalable URL shortening service like bit.ly

Key Requirements:

- Generate unique short URLs
- Redirect to original URL
- High availability
- Low latency

System Design:

- **URL Generation:** Base62 encoding or counter-based with distributed ID generator (Snowflake)
- **Storage:** NoSQL (DynamoDB/Cassandra) for URL mappings
- **Caching:** Redis/Memcached for hot URLs
- **Load Balancing:** Round-robin with consistent hashing

Sample Code (URL Generation):

```
def generate_short_url(long_url):
    url_hash = md5(long_url.encode()).hexdigest()
    return base62_encode(int(url_hash[:8], 16))
```

2. Design a real-time chat system supporting millions of users

Architecture Components:

- **WebSocket Server:** Node.js/Socket.io for real-time communication
- **Message Queue:** Kafka/RabbitMQ for async processing
- **Storage:** Cassandra for messages, Redis for presence
- **Service Discovery:** Zookeeper/Consul

Key Features:

- Message delivery guarantees
- Presence detection
- Message persistence
- Horizontal scaling

```
const ws = new WebSocket(url);
ws.on('message', async (msg) => {
  await kafka.produce('chat-messages', msg);
  redis.setex(`presence:${userId}`, 30, 'online');
});
```

3. Design a distributed task scheduler system

Core Components:

- **Job Queue:** Redis/RabbitMQ
- **Worker Nodes:** Distributed processing units
- **Coordinator:** Leader election with ZooKeeper
- **Storage:** PostgreSQL for job metadata

Features:

- Cron-like scheduling
- Retry mechanisms
- Dead letter queues
- Task prioritization

```
class TaskScheduler:
    def schedule(self, task, cron):
        job_id = generate_uuid()
        redis.zadd('scheduled_tasks', time.now(), job_id)
        postgres.save_metadata(job_id, task, cron)
```

4. Design a scalable social media feed system

System Components:

- **Feed Generation:** Fan-out on write vs read
- **Storage:** Cassandra for posts, Redis for feed cache
- **CDN:** For media content delivery
- **Load Balancing:** Consistent hashing

Optimization Techniques:

- Feed pre-computation
- Pagination with cursor-based approach
- Content ranking algorithms

```
async function generateFeed(userId) {
    const cachedFeed = await redis.get(`feed:${userId}`);
    return cachedFeed || computeFeed(userId);
}
```

5. Design a distributed rate limiter

Implementation Approaches:

- **Token Bucket Algorithm**
- **Sliding Window Counter**
- **Redis-based implementation**
- **Distributed consensus with etcd**

Key Considerations:

- Accuracy vs Performance
- Clock synchronization
- Race conditions

```
def check_rate_limit(user_id):
    key = f'ratelimit:{user_id}'
    current = redis.get(key) or 0
    return redis.set(key, current + 1, ex=3600) < LIMIT
```

6. Design a distributed caching system like Redis

Core Features:

- **Data Structures:** String, List, Hash, Set
- **Eviction Policies:** LRU, LFU
- **Persistence:** RDB and AOF
- **Cluster Management:** Master-Slave replication

Implementation Challenges:

- Consistency models
- Partition tolerance
- Network failures

```
class DistributedCache:
    def set(self, key, value, ttl=None):
```

```
        node = self.get_node(key)
        return node.set(key, value, ex=ttl)
```

7. Design a distributed search engine

Components:

- **Crawler:** URL frontier, robots.txt handling
- **Indexer:** Inverted index, document store
- **Query Engine:** Query parsing, ranking
- **Storage:** Elasticsearch/Lucene

Optimizations:

- Index sharding
- Query caching
- Document scoring

```
def search(query):
    tokens = tokenize(query)
    scores = calculate_tf_idf(tokens)
    return rank_documents(scores)
```

8. Design a distributed configuration management system

Key Features:

- **Configuration Storage:** etcd/ZooKeeper
- **Change Notification:** Watch mechanisms
- **Version Control:** Configuration history
- **Access Control:** RBAC

Implementation:

- Configuration namespaces
- Hot reload capability
- Audit logging

```
class ConfigManager:
    def watch_config(self, key):
        etcd_client.watch(key, callback=self.on_change)
        return self.get_config(key)
```

9. Design a distributed logging system

Components:

- **Log Collection:** Fluentd/Logstash
- **Transport:** Kafka/Kinesis
- **Storage:** Elasticsearch
- **Analysis:** Kibana/Grafana

Features:

- Log aggregation
- Real-time processing
- Search capabilities

```
def log_event(event):
    kafka.produce('logs', event)
    elasticsearch.index(index='logs', body=event)
```

10. Design a distributed transaction system

Key Concepts:

- **2PC/3PC Protocols**

- **Saga Pattern**
- **ACID Properties**
- **Consensus Algorithms**

Implementation Challenges:

- Partial failures
- Network partitions
- Deadlock prevention

```
class Transaction:
    async def execute(self, operations):
        coordinator = await elect_coordinator()
        return await coordinator.two_phase_commit(operations)
```

