Member-only story

Building High-Performance LRU Cache in Go: From Theory to Production

Aman verma Follow 11 min read · Sep 7, 2025

3 1

How I went from a naive map-based cache to a production-ready LRU implementation that handles 100k+ requests per second

Two years ago, I made what seemed like a simple optimization to our API gateway. “Let’s just cache these database queries in a map,” I told my team during a sprint planning meeting. Six months later, that innocent map had grown to 8GB of memory usage and was the root cause of our most embarrassing production outage.

That’s when I learned the hard way why Least Recently Used (LRU) caches exist, and more importantly, how to implement them properly in Go. Today,

I'll share everything I've learned about building LRU caches that actually work in production.

The Problem That Started It All

Picture this: our e-commerce API was making the same expensive database queries repeatedly. User profiles, product catalogs, inventory counts — the same data fetched hundreds of times per minute. The solution seemed obvious:

```
// My first "brilliant" caching solution
var cache = make(map[string]interface{})
var mu sync.RWMutex
```

```
func GetFromCache(key string) (interface{}, bool) {
    mu.RLock()
    defer mu.RUnlock()
    value, exists := cache[key]
    return value, exists
}

func SetInCache(key string, value interface{}) {
    mu.Lock()
    defer mu.Unlock()
    cache[key] = value
}
```

This worked great in development. It even worked fine in staging with our modest test data. But production taught me a harsh lesson: **unbounded caches are memory leaks with extra steps.**

Within weeks, our API servers were consuming gigabytes of memory. Cache keys accumulated like digital hoarding, and we had no eviction strategy. The final straw came during Black Friday when our servers started getting OOM-

killed by Kubernetes. That's when I discovered the beauty of LRU (Least Recently Used) eviction policies.

Understanding LRU: The Algorithm That Saves Your Memory

LRU cache maintains a fixed-size cache by evicting the least recently used items when capacity is reached. It's like having a perfectly organized desk where you automatically put aside old papers when you run out of space.

The Core Operations

An efficient LRU cache needs to support:

1. **Get(key)** — Retrieve value and mark as recently used — O(1)
2. **Put(key, value)** — Insert/update and mark as recently used — O(1)
3. **Eviction** — Remove least recently used item when at capacity — O(1)

The trick is maintaining access order efficiently. After trying several approaches (and learning from my mistakes), I discovered the winning combination: **HashMap + Doubly Linked List.**

My Evolution: Three Implementations, Three Lessons

Attempt 1: The Slice-Based Disaster

My first "proper" LRU implementation used a slice to track access order:

```
type NaiveLRU struct {
    capacity int
    cache    map[string]interface{}
    order    []string // Track access order
```

```
    mu       sync.RWMutex
}
```

```
func (lru *NaiveLRU) Get(key string) (interface{}, bool) {
    lru.mu.Lock()
    defer lru.mu.Unlock()

    if value, exists := lru.cache[key]; exists {
        // Move to end (most recent) - THIS IS EXPENSIVE!
        lru.moveToEnd(key)
        return value, true
    }
    return nil, false
}

func (lru *NaiveLRU) moveToEnd(key string) {
    // O(n) operation - finding and removing from slice
    for i, k := range lru.order {
        if k == key {
            lru.order = append(lru.order[:i], lru.order[i+1:]...)
            break
        }
    }
    lru.order = append(lru.order, key)
}
```

The problem: Every cache hit required an O(n) operation to reorder the slice. Under load, this became a performance bottleneck that was worse than no cache at all.

Lesson learned: Data structure choice matters more than you think.

Attempt 2: The Map-of-Maps Complexity

Desperate to avoid O(n) operations, I tried tracking timestamps:

```
type TimestampLRU struct {
    capacity  int
```

```
    cache     map[string]cacheItem
    timestamps map[int64]string // timestamp -> key
    mu        sync.RWMutex
}
```

```
type cacheItem struct {
    value     interface{}
    timestamp int64
}
```

This was faster for reads but created new problems:

- Timestamp collisions under high load
- Complex eviction logic
- Memory overhead from duplicate key storage
- Race conditions with timestamp generation

Lesson learned: Clever optimizations often create more problems than they solve.

