ClickUp Backend Engineer Interview Questions

Comprehensive Preparation Guide

November 30, 2025

Contents

1 Introduction

This document contains 14 backend engineering problems commonly asked in ClickUp interviews, based on real candidate experiences and ClickUp's technical stack.

ClickUp Tech Stack:

- Node.js / TypeScript / Express
- Transitioning from Monolith to Microservices
- Docker & Amazon EKS (Kubernetes)
- Focus on API design and scalability

Interview Process:

1. Technical Recruiter Screen
2. Take-Home Assignment (1hr Node.js API)
3. Technical Interview (LeetCode Medium + Discussion)
4. System Design Round
5. Cultural Fit Interview

2 Problem 1: Two Sum (Array Pairs)

Problem

Difficulty: Easy/Medium
Topics: Hash Map, Arrays
Asked at ClickUp: Yes (confirmed)

Given an array of integers `nums` and an integer `target`, return indices of the two numbers that add up to `target`. You may not use the same element twice.

Constraints:

- $2 \leq$ nums.length $\leq 10^4$
- $-10^9 \leq$ nums[i] $\leq 10^9$
- Exactly one solution exists
- Cannot use same index twice

Example 1:

```
Input: nums = [2, 7, 11, 15], target = 9
Output: [0, 1]
Explanation: nums[0] + nums[1] = 2 + 7 = 9
```

Example 2:

```
Input: nums = [3, 2, 4], target = 6
Output: [1, 2]
```

Solution

Approach: Hash Map (Optimal)
The key insight is to use a hash map to store numbers we've seen and check if the complement exists.
Algorithm:

1. Create empty hash map
2. For each number at index i:
   - Calculate complement = target - nums[i]
   - If complement exists in hash map, return [map[complement], i]
   - Otherwise, store nums[i] → i in map

Time Complexity: O(n)
Space Complexity: O(n)

```python
def two_sum(nums: List[int], target: int) -> List[int]:
    """
    Find indices of two numbers that sum to target.

    Args:
        nums: Array of integers
        target: Target sum

    Returns:
        List of two indices [i, j] where nums[i] + nums[j] =
            target
    """
    seen = {}  # Maps number -> index

    for i, num in enumerate(nums):
        complement = target - num

        if complement in seen:
            return [seen[complement], i]

        seen[num] = i

    return []  # No solution found
```

Test Cases

```python
def test_two_sum():
    # Test 1: Basic case
    assert two_sum([2, 7, 11, 15], 9) == [0, 1]

    # Test 2: Numbers not in order
    assert two_sum([3, 2, 4], 6) == [1, 2]

    # Test 3: Negative numbers
    assert two_sum([-1, -2, -3, -4, -5], -8) == [2, 4]

    # Test 4: Same number appears twice
    assert two_sum([3, 3], 6) == [0, 1]

    # Test 5: Larger array
    assert two_sum([1, 5, 3, 7, 9, 2], 10) == [1, 3]

    print("All test cases passed!")

if __name__ == "__main__":
    test_two_sum()
```

3 Problem 2: Combination Sum

Problem

Difficulty: Medium
Topics: Backtracking, Recursion
Asked at ClickUp: Yes (confirmed variation)

Given an array of distinct integers `candidates` and a target integer `target`, return all unique combinations where the chosen numbers sum to `target`. Numbers can be used unlimited times.

Constraints:

- $1 \leq$ candidates.length ≤ 30
- All elements in candidates are distinct
- $1 \leq$ candidates[i] ≤ 200
- $1 \leq$ target ≤ 500

Example:

```
Input: candidates = [2, 3, 6, 7], target = 7
Output: [[2, 2, 3], [7]]
Explanation:
2 + 2 + 3 = 7
7 = 7
```

Solution

Approach: Backtracking

Use recursion to explore all possible combinations, pruning when sum exceeds target.

```python
def combination_sum(candidates: List[int], target: int) -> List[List[int]]:
    """
    Find all unique combinations that sum to target.

    Args:
        candidates: Array of distinct integers
        target: Target sum

    Returns:
        List of all valid combinations
    """
    result = []

    def backtrack(start: int, current: List[int], current_sum: int):
        # Base cases
        if current_sum == target:
            result.append(current[:])  # Found valid combination
            return

        if current_sum > target:
            return  # Exceeded target, prune

        # Try each candidate starting from 'start' index
        for i in range(start, len(candidates)):
            current.append(candidates[i])
            # Can reuse same element, so pass i (not i+1)
            backtrack(i, current, current_sum + candidates[i])
            current.pop()  # Backtrack

    backtrack(0, [], 0)
    return result
```

