ClickUp Backend Interview
Document Event Processing

Complete Preparation Guide - November 30, 2024 (v2)

November 30, 2025

Contents

1 Interview Overview

1.1 What to Expect

Interview Type: Backend Live Coding (60 minutes)

Format:

- CodeSignal platform (browser-based IDE)
- No pre-written test suite
- You must test your own code
- Recommended language: Python
- Focus on correctness, edge cases, and code quality

Problem Domain:

- Event processing in memory
- Document state management (similar to Google Docs)
- Real-time collaborative editing
- ClickUp's core functionality

1.2 Success Criteria

1. **Correctness**: Handle all test cases including edge cases
2. **Code Quality**: Clean, readable, well-structured code
3. **Communication**: Think out loud, explain your approach
4. **Testing**: Demonstrate testing strategy
5. **Problem-Solving**: Handle follow-up questions and variations

2 Core Problem: Document Event Processor

Problem

Context: ClickUp has a feature that allows users to create documents similar to Google Docs. Changes are tracked through batches of events.

Task: Process events on a document and return the correctly updated document.

Data Structures:
Document:

```
{
    title: string,
    content: string,          // Lines separated by \n
    lastUpdated: timestamp,
    createdOn: timestamp
}
```

Event:

```
{
    event_id: number,
    event_name: string,       // Case-insensitive: "append", "APPEND", "Append"
    payload: object,
    timestamp: timestamp
}
```

Event Types:
1. **append** - Add content to a specific line

```
payload: {
    newContent: string,
    startLine: number        // 1-indexed (line 1 = first line)
}
```

Important: If content exists at that line, **append** newContent to the end of that line (don't replace!).
2. **delete** - Delete all content from document

```
payload: {}  // Empty
```

Requirements:

- Process events in timestamp order (may arrive out-of-order)
- Update document.lastUpdated to latest event timestamp
- Handle case-insensitive event names
- Return updated document object

Important Edge Cases & Gotchas

IMPORTANT NOTE: The provided examples show lastUpdated incrementing by 1 (123456789 → 123456790), but event timestamps are in the billions (1641024000001). This appears to be an inconsistency in the problem statement.

Most logical interpretation: Set lastUpdated to the timestamp of the most recent event.

Action: Ask the interviewer to clarify this behavior before coding!

3 CRITICAL: Questions to Ask First!

Pro Tips

Before you start coding, ask these clarifying questions:

1. **lastUpdated behavior:** "Should lastUpdated be set to the timestamp of the latest event, or should it be incremented by 1?"
   - The examples show inconsistent behavior - clarify this!
   - Most logical: use the latest event's timestamp
2. **Invalid line numbers:** "What should happen if startLine is 0 or negative?"
   - Skip the event? Return error? Assume line 1?
3. **Event validation:** "Should I validate event structure and handle malformed events?"
   - Missing required fields?
   - Unknown event types?
4. **Same timestamp:** "If multiple events have the same timestamp, what determines their order?"
   - Use event_id as tiebreaker?
   - Stable sort (preserve original order)?
5. **Document mutation:** "Should I modify the input document in place or return a new copy?"
   - Safer to return a copy
   - But in-place is more efficient
6. **Content initialization:** "If document doesn't have a content field initially, should I initialize it as empty string?"
7. **Trailing newlines:** "Should the final content have a trailing newline character?"

These questions show you think critically and catch ambiguities!

4 Examples with Step-by-Step Trace

4.1 Example 1: Multiple Appends

Input:

```
events = [
    {"event_id": 1, "event_name": "append",
     "payload": {"newContent": "Line 1 ", "startLine": 1},
     "timestamp": 1641024000001},
    {"event_id": 2, "event_name": "APPEND",
     "payload": {"newContent": "Line 2 ", "startLine": 2},
     "timestamp": 1641024000002},
    {"event_id": 3, "event_name": "APPEND",
     "payload": {"newContent": "Line 3 ", "startLine": 3},
     "timestamp": 1641024000003}
]

document = {"title": "Lorem Ipsum", "lastUpdated": 123456789, "
    createdOn": 123456789}
```

Step-by-Step Execution:

```
Initial state:
  lines = []
  lastUpdated = 123456789

After Event 1 (append to line 1):
  lines = ["Line 1 "]
  lastUpdated = 1641024000001

After Event 2 (append to line 2):
  lines = ["Line 1 ", "Line 2 "]
  lastUpdated = 1641024000002

After Event 3 (append to line 3):
  lines = ["Line 1 ", "Line 2 ", "Line 3 "]
  lastUpdated = 1641024000003

Final: Join lines with "\n"
  content = "Line 1 \nLine 2 \nLine 3 "
```

Expected Output:

```
{
    "title": "Lorem Ipsum",
    "content": "Line 1 \nLine 2 \nLine 3 ",
    "lastUpdated": 1641024000003,
    "createdOn": 123456789
}
```

4.2 Example 2: Delete Event

Input:

```
events = [{"event_id": 1, "event_name": "delete", "timestamp":
    1641024000000}]
```

```

document = {
    "title": "Lorem Ipsum",
    "content": "This is Lorem ipsum",
    "lastUpdated": 123456789,
    "createdOn": 123456789
}
```

Step-by-Step:

```
Initial state:
  lines = ["This is Lorem ipsum"]

After Event 1 (delete):
  lines = []
  content = ""
  lastUpdated = 1641024000000
```

Expected Output:

```
{
    "title": "Lorem Ipsum",
    "content": "",
    "lastUpdated": 1641024000000,
    "createdOn": 123456789
}
```

5 Solution Strategy

5.1 Approach

1. **Sort events by timestamp** (handle out-of-order delivery)
2. **Initialize content** as list of lines
3. **Process each event** in order:
   - For `append`: Update/create line at specified position
   - For `delete`: Clear all content
4. **Join lines** back into string with \n
5. **Update lastUpdated** to latest event timestamp
6. **Return updated document**

5.2 Key Insights

- Use **list for lines** - O(1) indexing, easy modification
- Handle **1-indexed lines** (convert to 0-indexed: `line_idx = start_line - 1`)
- **Extend list** with empty strings if startLine exceeds current length
- Event names are **case-insensitive** (use `.lower()`)
- **Append, don't replace** - use `lines[idx] += new_content`
- Return a **copy** to avoid modifying input (safer)

5.3 Why List Over String Manipulation?

- **Strings are immutable** in Python - expensive to modify
- **List operations** are O(1) for append and indexed assignment
- **Only convert** to string once at the end
- Much more efficient for multiple operations

