

ClickUp Backend Engineer Interview Questions

Comprehensive Preparation Guide

November 9, 2025

Contents

1	Introduction	2
2	Problem 1: Two Sum (Array Pairs)	3
3	Problem 2: Combination Sum	6
4	Problem 3: Task Dependency Resolution	9
5	Problem 4: API Rate Limiter	12
6	Problem 5: Event Stream Processor	15
7	Problem 6: Task Hierarchy Builder	18
8	Summary and Additional Tips	21
8.1	Key Topics for ClickUp Interviews	21
8.2	Interview Preparation Strategy	21
8.3	Common Interview Questions	22
8.4	Resources	22

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:

- $2 \leq \text{nums.length} \leq 10^4$
- $-10^9 \leq \text{nums}[i] \leq 10^9$
- 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)

```
1 def two_sum(nums: List[int], target: int) -> List[int]:
2     """
3         Find indices of two numbers that sum to target.
4
5         Args:
6             nums: Array of integers
7             target: Target sum
8
9         Returns:
10            List of two indices [i, j] where nums[i] + nums[j] =
11                target
12
13    seen = {} # Maps number -> index
14
15    for i, num in enumerate(nums):
16        complement = target - num
17
18        if complement in seen:
19            return [seen[complement], i]
20
21        seen[num] = i
22
23    return [] # No solution found
```

Test Cases

```
1 def test_two_sum():
2     # Test 1: Basic case
3     assert two_sum([2, 7, 11, 15], 9) == [0, 1]
4
5     # Test 2: Numbers not in order
6     assert two_sum([3, 2, 4], 6) == [1, 2]
7
8     # Test 3: Negative numbers
9     assert two_sum([-1, -2, -3, -4, -5], -8) == [2, 4]
10
11    # Test 4: Same number appears twice
12    assert two_sum([3, 3], 6) == [0, 1]
13
14    # Test 5: Larger array
15    assert two_sum([1, 5, 3, 7, 9, 2], 10) == [1, 3]
16
17    print("All test cases passed!")
18
19 if __name__ == "__main__":
20     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:

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

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.

```
1 def combination_sum(candidates: List[int], target: int) -> List[List[int]]:
2     """
3         Find all unique combinations that sum to target.
4
5     Args:
6         candidates: Array of distinct integers
7         target: Target sum
8
9     Returns:
10        List of all valid combinations
11    """
12    result = []
13
14    def backtrack(start: int, current: List[int], current_sum: int):
15        # Base cases
16        if current_sum == target:
17            result.append(current[:]) # Found valid combination
18            return
19
20        if current_sum > target:
21            return # Exceeded target, prune
22
23        # Try each candidate starting from 'start' index
24        for i in range(start, len(candidates)):
25            current.append(candidates[i])
26            # Can reuse same element, so pass i (not i+1)
27            backtrack(i, current, current_sum + candidates[i])
28            current.pop() # Backtrack
29
30    backtrack(0, [], 0)
31    return result
```

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

Space Complexity: $O(\text{target})$ for recursion depth

Test Cases

```
1 def test_combination_sum():
2     # Test 1: Basic case
3     result1 = combination_sum([2, 3, 6, 7], 7)
4     assert sorted(result1) == sorted([[2, 2, 3], [7]])
5
6     # Test 2: Multiple solutions
7     result2 = combination_sum([2, 3, 5], 8)
8     assert len(result2) == 3    # [2,2,2,2], [2,3,3], [3,5]
9
10    # Test 3: No solution
11    result3 = combination_sum([2, 4], 7)
12    assert result3 == []
13
14    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.

```
1 from collections import defaultdict
2 from typing import List
3
4 def find_task_order(num_tasks: int, dependencies: List[List[int]]) -> List[int]:
5     """
6         Find valid task execution order or detect circular
7         dependencies.
8
9     Args:
10        num_tasks: Total number of tasks
11        dependencies: List of [task, prerequisite] pairs
12
13    Returns:
14        Valid execution order, or empty list if circular
15        dependency exists
16    """
17
18    # Build adjacency list
19    graph = defaultdict(list)
20    for task, prereq in dependencies:
21        graph[prereq].append(task)
22
23    # States: 0 = unvisited, 1 = visiting, 2 = visited
24    state = [0] * num_tasks
25    result = []
26
27    def has_cycle(task: int) -> bool:
28        """
29            DFS to detect cycles.
30        """
31        if state[task] == 1: # Currently visiting - cycle
32            detected
33            return True
34        if state[task] == 2: # Already processed
35            return False
36
37        state[task] = 1 # Mark as visiting
38
39        # Visit all dependent tasks
40        for dependent in graph[task]:
41            if has_cycle(dependent):
42                return True
43
44        state[task] = 2 # Mark as visited
45        result.append(task) # Add to result in reverse order
46    return False
47
48    # Check all tasks
49    for task in range(num_tasks):
50        if has_cycle(task):
51            return [] # Circular dependency found
52
53    return result[::-1] # Reverse to get correct order
```