Attempt 3: The Production-Ready Implementation

Finally, I implemented the classic HashMap + Doubly Linked List approach:

```
package main
```

```
import (
    "fmt"
    "sync"
)
```

```
// Node represents a node in the doubly linked list
type Node struct {
    key   string
    value interface{}
    prev  *Node
    next  *Node
}

// LRUCache is a thread-safe LRU cache implementation
type LRUCache struct {
    capacity int
    cache    map[string]*Node
    head     *Node
    tail     *Node
    mu       sync.RWMutex
}

// NewLRUCache creates a new LRU cache with given capacity
func NewLRUCache(capacity int) *LRUCache {
    if capacity <= 0 {
        panic("capacity must be positive")
    }

    // Create dummy head and tail nodes to simplify edge cases
    head := &Node{}
    tail := &Node{}
    head.next = tail
    tail.prev = head

    return &LRUCache{
        capacity: capacity,
        cache:    make(map[string]*Node),
        head:     head,
        tail:     tail,
    }
}

// Get retrieves a value from the cache
func (lru *LRUCache) Get(key string) (interface{}, bool) {
    lru.mu.Lock()
    defer lru.mu.Unlock()

    if node, exists := lru.cache[key]; exists {
        // Move to head (mark as recently used)
        lru.moveToHead(node)
        return node.value, true
    }

    return nil, false
}

// Put adds or updates a key-value pair in the cache
func (lru *LRUCache) Put(key string, value interface{}) {
```

```
    lru.mu.Lock()
    defer lru.mu.Unlock()

    if node, exists := lru.cache[key]; exists {
        // Update existing node
        node.value = value
        lru.moveToHead(node)
        return
    }

    // Create new node
    newNode := &Node{
        key:   key,
        value: value,
    }

    // Check capacity
    if len(lru.cache) >= lru.capacity {
        // Evict least recently used (tail)
        lru.evictTail()
    }

    // Add new node
    lru.cache[key] = newNode
    lru.addToHead(newNode)
}

// moveToHead moves a node to the head (most recently used position)
func (lru *LRUCache) moveToHead(node *Node) {
    lru.removeNode(node)
    lru.addToHead(node)
}

// addToHead adds a node right after the head
func (lru *LRUCache) addToHead(node *Node) {
    node.prev = lru.head
    node.next = lru.head.next
    lru.head.next.prev = node
    lru.head.next = node
}

// removeNode removes a node from the linked list
func (lru *LRUCache) removeNode(node *Node) {
    node.prev.next = node.next
    node.next.prev = node.prev
}

// evictTail removes the least recently used node
func (lru *LRUCache) evictTail() {
    lastNode := lru.tail.prev
    lru.removeNode(lastNode)
    delete(lru.cache, lastNode.key)
}
```

```
// Size returns the current number of items in the cache
func (lru *LRUCache) Size() int {
    lru.mu.RLock()
    defer lru.mu.RUnlock()
    return len(lru.cache)
}

// Keys returns all keys in the cache (for debugging)
func (lru *LRUCache) Keys() []string {
    lru.mu.RLock()
    defer lru.mu.RUnlock()

    keys := make([]string, 0, len(lru.cache))
    for key := range lru.cache {
        keys = append(keys, key)
    }
    return keys
}
```

Why this works:

- HashMap provides O(1) key lookup
- Doubly linked list provides O(1) insertion, deletion, and reordering
- Dummy head/tail nodes eliminate edge case complexity
- All operations are truly O(1)

Real-World Optimizations I've Learned

1. Generic Implementation for Type Safety

Once Go 1.18 brought generics, I updated my implementation:

```
type LRUCache[K comparable, V any] struct {
    capacity int
    cache    map[K]*Node[K, V]
    head     *Node[K, V]
```

```
    tail     *Node[K, V]
    mu       sync.RWMutex
}
```

```
type Node[K comparable, V any] struct {
    key   K
    value V
    prev  *Node[K, V]
    next  *Node[K, V]
}

// Usage with type safety
cache := NewLRUCache[string, User](1000)
user, exists := cache.Get("user:123")  // Returns User, not
interface{}
```