6 Complete Solution

Solution

Clean Production-Ready Implementation

```python
def execute(events, document):
    """
    Process document events and return updated document.

    This solution handles:
    - Out-of-order events (sorts by timestamp)
    - Case-insensitive event names
    - Line gaps (fills with empty strings)
    - Multiple appends to same line
    - Document without initial content

    Args:
        events: List of event dictionaries
        document: Document dictionary

    Returns:
        New document dictionary with processed changes
    """
    # Handle empty events - return unchanged
    if not events:
        return document

    # Sort events by timestamp (handle out-of-order delivery)
    sorted_events = sorted(events, key=lambda e: e["timestamp"])

    # Initialize content as list of lines
    content = document.get("content", "")
    lines = content.split("\n") if content else []

    # Track latest timestamp for lastUpdated
    latest_timestamp = document.get("lastUpdated", 0)

    # Process each event in chronological order
    for event in sorted_events:
        event_name = event["event_name"].lower()
        timestamp = event["timestamp"]

        if event_name == "append":
            payload = event["payload"]
            new_content = payload["newContent"]
            start_line = payload["startLine"]  # 1-indexed

            # Convert to 0-indexed
            line_idx = start_line - 1

            # Extend lines list if necessary (fill gaps with empty
                strings)
            while len(lines) <= line_idx:
                lines.append("")

            # Append to existing line content (don't replace!)
            lines[line_idx] += new_content

        elif event_name == "delete":
            # Clear all content
            lines = []

        # Update to most recent timestamp
        latest_timestamp = max(latest_timestamp, timestamp)
```

7 Robust Version with Validation

Solution

Enterprise Version with Error Handling

```python
def execute_robust(events, document):
    """
    Robust version with comprehensive validation and error handling.
    Use this if interviewer asks about production considerations.
    """
    # Validate inputs
    if not events:
        return document.copy()

    if not isinstance(events, list):
        raise ValueError("events must be a list")

    if not isinstance(document, dict):
        raise ValueError("document must be a dictionary")

    # Sort by timestamp (with fallback for missing timestamps)
    sorted_events = sorted(events, key=lambda e: e.get("timestamp", 0))

    # Initialize content
    content = document.get("content", "")
    lines = content.split("\n") if content else []

    # Remove trailing empty line if present (from content ending in \n)
    if lines and lines[-1] == "":
        lines = lines[:-1]

    latest_timestamp = document.get("lastUpdated", 0)

    # Process each event with validation
    for event in sorted_events:
        # Validate event structure
        if not isinstance(event, dict):
            continue  # Skip malformed events

        event_name = event.get("event_name", "").lower()
        timestamp = event.get("timestamp", 0)
        payload = event.get("payload", {})

        if event_name == "append":
            # Validate payload
            new_content = payload.get("newContent", "")
            start_line = payload.get("startLine", 1)

            # Validate line number (must be positive)
            if start_line < 1:
                continue  # Skip invalid line numbers

            line_idx = start_line - 1

            # Extend lines if needed
            while len(lines) <= line_idx:
                lines.append("")

            # Append content
            lines[line_idx] += new_content
```

8 Critical Edge Cases

Important Edge Cases & Gotchas

YOU MUST HANDLE THESE:

1. **Empty Events List**

```
events = []
# Should return document unchanged
```

2. **Out-of-Order Events** - CRITICAL!

```
events = [
    {"event_id": 2, ..., "timestamp": 102},  # Second
    {"event_id": 1, ..., "timestamp": 101}   # First
]
# Must sort by timestamp before processing!
```

3. **Case-Insensitive Event Names**

```
"append", "APPEND", "Append", "aPpEnD"  # All valid
# Use event_name.lower() for comparison
```

4. **Multiple Appends to Same Line** - MUST APPEND, NOT REPLACE!

```
events = [
    {"payload": {"newContent": "Hello ", "startLine": 1},
        ...},
    {"payload": {"newContent": "World", "startLine": 1}, ...}
]
# Result: lines[0] = "Hello World" NOT "World"
```

5. **Gap in Line Numbers**

```
events = [{"payload": {"newContent": "Line5", "startLine": 5},
    ...}]
# Result: lines = ["", "", "", "", "Line5"]
# Fill gaps with empty strings!
```

6. **Delete Then Append**

```
events = [
    {"event_name": "delete", "timestamp": 100},
    {"event_name": "append", "payload": {..., "startLine": 1},
        "timestamp": 101}
]
# Delete clears lines = [], then append starts fresh
```

7. **Document Without Content Field**

```
document = {"title": "Test", "lastUpdated": 0, "createdOn": 0}
# No "content" key initially
# Use document.get("content", "") to handle safely
```

8. **Empty Payload or Missing Fields**

```
{"event_name": "delete", "timestamp": 100}
# Delete has no payload - that's valid
# Use payload.get("key", default) for safety
```

9. **Line Number 0 or Negative**

9 Comprehensive Test Suite

Test Cases

```
def test_document_processor():
    """Complete test suite covering all edge cases."""

    print("Running comprehensive test suite...")

    # Test 1: Basic append to empty document
    print("\n[Test 1] Basic append")
    events = [{
        "event_id": 1,
        "event_name": "append",
        "payload": {"newContent": "Hello", "startLine": 1},
        "timestamp": 100
    }]
    doc = {"title": "Test", "lastUpdated": 0, "createdOn": 0}
    result = execute(events, doc)
    assert result["content"] == "Hello", f"Expected 'Hello', got
       '{result['content']}'"
    assert result["lastUpdated"] == 100, f"Expected 100, got {
       result['lastUpdated']}"
    print("  PASSED")

    # Test 2: Multiple appends to different lines
    print("\n[Test 2] Multiple lines")
    events = [
        {"event_id": 1, "event_name": "append",
         "payload": {"newContent": "Line1", "startLine": 1},
         "timestamp": 100},
        {"event_id": 2, "event_name": "append",
         "payload": {"newContent": "Line2", "startLine": 2},
         "timestamp": 101}
    ]
    doc = {"title": "Test", "lastUpdated": 0, "createdOn": 0}
    result = execute(events, doc)
    assert result["content"] == "Line1\nLine2"
    print("  PASSED")

    # Test 3: Multiple appends to SAME line (critical!)
    print("\n[Test 3] Same line appends (CRITICAL)")
    events = [
        {"event_id": 1, "event_name": "append",
         "payload": {"newContent": "Hello ", "startLine": 1},
         "timestamp": 100},
        {"event_id": 2, "event_name": "append",
         "payload": {"newContent": "World", "startLine": 1},
         "timestamp": 101}
    ]
    doc = {"title": "Test", "lastUpdated": 0, "createdOn": 0}
    result = execute(events, doc)
    assert result["content"] == "Hello World", \
        f"Must APPEND not replace! Got: '{result['content']}'"
    print("  PASSED")

    # Test 4: Delete event
    print("\n[Test 4] Delete clears all content")
    events = [{"event_id": 1, "event_name": "delete", "timestamp":
        100}]
    doc = {"title": "Test", "content": "Some content",
           "lastUpdated": 0, "createdOn": 0}
    result = execute(events, doc)
    assert result["content"] == ""
```