Time Complexity: $O(k \cdot 2^t)$ where k is average combination length, t is target

Space Complexity: O(target) for recursion depth

Test Cases

```python
def test_combination_sum():
    # Test 1: Basic case
    result1 = combination_sum([2, 3, 6, 7], 7)
    assert sorted(result1) == sorted([[2, 2, 3], [7]])

    # Test 2: Multiple solutions
    result2 = combination_sum([2, 3, 5], 8)
    assert len(result2) == 3  # [2,2,2,2], [2,3,3], [3,5]

    # Test 3: No solution
    result3 = combination_sum([2, 4], 7)
    assert result3 == []

    print("All test cases passed!")
```

4 Problem 3: Task Dependency Resolution

Problem

Difficulty: Medium
Topics: Graphs, Topological Sort, DFS
ClickUp Context: Task hierarchies and dependencies

Given a list of tasks and their dependencies, determine if all tasks can be completed and return a valid execution order. Return empty list if there's a circular dependency.

Input Format:

- `num_tasks`: Number of tasks (labeled 0 to n-1)
- `dependencies`: List of [task, prerequisite] pairs

Example 1:

```
Input: num_tasks = 4, dependencies = [[1,0], [2,0], [3,1], [3,2]]
Output: [0, 1, 2, 3] or [0, 2, 1, 3]
Explanation:
Task 0 has no prerequisites
Tasks 1 and 2 depend on task 0
Task 3 depends on tasks 1 and 2
```

Example 2:

```
Input: num_tasks = 2, dependencies = [[1,0], [0,1]]
Output: []
Explanation: Circular dependency exists
```

Solution

Approach: Topological Sort using DFS

Use DFS with cycle detection to find valid task execution order.

```python
from collections import defaultdict
from typing import List

def find_task_order(num_tasks: int, dependencies: List[List[int]]) -> List[int]:
    """
    Find valid task execution order or detect circular dependencies.

    Args:
        num_tasks: Total number of tasks
        dependencies: List of [task, prerequisite] pairs

    Returns:
        Valid execution order, or empty list if circular dependency exists
    """
    # Build adjacency list
    graph = defaultdict(list)
    for task, prereq in dependencies:
        graph[prereq].append(task)

    # States: 0 = unvisited, 1 = visiting, 2 = visited
    state = [0] * num_tasks
    result = []

    def has_cycle(task: int) -> bool:
        """DFS to detect cycles."""
        if state[task] == 1:  # Currently visiting - cycle detected
            return True
        if state[task] == 2:  # Already processed
            return False

        state[task] = 1  # Mark as visiting

        # Visit all dependent tasks
        for dependent in graph[task]:
            if has_cycle(dependent):
                return True

        state[task] = 2  # Mark as visited
        result.append(task)  # Add to result in reverse order
        return False

    # Check all tasks
    for task in range(num_tasks):
        if has_cycle(task):
            return []  # Circular dependency found

    return result[::-1]  # Reverse to get correct order
```

Time Complexity: O(V + E) where V = tasks, E = dependencies
Space Complexity: O(V + E)

Test Cases

```python
def test_task_order():
    # Test 1: Valid dependency chain
    result1 = find_task_order(4, [[1,0], [2,0], [3,1], [3,2]])
    assert result1 in [[0,1,2,3], [0,2,1,3]]

    # Test 2: Circular dependency
    result2 = find_task_order(2, [[1,0], [0,1]])
    assert result2 == []

    # Test 3: No dependencies
    result3 = find_task_order(3, [])
    assert len(result3) == 3

    # Test 4: Linear chain
    result4 = find_task_order(3, [[1,0], [2,1]])
    assert result4 == [0, 1, 2]

    print("All test cases passed!")
```

5 Problem 4: API Rate Limiter

Problem

Difficulty: Medium
Topics: System Design, Sliding Window
ClickUp Context: API design and scalability

Design a rate limiter that restricts the number of API requests a user can make within a time window.

