Go

Interview Questions and Answers

Core Concepts

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

---

1. Explain the difference between goroutines and OS threads. How does the Go scheduler work?

Goroutines vs OS Threads: Goroutines are lightweight, user-space threads managed by the Go runtime, while OS threads are kernel-level entities managed by the operating system.

- **Memory:** Goroutines start with ~2KB stack (growable), OS threads typically use 1-2MB
- **Context switching:** Goroutines have faster context switches (software-level) vs OS threads (kernel-level)
- **Scheduling:** Goroutines use M:N scheduling (many goroutines on fewer OS threads)

Go Scheduler (GMP Model):

- **G (Goroutine):** The goroutine itself
- **M (Machine):** OS thread that executes goroutines
- **P (Processor):** Logical processor that holds context for executing goroutines (default: GOMAXPROCS = number of CPUs)

The scheduler uses work-stealing: when a P's local run queue is empty, it steals goroutines from other P's queues. This ensures efficient CPU utilization without manual thread pool management.

2. What are the key differences between buffered and unbuffered channels? When would you use each?

Unbuffered Channels:

```
ch := make(chan int)
// Sender blocks until receiver is ready
// Provides synchronization guarantee
```

Characteristics:

- Capacity of 0
- Send blocks until receive occurs (synchronous)
- Guarantees message delivery before sender continues

Buffered Channels:

```
ch := make(chan int, 5)
// Sender blocks only when buffer is full
// Receiver blocks only when buffer is empty
```

Characteristics:

- Has specified capacity
- Send blocks only when buffer is full
- Allows asynchronous communication

When to Use:

- **Unbuffered:** When you need strict synchronization, request-response patterns, or guaranteed handoff
- **Buffered:** When you want to decouple producer/consumer speeds, handle burst traffic, or implement worker pools with task queues

3. How does Go's garbage collector work? What are the trade-offs of its design?

Go uses a concurrent, tri-color mark-and-sweep garbage collector.

How It Works:

- **Mark Phase:** Identifies live objects using tri-color marking (white=unscanned, grey=scanned but references not processed, black=fully processed)
- **Sweep Phase:** Reclaims memory from unmarked (white) objects
- **Concurrent Execution:** GC runs concurrently with application goroutines, minimizing stop-the-world pauses
- **Write Barriers:** Tracks pointer modifications during concurrent marking

Key Features:

- Target pause time: <1ms (sub-millisecond latency)
- Pacer adjusts GC frequency based on heap growth
- GOGC environment variable controls GC aggressiveness (default: 100%)

Trade-offs:

- **Pros:** Low latency, predictable pauses, good for services requiring consistent response times
- **Cons:** Higher CPU overhead (~25% for GC), less throughput than generational GCs, not optimized for short-lived object patterns
- **Memory overhead:** Requires extra CPU to maintain low pause times

4. Explain interface satisfaction in Go. What is the empty interface and when should it be used?

Interface Satisfaction: Go uses structural typing (duck typing). A type satisfies an interface by implementing all its methods—no explicit declaration needed.

```
type Writer interface {
    Write([]byte) (int, error)
}
// Any type with Write method satisfies Writer
type MyWriter struct{}
func (m MyWriter) Write(p []byte) (int, error) {
    return len(p), nil
}
```

Empty Interface:

interface{} (or **any** in Go 1.18+) is satisfied by all types—it has zero methods.

```
func PrintAny(v interface{}) {
    fmt.Println(v)
}
```

When to Use Empty Interface:

- **Appropriate:** Generic containers (before generics), JSON unmarshaling, reflection-based libraries, truly polymorphic functions
- **Avoid:** When specific types are known—loses type safety and requires type assertions/switches
- **Better alternatives:** Use generics (Go 1.18+) for type-safe generic code, or define specific interfaces

Type Assertion:

```
val, ok := v.(string)
if !ok { /* handle type mismatch */ }
```

5. What are defer, panic, and recover? How do they interact and what are best practices?

defer:

Schedules a function call to execute after the surrounding function returns (LIFO order).

```
func example() {
    defer fmt.Println("third")
    defer fmt.Println("second")
    fmt.Println("first")
```

```
    // Output: first, second, third
}
```

Use cases: Resource cleanup (files, locks, connections), unlock mutexes, recover from panics.

panic:

Stops normal execution flow, begins unwinding the stack, executing deferred functions.

```
panic("critical error")
```

recover:

Regains control after a panic—only works inside deferred functions.

```
func safe() {
    defer func() {
        if r := recover(); r != nil {
            fmt.Println("Recovered:", r)
        }
    }()
    panic("oops")
}
```

Best Practices:

- **Don't use panic for normal errors:** Return error values instead
- **Panic only for:** Unrecoverable errors, programmer mistakes, initialization failures
- **Recover at boundaries:** HTTP handlers, goroutine tops to prevent crashes
- **Defer pitfalls:** Arguments are evaluated immediately, not when deferred function runs

6. How does Go handle memory allocation? Explain stack vs heap allocation and escape analysis.

Go automatically decides whether to allocate on stack or heap using escape analysis.

Stack Allocation:

- Fast allocation/deallocation (just move stack pointer)
- Automatic cleanup when function returns
- Limited size (~1MB default, growable)
- Used for variables that don't escape function scope

Heap Allocation:

- Slower allocation (involves memory allocator)
- Requires garbage collection
- Used when variables escape function scope

Escape Analysis:

Compiler determines if a variable's lifetime extends beyond function scope.

```
func noEscape() int {
    x := 42  // stack allocated
    return x
}

func escapes() *int {
    x := 42  // heap allocated (escapes)
    return &x
}
```

Common Escape Scenarios:

- Returning pointers to local variables
- Storing pointers in global variables or heap-allocated structures
- Sending pointers through channels

- Variables too large for stack
- Variables with dynamic size (slices, maps)

Check with: go build -gcflags='-m' to see escape analysis decisions.

