JANUARY 2026 SERIES

FROM GO BUILD TO GO RUN

GOLANG 2026 - NIV RAVE

#42

WORKER POOLS: ORCHESTRATING SCALE

MANAGING HIGH-THROUGHPUT CONCURRENCY WITHOUT CRASHING YOUR SYSTEM

Scaling Responsibly

The Problem with Unbounded Goroutines

If you launch a new goroutine for every incoming request or data point without a limit, you risk exhausting memory or hitting file descriptor limits.

Core Concept:
A Worker Pool allows you to process a massive queue of work using a fixed number of goroutines. This creates "Backpressure," ensuring your system only takes on what it can actually handle.

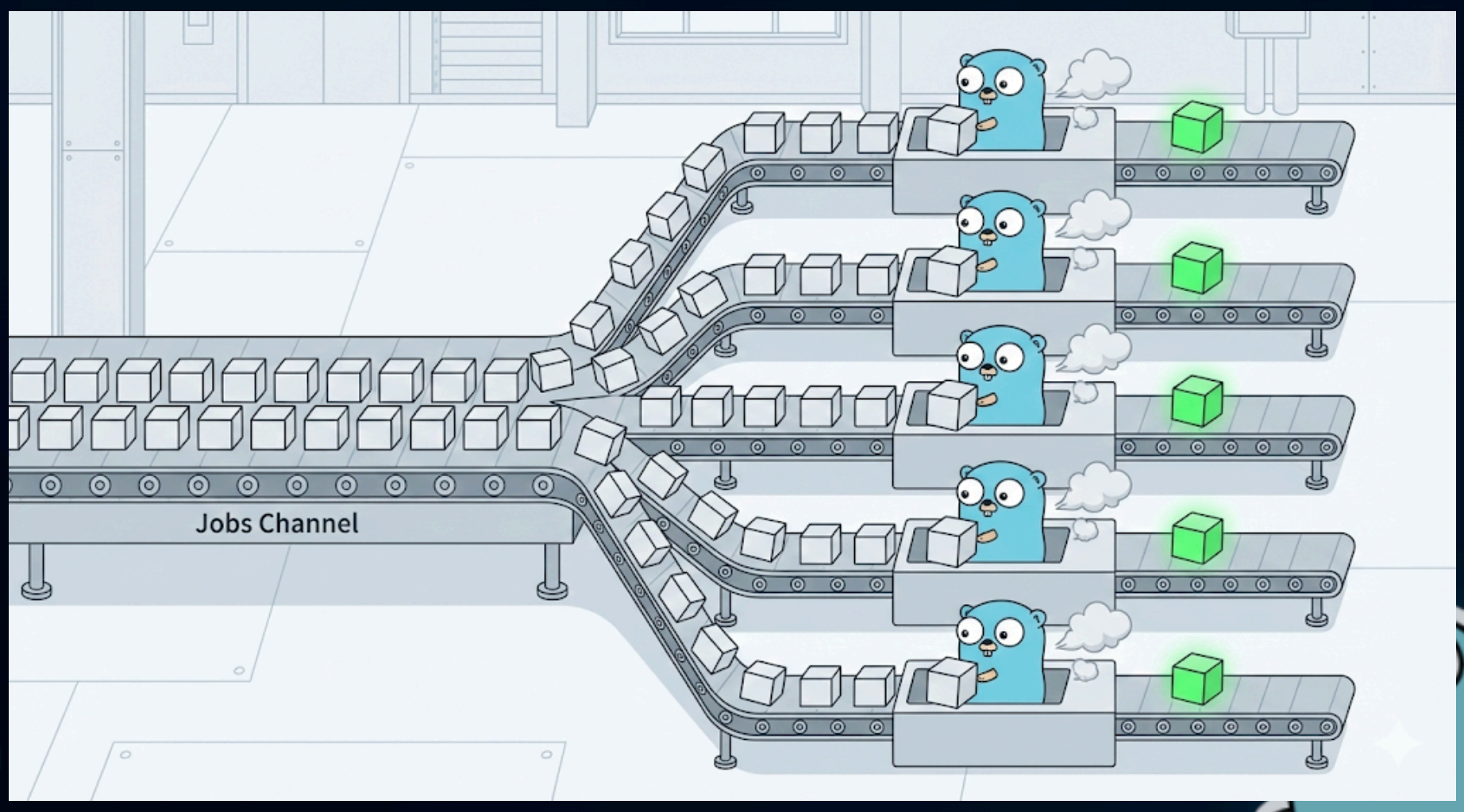
The Fan-Out Strategy

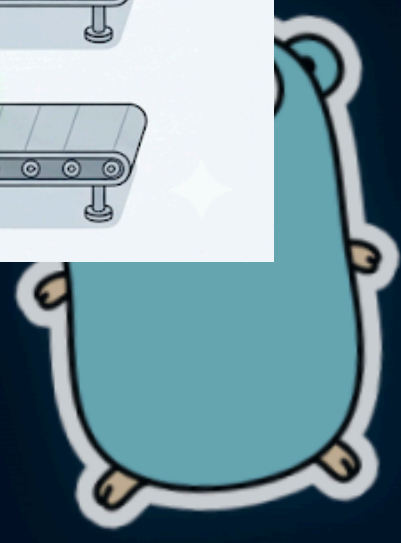
Fan-Out: Distributing the Work