Requirements:

- Support sliding window rate limiting
- `is_allowed(user_id, timestamp)`: Check if request is allowed
- Track requests per user
- Configure max requests per time window

Example:

```
limiter = RateLimiter(max_requests=3, window_seconds=60)
limiter.is_allowed("user1", 0)   # True  (1/3)
limiter.is_allowed("user1", 10)  # True  (2/3)
limiter.is_allowed("user1", 20)  # True  (3/3)
limiter.is_allowed("user1", 30)  # False (limit reached)
limiter.is_allowed("user1", 65)  # True  (first request expired)
```

Solution

Approach: Sliding Window with Deque

Maintain a queue of timestamps for each user and remove expired entries.

```python
from collections import defaultdict, deque
from typing import Dict

class RateLimiter:
    """
    Rate limiter using sliding window algorithm.
    """

    def __init__(self, max_requests: int, window_seconds: int):
        """
        Initialize rate limiter.

        Args:
            max_requests: Maximum requests allowed in window
            window_seconds: Time window in seconds
        """
        self.max_requests = max_requests
        self.window_seconds = window_seconds
        # Map user_id -> deque of timestamps
        self.requests: Dict[str, deque] = defaultdict(deque)

    def is_allowed(self, user_id: str, timestamp: int) -> bool:
        """
        Check if request is allowed for user at given timestamp.

        Args:
            user_id: User identifier
            timestamp: Current timestamp in seconds

        Returns:
            True if request is allowed, False otherwise
        """
        user_requests = self.requests[user_id]

        # Remove expired requests (outside sliding window)
        window_start = timestamp - self.window_seconds
        while user_requests and user_requests[0] <= window_start:
            user_requests.popleft()

        # Check if under limit
        if len(user_requests) < self.max_requests:
            user_requests.append(timestamp)
            return True

        return False

    def get_request_count(self, user_id: str, timestamp: int) ->
        int:
        """Get current request count in window."""
        user_requests = self.requests[user_id]
        window_start = timestamp - self.window_seconds

        # Remove expired
        while user_requests and user_requests[0] <= window_start:
            user_requests.popleft()

        return len(user_requests)
```

Time Complexity: O(n) worst case for removing expired entries

Space Complexity: O(u · r) where u = users, r = requests per window

Test Cases

```python
def test_rate_limiter():
    # Test 1: Basic rate limiting
    limiter = RateLimiter(max_requests=3, window_seconds=60)
    assert limiter.is_allowed("user1", 0) == True
    assert limiter.is_allowed("user1", 10) == True
    assert limiter.is_allowed("user1", 20) == True
    assert limiter.is_allowed("user1", 30) == False

    # Test 2: Sliding window expiration
    assert limiter.is_allowed("user1", 65) == True  # First
        request expired

    # Test 3: Multiple users
    limiter2 = RateLimiter(max_requests=2, window_seconds=10)
    assert limiter2.is_allowed("user1", 0) == True
    assert limiter2.is_allowed("user2", 0) == True
    assert limiter2.is_allowed("user1", 5) == True
    assert limiter2.is_allowed("user1", 7) == False
    assert limiter2.is_allowed("user2", 7) == True

    # Test 4: Request count
    assert limiter2.get_request_count("user1", 7) == 2

    print("All test cases passed!")
```

6 Problem 5: Event Stream Processor

Problem

Difficulty: Medium
Topics: Async, Streams, Data Processing
ClickUp Context: Real-time event processing

Implement an event stream processor that handles incoming events, applies transformations, and maintains windowed aggregations.

Requirements:

- Process events with `event_id`, `event_type`, `timestamp`, `data`
- Filter events by type
- Transform event data
- Calculate windowed aggregates (count, sum, avg)
- Handle out-of-order events

Example:

```
Events:
{"id": 1, "type": "click", "timestamp": 100, "value": 10}
{"id": 2, "type": "view", "timestamp": 101, "value": 5}
{"id": 3, "type": "click", "timestamp": 102, "value": 15}

Window (100-110):
  clicks: count=2, sum=25, avg=12.5
  views: count=1, sum=5, avg=5.0
```

