Concurrency in Action: Go Techniques and Tips

Gili Kamma@GopherCon Israel 2025

Intro

- I’ll be talking about concurrency and design patterns
- I will walk through 8 real-world problems and their solutions

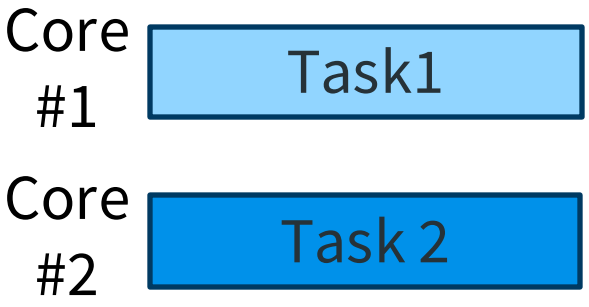
Hello!

- Over 20 years in the industry
- Experienced across diverse domains & languages
- Currently using Go in the Infrastructure domain
- Team leader @Priority

I am Gili Kamma

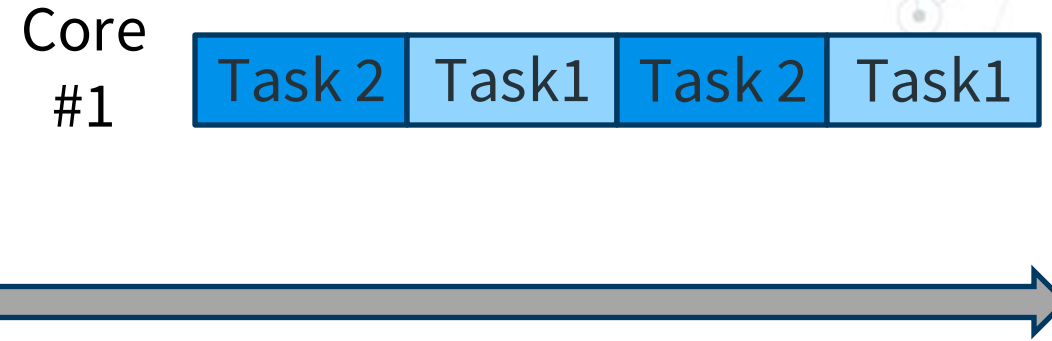
Parallelism Versus Concurrency

Parallelism



The ability to execute multiple tasks simultaneously on multiple processors/cores

Concurrency



The ability to handle multiple tasks by switching between them

Execution Order

```
 8  func GoroutineExample() {
 9      fmt.Println("Hello, World!")
10      defer fmt.Println("Goodbye, World!")
11      for i := 1; i <= 5; i++ {
12          go fmt.Printf("Message %d\n", i)
13      }
14      time.Sleep(3 * time.Second)
15  }
```

Execution Order

```
 8  func GoroutineExample() {
 9      fmt.Println("Hello, World!")
10      defer fmt.Println("Goodbye, World!")
11      for i := 1; i <= 5; i++ {
12          go fmt.Printf("Message %d\n", i)
13      }
14      time.Sleep(3 * time.Second)
15  }
```

Execution Order

```
 8  func GoroutineExample() {
 9      fmt.Println("Hello, World!")
10      defer fmt.Println("Goodbye, World!")
11      for i := 1; i <= 5; i++ {
12          go fmt.Printf("Message %d\n", i)
13      }
14      time.Sleep(3 * time.Second)
15  }
```

```
20  Hello, World!
21  Message 5
22  Message 4
23  Message 1
24  Message 2
25  Message 3
26  Goodbye, World!
```

Order is not kept

1

#1 Non-Blocking Logging

- The application generates continuous log streams
- Logs need to be shipped to a central logging service (e.g., Datadog)

#1 Non-Blocking Logging

- The application generates continuous log streams
- Logs need to be shipped to a central logging service (e.g., Datadog)

So, I need a queue

#1 Non-Blocking Logging

- The application generates continuous log streams
- Logs need to be shipped to a central logging service (e.g., Datadog)

Is the channel a good fit?

