DataHub Interview Preparation Guide

Alex Yang
Principal Software Engineer, Roblox

Day 1 (Coding): November 18, 2025
Day 2 (Onsite): November 20, 2025
Prepared: November 20, 2025

Contents

1 Interview Overview

1.1 Schedule Summary

- **Day 1**: Coding interview with Andrew (SharedPad, Python 3.8.10)
- **Day 2**: Onsite with Abe
  - Project deep dive (significant Roblox project)
  - Search architecture deep dive

1.2 Recruiter Contact

- **Recruiter**: Myra (DataHub)
- **Interviewer (Day 1)**: Andrew
- **Interviewer (Day 2)**: Abe

! Critical

Platform Note: SharedPad uses Python 3.8.10, not the latest version. Avoid Python 3.9+ features like `list[str]` type hints (use `List[str]` from `typing` instead) and `match/case` statements (use `if/elif` instead).

2 Day 1: Coding Interview (Andrew)

2.1 Problem Description

You will be asked to implement a **concurrent request executor system** that processes a stream of incoming requests using a pool of worker threads. This is essentially a producer-consumer pattern with a thread pool.

2.1.1 Core Requirements

1. Accept a stream of incoming requests/jobs
2. Process them using a pool of worker threads
3. Ensure thread safety across all operations
4. Handle request partitioning/routing to workers
5. Maintain correctness under concurrent execution

2.2 Follow-Up Questions (Expect These)

2.2.1 Thread Count Decisions

Q: How many threads should we use? How should we make that decision?
Answer Framework:

- **Starting point**: `os.cpu_count()` for CPU-bound, `cpu_count() * 2` for I/O-bound
- **Considerations**:
  - Workload type (CPU-bound vs I/O-bound)
  - Latency requirements (more threads = better throughput, but overhead)
  - Memory constraints (each thread has stack space)
  - External resource limits (DB connections, API rate limits)
- **Maximum**: Limited by memory, context switching overhead, OS file descriptor limits

> * Key Insight
>
> For this interview, assume I/O-bound workload (network requests, disk I/O). Start with 4-8 threads and explain you'd tune based on profiling.

2.2.2 Testing & Correctness

Q: How do we test this code? How do we ensure correctness?
Testing Strategy:

- **Unit tests**: Mock requests, verify processing logic
- **Race condition tests**: Submit many concurrent requests, verify no data corruption
- **Correctness invariants**:

  - No dropped requests (submitted count = processed count)
  - No duplicate processing (track request IDs)
  - Thread-safe state updates (atomic operations)

- **Shutdown tests**: Verify graceful termination, no orphaned threads
- **Stress tests**: Submit requests faster than processing capacity

2.2.3 Backpressure Detection

Q: How can we detect backpressure? What should we do if workers are overwhelmed?
Detection Mechanisms:

- Monitor queue depth (`queue.qsize()`)
- Track worker thread utilization/idle time
- Measure request latency (time from submit to completion)
- Calculate processing rate vs arrival rate

Response Actions:

- Use bounded queue (blocks submissions when full)
- Return backpressure signal to upstream
- Implement adaptive rate limiting
- Add more workers dynamically (if resources allow)
- Reject requests with appropriate error code

2.2.4 Worker Thread Failures

Q: How can we detect if a worker thread dies? How do we handle it?
Detection & Recovery:

- **Try/except in worker loop**: Catch exceptions, log, continue
- **Heartbeat mechanism**: Workers report liveness periodically
- **Monitoring**: Track last activity timestamp per worker
- **Recovery options**:
  - Restart dead worker thread
  - Requeue failed request to different worker
  - Send to dead letter queue for manual review
  - Alert/log for operator intervention

! Critical

Critical Python Gotcha: Exceptions in threads don't propagate to the main thread! Always wrap worker code in try/except.