Solution

Approach: Buffered Stream with Sliding Windows

```python
from collections import defaultdict, deque
from typing import Dict, List, Callable, Any

class Event:
    """Represents a single event."""
    def __init__(self, event_id: int, event_type: str,
                 timestamp: int, data: Dict[str, Any]):
        self.event_id = event_id
        self.event_type = event_type
        self.timestamp = timestamp
        self.data = data

class EventStreamProcessor:
    """
    Process streaming events with windowed aggregations.
    """

    def __init__(self, window_size: int = 60):
        """
        Initialize processor.

        Args:
            window_size: Window size in seconds
        """
        self.window_size = window_size
        # Store events by type
        self.events: Dict[str, deque] = defaultdict(deque)
        # Filters and transformers
        self.filters: List[Callable] = []
        self.transformers: List[Callable] = []

    def add_filter(self, filter_func: Callable[[Event], bool]):
        """Add event filter."""
        self.filters.append(filter_func)

    def add_transformer(self, transform_func: Callable[[Event],
        Event]):
        """Add event transformer."""
        self.transformers.append(transform_func)

    def process_event(self, event: Event) -> bool:
        """
        Process a single event.

        Args:
            event: Event to process

        Returns:
            True if event was accepted, False if filtered out
        """
        # Apply filters
        for filter_func in self.filters:
            if not filter_func(event):
                return False

        # Apply transformations
        for transform_func in self.transformers:
            event = transform_func(event)

        # Store event
        self.events[event.event_type].append(event)
```

Test Cases

```
def test_event_stream_processor():
    # Test 1: Basic event processing
    processor = EventStreamProcessor(window_size=60)

    e1 = Event(1, "click", 100, {"value": 10})
    e2 = Event(2, "view", 101, {"value": 5})
    e3 = Event(3, "click", 102, {"value": 15})

    assert processor.process_event(e1) == True
    assert processor.process_event(e2) == True
    assert processor.process_event(e3) == True

    # Test 2: Window statistics
    stats = processor.get_window_stats("click", 110)
    assert stats["count"] == 2
    assert stats["sum"] == 25
    assert stats["avg"] == 12.5

    # Test 3: Event filtering
    processor2 = EventStreamProcessor(window_size=60)
    processor2.add_filter(lambda e: e.data.get("value", 0) > 5)

    assert processor2.process_event(Event(1, "test", 100, {"value"
        : 3})) == False
    assert processor2.process_event(Event(2, "test", 100, {"value"
        : 10})) == True

    # Test 4: Event expiration
    stats = processor.get_window_stats("click", 200)
    assert stats["count"] == 0  # All events expired

    print("All test cases passed!")
```

7 Problem 6: Task Hierarchy Builder

Problem

Difficulty: Medium
Topics: Trees, Recursion, Data Structures
ClickUp Context: Task/subtask hierarchy system

Build a task hierarchy system that supports nested tasks (ClickUp's core feature).

Requirements:

- Create tasks and subtasks (unlimited nesting)
- Find all subtasks under a task (recursive)
- Calculate total estimated time for task + all subtasks
- Find task path from root to specific task
- Get tasks at specific depth level

Example:

```
Task Hierarchy:
Project Alpha (8h)
 Design Phase (3h)
    Wireframes (1h)
    Mockups (2h)
 Development (5h)
     Backend API (3h)
     Frontend (2h)
```

Solution

Approach: Tree with Recursive Operations

```python
from typing import List, Optional, Dict
from dataclasses import dataclass, field

@dataclass
class Task:
    """Represents a task in the hierarchy."""
    task_id: str
    name: str
    estimated_hours: float
    parent_id: Optional[str] = None
    subtasks: List['Task'] = field(default_factory=list)
    metadata: Dict = field(default_factory=dict)

class TaskHierarchy:
    """
    Manages hierarchical task structure.
    """

    def __init__(self):
        self.tasks: Dict[str, Task] = {}
        self.root_tasks: List[Task] = []

    def add_task(self, task_id: str, name: str,
                 estimated_hours: float,
                 parent_id: Optional[str] = None) -> Task:
        """
        Add a task to the hierarchy.

        Args:
            task_id: Unique task identifier
            name: Task name
            estimated_hours: Estimated completion time
            parent_id: Parent task ID (None for root tasks)

        Returns:
            Created Task object
        """
        task = Task(task_id, name, estimated_hours, parent_id)
        self.tasks[task_id] = task

        if parent_id:
            if parent_id not in self.tasks:
                raise ValueError(f"Parent task {parent_id} not
                    found")
            parent = self.tasks[parent_id]
            parent.subtasks.append(task)
        else:
            self.root_tasks.append(task)

        return task

    def get_all_subtasks(self, task_id: str) -> List[Task]:
        """
        Get all subtasks recursively.

        Args:
            task_id: Task ID to search from

        Returns:
            List of all descendant tasks
        """
```