Channel Versus Queue

Channel	Queue
Fixed/zero size	Dynamic
Blocking by default	Non-blocking

Channel

Channel Versus Queue

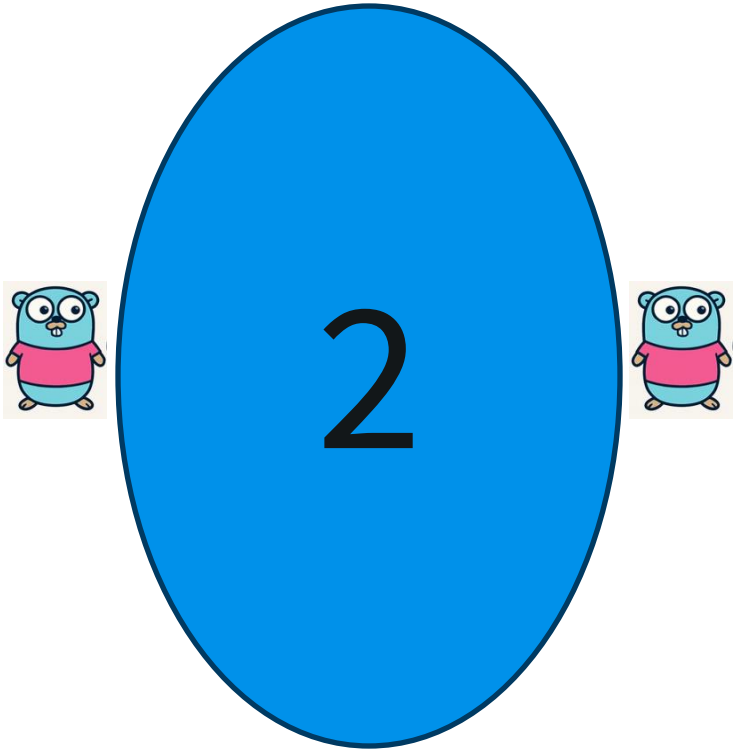
Channel	Queue
Fixed/zero size	Dynamic
Blocking by default	Non-blocking

Channel is not the right answer!

* https://pkg.go.dev/container/list

** list is not a thread safe

Channel is not a queue


2

#2 Email Scaling Challenge

- You have an email service that needs to send 100,000 marketing emails to customers
- Sending them sequentially is too
- But creating a goroutine for each email would consume too much memory
- You need to control the number of concurrent operations

Worker Pool

Use it when you need to manage and optimize the execution of multiple tasks concurrently

- Goroutines reuse
- Concurrency control

Worker Pool

```
// Worker function that processes tasks
func worker(id int, tasks <-chan int, results chan<- string) {
    for _ = range tasks {
        a := rand.IntN(1000) + rand.IntN(1000)
        results <- fmt.Sprintf("Result: %d, worker %d", a, id)
    }
}
```

Worker Pool

```
// Worker function that processes tasks
func worker(id int, tasks <-chan int, results chan<- string) {
    for _ = range tasks {
        a := rand.IntN(1000) + rand.IntN(1000)
        results <- fmt.Sprintf("Result: %d, worker %d", a, id)
    }
}
```

Worker Pool

```
// Worker function that processes tasks
func worker(id int, tasks <-chan int, results chan<- string) {
    for _ = range tasks {
        a := rand.IntN(1000) + rand.IntN(1000)
        results <- fmt.Sprintf("Result: %d, worker %d", a, id)
    }
}
```

```go
func WorkerPoolExample(numWorkers int, numTasks int) {
    // Channels for tasks and results
    tasks := make(chan int, numTasks)
    results := make(chan string, numTasks)

    // Start workers
    for i := 1; i <= numWorkers; i++ {
        go worker(i, tasks, results)
    }

    // Send tasks to the task channel
    for i := 1; i <= numTasks; i++ {
        tasks <- i
    }
    close(tasks)

    // Collect results
    var i int
    for i = 1; i <= numTasks; i++ {
        <-results
    }
}
```

```
16  func WorkerPoolExample(numWorkers int, numTasks int) {
17      // Channels for tasks and results
18      tasks := make(chan int, numTasks)
19      results := make(chan string, numTasks)
20
21      // Start workers
22      for i := 1; i <= numWorkers; i++ {
23          go worker(i, tasks, results)
24      }
```

```
16  func WorkerPoolExample(numWorkers int, numTasks int) {
17      // Channels for tasks and results
18      tasks := make(chan int, numTasks)
19      results := make(chan string, numTasks)
20
21      // Start workers
22      for i := 1; i <= numWorkers; i++ {
23          go worker(i, tasks, results)
24      }
```

```
16  func WorkerPoolExample(numWorkers int, numTasks int) {
17      // Channels for tasks and results
18      tasks := make(chan int, numTasks)
19      results := make(chan string, numTasks)
20
21      // Start workers
22      for i := 1; i <= numWorkers; i++ {
23          go worker(i, tasks, results)
24      }
```

```
26      // Send tasks to the task channel
27      for i := 1; i <= numTasks; i++ {
28          tasks <- i
29      }
30      close(tasks)
31
32      // Collect results
33      var i int
34      for i = 1; i <= numTasks; i++ {
35          <-results
36      }
37  }
```

```
26      // Send tasks to the task channel
27      for i := 1; i <= numTasks; i++ {
28          tasks <- i
29      }
30      close(tasks)
31
32      // Collect results
33      var i int
34      for i = 1; i <= numTasks; i++ {
35          <-results
36      }
37  }
```

```go
func WorkerPoolExample(numWorkers int, numTasks int) {
    // Channels for tasks and results
    tasks := make(chan int, numTasks)
    results := make(chan string, numTasks)

    // Start workers
    for i := 1; i <= numWorkers; i++ {
        go worker(i, tasks, results)
    }

    // Send tasks to the task channel
    for i := 1; i <= numTasks; i++ {
        tasks <- i
    }
    close(tasks)

    // Collect results
    var i int
    for i = 1; i <= numTasks; i++ {
        <-results
    }
}
```