10 Debugging Strategy

Pro Tips

If Your Tests Are Failing:

1. **Add Debug Prints**

```
for event in sorted_events:
    print(f"Processing: {event['event_name']} at line {event.
        get('payload', {}).get('startLine')}")
    print(f"Lines before: {lines}")
    # ... process event ...
    print(f"Lines after: {lines}")
    print()
```

2. **Check Event Sorting**

```
print("Events before sorting:")
for e in events:
    print(f"  {e['event_id']}: timestamp={e['timestamp']}")

sorted_events = sorted(events, key=lambda e: e["timestamp"])

print("\nEvents after sorting:")
for e in sorted_events:
    print(f"  {e['event_id']}: timestamp={e['timestamp']}")
```

3. **Verify Line Indexing**

```
print(f"startLine={start_line} (1-indexed)")
print(f"line_idx={line_idx} (0-indexed)")
print(f"lines length before: {len(lines)}")
# ... extend lines ...
print(f"lines length after: {len(lines)}")
```

4. **Check Append vs Replace**

```
print(f"Line {line_idx} before: '{lines[line_idx]}'")
lines[line_idx] += new_content  # Should use +=, not =
print(f"Line {line_idx} after: '{lines[line_idx]}'")
```

5. **Verify Final Join**

```
print(f"Lines array: {lines}")
content = "\n".join(lines)
print(f"Joined content: '{content}'")
print(f"Content length: {len(content)}")
```

Common Mistakes:

- Forgetting to sort events by timestamp
- Using = instead of += for append
- Off-by-one error with 1-indexed vs 0-indexed
- Not handling case-insensitive event names
- Creating extra empty lines with improper split/join

11 Follow-Up Questions & Variations

11.1 Expected Follow-Ups

1. **What if events can arrive significantly out of order?**
   - Current solution handles this with sorting
   - For streaming: use priority queue or buffering window
   - Trade-off: latency vs correctness
2. **How would you handle millions of events?**
   - Batch processing
   - Periodic snapshots + incremental updates
   - Event sourcing pattern
   - Compaction/aggregation of old events
3. **What about concurrent editing by multiple users?**
   - Operational Transform (OT)
   - Conflict-free Replicated Data Types (CRDTs)
   - Last-write-wins with timestamps
   - Use event_id as tiebreaker
4. **How would you optimize for very large documents?**
   - Rope data structure instead of list
   - Lazy loading of content
   - Chunk-based storage
   - Only process visible viewport
5. **What if we add more event types?**
   - `insert`: Insert at character position within line
   - `delete_range`: Delete specific line range
   - `replace`: Replace content at line
   - `format`: Apply formatting (bold, italic)

11.2 Variation 1: Insert at Character Position

New Event Type:

```
{
    "event_name": "insert",
    "payload": {
        "newContent": "text",
        "startLine": 2,
        "position": 5  # Character position in line (0-indexed)
    },
    "timestamp": 100
}
```

Implementation:

```
elif event_name == "insert":
    new_content = payload["newContent"]
    start_line = payload["startLine"]
    position = payload["position"]

    line_idx = start_line - 1
    while len(lines) <= line_idx:
        lines.append("")

    line = lines[line_idx]
    # Insert at character position
    lines[line_idx] = line[:position] + new_content + line[position:]
```

11.3 Variation 2: Delete Specific Lines

New Event Type:

```
{
    "event_name": "delete_range",
    "payload": {
        "startLine": 2,
        "endLine": 4    # Inclusive
    },
    "timestamp": 100
}
```

Implementation:

```
elif event_name == "delete_range":
    start_line = payload["startLine"]
    end_line = payload["endLine"]

    start_idx = start_line - 1
    end_idx = end_line - 1

    # Delete lines in range
    if start_idx < len(lines):
        del lines[start_idx:min(end_idx + 1, len(lines))]
```

11.4 Variation 3: Replace Line Content

New Event Type:

```
{
    "event_name": "replace",
    "payload": {
        "newContent": "completely new text",
        "startLine": 3
    },
    "timestamp": 100
}
```

Implementation:

```
elif event_name == "replace":
    new_content = payload["newContent"]
    start_line = payload["startLine"]

```

```
    line_idx = start_line - 1
    while len(lines) <= line_idx:
        lines.append("")

    # Replace entire line (use = instead of +=)
    lines[line_idx] = new_content
```

12 Alternative Implementations

12.1 Object-Oriented Approach (Python)

```python
class DocumentProcessor:
    """OOP approach for document event processing."""

    def __init__(self, document):
        self.document = document.copy()
        self.lines = self._init_lines()

    def _init_lines(self):
        """Initialize lines from document content."""
        content = self.document.get("content", "")
        return content.split("\n") if content else []

    def process_events(self, events):
        """Process all events and return updated document."""
        if not events:
            return self.document

        sorted_events = sorted(events, key=lambda e: e["timestamp"])

        for event in sorted_events:
            self._process_event(event)

        return self._finalize()

    def _process_event(self, event):
        """Process single event based on type."""
        event_name = event["event_name"].lower()

        if event_name == "append":
            self._handle_append(event)
        elif event_name == "delete":
            self._handle_delete(event)
        # Easy to add more event types here

    def _handle_append(self, event):
        """Handle append event."""
        payload = event["payload"]
        new_content = payload["newContent"]
        start_line = payload["startLine"]

        line_idx = start_line - 1

        # Extend lines if needed
        while len(self.lines) <= line_idx:
            self.lines.append("")

        self.lines[line_idx] += new_content
        self.document["lastUpdated"] = event["timestamp"]

    def _handle_delete(self, event):
        """Handle delete event."""
        self.lines = []
        self.document["lastUpdated"] = event["timestamp"]

```

```
    def _finalize(self):
        """Finalize and return document."""
        self.document["content"] = "\n".join(self.lines)
        return self.document


# Usage
def execute(events, document):
    processor = DocumentProcessor(document)
    return processor.process_events(events)
```

13 System Design Discussion

13.1 Real-World ClickUp Architecture

In production, ClickUp likely uses:

1. **Event Store / Event Sourcing**
    - Kafka or similar for event streaming
    - Permanent event log (source of truth)
    - Can replay events to rebuild state
    - Enables time-travel debugging
2. **CRDT (Conflict-free Replicated Data Types)**
    - Handle concurrent edits from multiple users
    - Eventual consistency without conflicts
    - Examples: Yjs, Automerge
    - Used by: Figma, Notion, Google Docs
3. **Operational Transform (OT)**
    - Transform conflicting operations
    - Maintain causal ordering
    - More complex but deterministic
    - Used by: Google Docs originally
4. **Snapshot + Delta Pattern**
    - Store periodic snapshots
    - Apply only recent events
    - Faster recovery and queries
    - Reduce memory usage
5. **WebSocket for Real-time Sync**
    - Push events to all connected clients
    - Low latency updates
    - Handle reconnection gracefully