2.3 Extension Problems

2.3.1 Execution Order Invariants

Q: How can we maintain an invariant that requests are always executed in order?
Solution: Request Partitioning

- Assign requests to workers based on partition key (e.g., user_id)
- Use consistent hashing: `hash(request.key) % num_workers`
- All requests with same key go to same worker thread
- Trade-off: Preserves order per partition, but reduces parallelism

2.3.2 Backpressure Reporting

Q: If we could tell upstream to slow down, when should we do that? How do we make an informed decision?
Signal Conditions:

- Queue depth > 80% capacity
- Request latency > acceptable threshold (e.g., p99 > 5s)
- Worker saturation > 90% (all workers busy)
- Queue growth rate trending upward

Decision Framework:

- Monitor queue depth over time window (avoid false alarms)
- Calculate: arrival_rate - processing_rate
- If negative trend persists for N seconds, signal backpressure
- Expose metrics endpoint for upstream monitoring

2.4 Deep Dive: Request Partitioning for Ordering

2.4.1 The Problem

Why do we need ordering?
Some requests must be processed in the order they arrive. For example:

- All requests from the same user (login → update → logout)
- Database operations on the same record (read → update → delete)
- Sequential workflow steps (step1 → step2 → step3)

Without partitioning (broken ordering):

```
Request(user="alice", action="login")   → Worker 2
Request(user="alice", action="update")  → Worker 1  ← Executes first!
Request(user="alice", action="logout")  → Worker 3

Problem: "update" might execute before "login"!
```

2.4.2 The Solution: Partition Key Routing

Core Insight: We only need order within *related* requests. Different users can process in parallel.

With partitioning (order guaranteed):

```
Request(user="alice", action="login")   → Worker 2
Request(user="alice", action="update")  → Worker 2  ← Same worker!
Request(user="alice", action="logout")  → Worker 2  ← Same worker!

Request(user="bob", action="login")     → Worker 0  ← Different user
Request(user="bob", action="update")    → Worker 0  ← Same worker!

Result:
- Alice's requests execute in order on Worker 2
- Bob's requests execute in order on Worker 0
- Alice and Bob process in parallel!
```

2.4.3 Implementation: Consistent Hashing

The Key Formula:

```
worker_id = hash(partition_key) % num_workers
```

Why this works:

- `hash("alice")` always returns the same value
- Same hash mod num_workers = same worker_id every time
- Different partition keys likely map to different workers
- Evenly distributed across workers (on average)

Example with 3 workers:

```
hash("alice") % 3 = 2  → Worker 2
hash("alice") % 3 = 2  → Worker 2 (always!)
hash("alice") % 3 = 2  → Worker 2 (always!)

hash("bob") % 3 = 0    → Worker 0
hash("bob") % 3 = 0    → Worker 0 (always!)

hash("carol") % 3 = 1  → Worker 1
```

2.4.4 Trade-offs Discussion

Benefits:

- Guarantees order within partition (same user)
- Parallel processing across partitions (different users)
- Simple implementation with consistent hashing

- Predictable routing behavior

Costs:

- **Hot partition problem**: If one partition is very active, that worker becomes bottleneck
- **Load imbalance**: Cannot redistribute work from busy to idle workers
- **More complex**: Multiple queues vs single shared queue

Hot Partition Example:

```
alice sends 1000 requests → Worker-2 (overloaded!)
bob sends 1 request       → Worker-0 (idle)
carol sends 1 request     → Worker-1 (idle)

Worker-2 is bottleneck while others sit idle!
```

* Key Insight

When asked about maintaining order:
"To maintain execution order, I'd use **request partitioning** with consistent hashing. The key insight is we only need order for related requests - like all requests from the same user - while different users can process in parallel.
I'd implement by:

1. Adding a partition_key field (e.g., user_id)
2. Using worker_id = hash(partition_key) mod num_workers to route
3. Having per-worker queues instead of a shared queue
4. Each worker processes its queue sequentially

Trade-off: This guarantees order per partition but reduces load balancing flexibility. If one user is very active (hot partition), that worker might be overloaded. For most use cases, this is acceptable since user activity is typically well-distributed."