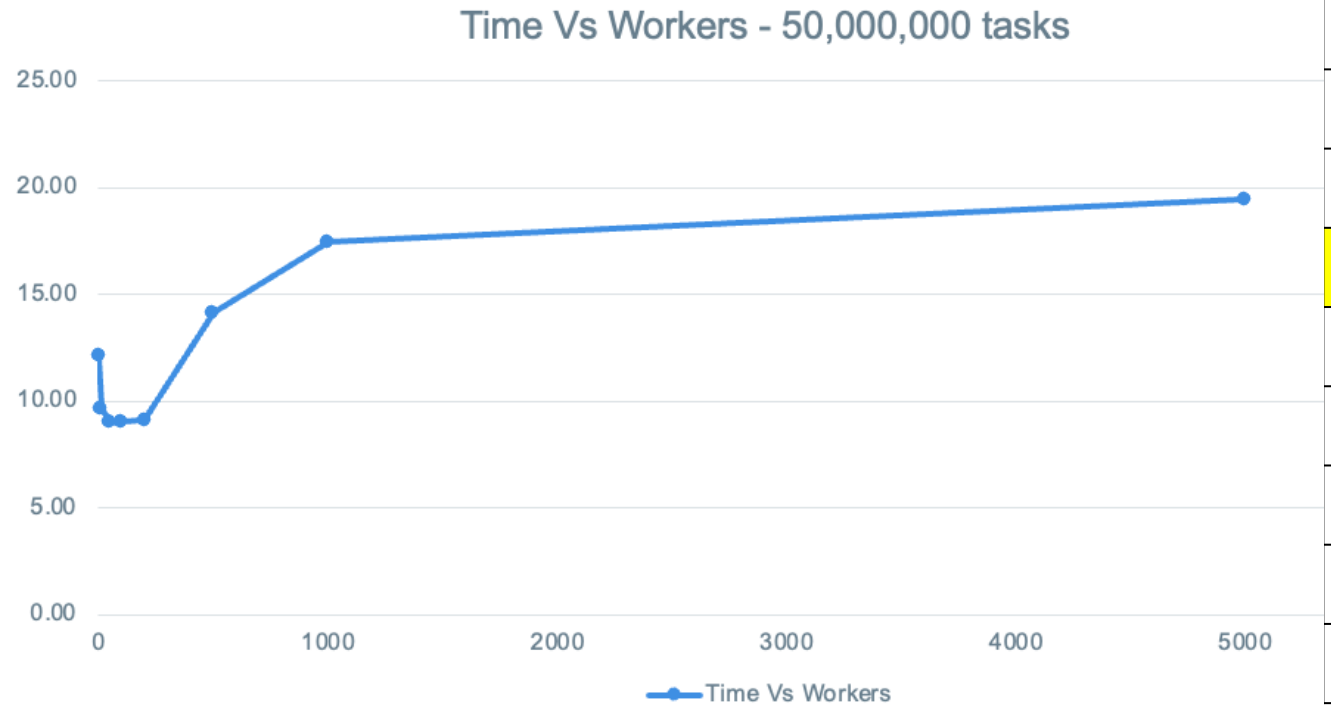
```go
func BenchmarkWorkerPoolExample(b *testing.B) {
    benchmarks := []struct {
        workers int
        tasks   int
    }{
        {workers: 1, tasks: 50_000_000},
        {workers: 10, tasks: 50_000_000},
        {workers: 50, tasks: 50_000_000},
        {workers: 100, tasks: 50_000_000},
        {workers: 200, tasks: 50_000_000},
        {workers: 500, tasks: 50_000_000},
        {workers: 1000, tasks: 50_000_000},
        {workers: 5000, tasks: 50_000_000},
    }
    for _, bm := range benchmarks {
        b.Run(
            fmt.Sprintf("%d_workers_%d_tasks", bm.workers, bm.tasks),
            func(b *testing.B) {
                for b.Loop() {
                    WorkerPoolExample(bm.workers, bm.tasks)
                }
            },
        )
    }
}
```

```
for _, bm := range benchmarks {
    b.Run(
        fmt.Sprintf("%d_workers_%d_tasks", bm.workers, bm.tasks),
        func(b *testing.B) {
            for b.Loop() {
                WorkerPoolExample(bm.workers, bm.tasks)
            }
        },
    )
}
```

```go
func BenchmarkWorkerPoolExample(b *testing.B) {
    benchmarks := []struct {
        workers int
        tasks   int
    }{
        {workers: 1, tasks: 50_000_000},
        {workers: 10, tasks: 50_000_000},
        {workers: 50, tasks: 50_000_000},
        {workers: 100, tasks: 50_000_000},
        {workers: 200, tasks: 50_000_000},
        {workers: 500, tasks: 50_000_000},
        {workers: 1000, tasks: 50_000_000},
        {workers: 5000, tasks: 50_000_000},
    }
```

```go
func BenchmarkWorkerPoolExample(b *testing.B) {
    benchmarks := []struct {
        workers int
        tasks   int
    }{
        {workers: 1, tasks: 50_000_000},
        {workers: 10, tasks: 50_000_000},
        {workers: 50, tasks: 50_000_000},
        {workers: 100, tasks: 50_000_000},
        {workers: 200, tasks: 50_000_000},
        {workers: 500, tasks: 50_000_000},
        {workers: 1000, tasks: 50_000_000},
        {workers: 5000, tasks: 50_000_000},
    }
```