Test Cases

```python
def test_task_hierarchy():
    # Test 1: Build hierarchy
    hierarchy = TaskHierarchy()

    hierarchy.add_task("proj", "Project Alpha", 8.0)
    hierarchy.add_task("design", "Design Phase", 3.0, "proj")
    hierarchy.add_task("wire", "Wireframes", 1.0, "design")
    hierarchy.add_task("mock", "Mockups", 2.0, "design")
    hierarchy.add_task("dev", "Development", 5.0, "proj")
    hierarchy.add_task("backend", "Backend API", 3.0, "dev")
    hierarchy.add_task("frontend", "Frontend", 2.0, "dev")

    # Test 2: Get all subtasks
    subtasks = hierarchy.get_all_subtasks("proj")
    assert len(subtasks) == 6  # All tasks except root

    # Test 3: Calculate total time
    total = hierarchy.calculate_total_time("proj")
    assert total == 16.0  # 8 + 3 + 1 + 2 + 5 + 3 + 2

    # Test 4: Find task path
    path = hierarchy.find_task_path("wire")
    assert path == ["proj", "design", "wire"]

    # Test 5: Get tasks at depth
    depth_0 = hierarchy.get_tasks_at_depth(0)
    assert len(depth_0) == 1  # Just "proj"

    depth_1 = hierarchy.get_tasks_at_depth(1)
    assert len(depth_1) == 2  # "design" and "dev"

    depth_2 = hierarchy.get_tasks_at_depth(2)
    assert len(depth_2) == 4  # All leaf tasks

    print("All test cases passed!")
```

8 Summary and Additional Tips

8.1 Key Topics for ClickUp Interviews

1. **Data Structures & Algorithms**
   - Hash maps and sets (Two Sum variations)
   - Backtracking (Combination problems)
   - Graph algorithms (Task dependencies, topological sort)
   - Trees (Task hierarchies)
2. **Backend System Design**
   - API design patterns
   - Rate limiting strategies
   - Caching mechanisms
   - Microservices architecture
3. **Node.js / JavaScript Concepts**
   - Event loop and async patterns
   - Promises and async/await
   - Stream processing
   - Error handling
4. **Database & Data Processing**
   - Query optimization
   - Data aggregation
   - Real-time data processing
   - Data integrity and validation

8.2 Interview Preparation Strategy

1. **Master the Fundamentals**
   - Practice LeetCode medium problems daily
   - Focus on hash maps, graphs, and trees
   - Understand time/space complexity analysis
2. **Study ClickUp's Tech Stack**
   - Review Node.js best practices
   - Understand Express middleware patterns
   - Study Docker and Kubernetes basics
   - Learn about microservices architecture
3. **Practice System Design**
   - Design scalable API systems
   - Plan database schemas

- Consider caching and rate limiting
- Think about real-world ClickUp features

4. **Prepare for Take-Home Assignment**
   - Build a simple Express API beforehand
   - Practice writing testable, modular code
   - Review testing frameworks (Jest, Mocha)
   - Focus on code organization and documentation

8.3 Common Interview Questions

Technical Discussion Topics:

- Explain the JavaScript event loop
- How does async/await work under the hood?
- Describe your experience with microservices
- How would you handle database migrations?
- Explain caching strategies you've used
- How do you ensure API security?

Behavioral Questions:

- Describe a challenging bug you fixed
- How do you handle tight deadlines?
- Tell me about a time you disagreed with a technical decision
- How do you stay current with technology?
- Describe your ideal development workflow

8.4 Resources

- **LeetCode:** Focus on medium difficulty, especially graphs and backtracking
- **Node.js Docs:** Deep dive into async patterns and streams
- **System Design:** "Designing Data-Intensive Applications" by Martin Kleppmann
- **ClickUp Blog:** Read about their engineering challenges and solutions