7. Explain the differences between sync.Mutex and sync.RWMutex. When should each be used?

sync.Mutex:

Provides exclusive access—only one goroutine can hold the lock.

```
var mu sync.Mutex
mu.Lock()
defer mu.Unlock()
// critical section
```

Use when: Modifying shared state, or when reads and writes occur with similar frequency.

sync.RWMutex:

Allows multiple concurrent readers OR one exclusive writer.

```
var mu sync.RWMutex
// For reads:
mu.RLock()
defer mu.RUnlock()

// For writes:
mu.Lock()
defer mu.Unlock()
```

Key Differences:

- **Concurrency:** RWMutex allows multiple RLock holders simultaneously; Mutex allows only one Lock holder
- **Performance:** RWMutex has higher overhead than Mutex
- **Writer priority:** RWMutex may starve readers if writers are frequent

When to Use RWMutex:

- Read-heavy workloads (reads >> writes)
- Critical sections are short
- Need to scale read throughput

Benchmark Before Choosing:

RWMutex only improves performance when read operations significantly outnumber writes (typically 10:1 or higher ratio). For balanced read/write workloads, sync.Mutex is often faster due to lower overhead.

8. What are Go's race conditions and how do you detect them? Explain the race detector's limitations.

Race conditions occur when multiple goroutines access shared memory concurrently and at least one access is a write.

Example Race Condition:

```
var counter int
go func() { counter++ }()  // write
go func() { fmt.Println(counter) }()  // read
// Undefined behavior!
```

Detection with Race Detector:

```
go test -race
go run -race main.go
```

```
go build -race
```

The race detector instruments memory accesses at runtime and detects unsynchronized access to shared variables.

How It Works:

- Uses ThreadSanitizer (TSan) technology
- Tracks happens-before relationships
- Reports races when conflicting accesses lack synchronization

Limitations:

- **Runtime only:** Only detects races in executed code paths—requires good test coverage
- **Performance:** ~10x slowdown, 10x memory overhead—not for production
- **False negatives:** May miss races in unexecuted code or timing-dependent scenarios
- **No false positives:** Reported races are real issues
- **Doesn't catch:** Logical race conditions or atomicity violations across multiple operations

Best Practices:

Run race detector in CI/CD, use channels or sync primitives, follow "share memory by communicating" principle.

9. How do context.Context work in Go? Explain cancellation, deadlines, and value propagation.

context.Context manages cancellation signals, deadlines, and request-scoped values across API boundaries and goroutines.

Core Functions:

```
ctx := context.Background()  // root
ctx, cancel := context.WithCancel(ctx)
defer cancel()

ctx, cancel := context.WithTimeout(ctx, 5*time.Second)
ctx, cancel := context.WithDeadline(ctx, time.Now().Add(5*time.Second))
```

Cancellation:

```
select {
case <-ctx.Done():
    return ctx.Err()  // context.Canceled or context.DeadlineExceeded
case result := <-ch:
    return result
}
```

Value Propagation:

```
ctx = context.WithValue(ctx, keyUser, "alice")
user := ctx.Value(keyUser).(string)
```

Best Practices:

- **First parameter:** Always pass context as first parameter named ctx
- **Don't store contexts:** Pass them explicitly through call chains
- **Always cancel:** Call cancel() to release resources (use defer)
- **Values for request-scoped data only:** Auth tokens, trace IDs—not for optional parameters
- **Check Done() channel:** In long-running operations and loops

Common Patterns:

HTTP handlers receive context, database queries accept context for cancellation, goroutines should respect context cancellation.

10. Explain Go's slice internals. What happens during append operations and how can you optimize slice usage?

A slice is a descriptor containing a pointer to underlying array, length, and capacity.

Internal Structure:

```
type slice struct {
    array unsafe.Pointer  // pointer to array
    len   int             // number of elements
    cap   int             // capacity of array
}
```

Append Behavior:

- If **len < cap:** Element added to existing array, len incremented
- If **len == cap:** New array allocated (typically 2x capacity), elements copied, new element added

```
s := make([]int, 0, 3)
s = append(s, 1)  // no allocation
s = append(s, 2, 3)  // no allocation
s = append(s, 4)  // allocates new array, copies data
```

Optimization Strategies:

- **Pre-allocate with capacity:** s := make([]int, 0, expectedSize) avoids reallocations
- **Use full slice expression:** s[low:high:max] to limit capacity and prevent unintended sharing
- **Avoid slice leaks:** When keeping small portion of large slice, copy to new slice
- **Reuse slices:** Reset length to 0 instead of creating new slices in loops

Pitfalls:

Slicing doesn't copy—multiple slices can share underlying array. Modifying one affects others if they overlap.

Data Structures and Algorithms

Questions in this section test your understanding of how to work with and manipulate data efficiently.

1. How do you implement a thread-safe LRU cache in Go?

Thread-Safe LRU Cache Implementation

An **LRU (Least Recently Used) cache** requires a combination of a doubly linked list and a hash map for O(1) operations. For thread-safety, wrap operations with a mutex.

```
type LRUCache struct {
    capacity int
    cache    map[int]*list.Element
    list     *list.List
    mu       sync.Mutex
}

func (c *LRUCache) Get(key int) int {
    c.mu.Lock()
    defer c.mu.Unlock()
    if node, ok := c.cache[key]; ok {
        c.list.MoveToFront(node)
        return node.Value.(pair).value
    }
    return -1
}
```

- **Time Complexity:** O(1) for Get and Put operations
- **Space Complexity:** O(capacity)
- Use **sync.Mutex** for concurrent access protection

2. Implement a stack that supports GetMin() in O(1) time complexity.

Min Stack Implementation

Use two stacks: one for regular values and another to track minimums at each level.

```
type MinStack struct {
    stack    []int
    minStack []int
}

func (s *MinStack) Push(val int) {
    s.stack = append(s.stack, val)
    if len(s.minStack) == 0 || val <= s.GetMin() {
        s.minStack = append(s.minStack, val)
    }
}
```

- **Push:** O(1) - append to both stacks conditionally
- **Pop:** O(1) - remove from both if value equals current min
- **GetMin:** O(1) - return top of minStack
- **Space trade-off:** Uses O(n) extra space in worst case

3. How do you find all pairs in a slice that sum to a target value?

Two Sum - All Pairs Solution

Use a **hash map** to track seen numbers and their indices for O(n) time complexity.

```
func findPairs(nums []int, target int) [][2]int {
    seen := make(map[int]int)
    var pairs [][2]int
    for i, num := range nums {
        complement := target - num
        if idx, ok := seen[complement]; ok {
            pairs = append(pairs, [2]int{idx, i})
        }
        seen[num] = i
    }
    return pairs
}
```

- **Time Complexity:** O(n) - single pass through array
- **Space Complexity:** O(n) - hash map storage
- For sorted arrays, use **two-pointer technique** for O(1) space