Time Complexity: $O(V + E)$ where $V = \text{tasks}$, $E = \text{dependencies}$

Space Complexity: $O(V + E)$

Test Cases

```
1 def test_task_order():
2     # Test 1: Valid dependency chain
3     result1 = find_task_order(4, [[1,0], [2,0], [3,1], [3,2]])
4     assert result1 in [[0,1,2,3], [0,2,1,3]]
5
6     # Test 2: Circular dependency
7     result2 = find_task_order(2, [[1,0], [0,1]])
8     assert result2 == []
9
10    # Test 3: No dependencies
11    result3 = find_task_order(3, [])
12    assert len(result3) == 3
13
14    # Test 4: Linear chain
15    result4 = find_task_order(3, [[1,0], [2,1]])
16    assert result4 == [0, 1, 2]
17
18    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.

```
1 from collections import defaultdict, deque
2 from typing import Dict
3
4 class RateLimiter:
5     """
6         Rate limiter using sliding window algorithm.
7     """
8
9     def __init__(self, max_requests: int, window_seconds: int):
10        """
11            Initialize rate limiter.
12
13        Args:
14            max_requests: Maximum requests allowed in window
15            window_seconds: Time window in seconds
16        """
17        self.max_requests = max_requests
18        self.window_seconds = window_seconds
19        # Map user_id -> deque of timestamps
20        self.requests: Dict[str, deque] = defaultdict(deque)
21
22    def is_allowed(self, user_id: str, timestamp: int) -> bool:
23        """
24            Check if request is allowed for user at given timestamp.
25
26        Args:
27            user_id: User identifier
28            timestamp: Current timestamp in seconds
29
30        Returns:
31            True if request is allowed, False otherwise
32        """
33        user_requests = self.requests[user_id]
34
35        # Remove expired requests (outside sliding window)
36        window_start = timestamp - self.window_seconds
37        while user_requests and user_requests[0] <= window_start:
38            user_requests.popleft()
39
40        # Check if under limit
41        if len(user_requests) < self.max_requests:
42            user_requests.append(timestamp)
43            return True
44
45        return False
46
47    def get_request_count(self, user_id: str, timestamp: int) ->
48        int:
49        """
50            Get current request count in window.
51        """
52        user_requests = self.requests[user_id]
53        window_start = timestamp - self.window_seconds
54
55        # Remove expired
56        while user_requests and user_requests[0] <= window_start:
57            user_requests.popleft()
58
59        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

```
1 def test_rate_limiter():
2     # Test 1: Basic rate limiting
3     limiter = RateLimiter(max_requests=3, window_seconds=60)
4     assert limiter.is_allowed("user1", 0) == True
5     assert limiter.is_allowed("user1", 10) == True
6     assert limiter.is_allowed("user1", 20) == True
7     assert limiter.is_allowed("user1", 30) == False
8
9     # Test 2: Sliding window expiration
10    assert limiter.is_allowed("user1", 65) == True # First
11        request expired
12
13    # Test 3: Multiple users
14    limiter2 = RateLimiter(max_requests=2, window_seconds=10)
15    assert limiter2.is_allowed("user1", 0) == True
16    assert limiter2.is_allowed("user2", 0) == True
17    assert limiter2.is_allowed("user1", 5) == True
18    assert limiter2.is_allowed("user1", 7) == False
19    assert limiter2.is_allowed("user2", 7) == True
20
21    # Test 4: Request count
22    assert limiter2.get_request_count("user1", 7) == 2
23
24    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

```
1  from collections import defaultdict, deque
2  from typing import Dict, List, Callable, Any
3
4  class Event:
5      """Represents a single event."""
6      def __init__(self, event_id: int, event_type: str,
7                   timestamp: int, data: Dict[str, Any]):
8          self.event_id = event_id
9          self.event_type = event_type
10         self.timestamp = timestamp
11         self.data = data
12
13  class EventStreamProcessor:
14      """
15          Process streaming events with windowed aggregations.
16      """
17
18      def __init__(self, window_size: int = 60):
19          """
20              Initialize processor.
21
22          Args:
23              window_size: Window size in seconds
24          """
25          self.window_size = window_size
26          # Store events by type
27          self.events: Dict[str, deque] = defaultdict(deque)
28          # Filters and transformers
29          self.filters: List[Callable] = []
30          self.transformers: List[Callable] = []
31
32      def add_filter(self, filter_func: Callable[[Event], bool]):
33          """
34              Add event filter.
35          """
36          self.filters.append(filter_func)
37
38      def add_transformer(self, transform_func: Callable[[Event], Event]):
39          """
40              Add event transformer.
41          """
42          self.transformers.append(transform_func)
43
44      def process_event(self, event: Event) -> bool:
45          """
46              Process a single event.
47
48          Args:
49              event: Event to process
50
51          Returns:
52              True if event was accepted, False if filtered out
53          """
54          # Apply filters
55          for filter_func in self.filters:
56              if not filter_func(event):
57                  return False
58
59          # Apply transformations
60          for transform_func in self.transformers:
61              event = transform_func(event)
62
63          # Store event
64          self.events[event.event_type].append(event)
```

Test Cases