3 Python Threading Fundamentals

3.1 Why Threading? The GIL Caveat

* Key Insight

Python's GIL (Global Interpreter Lock): Only one thread executes Python bytecode at a time.

- Threading helps: I/O-bound work (network, disk, sleep)
- Threading doesn't help: CPU-bound work (use `multiprocessing` instead)
- For your interview: Assume I/O-bound request processing

3.2 Core Concepts

3.2.1 Starting and Joining Threads

```python
import threading
import time

def worker(name, delay):
    print(f"{name} starting")
    time.sleep(delay)  # Simulate I/O work
    print(f"{name} done")

# Create threads
t1 = threading.Thread(target=worker, args=("Worker-1", 2))
t2 = threading.Thread(target=worker, args=("Worker-2", 1))

# Start threads (non-blocking)
t1.start()
t2.start()

# Wait for completion (blocking)
t1.join()
t2.join()

print("All workers finished")
```

Key Points:

- `start()`: Launches thread, returns immediately
- `join()`: Blocks until thread completes
- Without `join()`, main thread may exit while workers are running

3.2.2 Thread-Safe Shared State (Locks)

```python
import threading

```

```
counter = 0
lock = threading.Lock()

def increment():
    global counter
    for _ in range(100000):
        with lock:  # CRITICAL: prevents race conditions
            counter += 1

# Start 10 threads
threads = [threading.Thread(target=increment) for _ in range(10)]
for t in threads: t.start()
for t in threads: t.join()

print(f"Counter: {counter}")  # Should be 1,000,000
```

! Critical

Without the lock, you'll get wrong results due to race conditions! The += operation is not atomic:

1. Read counter value
2. Add 1
3. Write back

Multiple threads can interleave these steps, causing lost updates.

3.2.3 Producer-Consumer with Queue

```
import queue
import threading
import time

# Thread-safe queue
task_queue = queue.Queue(maxsize=100)  # Bounded queue

def producer():
    for i in range(20):
        task_queue.put(f"Task-{i}")
        time.sleep(0.1)
    print("Producer done")

def consumer(consumer_id):
    while True:
        try:
            task = task_queue.get(timeout=1)  # 1 second timeout
            print(f"Consumer-{consumer_id} processing {task}")
            time.sleep(0.3)  # Simulate work
            task_queue.task_done()
        except queue.Empty:
            break  # No more tasks
```

```
23
24 # Start producer
25 producer_thread = threading.Thread(target=producer)
26 producer_thread.start()
27
28 # Start 3 consumers
29 consumers = [threading.Thread(target=consumer, args=(i,))
30              for i in range(3)]
31 for c in consumers: c.start()
32
33 # Wait for completion
34 producer_thread.join()
35 task_queue.join()  # Wait until all tasks processed
36
37 for c in consumers: c.join()
38 print("All tasks completed")
```

Key Methods:

- `queue.Queue()`: Thread-safe FIFO queue
- `put(item, block=True)`: Add item (blocks if queue full)
- `get(timeout=None)`: Remove and return item (blocks if empty)
- `task_done()`: Signal task completion
- `join()`: Block until all tasks done
- `qsize()`: Approximate queue size (for monitoring)

3.2.4 Graceful Shutdown with Event

```
1  import threading
2  import queue
3  import time
4
5  class ThreadPool:
6      def __init__(self, num_workers):
7          self.task_queue = queue.Queue()
8          self.workers = []
9          self.shutdown_event = threading.Event()  # Signal flag
10
11         for i in range(num_workers):
12             worker = threading.Thread(target=self._worker, args=(i,))
13             worker.start()
14             self.workers.append(worker)
15
16     def _worker(self, worker_id):
17         while not self.shutdown_event.is_set():
18             try:
19                 task = self.task_queue.get(timeout=0.5)
20                 # Process task
21                 print(f"Worker-{worker_id}: {task}")
```

```
                self.task_queue.task_done()
            except queue.Empty:
                continue  # Check shutdown flag again

        print(f"Worker-{worker_id} shutting down")

    def submit(self, task):
        self.task_queue.put(task)

    def shutdown(self):
        print("Initiating shutdown...")
        self.shutdown_event.set()  # Signal all workers
        for worker in self.workers:
            worker.join()  # Wait for each to finish
        print("Shutdown complete")

# Usage
pool = ThreadPool(num_workers=3)
for i in range(10):
    pool.submit(f"Task-{i}")

time.sleep(2)
pool.shutdown()
```

Why Event instead of a boolean?