4. Implement a Trie (prefix tree) with insert, search, and startsWith methods.

Trie Data Structure

A **Trie** is efficient for prefix-based searches, autocomplete, and dictionary implementations.

```
type TrieNode struct {
    children map[rune]*TrieNode
    isEnd    bool
}

type Trie struct {
    root *TrieNode
}

func (t *Trie) Insert(word string) {
    node := t.root
    for _, ch := range word {
        if node.children[ch] == nil {
            node.children[ch] = &TrieNode{children: make(map[rune]*TrieNode)}
        }
        node = node.children[ch]
    }
    node.isEnd = true
}
```

- **Insert:** O(m) where m is word length
- **Search:** O(m) for exact match
- **StartsWith:** O(m) for prefix check
- **Space:** O(ALPHABET_SIZE * N * M) worst case

5. How do you implement a sliding window maximum for an array?

Sliding Window Maximum Using Deque

Use a **monotonic deque** to maintain indices of potential maximum elements in decreasing order.

```
func maxSlidingWindow(nums []int, k int) []int {
    deque := []int{}
    result := []int{}
    for i, num := range nums {
        for len(deque) > 0 && nums[deque[len(deque)-1]] < num {
            deque = deque[:len(deque)-1]
        }
        deque = append(deque, i)
        if deque[0] <= i-k {
            deque = deque[1:]
        }
        if i >= k-1 {
            result = append(result, nums[deque[0]])
        }
```

```
    }
    return result
}
```

- **Time Complexity:** O(n) - each element added/removed once
- **Space Complexity:** O(k) - deque size bounded by window
- Deque stores **indices**, not values

6. Implement a binary heap (priority queue) from scratch in Go.

Min Heap Implementation

A **binary heap** maintains heap property where parent is smaller than children (min heap).

```
type MinHeap []int

func (h *MinHeap) Push(x interface{}) {
    *h = append(*h, x.(int))
    h.heapifyUp(len(*h) - 1)
}

func (h *MinHeap) heapifyUp(i int) {
    for i > 0 {
        parent := (i - 1) / 2
        if (*h)[parent] <= (*h)[i] {
            break
        }
        (*h)[parent], (*h)[i] = (*h)[i], (*h)[parent]
        i = parent
    }
}
```

- **Push:** O(log n) - heapify up operation
- **Pop:** O(log n) - heapify down operation
- **Peek:** O(1) - access root element
- Go's **container/heap** package provides standard interface

7. How do you detect a cycle in a linked list and find the cycle start node?

Floyd's Cycle Detection Algorithm

Use **two pointers** (slow and fast) to detect cycles, then find the entry point.

```
func detectCycle(head *ListNode) *ListNode {
    slow, fast := head, head
    for fast != nil && fast.Next != nil {
        slow = slow.Next
        fast = fast.Next.Next
        if slow == fast {
            slow = head
            for slow != fast {
                slow = slow.Next
                fast = fast.Next
            }
            return slow
        }
    }
    return nil
}
```

- **Time Complexity:** O(n) - at most 2n iterations
- **Space Complexity:** O(1) - only two pointers
- **Phase 1:** Detect cycle existence
- **Phase 2:** Find cycle entry point by resetting one pointer

8. Implement a thread-safe bounded blocking queue in Go.

Bounded Blocking Queue

Use **channels** with buffering for natural blocking behavior and thread-safety.

```
type BlockingQueue struct {
    queue chan interface{}
}

func NewBlockingQueue(capacity int) *BlockingQueue {
    return &BlockingQueue{queue: make(chan interface{}, capacity)}
}

func (q *BlockingQueue) Put(item interface{}) {
    q.queue <- item
}

func (q *BlockingQueue) Take() interface{} {
    return <-q.queue
}
```

- **Channels** provide built-in blocking and synchronization
- **Put** blocks when queue is full
- **Take** blocks when queue is empty
- No explicit mutex needed - channels are thread-safe
- Use **context** for timeout operations

9. How do you merge K sorted linked lists efficiently?

Merge K Sorted Lists Using Min Heap

Use a **min heap** to efficiently select the smallest element among K lists.

```
type ListHeap []*ListNode

func mergeKLists(lists []*ListNode) *ListNode {
    h := &ListHeap{}
    heap.Init(h)
    for _, node := range lists {
        if node != nil {
            heap.Push(h, node)
        }
    }
    dummy := &ListNode{}
    curr := dummy
    for h.Len() > 0 {
        node := heap.Pop(h).(*ListNode)
        curr.Next = node
        curr = curr.Next
        if node.Next != nil {
            heap.Push(h, node.Next)
        }
    }
    return dummy.Next
}
```

- **Time Complexity:** O(N log K) where N is total nodes
- **Space Complexity:** O(K) for heap
- Alternative: **Divide and conquer** approach with O(log K) merges

10. Implement a data structure that supports insert, delete, and getRandom in O(1) average time.

RandomizedSet Implementation

Combine a **slice** for random access and a **map** for O(1) lookups.

```
type RandomizedSet struct {
    nums []int
    pos  map[int]int
}
```

```
func (s *RandomizedSet) Insert(val int) bool {
    if _, ok := s.pos[val]; ok {
        return false
    }
    s.pos[val] = len(s.nums)
    s.nums = append(s.nums, val)
    return true
}

func (s *RandomizedSet) Remove(val int) bool {
    if idx, ok := s.pos[val]; ok {
        last := s.nums[len(s.nums)-1]
        s.nums[idx] = last
        s.pos[last] = idx
        s.nums = s.nums[:len(s.nums)-1]
        delete(s.pos, val)
        return true
    }
    return false
}
```

- **Insert:** O(1) - append to slice and update map
- **Delete:** O(1) - swap with last element and remove
- **GetRandom:** O(1) - use rand.Intn(len(nums))
- Map tracks value-to-index mapping

System Design

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

1. Design a scalable URL shortener service like bit.ly. What are the key components and how would you handle high traffic?

