● JANUARY 2026 SERIES

FROM GO BUILD
TO GO RUN

GOLANG 2026 - NIV RAVE

#32

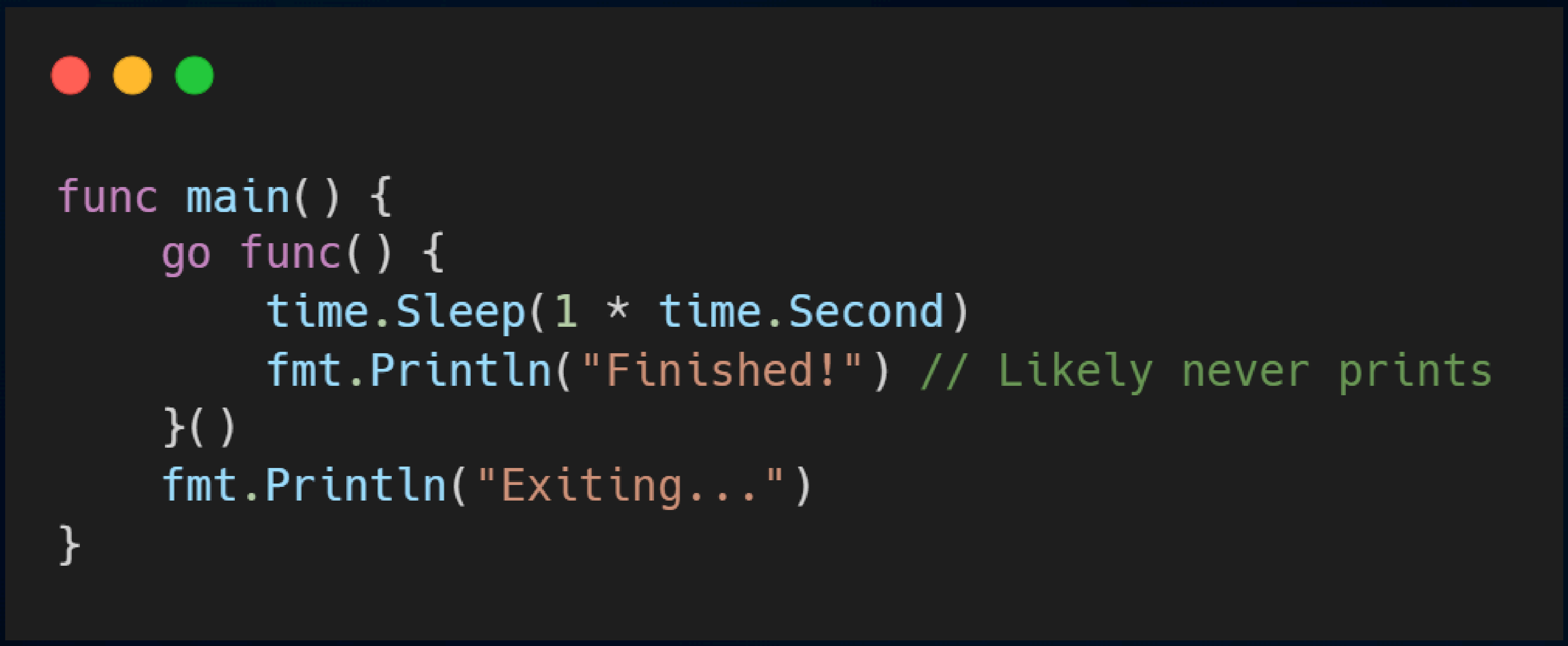
LAUNCHING TASKS SAFELY

ORCHESTRATING GOROUTINES WITH SYNC.WAITGROUP

The Race to the Exit

The "Main" Problem

When the main() function finishes, the program terminates immediately, regardless of how many background goroutines are still running.



```
func main() {
    go func() {
        time.Sleep(1 * time.Second)
        fmt.Println("Finished!") // Likely never prints
    }()
    fmt.Println("Exiting...")
}
```

Never rely on *time.Sleep* to wait for goroutines.
It’s a "non-deterministic" hack that leads to flaky tests and unstable production code.