- `Event` is thread-safe (no race conditions)
- `set()` and `is_set()` are atomic operations
- A simple boolean would need a lock

3.3 Common Pitfalls

1. **Forgetting join()**: Threads keep running after main exits
2. **Shared state without locks**: Data corruption, race conditions
3. **Ignoring thread exceptions**: Silent failures (wrap in try/except!)
4. **Infinite blocking**: Always use timeouts on `get()`
5. **Deadlocks**: Multiple locks acquired in different order

4 Interview-Ready Implementation

4.1 Complete Request Executor Template

```python
import queue
import threading
import time
from typing import Callable, Any, Dict
from dataclasses import dataclass

@dataclass
class Request:
    """Request object with metadata"""
    request_id: int
    partition_key: str  # For ordering within partition
    payload: Any
    timestamp: float = None

    def __post_init__(self):
        if self.timestamp is None:
            self.timestamp = time.time()

class RequestExecutor:
    """
    Thread-safe request executor with backpressure detection,
    graceful shutdown, and request partitioning.
    """

    def __init__(self, num_workers: int, max_queue_size: int = 1000):
        self.num_workers = num_workers
        self.max_queue_size = max_queue_size

        # Bounded queue for backpressure
        self.request_queue = queue.Queue(maxsize=max_queue_size)

        # Worker management
        self.workers = []
        self.shutdown_event = threading.Event()

        # Metrics (thread-safe with locks)
        self.metrics_lock = threading.Lock()
        self.total_processed = 0
        self.total_failed = 0
        self.worker_stats: Dict[int, int] = {}

        # Start workers
        self._start_workers()

    def _start_workers(self):
        """Initialize and start worker threads"""
        for worker_id in range(self.num_workers):
            worker = threading.Thread(
                target=self._worker_loop,
                args=(worker_id,),
```

```
                name=f"Worker-{worker_id}"
            )
            worker.start()
            self.workers.append(worker)
            self.worker_stats[worker_id] = 0

        print(f"Started {self.num_workers} worker threads")

    def _worker_loop(self, worker_id: int):
        """Main worker thread loop"""
        print(f"Worker-{worker_id} started")

        while not self.shutdown_event.is_set():
            try:
                # Get request with timeout to check shutdown flag
                request = self.request_queue.get(timeout=0.5)

                # Process the request
                try:
                    self._process_request(worker_id, request)

                    # Update metrics
                    with self.metrics_lock:
                        self.total_processed += 1
                        self.worker_stats[worker_id] += 1

                except Exception as e:
                    # Handle processing errors
                    print(f"Worker-{worker_id} error processing "
                          f"request {request.request_id}: {e}")

                    with self.metrics_lock:
                        self.total_failed += 1

                finally:
                    self.request_queue.task_done()

            except queue.Empty:
                # No request available, loop again
                continue

        print(f"Worker-{worker_id} shutting down gracefully")

    def _process_request(self, worker_id: int, request: Request):
        """Process a single request (override in subclass)"""
        # Simulate I/O-bound work
        print(f"Worker-{worker_id} processing request "
              f"{request.request_id} (key: {request.partition_key})")
        time.sleep(0.1)  # Simulated work

    def submit(self, request: Request, block: bool = True,
               timeout: float = None):
        """
        Submit a request for processing
```

```
105
106         Args:
107             request: Request to process
108             block: If True, blocks when queue is full
109             timeout: Timeout for blocking (None = infinite)
110
111         Raises:
112             queue.Full: If queue is full and block=False
113         """
114         if self.shutdown_event.is_set():
115             raise RuntimeError("Executor is shutting down")
116
117         try:
118             self.request_queue.put(request, block=block,
119                                    timeout=timeout)
120         except queue.Full:
121             print(f"BACKPRESSURE: Queue full, rejecting request "
122                   f"{request.request_id}")
123             raise
124
125     def get_queue_depth(self) -> int:
126         """Get current queue depth (for backpressure monitoring)"""
127         return self.request_queue.qsize()
128
129     def is_backpressure(self, threshold: float = 0.8) -> bool:
130         """
131         Check if system is under backpressure
132
133         Args:
134             threshold: Queue fullness ratio (0-1) to trigger
135
136         Returns:
137             True if queue depth exceeds threshold
138         """
139         current = self.get_queue_depth()
140         return current > (self.max_queue_size * threshold)
141
142     def get_metrics(self) -> Dict[str, Any]:
143         """Get executor metrics (thread-safe)"""
144         with self.metrics_lock:
145             return {
146                 "total_processed": self.total_processed,
147                 "total_failed": self.total_failed,
148                 "queue_depth": self.get_queue_depth(),
149                 "worker_stats": self.worker_stats.copy(),
150                 "backpressure": self.is_backpressure()
151             }
152
153     def shutdown(self, wait: bool = True, timeout: float = None):
154         """
155         Gracefully shutdown executor
156
157         Args:
158             wait: If True, wait for in-flight requests to complete
```

```
            timeout: Maximum time to wait (None = infinite)
        """
        print("Initiating executor shutdown...")

        if wait:
            # Wait for queue to drain
            print("Waiting for queue to drain...")
            self.request_queue.join()

        # Signal shutdown to all workers
        self.shutdown_event.set()

        # Wait for workers to terminate
        print("Waiting for workers to terminate...")
        for worker in self.workers:
            worker.join(timeout=timeout)

        print("Executor shutdown complete")
        print(f"Final metrics: {self.get_metrics()}")

# ============================================================
# Extension: Partitioned Executor (for ordering)
# ============================================================

class PartitionedExecutor:
    """
    Executor with request partitioning for per-partition ordering.
    Requests with same partition_key go to same worker thread.
    """

    def __init__(self, num_workers: int, max_queue_size: int = 1000):
        self.num_workers = num_workers

        # One queue per worker (partition)
        self.queues = [queue.Queue(maxsize=max_queue_size // num_workers)
                       for _ in range(num_workers)]

        self.workers = []
        self.shutdown_event = threading.Event()

        self._start_workers()

    def _start_workers(self):
        for worker_id in range(self.num_workers):
            worker = threading.Thread(
                target=self._worker_loop,
                args=(worker_id,)
            )
            worker.start()
            self.workers.append(worker)

    def _worker_loop(self, worker_id: int):
        my_queue = self.queues[worker_id]