13.2 Scalability Challenges

- **Large Documents**: Millions of characters, thousands of lines
- **High Event Rate**: Hundreds of events per second per document
- **Many Concurrent Users**: 10+ people editing simultaneously
- **Offline Support**: Sync when reconnected, handle conflicts
- **Undo/Redo**: Maintain operation history efficiently
- **Performance**: Sub-second response time even for large docs

13.3 Data Structures for Large Documents

- **Rope**: Tree-based string for efficient insertions/deletions
- **Gap Buffer**: Used by Emacs, good for cursor-based editing
- **Piece Table**: Used by VS Code, great for undo/redo
- **CRDT Text**: Yjs uses linked list with tombstones

14 Interview Execution Strategy

14.1 Time Allocation (60 minutes)

- **5 min**: Clarify requirements, ask questions, confirm understanding
- **3 min**: Discuss approach, mention data structures, complexity
- **2 min**: Write function signature and comments
- **25 min**: Implement core solution (clean, working code)
- **10 min**: Write and run tests (catch bugs early!)
- **5 min**: Add error handling and edge cases
- **10 min**: Follow-up questions, optimizations, discussion

14.2 Before You Code

Pro Tips

The First 5 Minutes Are Critical!

1. **Clarify Requirements**
   - "Should lastUpdated use the event timestamp or increment?"
   - "What happens with invalid line numbers?"
   - "Should I validate event structure?"
2. **Confirm Examples**
   - Walk through Example 1 verbally
   - Confirm expected output matches your understanding
   - Note any inconsistencies in examples
3. **Discuss Approach**
   - "I'll sort events by timestamp first..."
   - "I'll use a list for lines for O(1) indexing..."
   - "Time complexity will be O(n log n) for sorting..."
4. **Mention Edge Cases**
   - "I'll need to handle out-of-order events..."
   - "Case-insensitive event names..."
   - "Multiple appends to same line..."

This shows you're thinking critically and builds trust!

14.3 While Coding

1. **Think Out Loud**
   - "Now I'll sort the events by timestamp..."

- "Converting to 0-indexed here because Python lists..."
- "Using += here to append, not replace..."

2. **Write Clean Code**
   - Descriptive variable names: `line_idx`, not `i`
   - Add comments for non-obvious logic
   - Consistent spacing and formatting
3. **Handle Errors Gracefully**
   - Use `.get()` for optional dictionary keys
   - Check for None/empty before processing
   - Validate inputs if time permits
4. **Ask If Stuck**
   - "I'm debating between X and Y, which would you prefer?"
   - "Should I prioritize robustness or simplicity here?"
   - Don't sit in silence - communicate!

14.4 After Coding

1. **Test Immediately**
   - Run provided examples first
   - Test edge cases (empty, out-of-order, same line)
   - Fix any bugs found
2. **Walk Through Code**
   - Explain your solution at high level
   - Point out key design decisions
   - Mention trade-offs considered
3. **Discuss Improvements**
   - "For production, I'd add validation..."
   - "Could optimize with rope data structure..."
   - "Would need CRDT for real-time collaboration..."
4. **Handle Follow-Ups**
   - Be ready for variations (insert, delete range)
   - Explain how to extend solution
   - Discuss system design implications

15 Communication Checklist

15.1 Before Interview

- ☐ Practiced core solution multiple times (can code in 20-25 min)
- ☐ Memorized critical edge cases
- ☐ Tested with all provided examples
- ☐ Understand time/space complexity
- ☐ Reviewed follow-up variations
- ☐ Comfortable with Python
- ☐ Practiced explaining thought process out loud
- ☐ Prepared clarifying questions to ask

15.2 During Interview - Execution

- ☐ Asked clarifying questions upfront
- ☐ Confirmed understanding of examples
- ☐ Explained approach before coding
- ☐ Thought out loud while implementing
- ☐ Handled edge cases explicitly
- ☐ Wrote clean, readable code
- ☐ Tested code with examples
- ☐ Discussed complexity analysis
- ☐ Proposed optimizations
- ☐ Asked intelligent follow-up questions

15.3 During Interview - Communication

- ☐ Maintained conversational tone
- ☐ Explained reasoning for decisions
- ☐ Asked for feedback/hints when stuck
- ☐ Admitted when unsure (don't fake it)
- ☐ Showed enthusiasm and engagement
- ☐ Treated interviewer as collaborator

16 Quick Reference Card

16.1 Key Points to Remember

1. **ALWAYS sort events by timestamp first!**
2. **Use list for lines** (not string manipulation)
3. **Lines are 1-indexed** in problem, 0-indexed in Python
4. **Case-insensitive event names** - use `.lower()`
5. **APPEND to line** with `+=`, DON'T REPLACE with `=`
6. **Fill gaps** with empty strings when extending
7. **Update lastUpdated** to latest event timestamp
8. **Return a copy** - don't mutate input document

16.2 Common Mistakes to Avoid

- X Forgetting to sort events
- X Not handling case-insensitive event names
- X Using `=` instead of `+=` for append (replaces instead of appends!)
- X Off-by-one errors with 1-indexed vs 0-indexed
- X Not handling empty events or missing content
- X Modifying input document directly
- X Not testing edge cases

16.3 Python Quick Reference

```
# Safe dictionary access
content = document.get("content", "")  # Default empty string
payload = event.get("payload", {})     # Default empty dict

# Case-insensitive comparison
event_name = event["event_name"].lower()

# List operations
lines = []
lines.append("new")            # Add to end
lines[idx] += "text"           # Append to existing
while len(lines) <= idx:       # Extend with gaps
    lines.append("")

# String operations
lines = content.split("\n")    # Split into list
content = "\n".join(lines)     # Join back to string

# Sorting
sorted_events = sorted(events, key=lambda e: e["timestamp"])

# List comprehension
valid = [e for e in events if e.get("timestamp") is not None]
```

17 Final Preparation Checklist

17.1 Technical Readiness

- ☐ Can implement solution in under 25 minutes
- ☐ All 12+ test cases pass without bugs
- ☐ Understand why each edge case matters
- ☐ Can explain time/space complexity
- ☐ Know how to extend for new event types
- ☐ Comfortable with alternative approaches (OOP, JS)

17.2 Interview Skills

- ☐ Practiced asking clarifying questions
- ☐ Can explain approach before coding
- ☐ Comfortable thinking out loud
- ☐ Know how to debug when tests fail
- ☐ Can discuss system design implications
- ☐ Ready for follow-up variations

17.3 Day Before Interview

- ☐ Do one final practice run (timed, 30 min)
- ☐ Review edge cases one more time
- ☐ Read through this guide's key sections
- ☐ Prepare 2-3 questions to ask interviewer
- ☐ Get good sleep - fresh mind is critical!