Key Components

- **API Gateway:** Handle incoming requests for URL shortening and redirection
- **Application Servers:** Stateless Go services for business logic
- **Database:** Store URL mappings (original URL → short code)
- **Cache Layer:** Redis/Memcached for frequently accessed URLs
- **Load Balancer:** Distribute traffic across application servers

Architecture Approach

URL Generation: Use base62 encoding of auto-incrementing ID or hash-based approach with collision handling.

```
func generateShortURL(id uint64) string {
    const base62 = "0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
    encoded := ""
    for id > 0 {
        encoded = string(base62[id%62]) + encoded
        id /= 62
    }
    return encoded
}
```

Database Choice: Use PostgreSQL or MySQL for persistence with indexed short_code column. Consider sharding by hash of short code for horizontal scaling.

Caching Strategy: Cache popular URLs with TTL. Implement cache-aside pattern where reads check cache first, then DB.

High Availability: Multi-region deployment with geo-DNS routing. Use master-slave replication for reads. Write to master, read from replicas.

Rate Limiting: Implement token bucket algorithm per IP/user to prevent abuse.

Analytics: Use message queue (Kafka) to async process click events for analytics without blocking redirects.

2. How would you design a real-time chat system supporting millions of concurrent users? Discuss WebSocket management, message delivery, and scalability.

Architecture Overview

- **WebSocket Gateway:** Go servers managing WebSocket connections
- **Message Broker:** Kafka or RabbitMQ for message routing
- **Presence Service:** Track online/offline status
- **Message Storage:** Cassandra or MongoDB for chat history
- **Redis:** For user session and connection mapping

Connection Management

```
type ConnectionManager struct {
    connections map[string]*websocket.Conn
    mu sync.RWMutex
```

```
}

func (cm *ConnectionManager) AddConn(userID string, conn *websocket.Conn) {
    cm.mu.Lock()
    defer cm.mu.Unlock()
    cm.connections[userID] = conn
}
```

Scalability Strategy:

- Horizontal scaling of WebSocket servers behind load balancer with sticky sessions
- Store user-to-server mapping in Redis for message routing
- Use consistent hashing to distribute users across servers

Message Delivery: Implement at-least-once delivery with message acknowledgments. Use message IDs and store undelivered messages in queue.

Presence Service: Use Redis with TTL for heartbeat mechanism. WebSocket servers update presence every 30 seconds.

Group Chat: Use pub/sub pattern with Redis or dedicated message broker. Each room is a topic.

Message Persistence: Async write to Cassandra partitioned by conversation_id and timestamp for efficient range queries.

CAP Theorem: Favor availability and partition tolerance (AP). Accept eventual consistency for message ordering across partitions.

3. Design a distributed rate limiter that can be used across multiple services. How would you implement it in Go?

Design Approaches

1. Token Bucket Algorithm: Most flexible, allows bursts
2. Sliding Window Log: Accurate but memory intensive
3. Sliding Window Counter: Balance of accuracy and efficiency

Distributed Implementation Using Redis

```
type RateLimiter struct {
    redisClient *redis.Client
    limit int
    window time.Duration
}

func (rl *RateLimiter) Allow(key string) bool {
    now := time.Now().Unix()
    pipe := rl.redisClient.Pipeline()
    pipe.ZRemRangeByScore(ctx, key, "0", fmt.Sprint(now-int64(rl.window.Seconds())))
    pipe.ZCard(ctx, key)
    _, err := pipe.Exec(ctx)
}
```

Architecture Components

- **Redis Cluster:** Centralized state for rate limit counters
- **Lua Scripts:** Atomic operations to check and increment counters
- **Local Cache:** Short-lived cache to reduce Redis calls

Sliding Window Counter Implementation: Divide time into fixed windows. Use weighted count from current and previous window for smoother limiting.

Token Bucket with Redis:

```
local tokens = redis.call('get', KEYS[1])
if tokens == false then
  tokens = ARGV[1]
end
```

```
if tonumber(tokens) >= 1 then
  redis.call('decr', KEYS[1])
  return 1
end
return 0
```

Scalability: Shard by user ID across Redis cluster. Use Redis Cluster for automatic sharding and failover.

Fallback Strategy: If Redis is down, implement local in-memory rate limiting or fail-open based on business requirements.

Multi-tier Limiting: Support per-user, per-IP, and global rate limits with different keys and limits.

4. Design a news feed system like Twitter or Facebook. How would you handle feed generation, ranking, and real-time updates?

Feed Generation Strategies

1. Fan-out on Write (Push Model): Pre-compute feeds when post is created
2. Fan-out on Read (Pull Model): Compute feed when user requests
3. Hybrid Approach: Push for most users, pull for celebrities

Architecture Components

- **Post Service:** Handle post creation and storage
- **Feed Service:** Generate and serve user feeds
- **Graph Service:** Manage follower/following relationships
- **Ranking Service:** Score and order feed items
- **Cache Layer:** Redis for pre-computed feeds

Feed Generation in Go

```
type FeedGenerator struct {
    postStore PostStore
    graphStore GraphStore
    cache Cache
}

func (fg *FeedGenerator) GetFeed(userID string, limit int) []Post {
    following := fg.graphStore.GetFollowing(userID)
    posts := fg.postStore.GetRecentPosts(following, limit*2)
    return fg.rankPosts(posts)[:limit]
}
```

Fan-out on Write: When user posts, async workers fetch followers and write post ID to their feed cache (Redis list). Scales well for users with moderate followers.

Hybrid Approach: Users with >1M followers use pull model. Regular users use push model. Detect celebrities and handle differently.

Ranking Algorithm: Score based on engagement (likes, comments), recency, relationship strength. Use ML model for personalization.

Real-time Updates: Use WebSockets or Server-Sent Events. When new post arrives, push notification to online followers.

Database Design: Cassandra for posts (partition by user_id, sort by timestamp). Redis sorted sets for feeds (score = timestamp or rank).

Scalability: Partition users across multiple feed services. Use message queue (Kafka) for async fan-out processing.

5. How would you design a distributed caching system? Discuss cache invalidation strategies, consistency, and handling cache stampede.