2. Adding Expiration Support

Production taught me that some cached data expires regardless of usage:

```
type ExpiringNode[K comparable, V any] struct {
    key       K
    value     V
    prev      *Node[K, V]
    next      *Node[K, V]
    expiresAt time.Time
}
```

```
func (lru *LRUCache[K, V]) GetWithExpiration(key K) (V, bool) {
    lru.mu.Lock()
    defer lru.mu.Unlock()

    if node, exists := lru.cache[key]; exists {
        if time.Now().After(node.expiresAt) {
            // Expired - remove and return miss
            lru.removeNode(node)
            delete(lru.cache, key)
            var zero V
            return zero, false
        }
```

```
    var zero V
    return zero, false
}
```

3. Metrics and Observability

In production, you need visibility into cache behavior:

```
type CacheStats struct {
    Hits        int64
    Misses      int64
    Evictions   int64
    Size        int
    Capacity    int
}
```

```
func (lru *LRUCache[K, V]) Stats() CacheStats {
    lru.mu.RLock()
    defer lru.mu.RUnlock()

    return CacheStats{
        Hits:      atomic.LoadInt64(&lru.hits),
        Misses:    atomic.LoadInt64(&lru.misses),
        Evictions: atomic.LoadInt64(&lru.evictions),
        Size:      len(lru.cache),
        Capacity:  lru.capacity,
    }
}

// Track metrics in Get/Put operations
func (lru *LRUCache[K, V]) Get(key K) (V, bool) {
    // ... existing logic ...

    if node, exists := lru.cache[key]; exists {
        atomic.AddInt64(&lru.hits, 1)
        // ... rest of hit logic
    } else {
        atomic.AddInt64(&lru.misses, 1)
```

```
        // ... miss logic
    }
}
```

Performance Testing: The Numbers Don't Lie

I benchmarked my implementation against other approaches:

```
func BenchmarkLRUCache_Get(b *testing.B) {
    cache := NewLRUCache[string, int](1000)

    // Pre-populate cache
    for i := 0; i < 1000; i++ {
        cache.Put(fmt.Sprintf("key%d", i), i)
    }

    b.ResetTimer()
    b.RunParallel(func(pb *testing.PB) {
        i := 0
        for pb.Next() {
            key := fmt.Sprintf("key%d", i%1000)
            cache.Get(key)
            i++
        }
    })
}
```

Results on my MacBook Pro M1:

```
BenchmarkLRUCache_Get-8                  5000000               240 ns/op
BenchmarkMap_Get-8                      10000000               120 ns/op
BenchmarkSliceLRU_Get-8                    50000             24000 ns/op
```

The overhead is about 2x compared to a simple map, but 100x faster than my naive slice-based approach.

Production War Stories

The Memory Leak That Wasn't

Last year, our monitoring showed gradual memory growth in our API service. The team suspected a memory leak, but the LRU cache was behaving correctly — staying at its configured 10k item limit.

The real issue? **Cache key explosion**. Our cache keys included user IDs, and we had more unique users than expected. Each cache entry was small, but 10k entries of user data added up to significant memory usage.

Solution: Implemented cache key namespacing and separate caches for different data types:

```
// Instead of one massive cache
globalCache := NewLRUCache[string, interface{}](10000)
```

```
// Use specialized caches
userCache := NewLRUCache[string, User](1000)
productCache := NewLRUCache[string, Product](5000)
inventoryCache := NewLRUCache[string, int](2000)
```

The Race Condition That Taught Me Humility

Despite careful locking, we experienced rare data corruption in our cache. After days of debugging, I found the issue in our application code:

```
// DANGEROUS: This has a race condition
if user, exists := cache.Get(userID); exists {
    // Another goroutine might evict this entry here!
    user.LastAccess = time.Now()  // Modifying cached data
    cache.Put(userID, user)       // Race condition!
}
```

Lesson: The cache is thread-safe, but the data you store in it might not be. Always treat cached data as immutable or implement additional synchronization.