Coding and Debugging

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

1. Implement a function to flatten a nested list of integers in Python without using built-in flatten methods.

Solution:

Here's an efficient recursive implementation:

```
def flatten(lst):
    result = []
    for item in lst:
        if isinstance(item, list):
            result.extend(flatten(item))
        else:
            result.append(item)
    return result
```

Key points:

- Handles arbitrary nesting levels
- Time complexity: O(n) where n is total number of elements
- Space complexity: O(d) where d is maximum nesting depth

2. How would you implement a custom memory profiler decorator to track function memory usage?

Implementation:

```
import memory_profiler
import functools

def track_memory(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        mem_before = memory_profiler.memory_usage()[0]
        result = func(*args, **kwargs)
        mem_after = memory_profiler.memory_usage()[0]
        print(f'Memory delta: {mem_after - mem_before} MB')
        return result
    return wrapper
```

Usage considerations:

- Requires memory_profiler package
- Measures peak memory usage
- Minimal overhead for production code

3. Design a rate limiter class for an LLM API with a sliding window approach.

Solution:

```
from collections import deque
from time import time

class RateLimiter:
    def __init__(self, max_requests, window_sec):
        self.max_requests = max_requests
        self.window_sec = window_sec
```

```
        self.requests = deque()

    def allow_request(self):
        now = time()
        while self.requests and self.requests[0] <= now - self.window_sec:
            self.requests.popleft()
        if len(self.requests) < self.max_requests:
            self.requests.append(now)
            return True
        return False
```

Features:

- Sliding window implementation
- O(1) time complexity for checks
- Thread-safe design possible with locks

4. Implement a custom context manager for handling LLM API timeouts.

Implementation:

```
import signal
from contextlib import contextmanager

@contextmanager
def timeout(seconds):
    def handler(signum, frame):
        raise TimeoutError('API call timed out')

    signal.signal(signal.SIGALRM, handler)
    signal.alarm(seconds)
    try:
        yield
    finally:
        signal.alarm(0)
```

Usage:

- Handles API timeout gracefully
- Works on Unix-based systems
- Clean resource management

5. Create a caching decorator for LLM API responses with TTL support.

Solution:

```
from functools import wraps
from time import time
import json

def cache_with_ttl(ttl_seconds):
    cache = {}
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            key = json.dumps((args, kwargs))
            if key in cache:
                result, timestamp = cache[key]
                if time() - timestamp < ttl_seconds:
                    return result
            result = func(*args, **kwargs)
            cache[key] = (result, time())
            return result
        return wrapper
    return decorator
```

6. Implement a retry mechanism with exponential backoff for LLM API calls.

Implementation:

```
import time
from functools import wraps

def retry_with_backoff(max_retries=3, base_delay=1):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            for attempt in range(max_retries):
                try:
                    return func(*args, **kwargs)
                except Exception as e:
                    if attempt == max_retries - 1:
                        raise e
                    time.sleep(base_delay * (2 ** attempt))
        return wrapper
    return decorator
```

7. Design a token bucket algorithm for LLM API rate limiting.

Solution:

```
class TokenBucket:
    def __init__(self, capacity, fill_rate):
        self.capacity = capacity
        self.fill_rate = fill_rate
        self.tokens = capacity
        self.last_fill = time.time()

    def consume(self, tokens=1):
        now = time.time()
        self.tokens += (now - self.last_fill) * self.fill_rate
        self.tokens = min(self.tokens, self.capacity)
        self.last_fill = now
        if self.tokens >= tokens:
            self.tokens -= tokens
            return True
        return False
```

8. Implement a concurrent request handler for multiple LLM API calls.

Implementation:

```
import asyncio
from typing import List

async def process_llm_requests(requests: List[str]):
    async def call_api(request):
        # Simulated API call
        await asyncio.sleep(1)
        return f'Response for {request}'

    tasks = [call_api(req) for req in requests]
    results = await asyncio.gather(*tasks, return_exceptions=True)
    return results
```

Features:

- Concurrent execution
- Exception handling
- Resource management

9. Create a custom exception hierarchy for LLM API error handling.

Implementation:

```
class LLMError(Exception):
```

```
    pass

class TokenLimitError(LLMError):
    pass

class ContextLengthError(LLMError):
    pass

class APIRateLimitError(LLMError):
    pass

class ModelNotFoundError(LLMError):
    pass
```

Benefits:

- Structured error handling
- Clear error hierarchy
- Specific error types

10. Implement a prompt template validator with regex pattern matching.

Solution:

```
import re
from typing import Dict

def validate_prompt_template(template: str, variables: Dict[str, str]):
    pattern = r'\{([^}]+)\}'
    required_vars = set(re.findall(pattern, template))
    provided_vars = set(variables.keys())

    if not required_vars.issubset(provided_vars):
        missing = required_vars - provided_vars
        raise ValueError(f'Missing variables: {missing}')
```

Features:

- Variable validation
- Pattern matching
- Error reporting