Fan-out occurs when multiple goroutines (workers) read from the same channel until it is closed. This is how we distribute a single stream of tasks across multiple CPU cores.

Pattern:

You create a "Jobs" channel and a set number of worker goroutines. Each worker stays alive, waiting for a job, processing it, and then waiting for the next.





Implementation: The Worker

Defining the Worker Logic

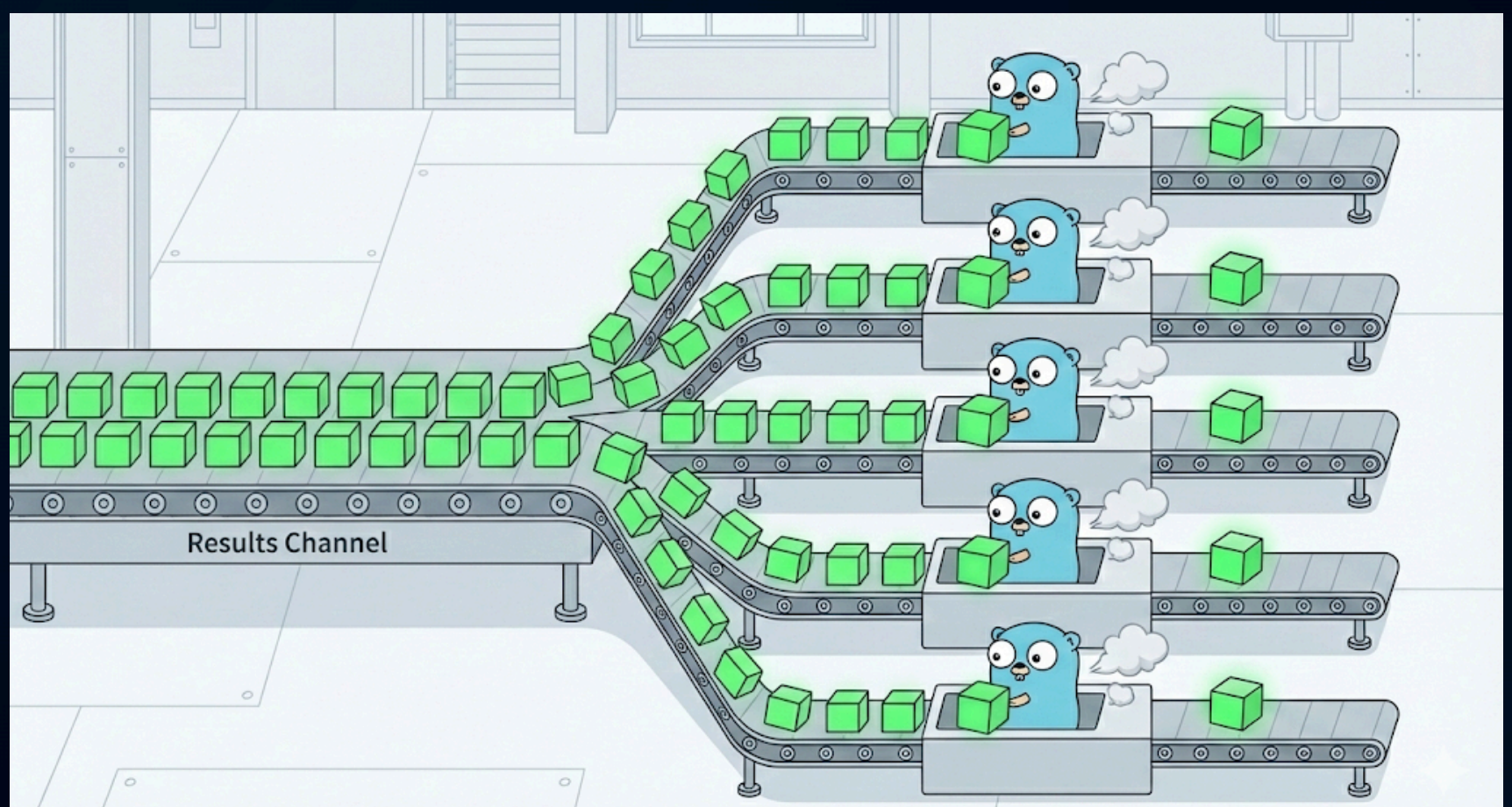
The worker is a simple loop that listens on a channel. It should be designed to handle work continuously until the channel is closed.

```go
func worker(id int, jobs <-chan int, results chan<- int) {
    for j := range jobs {
        // Simulating work
        fmt.Printf("worker %d processing job %d\n", id, j)
        results <- j * 2
    }
}
```