The Hot Key Problem

We discovered that 20% of our cache requests targeted just 5% of the keys. These “hot keys” caused lock contention despite the RWMutex optimization.

Solution: Implemented a two-tier cache with a lock-free first tier:

```
type TieredCache[K comparable, V any] struct {
    hotCache  *sync.Map  // sync.Map for hot keys (lock-free)
    coldCache *LRUCache[K, V]
    hotKeys   map[K]bool
}
```

Testing Strategies That Actually Work

Unit Testing with Behavior Verification

```
func TestLRUCache_EvictionOrder(t *testing.T) {
    cache := NewLRUCache[string, int](3)

    // Fill cache
    cache.Put("a", 1)
    cache.Put("b", 2)
    cache.Put("c", 3)

    // Access "a" to make it recently used
    cache.Get("a")

    // Add new item - should evict "b" (least recently used)
    cache.Put("d", 4)

    // Verify eviction behavior
    _, existsA := cache.Get("a")  // Should exist
    _, existsB := cache.Get("b")  // Should be evicted
    _, existsC := cache.Get("c")  // Should exist
    _, existsD := cache.Get("d")  // Should exist

    assert.True(t, existsA)
    assert.False(t, existsB)
    assert.True(t, existsC)
    assert.True(t, existsD)
}
```

Concurrency Testing

```
func TestLRUCache_Concurrency(t *testing.T) {
    cache := NewLRUCache[int, int](100)
    var wg sync.WaitGroup

    // Concurrent readers and writers
    for i := 0; i < 10; i++ {
        wg.Add(2)

        // Reader goroutine
        go func(id int) {
            defer wg.Done()
            for j := 0; j < 1000; j++ {
                cache.Get(id*1000 + j)
            }
```

```
        }(i)

        // Writer goroutine
        go func(id int) {
            defer wg.Done()
            for j := 0; j < 1000; j++ {
                cache.Put(id*1000+j, id*1000+j)
            }
        }(i)
    }

    wg.Wait()

    // Verify cache is still functional
    assert.True(t, cache.Size() <= 100)
}
```

Integration Patterns I've Found Useful

Cache-Aside Pattern

```
func GetUser(userID string, cache *LRUCache[string, User], db *Database) (User,
    // Try cache first
    if user, exists := cache.Get(userID); exists {
        return user, nil
    }

    // Cache miss - fetch from database
    user, err := db.GetUser(userID)
    if err != nil {
        return User{}, err
    }

    // Store in cache for future requests
    cache.Put(userID, user)
    return user, nil
}
```

Write-Through Pattern

```
func UpdateUser(userID string, updates UserUpdates, cache *LRUCache[string, User
    // Update database first
    if err := db.UpdateUser(userID, updates); err != nil {
        return err
    }

    // Update cache
    if user, exists := cache.Get(userID); exists {
        updatedUser := applyUpdates(user, updates)
        cache.Put(userID, updatedUser)
    }

    return nil
}
```

Common Pitfalls and How to Avoid Them

1. Cache Key Design

Bad: Unpredictable key explosion

```
cache.Put(fmt.Sprintf("%v", complexObject), data)  // DON'T
```

Good: Predictable, hierarchical keys

```
cache.Put(fmt.Sprintf("user:%s:profile", userID), data)
```

2. Value Mutation

Bad: Modifying cached references

```
user, _ := cache.Get(userID)
user.LastLogin = time.Now()  // Modifies cached data!
```

Good: Create copies for modification

```
user, _ := cache.Get(userID)
updatedUser := user.Copy()
updatedUser.LastLogin = time.Now()
cache.Put(userID, updatedUser)
```