```

```
        while not self.shutdown_event.is_set():
            try:
                request = my_queue.get(timeout=0.5)
                self._process_request(worker_id, request)
                my_queue.task_done()
            except queue.Empty:
                continue
            except Exception as e:
                print(f"Worker-{worker_id} error: {e}")

    def _process_request(self, worker_id: int, request: Request):
        print(f"Worker-{worker_id} processing request "
              f"{request.request_id} (key: {request.partition_key})")
        time.sleep(0.1)

    def submit(self, request: Request):
        """Route request to worker based on partition key"""
        worker_id = hash(request.partition_key) % self.num_workers
        self.queues[worker_id].put(request)
        print(f"Request {request.request_id} routed to "
              f"Worker-{worker_id} (key: {request.partition_key})")

    def shutdown(self):
        self.shutdown_event.set()
        for worker in self.workers:
            worker.join()
        print("Partitioned executor shutdown complete")

# ============================================================
# Example Usage & Testing
# ============================================================

if __name__ == "__main__":
    print("=== Testing Basic Executor ===")
    executor = RequestExecutor(num_workers=3, max_queue_size=10)

    # Submit requests
    for i in range(15):
        try:
            req = Request(
                request_id=i,
                partition_key=f"user-{i % 3}",
                payload={"data": f"payload-{i}"}
            )
            executor.submit(req, block=False)  # Non-blocking
        except queue.Full:
            print(f"Request {i} rejected due to backpressure")

    # Monitor metrics
    time.sleep(1)
    print(f"Metrics: {executor.get_metrics()}")

    # Graceful shutdown
    executor.shutdown()
```

```
267
268     print("\n=== Testing Partitioned Executor ===")
269     partitioned = PartitionedExecutor(num_workers=3)
270
271     # Submit requests with same key (will be ordered)
272     for i in range(9):
273         req = Request(
274             request_id=i,
275             partition_key=f"user-{i % 3}",
276             payload={"data": f"payload-{i}"}
277         )
278         partitioned.submit(req)
279
280     time.sleep(2)
281     partitioned.shutdown()
```

