● JANUARY 2026 SERIES

FROM GO BUILD
TO GO RUN

GOLANG 2026 - NIV RAVE

#34

THE PRODUCER/CONSUMER PATTERN

BUILDING RESILIENT PIPELINES

Why Decouple?

Handling Variable Load

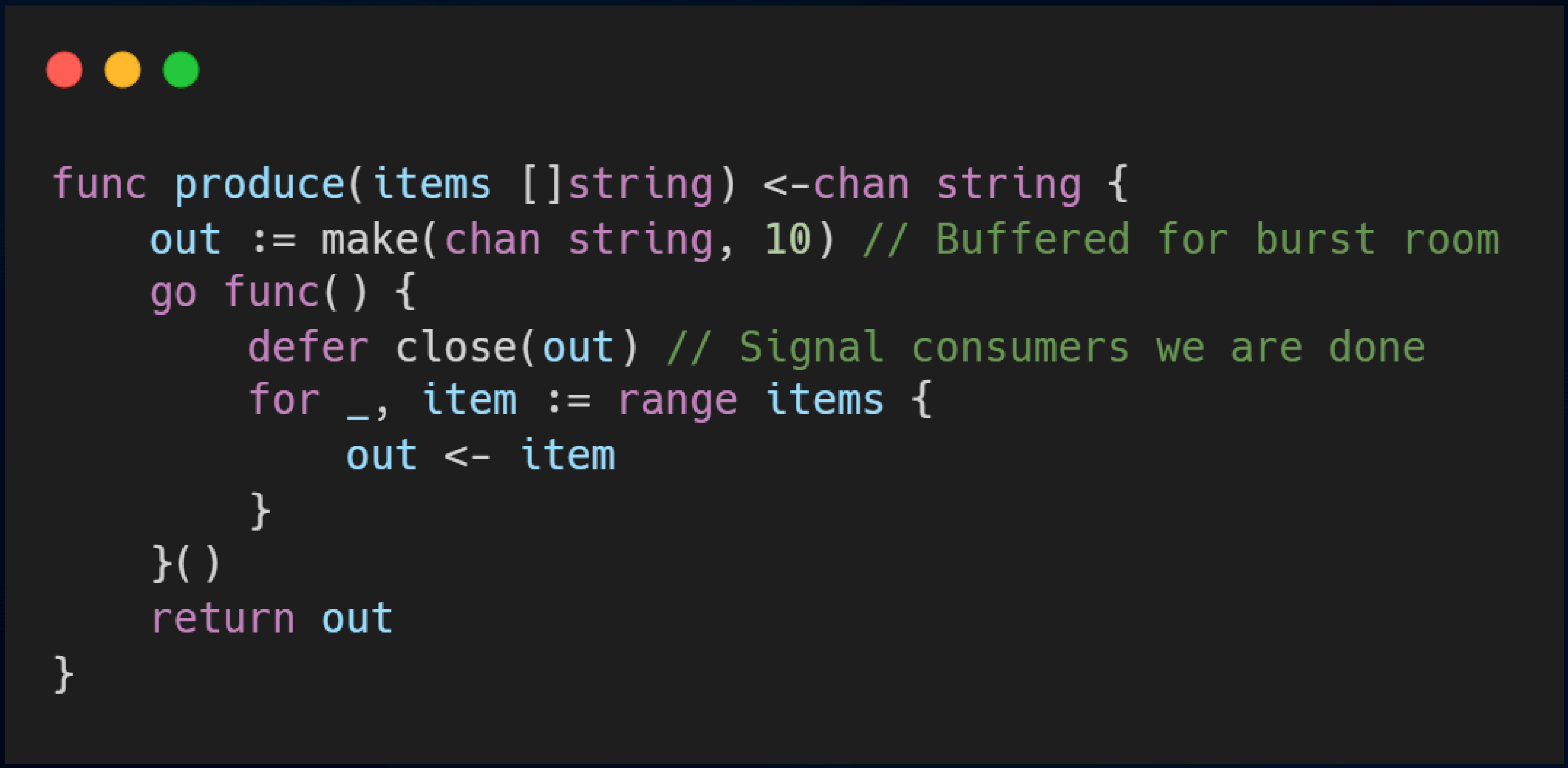
In a synchronous system, if your "Process" step takes 100ms, your "Ingest" step is stuck waiting.
The Solution: The Producer puts work into a channel and immediately returns. The Consumers pull from that channel at their own pace.

This pattern is the primary way we implement Asynchronous Processing within a single Go service. It transforms a blocking system into a non-blocking one.