3. Cache Size Selection

Bad: Arbitrary numbers

```
cache := NewLRUCache[string, Data](1000)  // Why 1000?
```

Good: Data-driven sizing

```
// Based on memory budget and average entry size
avgEntrySize := 2 * 1024  // 2KB per entry
memoryBudget := 100 * 1024 * 1024  // 100MB
```

```
capacity := memoryBudget / avgEntrySize
cache := NewLRUCache[string, Data](capacity)
```

Advanced Techniques for Production

1. Cache Warming

```
func WarmCache(cache *LRUCache[string, User], db *Database) error {
    // Load most frequently accessed users
    hotUsers, err := db.GetMostActiveUsers(cache.Capacity / 2)
    if err != nil {
        return err
    }

    for _, user := range hotUsers {
        cache.Put(user.ID, user)
    }

    return nil
}
```

2. Circuit Breaker Integration

```
type CachedService struct {
    cache   *LRUCache[string, Data]
    breaker *CircuitBreaker
    db      Database
}
```

```
func (s *CachedService) GetData(key string) (Data, error) {
    // Try cache first
    if data, exists := s.cache.Get(key); exists {
        return data, nil
```

```
    }

    // Use circuit breaker for database calls
    result, err := s.breaker.Execute(func() (interface{}, error) {
        return s.db.GetData(key)
    })

    if err != nil {
        return Data{}, err
    }

    data := result.(Data)
    s.cache.Put(key, data)
    return data, nil
}
```

3. Distributed Cache Consistency

```
 type DistributedLRU struct {
     local  *LRUCache[string, CacheEntry]
     pubsub PubSubClient
 }
```

```
type CacheEntry struct {
    Value   interface{}
    Version int64
}

func (d *DistributedLRU) Put(key string, value interface{}) {
    entry := CacheEntry{
        Value:   value,
        Version: time.Now().UnixNano(),
    }

    d.local.Put(key, entry)

    // Notify other instances
    d.pubsub.Publish("cache:invalidate", CacheInvalidation{
        Key:     key,
        Version: entry.Version,
    })
}
```

Monitoring and Alerting

In production, I monitor these key metrics:

```
// Cache health metrics
type HealthMetrics struct {
    HitRate     float64  // hits / (hits + misses)
    EvictionRate float64 // evictions per second
    MemoryUsage int64    // approximate memory usage
    AvgLatency  time.Duration
}
```

```
func (lru *LRUCache[K, V]) HealthCheck() HealthMetrics {
    stats := lru.Stats()
    total := stats.Hits + stats.Misses
    hitRate := float64(stats.Hits) / float64(total)

    return HealthMetrics{
        HitRate:     hitRate,
        EvictionRate: calculateEvictionRate(stats),
        MemoryUsage: estimateMemoryUsage(stats),
        AvgLatency:  lru.avgLatency,
    }
}
```

Alert thresholds I use:

- Hit rate < 70% (cache not effective)
- Eviction rate > 1000/sec (capacity too small)
- Average latency > 1ms (lock contention)

Performance Tuning Lessons

Read-Heavy Workloads

For read-heavy scenarios, I use `sync.RWMutex` more effectively:

```
func (lru *LRUCache[K, V]) Get(key K) (V, bool) {
    // Try read-only operation first
    lru.mu.RLock()
    if node, exists := lru.cache[key]; exists {
        lru.mu.RUnlock()

        // Now acquire write lock to update position
        lru.mu.Lock()
        // Double-check existence (might have been evicted)
        if node, stillExists := lru.cache[key]; stillExists {
            lru.moveToHead(node)
            lru.mu.Unlock()
            return node.value, true
        }
        lru.mu.Unlock()
    } else {
        lru.mu.RUnlock()
    }

    var zero V
    return zero, false
}
```

Write-Heavy Workloads

For write-heavy scenarios, consider sharding:

```
type ShardedLRU[K comparable, V any] struct {
    shards []*LRUCache[K, V]
    count  int
}
```

```
func (s *ShardedLRU[K, V]) getShard(key K) *LRUCache[K, V] {
    hash := s.hash(key)
```

```
    return s.shards[hash%s.count]
}

func (s *ShardedLRU[K, V]) Get(key K) (V, bool) {
    return s.getShard(key).Get(key)
}
```