The Counter Mechanism

Enter *sync.WaitGroup*

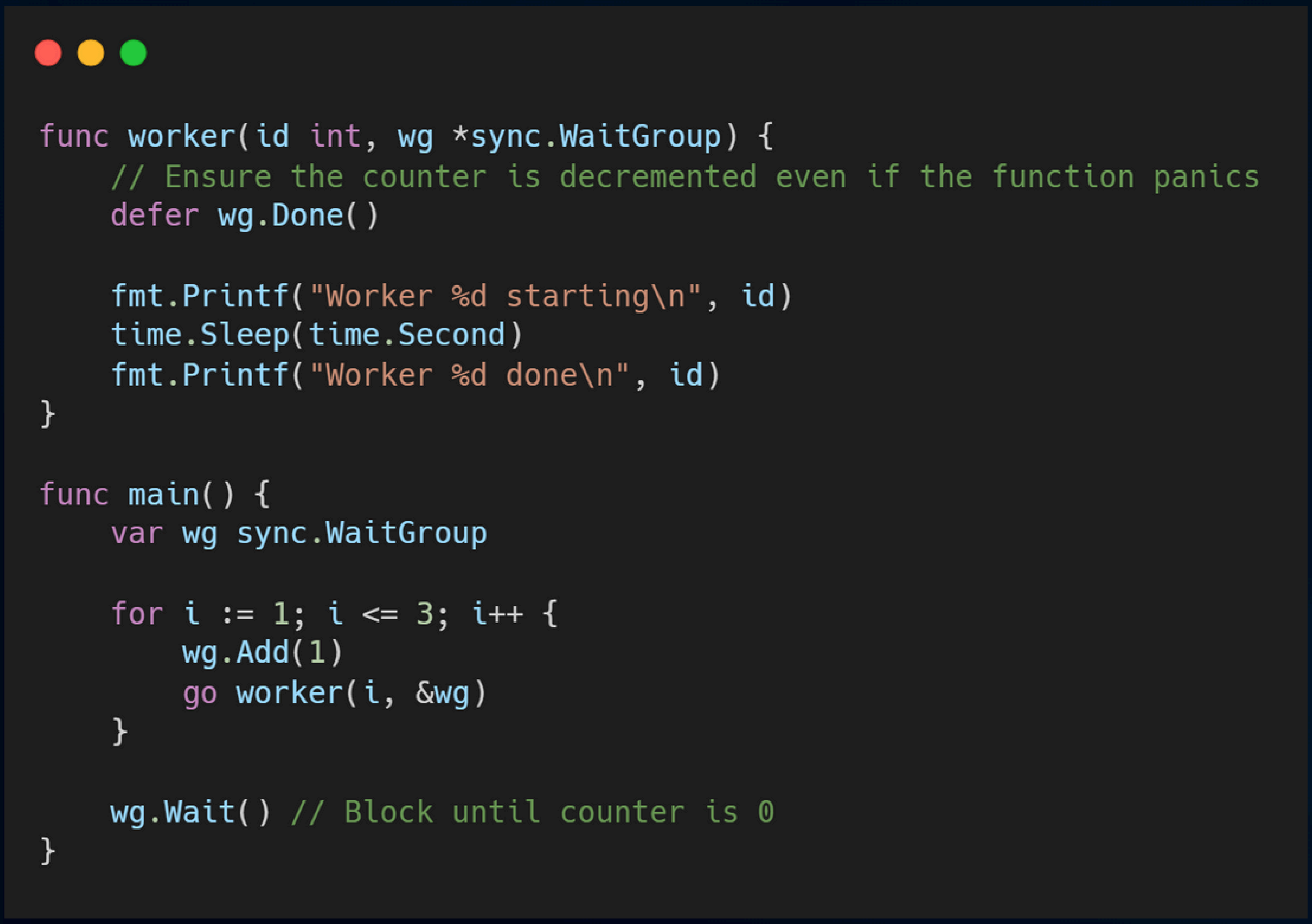
A WaitGroup is an opaque struct containing a counter and a wait-set. It allows you to synchronize the completion of multiple goroutines without the overhead of channel communication.