Cache Architecture

- **Application-level Cache:** In-memory Go cache per service instance
- **Distributed Cache:** Redis Cluster or Memcached
- **CDN:** For static assets and edge caching

Cache Patterns

1. Cache-Aside: App checks cache, fetches from DB on miss, updates cache
2. Write-Through: Write to cache and DB synchronously
3. Write-Behind: Write to cache, async update DB

```
type CacheService struct {
    cache *redis.Client
    db Database
}

func (cs *CacheService) Get(key string) (interface{}, error) {
    val, err := cs.cache.Get(ctx, key).Result()
    if err == redis.Nil {
        val, err = cs.db.Query(key)
        cs.cache.Set(ctx, key, val, time.Hour)
    }
    return val, err
}
```

Cache Invalidation Strategies

1. TTL-based: Set expiration time on all cache entries
2. Event-based: Invalidate on data updates via pub/sub
3. Version-based: Include version in cache key

Cache Stampede Prevention: When cache expires and multiple requests hit DB simultaneously:

```
func (cs *CacheService) GetWithLock(key string) (interface{}, error) {
    lock := cs.acquireLock(key)
    defer lock.Release()
    val, err := cs.cache.Get(ctx, key).Result()
    if err == redis.Nil {
        val, err = cs.db.Query(key)
        cs.cache.Set(ctx, key, val, time.Hour)
    }
    return val, err
}
```

Probabilistic Early Expiration: Refresh cache before expiry with probability inversely proportional to remaining TTL.

Consistency: Use cache versioning or two-phase commit for strong consistency. Accept eventual consistency for better performance.

Eviction Policies: LRU for memory-constrained scenarios. Consider LFU for frequency-based eviction.

6. Design a distributed task scheduler that can execute millions of tasks reliably. How would you handle task persistence, execution, and failure recovery?

System Components

- **Task Queue:** Kafka or RabbitMQ for task distribution
- **Worker Pool:** Go workers consuming from queue
- **Task Store:** PostgreSQL for task metadata and state
- **Scheduler:** Service for time-based task triggering
- **Coordinator:** Distributed consensus using etcd or Consul

Worker Implementation

```
type TaskWorker struct {
    queue Queue
```

```
    executor TaskExecutor
}

func (tw *TaskWorker) Start(concurrency int) {
    for i := 0; i < concurrency; i++ {
        go func() {
            for task := range tw.queue.Consume() {
                tw.executeWithRetry(task)
            }
        }()
    }
}
```

Task Persistence: Store task definition, schedule, status, and result in PostgreSQL. Use task_id as primary key with indexes on status and scheduled_time.

Scheduling: Use cron-like expressions. Scheduler service polls DB for due tasks and publishes to queue. Use database-level locking to prevent duplicate scheduling:

```
UPDATE tasks SET status='queued', locked_by='scheduler-1'
WHERE scheduled_time <= NOW() AND status='pending'
LIMIT 1000 FOR UPDATE SKIP LOCKED
```

Execution Guarantees: At-least-once delivery with idempotent task handlers. Store execution results with task_id to detect duplicates.

Failure Recovery: Workers send heartbeats. Coordinator detects dead workers and requeues their tasks. Use visibility timeout on messages.

Retry Strategy: Exponential backoff with max attempts. Dead letter queue for permanently failed tasks.

Scalability: Horizontal scaling of workers. Partition tasks by task_type or priority. Use priority queues for different task classes.

Monitoring: Track queue depth, task latency, failure rates. Alert on anomalies.

7. Design a video streaming platform like YouTube. Focus on video upload, transcoding, storage, and delivery at scale.

System Architecture

- **Upload Service:** Handle multipart uploads with resumability
- **Transcoding Pipeline:** Convert videos to multiple formats/resolutions
- **Storage:** Object storage (S3) for video files
- **CDN:** CloudFront or Akamai for content delivery
- **Metadata Store:** PostgreSQL for video metadata
- **Streaming Service:** Serve video segments via HLS/DASH

Upload Handling in Go

```
func (us *UploadService) HandleUpload(w http.ResponseWriter, r *http.Request) {
    file, header, _ := r.FormFile("video")
    uploadID := generateUploadID()

    go func() {
        us.storage.Upload(uploadID, file)
        us.queue.Publish(TranscodeTask{VideoID: uploadID})
    }()

    json.NewEncoder(w).Encode(map[string]string{"upload_id": uploadID})
}
```

Transcoding Pipeline: Use message queue (Kafka) to distribute transcoding jobs. Workers use FFmpeg to create multiple resolutions (1080p, 720p, 480p, 360p) and formats (HLS, DASH).

Adaptive Bitrate Streaming: Generate HLS playlist with multiple quality variants. Client adapts based on bandwidth:

```
#EXTM3U
#EXT-X-STREAM-INF:BANDWIDTH=2000000,RESOLUTION=1920x1080
1080p.m3u8
#EXT-X-STREAM-INF:BANDWIDTH=1000000,RESOLUTION=1280x720
720p.m3u8
```

Storage Strategy: Store original video in S3 with lifecycle policy to move to Glacier after 90 days. Store transcoded segments in S3 with CDN caching.

Content Delivery: Use CDN with origin shield. Implement signed URLs for private videos with expiration.

Scalability: Horizontal scaling of upload and transcoding services. Use S3 multipart upload for large files. Partition metadata by video_id.

Monitoring: Track upload success rate, transcoding time, CDN hit ratio, buffering events.

8. How would you design a distributed search engine? Discuss indexing, query processing, ranking, and handling billions of documents.

Architecture Components

- **Crawler:** Fetch and parse web pages
- **Indexer:** Build inverted index from documents
- **Query Processor:** Parse and execute search queries
- **Ranker:** Score and order results
- **Storage:** Distributed index storage (Elasticsearch or custom)

Inverted Index Structure

Term → [DocID1, DocID2, ...] with positions and frequencies

```
type InvertedIndex struct {
    index map[string][]Posting
    mu sync.RWMutex
}

type Posting struct {
    DocID uint64
    Positions []int
    Frequency int
}
```

Indexing Pipeline: Crawler feeds documents to message queue. Indexer workers consume, tokenize, remove stop words, stem, and build index. Batch updates to reduce I/O.

Distributed Index: Partition index by term hash across multiple shards. Each shard is replicated for availability. Use consistent hashing for shard assignment.

Query Processing:

```
func (qp *QueryProcessor) Search(query string) []Document {
    terms := tokenize(query)
    postingsLists := qp.fetchPostings(terms)
    docIDs := intersect(postingsLists)
    docs := qp.fetchDocuments(docIDs)
    return qp.rank(docs, terms)
}
```

Ranking Algorithm: TF-IDF for basic relevance. PageRank for authority. Machine learning models for personalization. Consider query intent, document freshness, user location.

Performance Optimization: Cache popular queries. Use skip lists in posting lists for faster intersection. Implement early termination for top-k results.