```
1 def test_event_stream_processor():
2     # Test 1: Basic event processing
3     processor = EventStreamProcessor(window_size=60)
4
5     e1 = Event(1, "click", 100, {"value": 10})
6     e2 = Event(2, "view", 101, {"value": 5})
7     e3 = Event(3, "click", 102, {"value": 15})
8
9     assert processor.process_event(e1) == True
10    assert processor.process_event(e2) == True
11    assert processor.process_event(e3) == True
12
13    # Test 2: Window statistics
14    stats = processor.get_window_stats("click", 110)
15    assert stats["count"] == 2
16    assert stats["sum"] == 25
17    assert stats["avg"] == 12.5
18
19    # Test 3: Event filtering
20    processor2 = EventStreamProcessor(window_size=60)
21    processor2.add_filter(lambda e: e.data.get("value", 0) > 5)
22
23    assert processor2.process_event(Event(1, "test", 100, {"value": 3})) == False
24    assert processor2.process_event(Event(2, "test", 100, {"value": 10})) == True
25
26    # Test 4: Event expiration
27    stats = processor.get_window_stats("click", 200)
28    assert stats["count"] == 0 # All events expired
29
30    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

```
1  from typing import List, Optional, Dict
2  from dataclasses import dataclass, field
3
4  @dataclass
5  class Task:
6      """Represents a task in the hierarchy."""
7      task_id: str
8      name: str
9      estimated_hours: float
10     parent_id: Optional[str] = None
11     subtasks: List['Task'] = field(default_factory=list)
12     metadata: Dict = field(default_factory=dict)
13
14  class TaskHierarchy:
15      """
16          Manages hierarchical task structure.
17      """
18
19      def __init__(self):
20          self.tasks: Dict[str, Task] = {}
21          self.root_tasks: List[Task] = []
22
23      def add_task(self, task_id: str, name: str,
24                  estimated_hours: float,
25                  parent_id: Optional[str] = None) -> Task:
26          """
27              Add a task to the hierarchy.
28
29          Args:
30              task_id: Unique task identifier
31              name: Task name
32              estimated_hours: Estimated completion time
33              parent_id: Parent task ID (None for root tasks)
34
35          Returns:
36              Created Task object
37          """
38
39          task = Task(task_id, name, estimated_hours, parent_id)
40          self.tasks[task_id] = task
41
42          if parent_id:
43              if parent_id not in self.tasks:
44                  raise ValueError(f"Parent task {parent_id} not found")
45              parent = self.tasks[parent_id]
46              parent.subtasks.append(task)
47          else:
48              self.root_tasks.append(task)
49
50
51      def get_all_subtasks(self, task_id: str) -> List[Task]:
52          """
53              Get all subtasks recursively.
54
55          Args:
56              task_id: Task ID to search from
57
58          Returns:
59              List of all descendant tasks
60          """
61
```

Test Cases

```
1 def test_task_hierarchy():
2     # Test 1: Build hierarchy
3     hierarchy = TaskHierarchy()
4
5     hierarchy.add_task("proj", "Project Alpha", 8.0)
6     hierarchy.add_task("design", "Design Phase", 3.0, "proj")
7     hierarchy.add_task("wire", "Wireframes", 1.0, "design")
8     hierarchy.add_task("mock", "Mockups", 2.0, "design")
9     hierarchy.add_task("dev", "Development", 5.0, "proj")
10    hierarchy.add_task("backend", "Backend API", 3.0, "dev")
11    hierarchy.add_task("frontend", "Frontend", 2.0, "dev")
12
13    # Test 2: Get all subtasks
14    subtasks = hierarchy.get_all_subtasks("proj")
15    assert len(subtasks) == 6 # All tasks except root
16
17    # Test 3: Calculate total time
18    total = hierarchy.calculate_total_time("proj")
19    assert total == 16.0 # 8 + 3 + 1 + 2 + 5 + 3 + 2
20
21    # Test 4: Find task path
22    path = hierarchy.find_task_path("wire")
23    assert path == ["proj", "design", "wire"]
24
25    # Test 5: Get tasks at depth
26    depth_0 = hierarchy.get_tasks_at_depth(0)
27    assert len(depth_0) == 1 # Just "proj"
28
29    depth_1 = hierarchy.get_tasks_at_depth(1)
30    assert len(depth_1) == 2 # "design" and "dev"
31
32    depth_2 = hierarchy.get_tasks_at_depth(2)
33    assert len(depth_2) == 4 # All leaf tasks
34
35    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