The Main Methods:

Add(int): Increments (or decrements) the internal counter.

Done(): Decrements the counter by exactly 1 (sugar for Add(-1)).

Wait(): Blocks the calling goroutine until the internal counter hits zero.

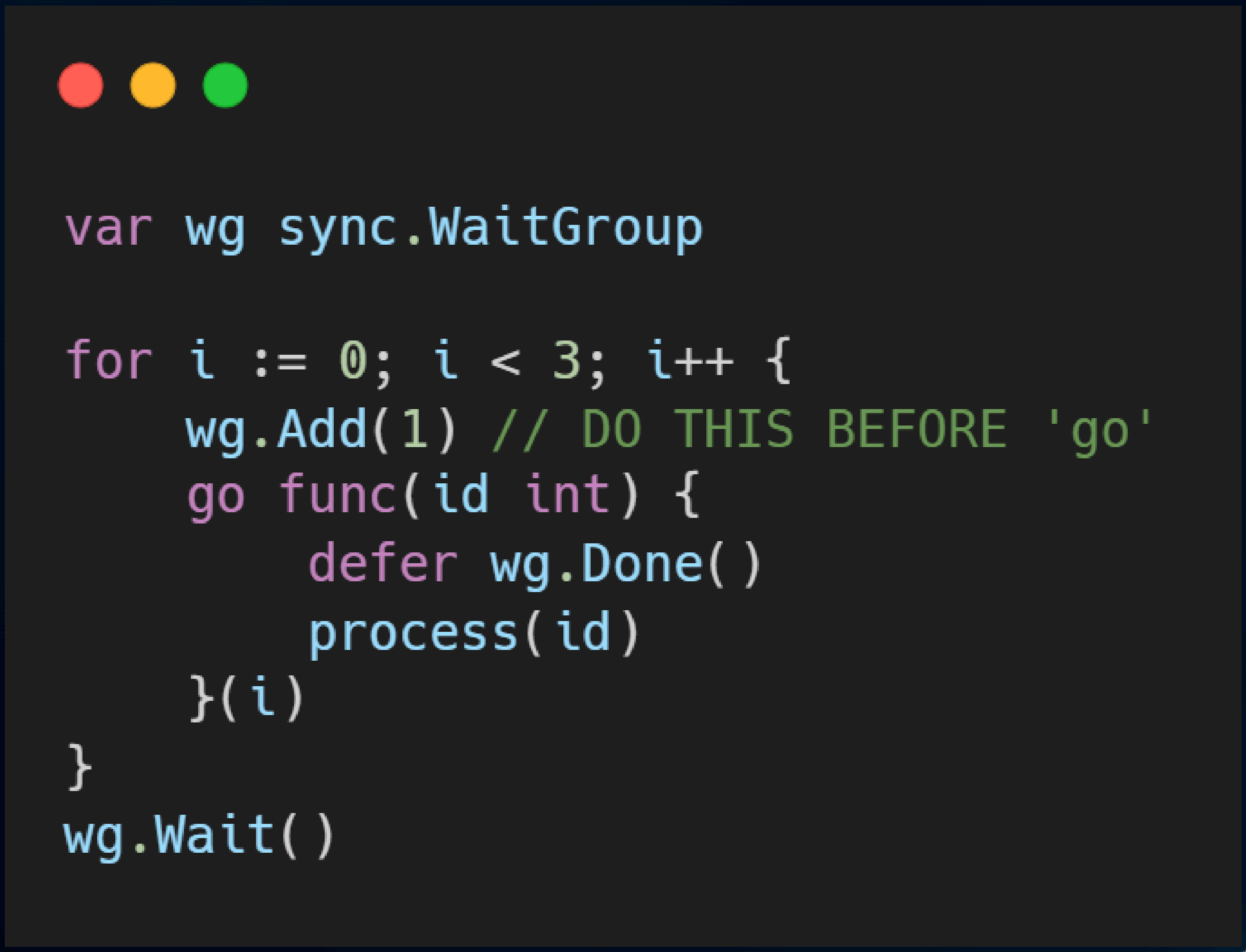


```go
func worker(id int, wg *sync.WaitGroup) {
    // Ensure the counter is decremented even if the function panics
    defer wg.Done()

    fmt.Printf("Worker %d starting\n", id)
    time.Sleep(time.Second)
    fmt.Printf("Worker %d done\n", id)
}

func main() {
    var wg sync.WaitGroup

    for i := 1; i <= 3; i++ {
        wg.Add(1)
        go worker(i, &wg)
    }

    wg.Wait() // Block until counter is 0
}
```