4.2 Key Implementation Points

+ Action Item

What to Emphasize During Interview:

1. **Thread safety**: All shared state protected by locks
2. **Graceful shutdown**: Event-based signaling, wait for completion
3. **Backpressure**: Bounded queue, monitoring, rejection
4. **Error handling**: Try/except in worker loop, no silent failures
5. **Metrics**: Track processed, failed, per-worker stats
6. **Partitioning**: Consistent hashing for per-key ordering

5 3-Day Study Plan

5.1 Day 1: Foundation (3-4 hours)

5.1.1 Morning (2 hours)

1. Read Python `threading` docs: https://docs.python.org/3/library/threading.html
2. Read Python `queue` docs: https://docs.python.org/3/library/queue.html
3. Watch: Corey Schafer's Python Threading tutorial (25 min)
   https://www.youtube.com/watch?v=IEEhzQoKtQU

5.1.2 Afternoon (2 hours)

1. Code Exercise 1: Simple thread creation and joining
2. Code Exercise 2: Shared counter with Lock (test with/without lock)
3. Code Exercise 3: Producer-consumer with Queue

Success Criteria:

- Can explain why locks are needed
- Can start/join threads correctly
- Understand `queue.Queue` is thread-safe

5.2 Day 2: Thread Pool Implementation (4-5 hours)

5.2.1 Morning (2.5 hours)

1. Implement basic thread pool from scratch (no `concurrent.futures`)
2. Add graceful shutdown with `Event`
3. Add metrics tracking with locks

5.2.2 Afternoon (2 hours)

1. Add backpressure detection (queue monitoring)
2. Add exception handling in workers
3. Test with different worker counts and load

Success Criteria:

- Can implement thread pool in 30 minutes
- Can explain every line of code
- Can handle worker failures gracefully

5.3 Day 3: Polish & Extensions (4-5 hours)

5.3.1 Morning (2.5 hours)

1. Implement request partitioning for ordering
2. Add backpressure reporting logic
3. Review `concurrent.futures.ThreadPoolExecutor` source

5.3.2 Afternoon (2 hours)

1. Timed practice: Implement basic executor in 30 min
2. Practice explaining follow-up questions out loud
3. Review common pitfalls and gotchas

Success Criteria:

- Can solve problem end-to-end in 30-40 minutes
- Can confidently answer all follow-up questions
- Can discuss trade-offs clearly

5.4 Python 3.8 Compatibility Checklist

! Critical

Avoid These (Python 3.9+ features):

- Type hints: `list[str]` → Use `List[str]` from `typing`
- Type hints: `dict[str, int]` → Use `Dict[str, int]`
- `match/case` statements → Use `if/elif`
- Walrus operator in complex contexts (available in 3.8, but be careful)

Safe to Use:

- All `threading` and `queue` features
- Type hints from `typing` module
- `dataclasses` (Python 3.7+)
- `concurrent.futures`
- All patterns in this guide