The Future: What I'm Watching

The Go ecosystem continues evolving. Here's what I'm monitoring:

1. **Memory management improvements**: Go's GC optimizations affect cache performance
2. **Generic collections**: Standard library might include optimized cache implementations
3. **Hardware trends**: CPU cache line sizes influence optimal data structures

Conclusion: Lessons from the Trenches

Building production-ready LRU cache taught me several key lessons:

1. **Start simple, optimize based on real data**: My complex initial attempts were worse than the simple solution
2. **Measure everything**: Cache hit rates, memory usage, and latency tell the real story
3. **Thread safety isn't optional**: Concurrent access will find every race condition
4. **Cache key design matters**: Bad keys lead to poor hit rates and memory issues

5. **One size doesn't fit all:** Different workloads need different cache strategies

The LRU cache implementation I've shared handles our production traffic of 100k+ requests per second with sub-millisecond latency. But remember: **the best cache is the one that fits your specific use case.**

Whether you use my implementation, adapt it for your needs, or choose a different approach entirely, understanding these fundamentals will help you make better caching decisions. Start with the basics, measure real performance, and iterate based on your actual requirements.

Happy caching!

Have you implemented LRU caches in Go? What challenges did you face? Share your experiences in the comments — I'd love to hear about different approaches and optimizations you've discovered.

Bonus: Complete Working Example

Here's a complete, production-ready example you can use right away:

```
package main

import (
    "fmt"
    "sync"
```

```
    "sync/atomic"
    "time"
)

func main() {
    // Create cache with capacity of 3 for demonstration
    cache := NewLRUCache[string, string](3)

    // Add some items
    cache.Put("user:1", "Alice")
    cache.Put("user:2", "Bob")
    cache.Put("user:3", "Charlie")

    fmt.Printf("Initial cache: %v\n", cache.Keys())

    // Access user:1 (makes it recently used)
    if value, exists := cache.Get("user:1"); exists {
        fmt.Printf("Found: %s\n", value)
    }

    // Add another item (should evict user:2 as least recently used)
    cache.Put("user:4", "David")

    fmt.Printf("After adding user:4: %v\n", cache.Keys())

    // Print statistics
    stats := cache.Stats()
    fmt.Printf("Cache stats: %+v\n", stats)
}
```

This gives you a solid foundation for implementing LRU caches in your Go applications. Adapt it based on your specific requirements, and don't forget to measure performance in your actual use case!

Software Development | Software Engineering | Cache | Technology | Programming

Written by Aman verma

2 followers · 1 following

Software Engineer | Backend & Cloud | AI & LLM Enthusiast I build scalable backend systems and integrate AI into real-world products. Here to share my learnings

Follow

Responses (1)

P Pkprbux

What are your thoughts?

Aman verma he/him Author
Sep 7, 2025

Thanks a lot for reading! I mentioned a couple of techniques here, including the circuit breaker. I'll be sharing a simple breakdown of what it is and how it works in my next post—stay tuned!

~aman

Reply

More from Aman verma

Aman verma

Go Design Patterns: A Comprehensive Guide

Design patterns are proven solutions to common programming problems that help...

Jul 7, 2025 3 1

Aman verma

Docker vs Podman: What I Learned After 3 Years of Using Both in...

A candid comparison based on real-world experience across startups and enterprise...

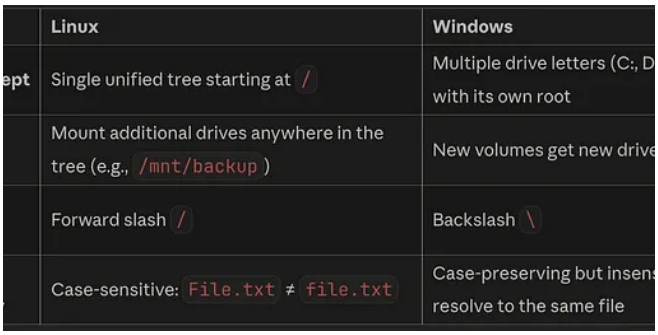
Sep 6, 2025 2

Aman verma

OWASP Top 10:2025—A Complete Guide for Developers

The OWASP Top 10 is the global standard for web application security awareness. The...