The Producer’s Responsibility

Ownership of the Channel

The Producer is the "Source of Truth." It creates the channel, starts the work, and most importantly – is responsible for closing the channel when no more data is coming.

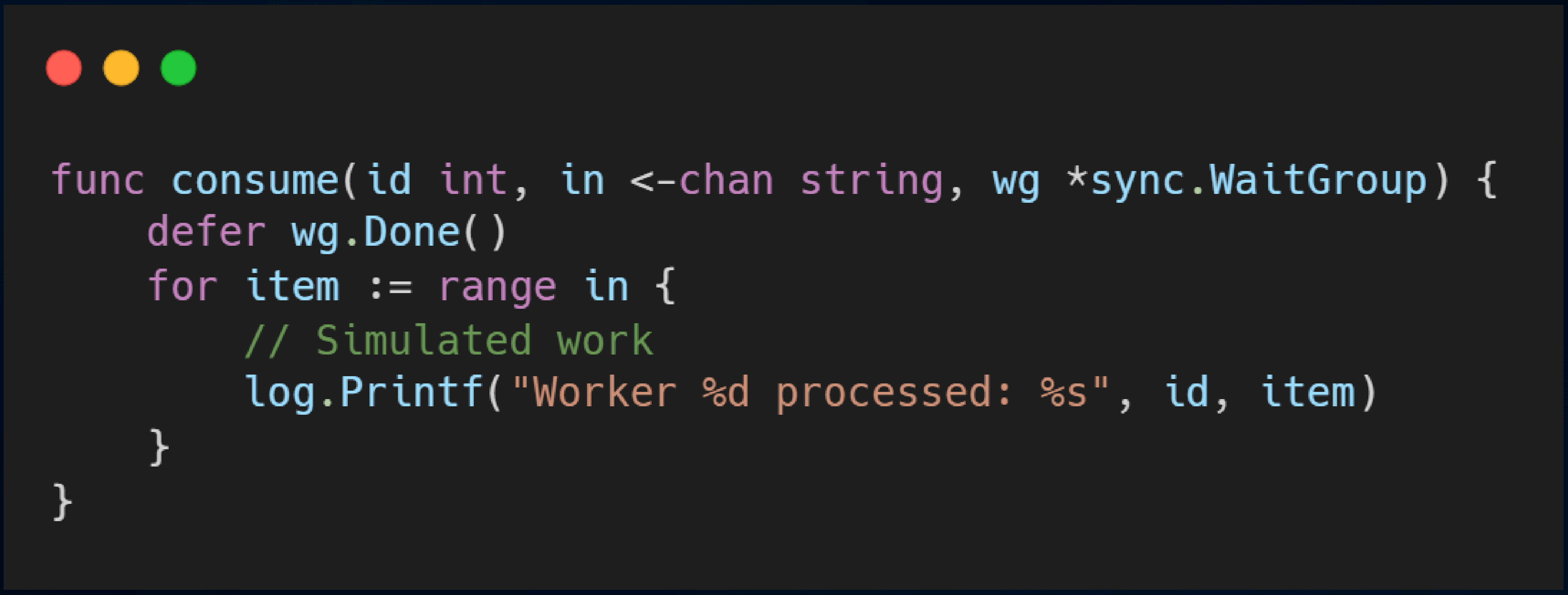


```go
func produce(items []string) <-chan string {
    out := make(chan string, 10) // Buffered for burst room
    go func() {
        defer close(out) // Signal consumers we are done
        for _, item := range items {
            out <- item
        }
    }()
    return out
}
```

The Consumer’s Loop

Efficient Processing

Consumers should use the *for range* syntax. It’s the cleanest way to say "keep working until the producer tells us to stop".