The Golden Rule of Add()

Avoid the Scheduling Race

Always call *wg.Add()* in the parent goroutine before the *go* statement. Why? If you call it inside the goroutine, there is a chance the parent hits *wg.Wait()* before the scheduler even starts the new goroutine, causing the program to exit prematurely.

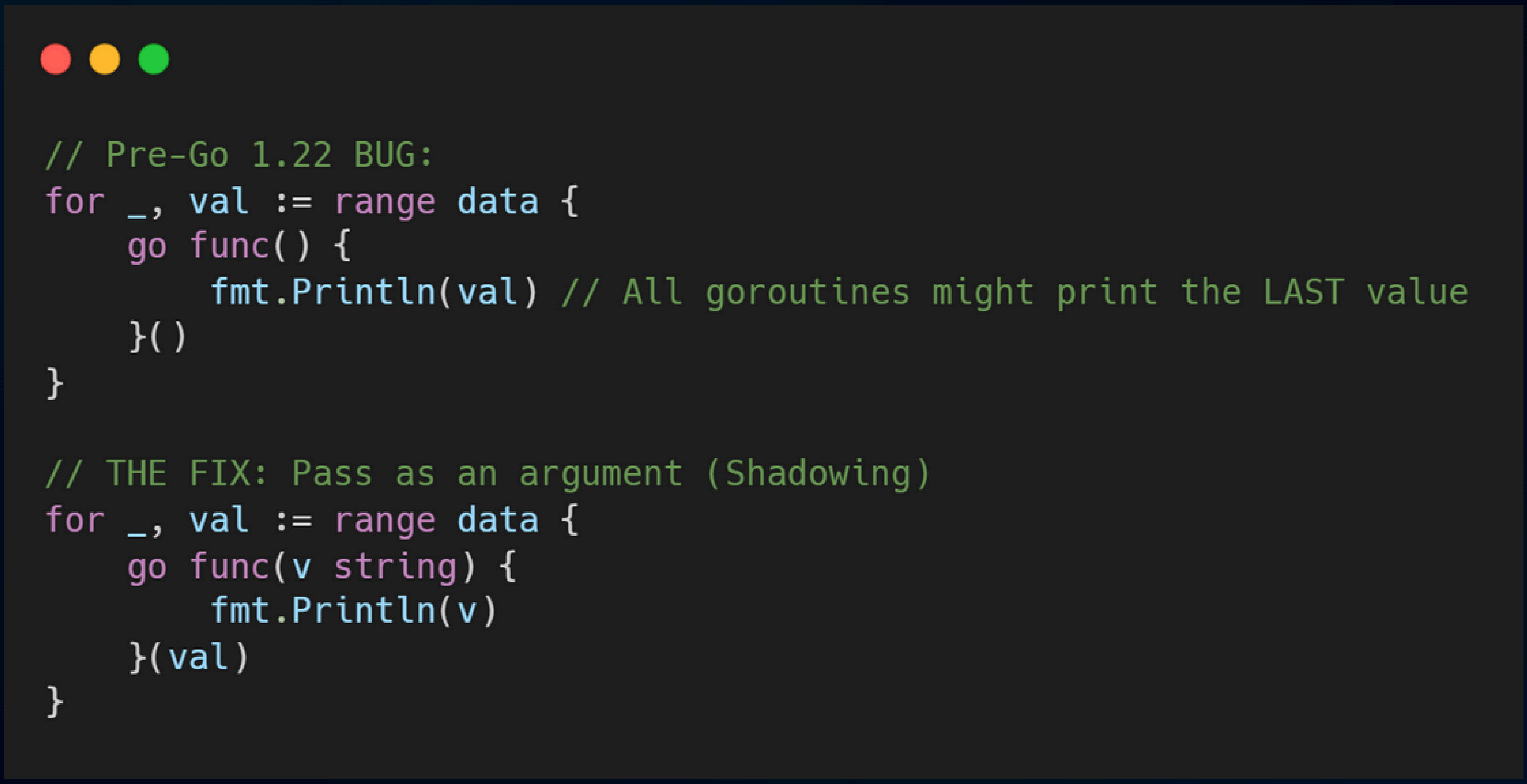


```go
var wg sync.WaitGroup

for i := 0; i < 3; i++ {
    wg.Add(1) // DO THIS BEFORE 'go'
    go func(id int) {
        defer wg.Done()
        process(id)
    }(i)
}
wg.Wait()
```