17.4 Interview Day

- ☐ Test internet connection and mic/camera
- ☐ Have this guide open for reference (if allowed)
- ☐ Water nearby, comfortable environment
- ☐ Positive mindset - you've got this!

18 PART 2: Search Architecture Deep Dive Prep

18.1 Interview Format

Session: Search Architecture Deep Dive (60 minutes)

Focus Areas:

- Multi-tenanted Search (ClickUp has workspaces - critical!)
- Search Ranking (BM25, field boosting, relevance)
- ElasticSearch/OpenSearch architecture
- Vector Search (semantic search for tasks/docs)
- Query DSL and practical search scenarios

Your Knowledge Level:

- Strong: Vector search concepts, high-level architecture, autocomplete
- Need refresh: BM25 details, field boosting, relevance scoring
- Gap: Multi-tenancy strategies, cross-tenant data leaks, Query DSL

19 Priority 1: Multi-Tenanted Search (MUST KNOW)

19.1 The Problem

ClickUp has multiple workspaces (tenants). When User A in Workspace 1 searches "project update", they should ONLY see results from their workspace, never from Workspace 2.

Critical Requirements:

- **Data Isolation**: Tenant A can't see Tenant B's data
- **Performance**: Don't slow down search for isolation
- **Scalability**: Support millions of tenants
- **Cost**: Balance resources vs isolation

19.2 Strategy 1: Index-Per-Tenant

Concept: Each tenant gets their own dedicated index.

```
workspace_123_tasks  # Workspace 123's task index
workspace_456_tasks  # Workspace 456's task index
workspace_789_tasks  # Workspace 789's task index
```

Pros:

- Perfect isolation (impossible to leak data)
- Easy to delete tenant data (drop the index)
- Can tune per-tenant settings
- Clear billing/resource tracking

Cons:

- ES has limits (1000s of indices max)
- Overhead per index (shards, mappings)
- Can't search across tenants (admin features)
- Cluster management complexity

When to use:

- Small number of large tenants (B2B SaaS)
- Strict compliance requirements
- Tenants need custom settings

19.3 Strategy 2: Shared Index with Filtering

Concept: All tenants share one index, filter by tenant_id field.

```
tasks_index:
{
  "task_id": "t123",
  "workspace_id": "ws_123",  # <-- CRITICAL field
  "title": "Project update",
  "description": "..."
}
```

Query Pattern:

```
{
  "query": {
    "bool": {
      "must": [
        {"match": {"title": "project update"}}
      ],
      "filter": [
        {"term": {"workspace_id": "ws_123"}}  // ALWAYS filter!
      ]
    }
  }
}
```

Pros:

- Scales to millions of tenants
- Lower overhead (fewer indices)
- Can search across tenants (admin)
- Easier cluster management

Cons:

- Data leak risk if filter missed
- Noisy neighbor problem
- Can't delete tenant data easily
- Query complexity increases

When to use:

- Many small-medium tenants (ClickUp likely uses this)
- Need cross-tenant analytics
- Cost optimization critical

19.4 Strategy 3: Hybrid (Index Pools)

Concept: Group tenants into pools, multiple tenants per index.

```
pool_1_tasks   # Tenants 1-1000
pool_2_tasks   # Tenants 1001-2000
pool_3_tasks   # Tenants 2001-3000
```

Balance between isolation and scalability.

19.5 Preventing Cross-Tenant Data Leaks

Important Edge Cases & Gotchas

CRITICAL: How to Prevent Leaks

1. **ALWAYS Filter at Query Time**

```
// WRONG - can leak data!
{"query": {"match": {"title": "update"}}}

// CORRECT - always filter
{
  "query": {
    "bool": {
      "must": [{"match": {"title": "update"}}],
      "filter": [{"term": {"workspace_id": "ws_123"}}]
    }
  }
}
```

2. **Middleware Enforcement**

```
class SearchService:
    def search(self, user, query):
        # ALWAYS inject workspace_id from auth context
        workspace_id = user.workspace_id

        # Force filter into query
        query["query"]["bool"]["filter"].append({
            "term": {"workspace_id": workspace_id}
        })

        return es.search(query)
```

3. **Index-Time Validation**
   - Verify workspace_id exists in every document
   - Reject documents without workspace_id
   - Validate workspace_id format
4. **Security Audits**
   - Log all search queries
   - Alert on queries without tenant filter
   - Periodic security reviews
   - Test with penetration testing
5. **Query Templates**
   - Use parameterized query templates
   - Never build queries with string concatenation
   - workspace_id as required parameter

19.6 ClickUp-Specific Multi-Tenancy

ClickUp's Architecture (likely):

- Shared index with workspace_id filtering
- Tasks, docs, comments in same index (tagged by type)
- High-traffic workspaces might get dedicated indices
- Filter: `workspace_id AND user_permissions`

Interview Question: "How would you design search for ClickUp with 1M+ workspaces?"
Your Answer:

1. Start with shared index (scales to millions)
2. Every document: workspace_id (indexed, not analyzed)
3. Middleware ALWAYS injects workspace filter
4. Large workspaces (¿10K users) → dedicated indices
5. Route queries to correct index pool
6. Security: audit logs, query templates, validation

19.7 Performance Optimizations (HIGH IMPACT)

1. Routing by workspace_id
Problem: Query searches ALL shards even if workspace data is on one shard.
Solution: Route documents and queries by workspace_id

```
# Index with routing
PUT /tasks/_doc/task_123?routing=ws_456

# Query with routing (searches only relevant shard!)
GET /tasks/_search?routing=ws_456

# Benefit: If workspace has 1M docs across 10 shards,
# query searches 1 shard instead of all 10!
# 10x performance improvement!
```

2. Hierarchical Filtering (ClickUp Realistic)
Beyond just workspace_id, ClickUp has complex permissions:

```
# Document structure
{
  "task_id": "t123",
  "workspace_id": "ws_456",
  "accessible_by_user_ids": ["user_1", "user_2", "user_3"],
  "accessible_by_team_ids": ["team_10", "team_15"],
  "is_public": false
}

# Query: workspace AND (user OR team OR public)
{
  "query": {
    "bool": {
      "must": [{"match": {"title": "project"}}],
```

```
      "filter": [
        {"term": {"workspace_id": "ws_456"}},
        {
          "bool": {
            "should": [
              {"term": {"accessible_by_user_ids": "user_1"}},
              {"terms": {"accessible_by_team_ids": ["team_10", "team_15
                  "]}},
              {"term": {"is_public": true}}
            ],
            "minimum_should_match": 1
          }
        }
      ]
    }
  }
}
```