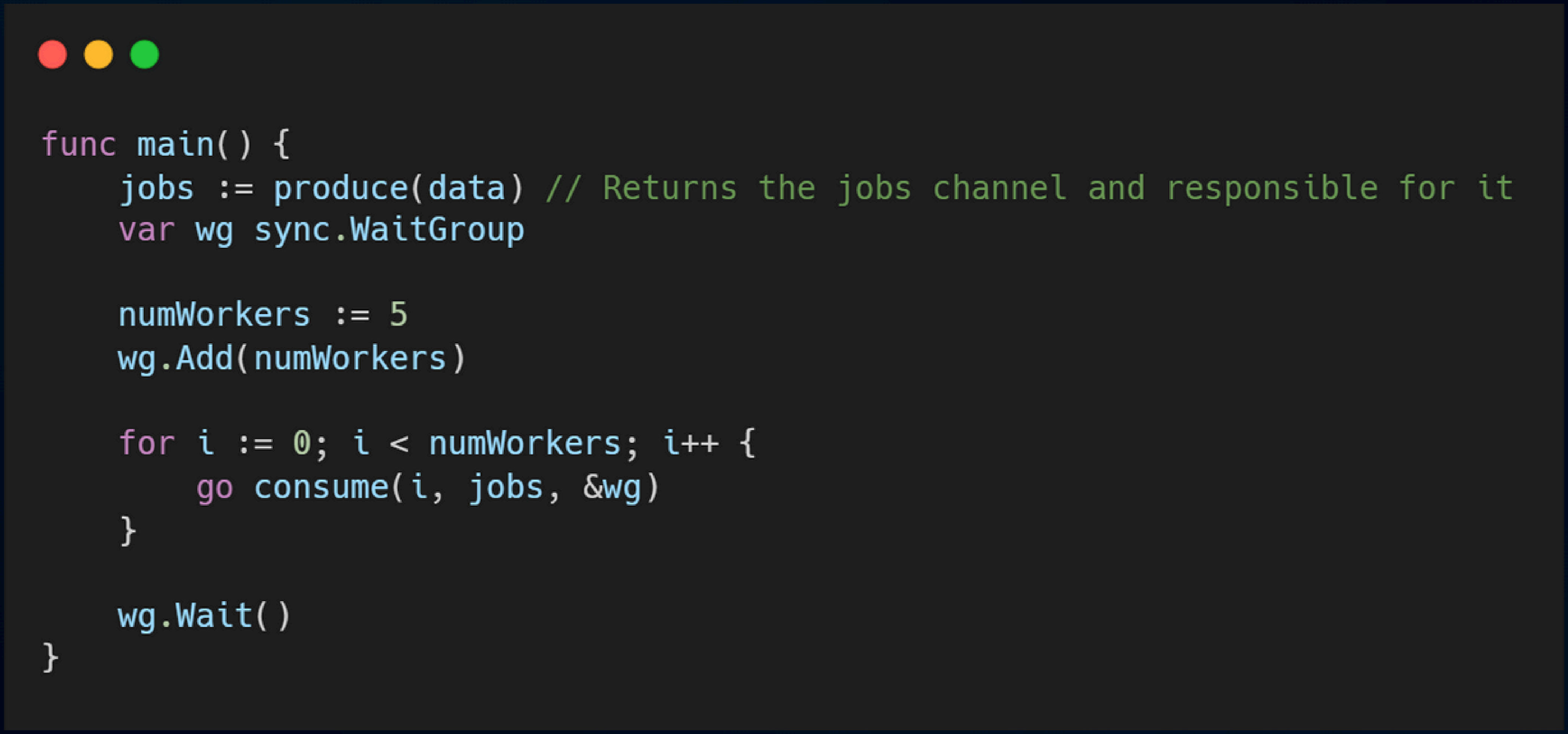


```go
func consume(id int, in <-chan string, wg *sync.WaitGroup) {
    defer wg.Done()
    for item := range in {
        // Simulated work
        log.Printf("Worker %d processed: %s", id, item)
    }
}
```

Scaling with Worker Pools

Parallelizing the Work

One producer can feed many consumers to process incoming data concurrently. This is how we utilize multi-core CPUs.



```go
func main() {
    jobs := produce(data) // Returns the jobs channel and responsible for it
    var wg sync.WaitGroup

    numWorkers := 5
    wg.Add(numWorkers)

    for i := 0; i < numWorkers; i++ {
        go consume(i, jobs, &wg)
    }

    wg.Wait()
}
```

Managing Backpressure

What happens when the buffer is full?