The Loop Trap

Closure Capture & Loop Variables

In older versions of Go (pre-1.22), failing to pass the loop variable into the goroutine was the #1 source of concurrency bugs.



```go
// Pre-Go 1.22 BUG:
for _, val := range data {
    go func() {
        fmt.Println(val) // All goroutines might print the LAST value
    }()
}

// THE FIX: Pass as an argument (Shadowing)
for _, val := range data {
    go func(v string) {
        fmt.Println(v)
    }(val)
}
```

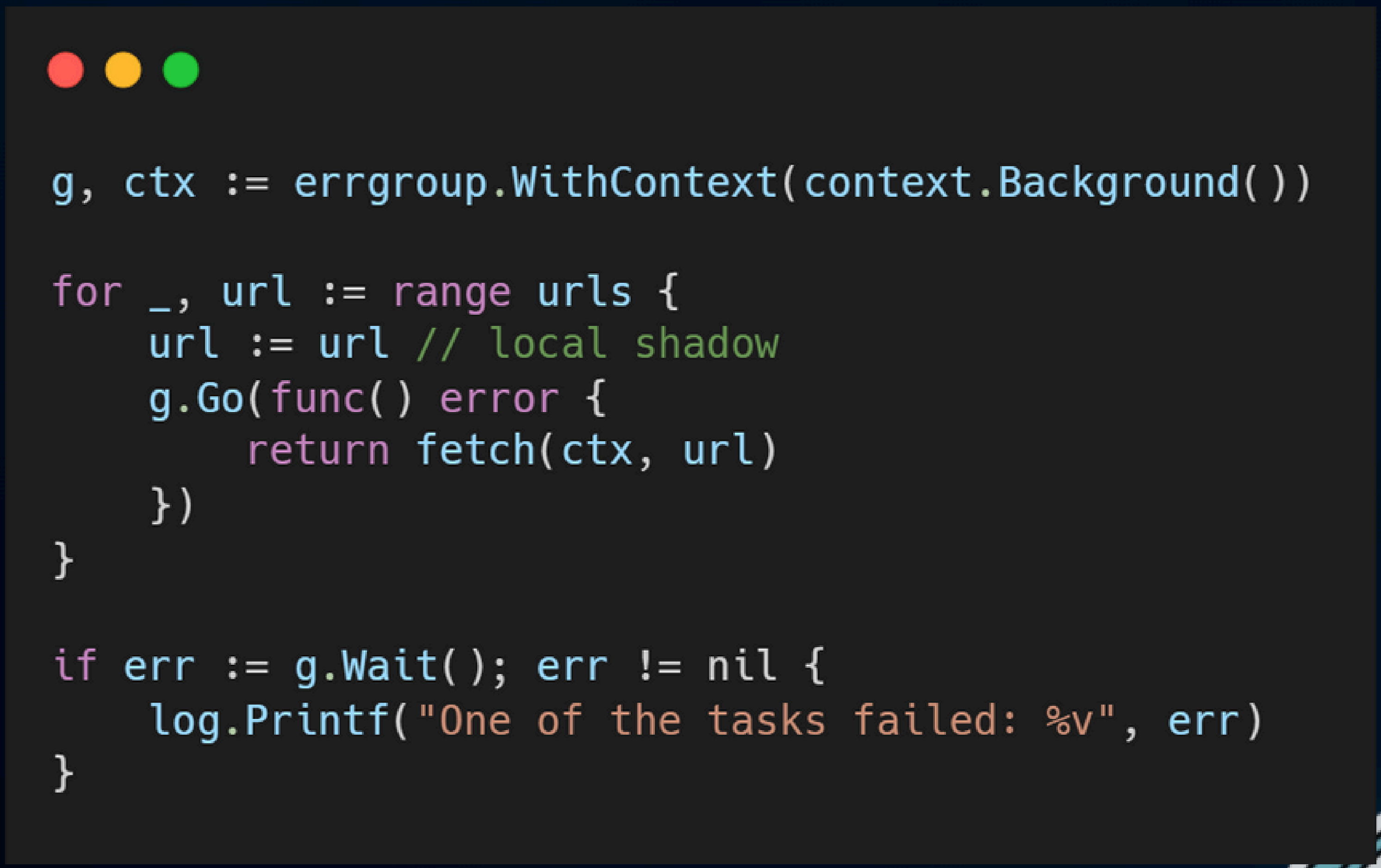
Note: Even though Go 1.22+ fixes the loop variable semantics, passing variables explicitly as arguments remains a best practice for clarity and testability.