Scalability: Horizontal scaling of query processors. Partition by query hash for cache locality. Use scatter-gather pattern for distributed search.

Real-time Indexing: Use incremental indexing with merge strategy. New documents go to smaller index, periodically merged with main index.

9. Design a ride-sharing platform like Uber. Focus on matching drivers with riders, location tracking, and handling surge pricing.

Core Services

- **Location Service:** Track real-time driver/rider locations
- **Matching Service:** Match riders with nearby drivers
- **Trip Service:** Manage trip lifecycle
- **Pricing Service:** Calculate fares with surge pricing
- **Payment Service:** Handle transactions
- **Notification Service:** Real-time updates via WebSocket/push

Location Tracking

```
type LocationService struct {
    redis *redis.Client
}

func (ls *LocationService) UpdateLocation(driverID string, lat, lng float64) {
    ls.redis.GeoAdd(ctx, "drivers", &redis.GeoLocation{
        Name: driverID,
        Latitude: lat,
        Longitude: lng,
    })
}
```

Geospatial Indexing: Use Redis GeoHash or QuadTree for efficient proximity search. Drivers update location every 4 seconds.

Matching Algorithm:

```
func (ms *MatchingService) FindDriver(riderLat, riderLng float64) string {
    nearby := ms.locationService.GetNearbyDrivers(riderLat, riderLng, 5.0)
    available := ms.filterAvailable(nearby)
    return ms.selectBest(available)
}
```

Select based on: distance, driver rating, acceptance rate, ETA. Use scoring function to balance factors.

Surge Pricing: Calculate supply/demand ratio in geohash cells. Increase multiplier when demand exceeds supply:

```
surgeMultiplier = 1.0 + (demandCount - supplyCount) / supplyCount * 0.5
```

Trip State Machine: requested → matched → accepted → arrived → started → completed → paid. Use database transactions for state transitions.

Scalability: Partition by geographic region (city/zone). Each region has dedicated matching service instance. Use event sourcing for trip history.

Consistency: Use distributed locks (Redis) to prevent double-matching. Implement timeout and retry for driver acceptance.

Real-time Updates: WebSocket connections for live location sharing. Publish trip events to message broker for analytics.

10. Design a distributed logging and monitoring system that can handle millions of log events per second. Discuss ingestion, storage, querying, and alerting.

System Architecture

- **Log Collectors:** Agents on each service instance
- **Ingestion Pipeline:** Kafka for buffering and distribution
- **Processing:** Stream processors for parsing and enrichment
- **Storage:** Elasticsearch for indexing, S3 for archival

- **Query API:** Service for log search and aggregation
- **Alerting:** Rule engine for threshold-based alerts

Log Ingestion in Go

```
type LogCollector struct {
    producer *kafka.Producer
    buffer chan LogEntry
}

func (lc *LogCollector) Collect(entry LogEntry) {
    lc.buffer <- entry
}

func (lc *LogCollector) Flush() {
    for entry := range lc.buffer {
        lc.producer.Produce("logs", entry)
    }
}
```

Ingestion Strategy: Batch logs locally before sending. Use async I/O. Compress with gzip. Handle backpressure with local disk buffering.

Stream Processing: Kafka Streams or Flink for real-time parsing. Extract structured fields (timestamp, level, service, message). Enrich with metadata (region, environment).

Storage Strategy: Hot data (last 7 days) in Elasticsearch for fast queries. Warm data (7-30 days) in cheaper ES tier. Cold data (>30 days) in S3 with Parquet format.

Indexing: Create time-based indices (logs-2024-01-15). Use index templates for consistent mapping. Shard by time and service for parallel queries.

Query Optimization:

```
func (qs *QueryService) Search(query string, from, to time.Time) []LogEntry {
    indices := qs.getIndicesForRange(from, to)
    return qs.es.Search(indices, query)
}
```

Use index patterns to limit search scope. Implement query caching for repeated searches.

Alerting: Define rules on metrics derived from logs (error rate, latency p99). Evaluate rules in real-time stream processor. Use time windows and thresholds.

Scalability: Partition Kafka topics by service or log level. Scale ES cluster horizontally. Use read replicas for query load.

Coding and Debugging

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

1. Write a Go function to reverse a string that properly handles Unicode characters.

Solution

In Go, strings are UTF-8 encoded byte sequences. To properly reverse Unicode characters, convert the string to a **rune slice**:

```
func reverseString(s string) string {
    runes := []rune(s)
    for i, j := 0, len(runes)-1; i < j; i, j = i+1, j-1 {
        runes[i], runes[j] = runes[j], runes[i]
    }
    return string(runes)
}
```

Key points:

- Converting to **[]rune** ensures proper handling of multi-byte Unicode characters
- Two-pointer technique swaps characters from both ends
- Time complexity: O(n), Space complexity: O(n)

2. How do you detect and fix a goroutine leak in a Go application?

Detection Methods

- **runtime.NumGoroutine()**: Monitor goroutine count over time
- **pprof goroutine profile**: Use import _ "net/http/pprof" and access /debug/pprof/goroutine
- **GODEBUG=schedtrace**: Track scheduler statistics

Common Causes & Fixes

- **Blocked channel operations**: Always use timeouts with select statements
- **Missing context cancellation**: Pass context.Context and listen for Done() signals
- **Infinite loops without exit conditions**: Ensure goroutines have termination logic

```
func worker(ctx context.Context) {
    for {
        select {
        case <-ctx.Done():
            return
        case work := <-workChan:
            process(work)
        }
    }
}
```

3. Implement a function to flatten a nested slice of integers in Go.

Recursive Solution

Since Go doesn't have generics for arbitrary nested structures, we use **interface{}** with type assertions:

```
func flatten(input interface{}) []int {
    result := []int{}
    switch v := input.(type) {
    case int:
```

```
        result = append(result, v)
    case []interface{}:
        for _, item := range v {
            result = append(result, flatten(item)...)
        }
    }
    return result
}
```

Usage: flatten([]interface{}{1, []interface{}{2, 3}, 4}) returns [1, 2, 3, 4]

Note: With Go 1.18+ generics, you can create type-safe versions for specific nested slice depths.