The Fan-In Strategy

Fan-In: Consolidating Results

Fan-in is the inverse: it’s the process of collecting results from multiple worker goroutines and merging them into a single channel for the final consumer.



While the workers are fanned out, a separate "Results" channel collects their output. The main thread (or a dedicated collector) then "fans in" these results to produce the final output.

Synchronizing with WaitGroups

Knowing When the Work is Done

To perform a clean Fan-in, you must know when all workers have finished so you can safely close the results channel.

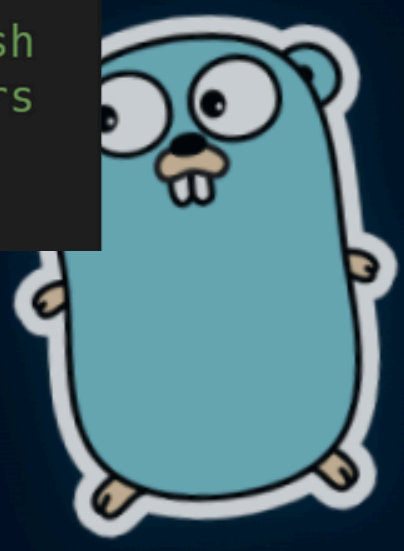
The Strategy:

- Increment a *sync.WaitGroup* for every worker started.
- Each worker calls *wg.Done()* when its job channel is exhausted.
- A separate goroutine waits on the *WaitGroup* and closes the results channel.

```go
var wg sync.WaitGroup

for w := 1; w <= 3; w++ {
    wg.Add(1) // Track each worker
    go func() {
        defer wg.Done() // Signal completion
        worker(jobs, results)
    }()
}

// Collector goroutine
go func() {
    wg.Wait()       // Block until all workers finish
    close(results)  // Safe to close; no more senders
}()
```



Putting it All Together

Implementation: The Orchestrator

The "Orchestrator" sets up the channels, launches the pool, and manages the lifecycle of the entire operation.

```go
jobs := make(chan int, 100)
results := make(chan int, 100)
var wg sync.WaitGroup

// Start 3 workers
for w := 1; w <= 3; w++ {
    wg.Add(1)
    go func(id int) {
        defer wg.Done()
        worker(id, jobs, results)
    }(w)
}

// Close results once all workers are done
go func() {
    wg.Wait()
    close(results)
}()
```

Balancing the Pool Size

Sizing Your Infrastructure

The "ideal" number of workers isn't a magic number; it depends on the nature of the work.

The Logic:

- **CPU-Bound**: Aim for the number of logical CPU cores (*runtime.NumCPU()*).
- **I/O-Bound**: You can scale much higher (hundreds or thousands) since goroutines spend most of their time waiting for network responses.

Tip:

Always make your worker count configurable. What works in development might fail under production load.

Summary:

- **Fan-out**: Spread work across a fixed pool of workers.
- **Fan-in**: Merge results into a single stream.
- **Backpressure**: Use buffered channels to prevent memory spikes.
- **WaitGroups**: Ensure clean shutdowns and channel closures.

Tomorrow, we move to the web: Building an HTTP Server from the ground up - and if you haven’t experienced it with Go yet, you will be amazed with how simple and intuitive it is over here :)