Handling Errors in Groups

Beyond *sync.WaitGroup*

WaitGroup is great for waiting, but it doesn't return errors. If one of your concurrent tasks fails, how do you catch it?

The Senior Move: Use *golang.org/x/sync/errgroup*. It manages a group of goroutines and returns the first error encountered by any of them.



```go
g, ctx := errgroup.WithContext(context.Background())

for _, url := range urls {
    url := url // local shadow
    g.Go(func() error {
        return fetch(ctx, url)
    })
}

if err := g.Wait(); err != nil {
    log.Printf("One of the tasks failed: %v", err)
}
```

Avoiding Goroutine Leaks

The Memory Silent Killer

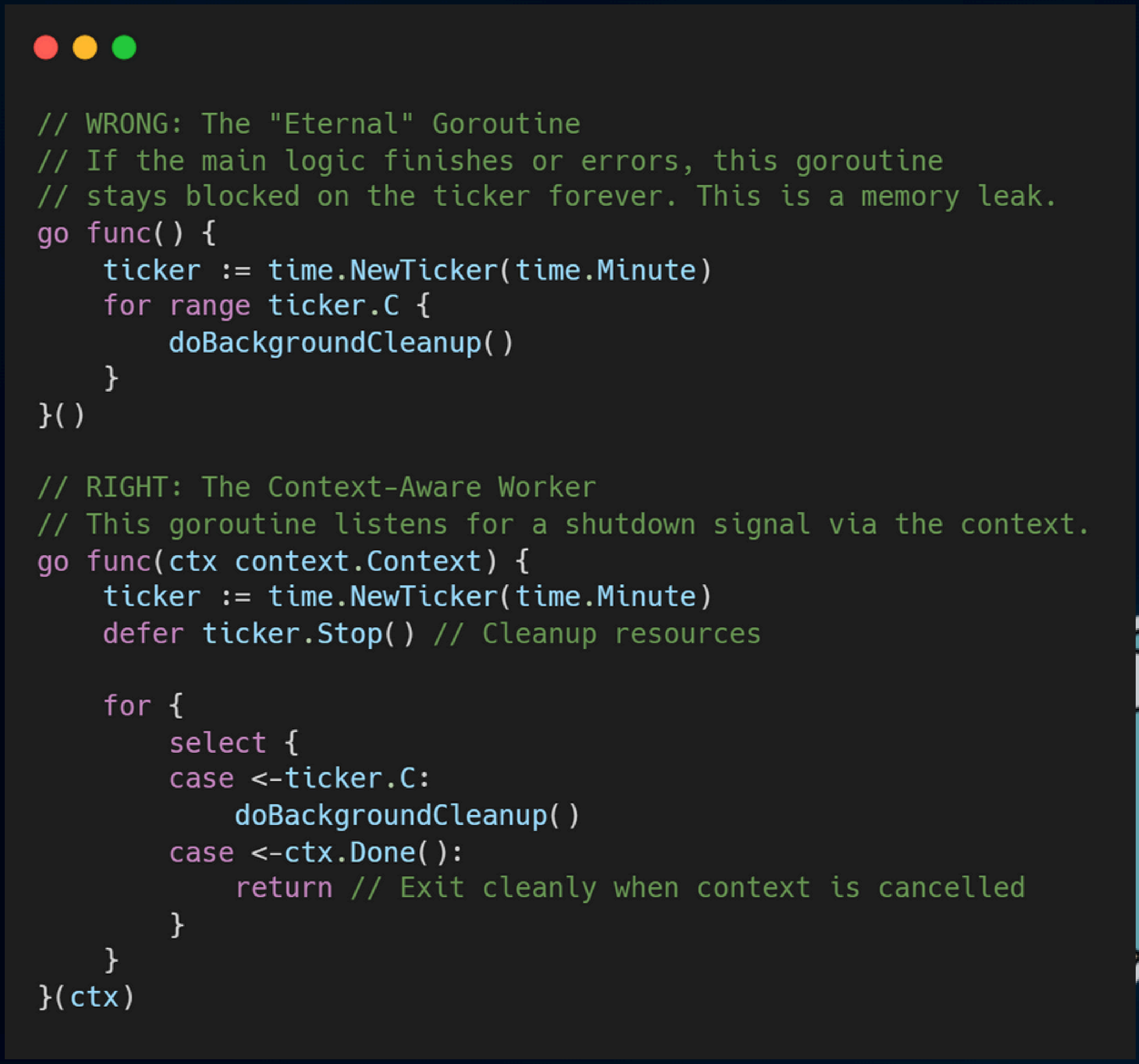
A goroutine that is blocked forever (e.g., waiting for a channel that never closes) is a memory leak.