4. What tools and techniques do you use for memory profiling in Go?

Primary Tools

- **pprof**: Built-in profiling tool for heap, CPU, and goroutine analysis
- **go tool trace**: Visualize execution traces and GC events
- **runtime/metrics**: Programmatic access to runtime statistics

Profiling Workflow

```
import _ "net/http/pprof"
import "runtime/pprof"

// Heap profile
f, _ := os.Create("heap.prof")
pprof.WriteHeapProfile(f)
f.Close()

// Analyze: go tool pprof heap.prof
```

Key Metrics to Monitor

- **Alloc**: Bytes allocated and still in use
- **TotalAlloc**: Cumulative bytes allocated
- **Sys**: Total memory obtained from OS
- **NumGC**: Number of completed GC cycles

5. Write a concurrent-safe LRU cache implementation in Go.

Implementation with sync.Mutex

```
type LRUCache struct {
    capacity int
    cache    map[int]*list.Element
    list     *list.List
    mu       sync.Mutex
}

func (c *LRUCache) Get(key int) int {
    c.mu.Lock()
    defer c.mu.Unlock()
    if elem, ok := c.cache[key]; ok {
        c.list.MoveToFront(elem)
        return elem.Value.(int)
    }
    return -1
}
```

Design considerations:

- Use **container/list** for O(1) move operations
- Combine map for O(1) lookups with doubly-linked list for ordering
- **sync.Mutex** protects concurrent access
- Consider **sync.Map** for read-heavy workloads

6. How do you handle panics and implement recovery in production Go code?

Recovery Pattern

```
func safeHandler(w http.ResponseWriter, r *http.Request) {
    defer func() {
        if err := recover(); err != nil {
            log.Printf("panic: %v\n%s", err, debug.Stack())
            http.Error(w, "Internal error", 500)
        }
    }()
    // handler logic
}
```

Best Practices

- **Don't recover from panics you can't handle**: Let the program crash for programmer errors
- **Use defer + recover only at boundaries**: HTTP handlers, goroutine entry points
- **Log stack traces**: Use runtime/debug.Stack() for diagnostics
- **Prefer errors over panics**: Reserve panics for truly exceptional situations
- **Never recover and continue silently**: Always log or report recovered panics

7. Explain race conditions in Go and how to detect them. Provide an example.

Race Condition Example

```
var counter int
func increment() {
    for i := 0; i < 1000; i++ {
        counter++ // RACE: non-atomic read-modify-write
    }
}
go increment()
go increment()
```

Detection

Run with the **race detector**: go run -race main.go or go test -race

Solutions

- **Mutex**: Use sync.Mutex for mutual exclusion
- **Atomic operations**: Use sync/atomic for simple counters
- **Channels**: Communicate through channels instead of shared memory

```
var counter int64
atomic.AddInt64(&counter, 1) // Thread-safe
```

8. Write a function to check if a string is a palindrome, optimized for performance.

Optimized Solution

```
func isPalindrome(s string) bool {
    runes := []rune(s)
    left, right := 0, len(runes)-1
    for left < right {
        if runes[left] != runes[right] {
            return false
        }
        left++
        right--
    }
    return true
}
```

Optimizations:

- **Single pass**: O(n/2) comparisons with two pointers

- **Early exit**: Returns false immediately on mismatch
- **Unicode-safe**: Converts to runes for proper character comparison
- **Space complexity**: O(n) for rune slice (unavoidable for Unicode)

For ASCII-only strings, iterate bytes directly: s[left] != s[right] to achieve O(1) space.

9. How do you profile CPU usage in a Go application and identify bottlenecks?

CPU Profiling Steps

```
import "runtime/pprof"

f, _ := os.Create("cpu.prof")
pprof.StartCPUProfile(f)
defer pprof.StopCPUProfile()

// Run your code here
```

Analysis

Analyze with: go tool pprof cpu.prof

Interactive commands:

- **top**: Show functions consuming most CPU time
- **list funcName**: Show source code with CPU samples
- **web**: Generate call graph visualization (requires Graphviz)
- **peek**: Show callers and callees

Production Profiling

Use net/http/pprof for live profiling: go tool pprof http://localhost:6060/debug/pprof/profile?seconds=30

10. Explain the difference between buffered and unbuffered channels, and when to use each.

Unbuffered Channels

```
ch := make(chan int)
go func() { ch <- 42 }() // Blocks until received
val := <-ch
```

Characteristics: Synchronous, sender blocks until receiver is ready (rendezvous)

Buffered Channels

```
ch := make(chan int, 3)
ch <- 1 // Doesn't block
ch <- 2
ch <- 3
ch <- 4 // Blocks: buffer full
```

Characteristics: Asynchronous until buffer is full

When to Use

- **Unbuffered**: Synchronization points, guaranteed handoff, request-response patterns
- **Buffered**: Producer-consumer with different rates, reducing goroutine blocking, worker pools
- **Caution**: Buffered channels can hide deadlocks and make reasoning harder

Behavioral Questions

These questions assess your soft skills, problem-solving approach, and how you work in a team.

1. Tell me about a time when you had to optimize a Go application that was experiencing performance issues.

Situation: Our microservice handling user authentication was experiencing high latency (>500ms) during peak hours, affecting 10,000+ concurrent users.

Task: I was responsible for identifying bottlenecks and reducing response time to under 100ms.

Action: I used pprof to profile the application and discovered excessive memory allocations in JSON marshaling. I implemented sync.Pool for reusing buffers, added caching with Redis for frequently accessed data, and optimized database queries by introducing connection pooling with appropriate limits. I also replaced reflection-heavy operations with code generation.

Result: Reduced average latency to 75ms, decreased memory usage by 40%, and improved throughput by 3x. The service handled peak loads without degradation.

2. Describe a situation where you had to debug a complex concurrency issue in Go.

Situation: Our data processing pipeline was experiencing random panics and data corruption in production, occurring sporadically under heavy load.

Task: I needed to identify and fix the race condition without causing downtime or data loss.

Action: I enabled the race detector in our staging environment and reproduced the issue with increased load. Discovered multiple goroutines were writing to a shared map without synchronization. I refactored the code to use channels for communication and implemented proper mutex locks where shared state was necessary. Added comprehensive unit tests with the -race flag and introduced context-based cancellation for graceful shutdowns.

Result: Eliminated all race conditions, achieved zero panics over the next 3 months in production, and established team guidelines for concurrent code reviews.

3. Can you share an example of when you had to make a difficult technical decision between two Go design patterns or approaches?