3. Caching Strategy

- **Query Cache**: ES caches entire query results
  - Best for: Repeated identical queries
  - Example: "status:open" query cached per workspace
- **Filter Cache**: Caches filter bit sets
  - `workspace_id` filters cached (high reuse!)
  - `status`, `priority` filters cached
  - Filters in "filter" context are cached (not "must")
- **Request Cache**: Caches aggregations
  - Dashboard: "How many tasks per status?"
  - Cached at shard level

4. Rate Limiting & Noisy Neighbors

```python
# Prevent one workspace from overwhelming cluster
class SearchService:
    def search(self, user, query):
        workspace_id = user.workspace_id

        # Rate limit per workspace (e.g., 100 QPS)
        if not rate_limiter.allow(workspace_id, max_qps=100):
            raise RateLimitError("Too many requests")

        # Inject mandatory filter
        query = inject_workspace_filter(query, workspace_id)
        return es.search(query)
```

5. Tier-Based Architecture

```
Small workspaces (< 10K docs):
  → shared_index_pool_1 (10K workspaces)

Medium workspaces (10K-100K docs):
```

```
  → shared_index_pool_2 (1K workspaces)

Large enterprise (> 100K docs):
  → dedicated_index_enterprise_A
  → dedicated_index_enterprise_B

Benefits:
- Large customers get guaranteed performance
- Small customers share resources efficiently
- Can migrate between tiers as workspace grows
```

Interview Impact: Mentioning routing + caching shows deep understanding!

20 Priority 2: Relevance Scoring & Ranking

20.1 BM25 Algorithm (ElasticSearch Default)

BM25 Formula (you should know this):

```
score(D,Q) =  IDF(q_i) × [f(q_i,D) × (k1 + 1)] /
                          [f(q_i,D) + k1 × (1 - b + b × |D|/avgdl)]

Where:
- D = document
- Q = query
- q_i = query term i
- f(q_i,D) = frequency of term q_i in document D
- |D| = length of document D (in words)
- avgdl = average document length in collection
- k1 = term frequency saturation parameter (default: 1.2)
- b = length normalization parameter (default: 0.75)
- IDF(q_i) = inverse document frequency of term q_i
```

Key Concepts:

1. **TF (Term Frequency)**: More occurrences = higher score
   - But with diminishing returns (saturation)
   - "project project project" isn't 3x better than "project"
2. **IDF (Inverse Document Frequency)**: Rare terms matter more
   - "the" appears everywhere → low IDF → low weight
   - "kubernetes" is rare → high IDF → high weight
3. **Document Length Normalization**: Longer docs penalized
   - Short doc with "project" beats long doc with same term
   - Parameter b controls strength (0=off, 1=full)
4. **Saturation (k1)**: Diminishing returns on term frequency
   - 2nd occurrence matters less than 1st
   - 10th occurrence barely matters

BM25 vs TF-IDF:

- TF-IDF: Linear relationship (2x frequency = 2x score)
- BM25: Logarithmic (2x frequency ≈ 1.2x score)
- BM25 handles document length better
- BM25 is more robust to keyword stuffing

20.2 Field Boosting

Problem: Title matches should rank higher than body matches.

ElasticSearch Syntax:

```
{
  "query": {
    "multi_match": {
      "query": "project update",
      "fields": [
        "title^3",         // 3x boost
        "description^1",   // 1x (normal)
        "comments^0.5"     // 0.5x (less important)
      ]
    }
  }
}
```

How Boosting Works:

- Base score calculated per field
- Final score = title_score × 3 + desc_score × 1 + comments_score × 0.5
- Title match contributes 3x more than description

ClickUp Example:

```
{
  "query": {
    "multi_match": {
      "query": "project update",
      "fields": [
        "task_name^5",      // Task name most important
        "task_description^2",
        "comments^1",
        "attachments.filename^1.5"
      ]
    }
  }
}
```

20.3 Custom Scoring

Beyond BM25: Incorporate business logic.

Function Score Query:

```
{
  "query": {
    "function_score": {
      "query": {"match": {"title": "project"}},
      "functions": [
        {
          "filter": {"term": {"priority": "high"}},
          "weight": 2  // Boost high priority tasks
        },
        {
          "gauss": {  // Decay based on date
            "created_date": {
```

```
                "origin": "now",
                "scale": "30d",
                "decay": 0.5
              }
            }
          },
          {
            "field_value_factor": {
              "field": "likes_count",  // Popular tasks rank higher
              "modifier": "log1p"
            }
          }
        ],
        "boost_mode": "multiply"
      }
    }
}
```

ClickUp Ranking Factors:

1. Text relevance (BM25)
2. Task priority (high/medium/low)
3. Recency (newer tasks slightly boosted)
4. User activity (tasks user interacted with)
5. Completion status (open ¿ completed)
6. Assignee (user's own tasks boosted)

20.4 Interview Question: Rank "project update"

Scenario: User searches "project update" in ClickUp.

Your Answer:

1. **Base Score**: BM25 on task name, description
2. **Field Boosts**: task_name^5, $description^2$**Priority** : $Highprioritytasks + 50\%boost$
3. **Recency**: Exponential decay over 90 days
4. **Status**: Open tasks +30%, completed -20%
5. **Personalization**: User's assigned tasks +40%
6. **Workspace Activity**: Recently viewed/edited +20%

```
{
  "query": {
    "function_score": {
      "query": {
        "bool": {
          "must": [{
            "multi_match": {
              "query": "project update",
              "fields": ["task_name^5", "description^2"]
            }
```

```
            }],
            "filter": [{"term": {"workspace_id": "ws_123"}}]
          }
        },
        "functions": [
          {"filter": {"term": {"priority": "high"}}, "weight": 1.5},
          {"filter": {"term": {"status": "open"}}, "weight": 1.3},
          {"filter": {"term": {"assignee_id": "user_abc"}}, "weight":
              1.4},
          {"gauss": {"updated_at": {"origin": "now", "scale": "90d"}}}
        ]
      }
    }
}
```

21 Priority 3: Query DSL Essentials

21.1 Core Query Types

1. Match Query (Full-text search)

```
{"query": {"match": {"title": "project update"}}}
// Analyzes text, finds "project" OR "update"
```

2. Term Query (Exact match, no analysis)

```
{"query": {"term": {"status": "open"}}}
// Exact match, case-sensitive, used for IDs, enums
```

3. Bool Query (Combine multiple conditions)

```
{
  "query": {
    "bool": {
      "must": [{"match": {"title": "project"}}],      // AND
      "should": [{"match": {"tags": "urgent"}}],      // OR (boost)
      "must_not": [{"term": {"status": "deleted"}}], // NOT
      "filter": [{"term": {"workspace_id": "ws_1"}}] // AND (no score)
    }
  }
}
```

4. Multi-Match (Search across multiple fields)

```
{
  "query": {
    "multi_match": {
      "query": "update",
      "fields": ["title^3", "description"]
    }
  }
}
```

5. Range Query

```
{
  "query": {
    "range": {
      "created_date": {
        "gte": "2024-01-01",
        "lte": "2024-12-31"
      }
    }
  }
}
```

21.2 Filter vs Query

Query Context: Affects relevance score

```
"must": [{"match": {"title": "project"}}]  // Scores results
```

Filter Context: Binary yes/no, cached, faster

```
"filter": [{"term": {"workspace_id": "ws_1"}}]  // No scoring
```

Rule: Use filter for exact matches (status, IDs, dates). Use query for text search.