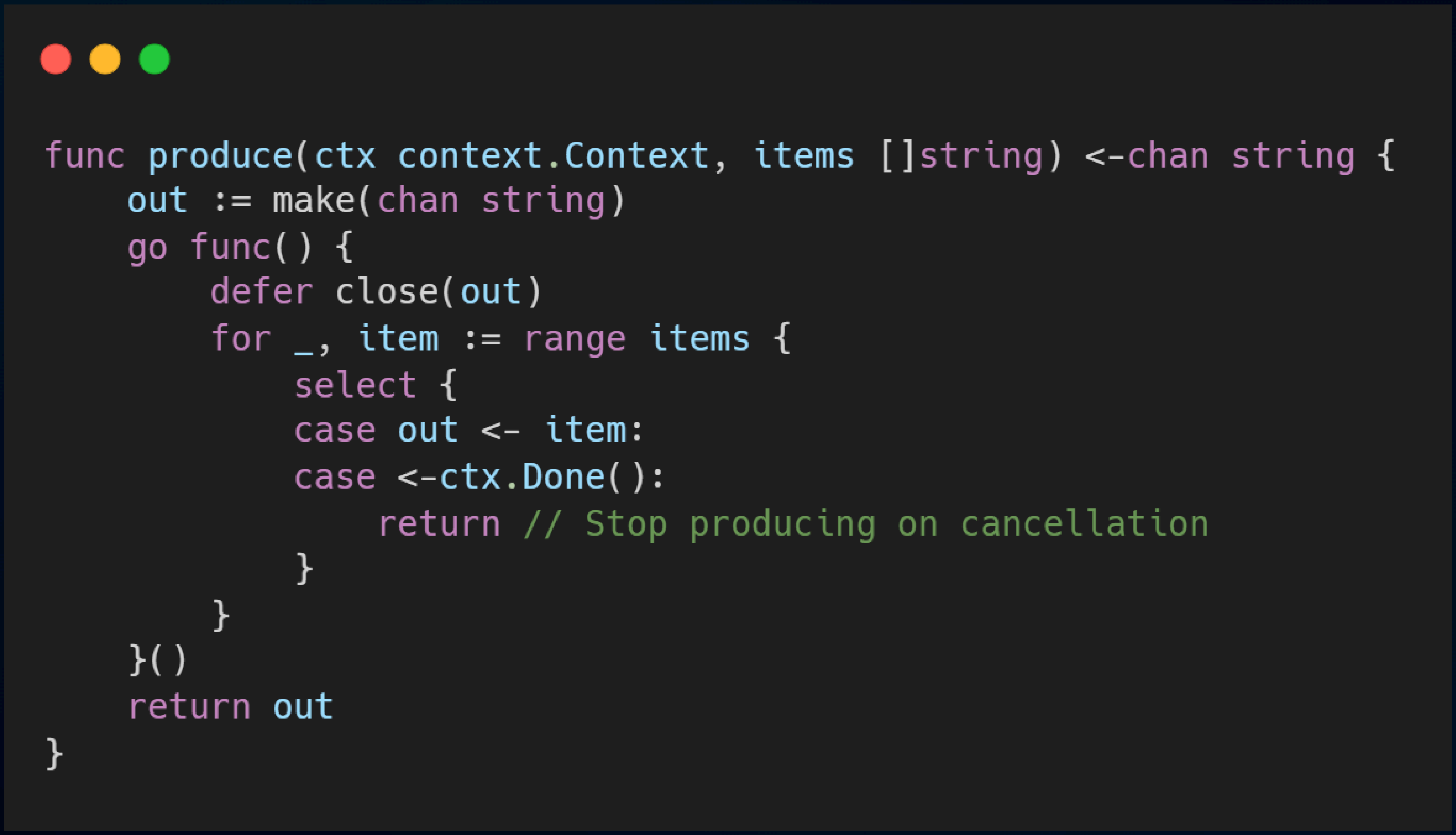
If your buffer fills up, your Producer will block. This is Backpressure.
It prevents your app from running out of memory by forcing the ingestion to slow down to the speed of the consumers.

Memory is never infinite and we can’t always determine processing (consumers’) speed - **never** use an "unlimited" buffer (like a growing slice). You want the system to block or error out eventually so you can see the bottleneck in your metrics.

Graceful Shutdown with Context

Respecting the Lifecycle

What if the user hits Ctrl+C or just cancels while the channel is full? You need to stop producing and allow consumers to finish their current task.



```go
func produce(ctx context.Context, items []string) <-chan string {
    out := make(chan string)
    go func() {
        defer close(out)
        for _, item := range items {
            select {
            case out <- item:
            case <-ctx.Done():
                return // Stop producing on cancellation
            }
        }
    }()
    return out
}
```

The "Poison Pill" Pattern

Advanced: Signalling Stop

Sometimes you need to tell consumers to stop before the channel is empty. A "Poison Pill" is a special value (like an empty string or a specific struct) that tells a worker to exit immediately.

In Go, closing the channel is usually the better "poison pill" because it's built into the language and unblocks every listener at once. Use explicit poison pills only for complex state-machine workers.

Summary:

- **Decouple** ingest from processing to handle spikes.
- **Producer** owns the close().
- **WaitGroup** ensures all work is finished before exit.
- **Backpressure** (buffered channels) protects your memory.

Question: In your pipelines, do you prefer a fixed number of workers (Worker Pool) or spinning up a new goroutine for every single task? What's your performance threshold?