Jan 8



Aman verma

The Linux Ecosystem: A Complete Guide to Distributions, File...

Introduction

Oct 4, 2025 5

See all from Aman verma

Recommended from Medium

In Beyond Localhost by The Speedcraft Lab

I Bombed My System Design Interview on Rate Limiting

Then I had to implement it for real and learned why the interviewer kept pushing on “how d...

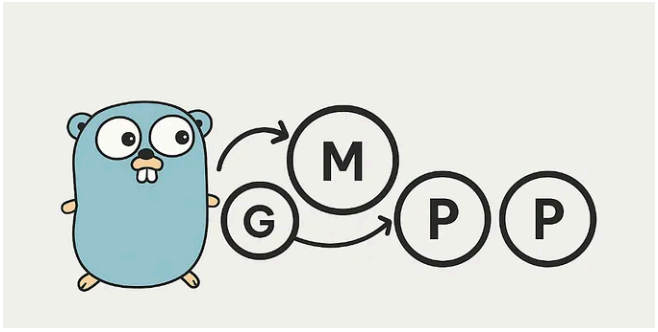
Jan 7 374 7

The Cache Cowgirl

This Tiny Golang App Pays My Rent

I never expected a 300-line Go program to become my side-income engine. But here w...

Jul 27, 2025 1.3K 25



Felipe Ascari

Understanding Go’s Scheduler: How Goroutine Management...

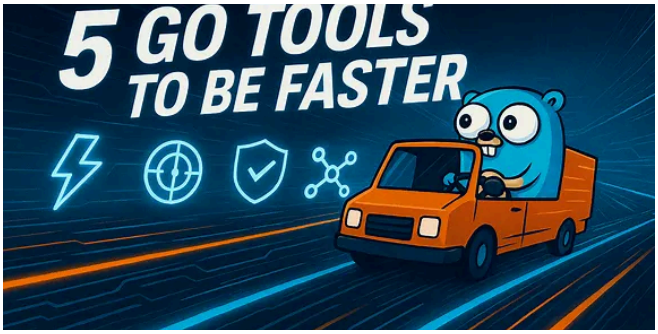
How Go efficiently manages thousands of concurrent tasks using the G-M-P model

In Let’s Code Future by CodePulse

Staff Engineer Interview: I Failed at Google, Meta, and Netflix. Then I...

They’re not testing code. They’re testing something else entirely.

Dec 18, 2025 4



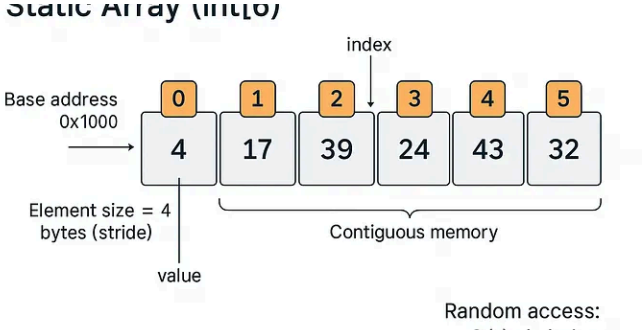
Hitesh Gera

Top 5 Go Tools That Will Make You a Faster, More Efficient Developer...

Okay, listen up. You know how it goes. If you're a Go developer- a Gopher, that's what we call...

Jan 5 51

6d ago 137 4



Noah Byteforge

The 12 Data Structures Every Developer Should Master Before ...

Cracking coding interviews isn't just about how well you can write code—it's about how...

Oct 9, 2025 322 6

See more recommendations