Situation: We were designing a new event processing system and debating between using channels with worker pools versus a callback-based approach with interfaces.

Task: I needed to evaluate both approaches and recommend the best solution for our scalability and maintainability requirements.

Action: I created proof-of-concept implementations for both approaches, benchmarked them under various load conditions, and evaluated code complexity, testability, and team familiarity. I documented trade-offs including performance characteristics, error handling patterns, and future extensibility. Presented findings to the team with metrics showing the channel-based approach had better throughput and clearer error propagation.

Result: Team adopted the channel-based worker pool pattern, which successfully processed 50M+ events daily with predictable performance and simplified our testing strategy.

4. Tell me about a time when you had to refactor legacy Go code to improve maintainability.

Situation: Inherited a 15,000-line monolithic Go service with no tests, deeply nested functions, and global state that was becoming impossible to maintain.

Task: Refactor the codebase to improve testability and maintainability while ensuring zero downtime and no regression bugs.

Action: I created a comprehensive test suite using table-driven tests to establish baseline behavior. Incrementally extracted business logic into smaller packages with clear interfaces, introduced dependency injection to eliminate global state, and applied the repository pattern for data access. Used feature flags to gradually roll out refactored components. Conducted code reviews with the team and created documentation for new patterns.

Result: Increased test coverage from 0% to 85%, reduced average function complexity by 60%, and decreased bug reports by 70% over the following quarter. The refactored codebase enabled the team to deliver features 2x faster.

5. Describe a situation where you had to mentor junior developers on Go best practices.

Situation: Three junior developers joined our team with limited Go experience, and their code reviews were taking excessive time due to common mistakes like improper error handling and goroutine leaks.

Task: I was asked to accelerate their onboarding and establish consistent coding standards across the team.

Action: I created a Go best practices guide specific to our codebase, conducted weekly code review sessions explaining idiomatic patterns, and set up pair programming sessions for complex features. Introduced static analysis tools (golangci-lint, staticcheck) in our CI pipeline with clear explanations for each rule. Created reusable code templates and examples for common patterns like graceful shutdowns, context propagation, and error wrapping.

Result: Junior developers became productive contributors within 6 weeks, code review time decreased by 50%, and the team adopted consistent patterns that reduced production issues by 40%.

6. Tell me about a time when you had to handle a critical production incident involving a Go service.

Situation: Our payment processing service written in Go started failing with timeout errors at 2 AM, affecting transaction processing worth $50K per hour.

Task: I was on-call and needed to quickly identify the root cause and restore service functionality.

Action: I immediately checked monitoring dashboards and logs, discovering goroutine count had grown to 100K+ indicating a goroutine leak. Identified that HTTP clients weren't setting timeouts, causing connections to hang indefinitely. Deployed a hotfix adding proper timeouts and context cancellation, then implemented graceful connection draining. Set up alerts for goroutine count thresholds and added circuit breaker patterns to prevent cascade failures.

Result: Restored service within 45 minutes, preventing $500K in potential lost revenue. Conducted a postmortem, updated our service template with proper timeout configurations, and implemented automated checks to prevent similar issues.

7. Describe a time when you had to integrate a Go service with a complex third-party API or legacy system.

Situation: We needed to integrate our Go-based order management system with a legacy SOAP-based inventory system that had inconsistent response formats and frequent timeouts.

Task: Build a reliable integration that could handle the legacy system's quirks while maintaining high availability for our service.

Action: I designed an adapter layer with retry logic using exponential backoff, implemented circuit breaker pattern to prevent cascade failures, and added comprehensive logging for debugging. Created a message queue (RabbitMQ) for asynchronous processing to decouple our system from the legacy one. Built thorough integration tests with mocked responses covering edge cases and implemented health checks to monitor integration status.

Result: Achieved 99.9% uptime despite legacy system instability, reduced integration-related incidents by 90%, and the architecture became a template for other legacy integrations across the organization.

8. Can you share an experience where you had to advocate for adopting Go in a project or organization?

Situation: Our team was building a new real-time analytics service, and there was debate between using Node.js (team's comfort zone) versus Go for better performance.

Task: I needed to present a compelling case for Go while addressing team concerns about learning curve and ecosystem maturity.

Action: I built a prototype in both languages demonstrating Go's superior performance (3x faster, 5x less memory) for our use case. Created a migration plan with training resources, identified team members interested in learning Go as champions, and addressed concerns about library availability by auditing required dependencies. Proposed a gradual adoption starting with new services while maintaining Node.js for existing ones.

Result: Management approved Go adoption. The analytics service handled 100K requests/second with minimal resources. Team proficiency increased within 2 months, and Go became the standard for new performance-critical services.

9. Tell me about a time when you had to balance technical debt with feature delivery in a Go project.

Situation: Our Go-based API gateway had accumulated significant technical debt (outdated dependencies, poor test coverage, complex routing logic) while pressure mounted to deliver new features quickly.

Task: I needed to address critical technical debt without derailing feature delivery timelines.

Action: I categorized technical debt by risk and impact, prioritizing security vulnerabilities and performance bottlenecks. Negotiated with product management to allocate 20% of each sprint to debt reduction. Implemented incremental improvements: updated critical dependencies, added tests for high-risk paths, and refactored the most problematic modules. Automated dependency updates with Dependabot and established a definition of done that included test coverage requirements.

Result: Reduced critical vulnerabilities to zero within 2 months, increased test coverage from 45% to 78%, and actually improved feature velocity by 25% due to fewer bugs and easier modifications.

10. Describe a situation where you had to design a Go service architecture to meet specific scalability requirements.

Situation: We needed to build a notification service in Go capable of handling 1M+ notifications per minute with delivery guarantees and minimal latency.

Task: Design and implement a horizontally scalable architecture that could meet these requirements while staying within budget constraints.

Action: I designed a microservices architecture using Go with message queues (Kafka) for reliable delivery, implemented worker pools with configurable concurrency, and used Redis for rate limiting and deduplication. Applied the bulkhead pattern to isolate different notification types (email, SMS, push), implemented graceful degradation with circuit breakers, and used Kubernetes for auto-scaling based on queue depth. Load tested the system to validate performance and tuned garbage collection parameters.

Result: Successfully deployed a system processing 1.5M notifications/minute with 99.95% delivery success rate and <100ms processing latency. The architecture scaled linearly with added instances and cost 40% less than initial estimates.