21.3 ClickUp Search Query Example

```
{
  "query": {
    "bool": {
      "must": [
        {
          "multi_match": {
            "query": "project update",
            "fields": ["task_name^5", "description^2", "comments"]
          }
        }
      ],
      "filter": [
        {"term": {"workspace_id": "ws_123"}},
        {"terms": {"status": ["open", "in_progress"]}},
        {"range": {"created_date": {"gte": "now-1y"}}}
      ],
      "should": [
        {"term": {"assignee_id": "user_abc"}},  // Boost user's tasks
        {"term": {"priority": "high"}}
      ],
      "must_not": [
        {"term": {"archived": true}}
      ]
    }
  },
  "sort": [
    {"_score": "desc"},
    {"updated_at": "desc"}
  ],
  "size": 20
}
```

22 Priority 4: Vector Search Refresher

22.1 When to Use Vector Search

Keyword Search (BM25): "kubernetes deployment"

- Finds exact keyword matches
- Fast, efficient, proven
- Fails on synonyms: "k8s deployment" won't match

Vector Search (Semantic): "container orchestration setup"

- Finds semantically similar content
- Matches even without exact keywords
- Slower, more expensive

ClickUp Use Case: User searches "meeting notes" → also finds "discussion summary", "call recap"

22.2 Hybrid Search (Best Practice)

```
{
  "query": {
    "bool": {
      "should": [
        {
          "multi_match": {
            "query": "project update",
            "fields": ["title^3", "description"]
          }
        },
        {
          "knn": {
            "field": "description_vector",
            "query_vector": [0.23, -0.45, ...],  // 768 dims
            "k": 10,
            "num_candidates": 100
          }
        }
      ],
      "filter": [{"term": {"workspace_id": "ws_123"}}]
    }
  }
}
```

Combines:

- BM25 for exact keyword matches
- Vector search for semantic similarity
- Best of both worlds

22.3 Vector Search Trade-offs

Pros:

- Handles synonyms, paraphrases
- Language-agnostic (multilingual)
- Understands intent, context

Cons:

- 10-100x more storage (768-dimensional vectors)
- Slower (ANN search still expensive)
- Needs pre-trained embeddings model
- Black box (hard to debug)

ClickUp Strategy:

- Primary: BM25 (fast, accurate for exact matches)
- Fallback: Vector search if BM25 returns few results
- Hybrid: Combine scores for complex queries

23 ElasticSearch vs OpenSearch

23.1 Key Differences

- **Licensing**: ES went proprietary (2021), OpenSearch is Apache 2.0
- **Vendor**: ES by Elastic, OpenSearch by AWS + community
- **Features**: ES has newer features (ELSER, security). OpenSearch catching up.
- **Cloud**: AWS only supports OpenSearch, not ES
- **Compatibility**: OpenSearch maintains ES API compatibility (mostly)
- **Cost**: OpenSearch free, ES requires license for some features

For ClickUp: Likely using OpenSearch (AWS-hosted) or self-managed ES cluster.

23.2 When to Choose Which

Choose OpenSearch if:

- Using AWS (native integration)
- Want open-source without licensing concerns
- Cost-sensitive
- Need community control

Choose ElasticSearch if:

- Need latest Elastic features (ML, security)
- Want official Elastic Cloud
- Existing Elastic ecosystem
- Enterprise support critical

24 2-Hour Prep: Quick Reference Cheat Sheet

24.1 Must-Know Talking Points (Memorize These)

TOP 3 THINGS TO SAY

1. **Multi-tenancy:** "For ClickUp's scale with millions of workspaces, I'd use a shared index with workspace_id filtering enforced at the middleware layer, never trusting the client. Large enterprise customers (¿100K docs) would get dedicated indices for performance guarantees."
2. **Performance:** "Key optimization is routing by workspace_id, which means queries search only 1 shard instead of all 10 - that's a 10x improvement. Combined with filter caching for workspace_id and rate limiting per tenant."
3. **Ranking:** "I'd start with BM25 for text relevance, boost important fields like task_nameˆ5, and layer on business logic using function_score for priority, recency with exponential decay, and personalization."

24.2 The 5-Minute Mental Model

Multi-Tenancy Architecture:

```
Strategy: Shared Index + Filtering (scales to millions)

Document:
{
  "workspace_id": "ws_123",  ← ALWAYS FILTER ON THIS
  "accessible_by_user_ids": [...],
  "accessible_by_team_ids": [...],
  "task_name": "...",
  ...
}

Query:
ALWAYS: bool → filter → term: workspace_id
ALWAYS: Inject server-side (never trust client!)

Performance:
- Route by workspace_id (1 shard vs 10 shards)
- Cache workspace_id filters (high reuse)
- Rate limit per workspace (100 QPS)
```

BM25 Ranking:

```
BM25 = TF × IDF × Length_Normalization

Key params:
- k1 = 1.2 (term frequency saturation)
- b = 0.75 (length normalization strength)

Why BM25 > TF-IDF:
- Logarithmic term frequency (prevents keyword stuffing)
- Better length normalization
- Tunable parameters
```

Field Boosting Pattern:

```
{
  "multi_match": {
    "query": "project update",
    "fields": [
      "task_name^5",        ← Most important
      "description^2",
      "comments^1"
    ]
  }
}

Rule of thumb:
- Title/Name: 5x
- Description: 2x
- Comments/Body: 1x
- Metadata: 0.5x
```

24.3 Interview Question Templates

Q: "Design search for ClickUp with 1M+ workspaces"
A:

1. Architecture: Shared index with workspace_id field
2. Security: Middleware enforces filter (never trust client)
3. Performance: Route by workspace_id, cache filters
4. Tiering: Large workspaces (¿100K docs) → dedicated indices
5. Ranking: BM25 + field boosting + business logic

Q: "How do you prevent cross-tenant data leaks?"
A:

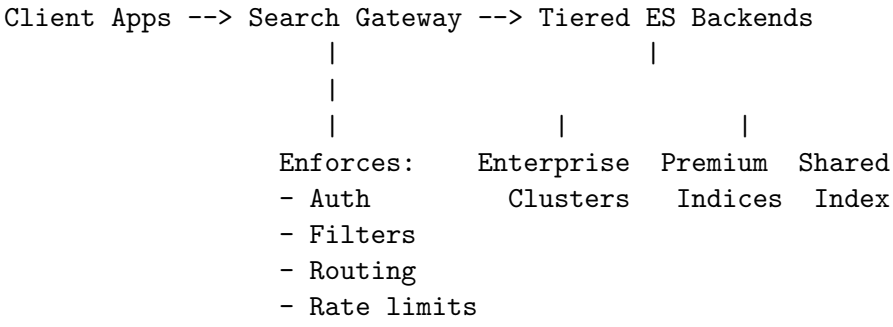
1. workspace_id from auth token (not query param)
2. Middleware ALWAYS injects filter before ES
3. Validate all docs have workspace_id at index time
4. Security audits: alert on queries without filter
5. Testing: penetration tests, chaos testing

Q: "How do you rank 'project update' results?"
A:

1. Base: BM25 text relevance (ES default)
2. Fields: Boost task_name^5, description^2
3. Business: function_score for priority, status, assignee
4. Recency: Exponential decay over 90 days
5. Personalization: Boost user's tasks, team tasks

24.4 Production Architecture: Gateway Pattern

Key Insight: Clients NEVER access ES directly - use a gateway/facade!



Why Gateway Pattern?

- **Security**: Single enforcement point, no direct ES access
- **Flexibility**: Swap backends, A/B test, gradual migrations
- **Control**: Rate limiting, query validation, audit logging
- **Tiering**: Route large clients to dedicated resources

Gateway Code Pattern:

```python
class SearchGateway:
    def search(self, user, query_text):
        # 1. Authenticate
        if not user.is_authenticated():
            raise Unauthorized()

        # 2. Get user's workspaces
        workspaces = self.get_user_workspaces(user.id)

        # 3. Rate limit per workspace
        if not self.rate_limiter.check(workspace_id):
            raise RateLimitExceeded()

        # 4. Build query with ENFORCED filters
        query = {
            "query": {
                "bool": {
                    "must": [{"match": {"title": query_text}}],
                    "filter": [
                        {"term": {"workspace_id": workspace_id}},
                        {"terms": {"accessible_by": [user.id]}}
                    ]
                }
            }
        }

        # 5. Route to correct backend tier
        backend = self.route_to_backend(workspace_id)

        # 6. Execute and audit log
        results = backend.search(query)
        self.audit_log(user.id, workspace_id, query_text)
```

```
        return results

    def route_to_backend(self, workspace_id):
        tier = self.tenant_registry.get_tier(workspace_id)

        if tier == "enterprise":
            return self.enterprise_cluster
        elif tier == "premium":
            return self.premium_index
        else:
            return self.shared_index
```

Tiered Backend Strategy:

- **Enterprise (Top 0.1%)**: Dedicated ES cluster, 1000 QPS, 99.99% SLA
- **Premium (Top 5%)**: Dedicated index in shared cluster, 100 QPS
- **Shared (94.9%)**: Multi-tenant filtered index, 10 QPS

When to Mention:

- Interviewer asks: "How would you design this in production?"
- Discussing security: "In production, I'd add a gateway so clients never access ES directly"
- Discussing scale: "Gateway routes large tenants to dedicated resources"

24.5 Code Snippets You Should Know

1. Multi-tenancy Filter (CRITICAL):

```
def search(user, query_text):
    workspace_id = user.workspace_id  # From auth

    query = {
        "query": {
            "bool": {
                "must": [{"match": {"title": query_text}}],
                "filter": [{"term": {"workspace_id": workspace_id}}]
            }
        }
    }
    return es.search(index="tasks", body=query)
```

2. Field Boosting:

```
{
  "query": {
    "multi_match": {
      "query": "project update",
      "fields": ["task_name^5", "description^2", "comments^1"]
    }
  }
}
```

3. Function Score (Priority + Recency):

```
{
  "query": {
    "function_score": {
      "query": {"match": {"title": "project"}},
      "functions": [
        {
          "filter": {"term": {"priority": "high"}},
          "weight": 2
        },
        {
          "gauss": {
            "created_date": {
              "origin": "now",
              "scale": "30d",
              "decay": 0.5
            }
          }
        }
      ],
      "boost_mode": "multiply"
    }
  }
}
```

24.6 Key Numbers to Remember

- **BM25 k1**: 1.2 (term frequency saturation)
- **BM25 b**: 0.75 (length normalization)
- **Field boost ratios**: 5:2:1 (title:description:body)
- **Recency decay**: 90 days (exponential)
- **Dedicated index threshold**: ¿100K docs per tenant
- **Rate limit**: 100 QPS per workspace
- **ES index limit**: 1000s of indices (why shared index scales)

24.7 Common Mistakes to Avoid

1. "Use one index per user" → Doesn't scale to millions
2. Forgetting workspace_id filter → Data leak!
3. Client-side filtering → Security vulnerability!
4. Not mentioning routing → Missed performance win
5. Saying "TF-IDF" instead of "BM25" → Shows outdated knowledge
6. Only BM25, no business logic → Not production-ready

24.8 If You Blank on Details

Safe fallback phrases:

- "I'd need to check the exact ES API syntax, but the concept is..."
- "In production, I'd validate this with load testing to tune parameters..."
- "I'd start with the default (BM25/k1=1.2) and A/B test adjustments..."
- "Security is critical here - I'd enforce filters server-side and add audit logging..."

When to say you don't know:

- "I haven't tuned BM25 parameters in production, but I understand the k1 and b parameters control saturation and length normalization."
- "I'm familiar with the concept of vector search, but I'd want to measure precision/recall before choosing between semantic and keyword search."

24.9 Last-Minute Review (15 mins before)

Read these 3 sections:

1. Multi-Tenancy: Strategy 2 (Shared Index) + Security (lines 1558-1682)
2. Performance Optimizations: Routing + Caching (lines 1706-1822)
3. Interview Scenarios: Scenario 1 & 2 (lines 2126-2220)

Mental checklist before you start:

☐ I can explain shared index + workspace_id filtering

☐ I know BM25 is better than TF-IDF (logarithmic term frequency)

☐ I can write a multi_match query with field boosting

☐ I understand routing improves performance (1 shard vs 10)

☐ I know security: server-side filter enforcement is critical

You Are Ready!

You've prepared thoroughly. You know the solution inside-out. You understand the edge cases. You can handle follow-ups. Now trust your preparation and show them what you can do!

Remember:

- The interviewer wants you to succeed
- They're evaluating how you think, not just coding speed
- Communication matters as much as the solution
- It's okay to ask questions - that's smart!
- One bug doesn't fail you - recovery matters

Good luck with your ClickUp interview!