6 Day 2: Onsite Interview (Abe)

6.1 Part 1: Project Deep Dive

6.1.1 Select Your Project

Choose ONE significant Roblox project from the last 2 years. Recommended options based on your background:

1. **BuilderAI / LMaaS Platform**
   - Language Model as a Service platform
   - Multi-LLM evaluation systems
   - Dynamic NPC dialogue generation
2. **Search Infrastructure Modernization**
   - Vector database implementation
   - Semantic search with embeddings
   - RAG (Retrieval-Augmented Generation) platform
3. **AI Safety Model Development**
   - Content classification with BERT
   - Community feedback systems
   - Safety model evaluation framework

+ Action Item

Preparation Task: Write a 2-page document covering your chosen project with these sections:

1. Problem statement (2-3 sentences)
2. Technical approach & architecture
3. Your specific contributions
4. Collaborators & team dynamics
5. Impact & results (quantitative if possible)
6. Challenges & lessons learned
7. What you'd do differently

6.1.2 Discussion Framework

Expect Abe to probe on:

1. **Problem & Context**
   - What was the business/technical problem?

- Why was it important? What was at stake?
- What constraints did you face (time, resources, tech debt)?

2. **Technical Architecture**
   - High-level design diagram (draw on whiteboard)
   - Key components and their interactions
   - Technology choices and trade-offs
   - Scalability considerations
3. **Your Contributions**
   - What did YOU specifically build/design/own?
   - What was most technically challenging?
   - How did you make key technical decisions?
4. **Collaboration**
   - Who did you work with? What were their roles?
   - How did you coordinate across teams?
   - How did you handle disagreements?
5. **Impact**
   - What metrics improved? (latency, accuracy, adoption, etc.)
   - How did you measure success?
   - What was the business impact?
6. **Retrospective**
   - What went well? Why?
   - What didn't go well? Why?
   - What would you do differently knowing what you know now?
   - What did you learn?

* Key Insight

Interview Tip: Use the STAR method (Situation, Task, Action, Result) for structuring answers. Be specific with numbers when possible ("reduced latency from 500ms to 50ms" beats "made it faster").

6.2 Part 2: Search Deep Dive

6.2.1 Roblox Search Architecture Overview

Prepare to discuss:

1. **High-Level Architecture**

- Indexing pipeline (data ingestion → processing → index building)
- Query processing flow
- Ranking & relevance
- Serving infrastructure

2. **Technical Components**
   - Search index technology (Elasticsearch? Custom?)
   - Vector database for semantic search
   - Query understanding (parsing, intent detection)
   - Ranking models (traditional vs ML-based)
   - Autocomplete/suggestions
3. **Scale & Performance**
   - Index size, query volume
   - Latency requirements (p50, p99)
   - Freshness requirements
   - Availability targets

6.2.2 Key Topics to Master

Indexing

- How do you build and update the search index?
- How do you handle schema changes?
- What's your strategy for freshness vs consistency?
- How do you handle large documents or assets?

Query Understanding

- How do you parse and normalize queries?
- How do you handle typos and synonyms?
- Do you do query rewriting or expansion?
- How do you detect user intent?

Ranking & Relevance

- What signals do you use for ranking? (textual relevance, popularity, personalization)
- How do you combine multiple signals?
- Do you use machine learning for ranking? What features?
- How do you evaluate ranking quality?

Semantic Search

- How do you generate embeddings? (model, training data)
- Vector database technology and configuration
- Hybrid search: combining keyword + semantic
- Trade-offs: latency, recall, precision

Personalization

- What personalization signals do you use?
- How do you balance personalization vs diversity?
- Cold start problem: new users, new content
- Privacy considerations

Evaluation & Metrics

- How do you measure search quality? (NDCG, MRR, precision@k)
- A/B testing strategy
- Offline evaluation datasets
- Human evaluation process

6.2.3 Trade-Offs Discussion

Be ready to articulate these common search trade-offs:

Dimension	Trade-Off
Latency vs Quality	Deeper ranking improves quality but increases latency
Freshness vs Consistency	Real-time indexing vs eventually consistent
Recall vs Precision	Cast wide net vs return only high-confidence results
Personalization vs Diversity	User preferences vs exploration/discovery
Keyword vs Semantic	Exact matches vs conceptual similarity
Compute Cost vs Quality	Expensive ML models vs simpler heuristics

* Key Insight

Interview Approach: Don't just describe what Roblox does. Explain WHY those decisions were made, what alternatives you considered, and what you learned. Show systems thinking by discussing upstream/downstream dependencies.