Unlike unused structs, the GC cannot clean up a blocked goroutine.

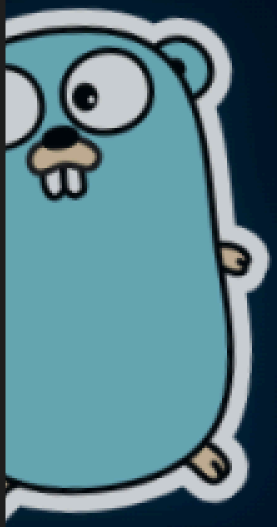
A Senior’s Strategy:

Every goroutine you start must have a clear exit path.

Use *context.Context* to signal timeouts or cancellations to your background workers.



```go
// WRONG: The "Eternal" Goroutine
// If the main logic finishes or errors, this goroutine
// stays blocked on the ticker forever. This is a memory leak.
go func() {
    ticker := time.NewTicker(time.Minute)
    for range ticker.C {
        doBackgroundCleanup()
    }
}()

// RIGHT: The Context-Aware Worker
// This goroutine listens for a shutdown signal via the context.
go func(ctx context.Context) {
    ticker := time.NewTicker(time.Minute)
    defer ticker.Stop() // Cleanup resources

    for {
        select {
        case <-ticker.C:
            doBackgroundCleanup()
        case <-ctx.Done():
            return // Exit cleanly when context is cancelled
        }
    }
}(ctx)
```



The Cost of Over-Concurrency

Throttling and Resource Limits

Just because you can launch 100,000 goroutines doesn't mean you should. Each goroutine takes ~2KB of memory, meaning that 100k goroutines = 200MB of overhead before any logic runs.

Rule:

If you are interacting with external resources (DBs, APIs), use a Worker Pool or a Semaphore to limit the number of concurrent goroutines. Don't DOS your own database.

Summary:

- Call Add() before go.
- Use defer wg.Done() to ensure cleanup even if the task panics.
- Use errgroup if you care about error reporting.
- Never let a goroutine run without an exit strategy (context).

Tomorrow, we move into the "Plumbing" of Go: Channels. We’ll compare Buffered vs. Unbuffered and when to use each.