6.2.4 Potential Probing Questions

1. "Walk me through what happens when a user types a search query."
2. "How do you handle very popular queries that could overwhelm the system?"
3. "What's your strategy for ranking results for a brand new user?"
4. "How do you detect and fix search quality issues?"
5. "Describe a time when search relevance was particularly bad. What did you do?"
6. "How would you add a new ranking signal to the system?"
7. "What's your biggest challenge with search at Roblox's scale?"

6.3 General Onsite Tips

+ Action Item

Day Before Interview:

- Review your project document (know it cold)
- Practice drawing architecture diagrams
- Prepare 3-5 questions to ask Abe about DataHub
- Get good sleep

* Key Insight

During Interview:

- Start with high-level, then drill into details
- Use whiteboard/paper to draw diagrams
- Be honest about challenges and failures
- Show enthusiasm for technical problem-solving
- Ask clarifying questions if needed
- Connect your experience to DataHub's problems

7 Quick Reference Cheat Sheet

7.1 Python Threading API

Class/Method	Purpose
`threading.Thread`	Create a thread
`thread.start()`	Begin thread execution
`thread.join()`	Wait for thread to finish
`threading.Lock()`	Mutual exclusion lock
`with lock:`	Acquire and release lock (RAII)
`threading.Event()`	Thread signaling flag
`event.set()`	Set flag to True
`event.is_set()`	Check if flag is True
`queue.Queue()`	Thread-safe FIFO queue
`queue.put(item)`	Add item to queue
`queue.get(timeout)`	Remove item from queue
`queue.task_done()`	Signal task completion
`queue.join()`	Wait for all tasks done
`queue.qsize()`	Approximate queue size

7.2 Common Interview Patterns

1. Worker Thread Pattern

```
def worker_loop():
    while not shutdown_event.is_set():
        try:
            item = queue.get(timeout=0.5)
            process(item)
            queue.task_done()
        except queue.Empty:
            continue
```

2. Thread-Safe Update Pattern

```
with metrics_lock:
    self.counter += 1
```

3. Graceful Shutdown Pattern

```
def shutdown():
    shutdown_event.set()  # Signal workers
    for worker in workers:
        worker.join()  # Wait for completion
```

4. Backpressure Detection Pattern

```
if queue.qsize() > threshold:
    raise BackpressureError("System overwhelmed")
```

7.3 Key Concepts Checklist

☐ Understand the GIL and when threading helps

☐ Can implement basic thread pool from scratch

☐ Know when to use Lock vs Event vs Queue

☐ Can explain race conditions with examples

☐ Understand bounded queues for backpressure

☐ Can implement graceful shutdown

☐ Can handle exceptions in worker threads

☐ Understand request partitioning for ordering

☐ Can discuss thread count trade-offs

☐ Know how to monitor queue depth

7.4 Before Interview Checklist

+ Action Item

Final Prep (Morning of Interview):

1. Review this guide's key sections (30 min)
2. Practice drawing thread pool architecture (10 min)
3. Review your project document (20 min)
4. Practice explaining one technical decision (10 min)
5. Prepare 3 questions for interviewer (5 min)

7.5 Contact Information

- **Recruiter**: Myra (DataHub)
- **Day 1 Interviewer**: Andrew (Coding)
- **Day 2 Interviewer**: Abe (Project & Search)
- **Platform**: SharedPad (Python 3.8.10)

Good luck with your DataHub interview!
You've got this. Trust your experience and preparation.