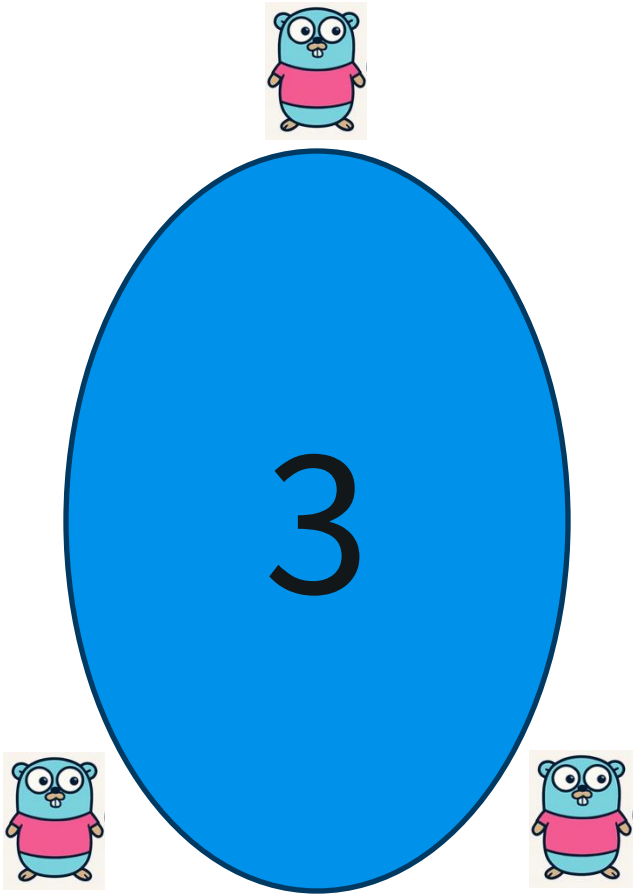
Worker Pool Performance – (11 cores)



Workers	Time[sec]
1	12.14
10	9.69
50	9.02
100	9.06
200	9.13
500	14.13
1000	17.47
5000	19.47

Measure Everything – Avoid Assumptions

*Use **Worker Pool** pattern when you want concurrency control and goroutine reuse*


3

#3 Connection Thundering Herd

- Service manages thousands of endpoints - client connections
- On startup/restart, all endpoints attempt to connect simultaneously
- Each connection requires configuration data from external service
- Result: the external service is bombarded with thousands of concurrent requests and crash

Semaphore Pattern

- A semaphore manages access to a shared resource
- We can use it to limit concurrency in a simple way

```
55 // Semaphore with 5 slots
56 var sem = make(chan struct{}, 5)
57 sem <- struct{}{} // Acquire
58 // Access to limited resource
59 <-sem // Release
```

```
55 // Semaphore with 5 slots
56 var sem = make(chan struct{}, 5)
57 sem <- struct{}{} // Acquire
58 // Access to limited resource
59 <-sem // Release
```

```
55 // Semaphore with 5 slots
56 var sem = make(chan struct{}, 5)
57 sem <- struct{}{} // Acquire
58 // Access to limited resource
59 <-sem // Release
```

```
55 // Semaphore with 5 slots
56 var sem = make(chan struct{}, 5)
57 sem <- struct{}{} // Acquire
58 // Access to limited resource
59 <-sem // Release
```

```
55 // Semaphore with 5 slots
56 var sem = make(chan struct{}, 5)
57 sem <- struct{}{} // Acquire
58 // Access to limited resource
59 <-sem // Release
```

```
55 // Semaphore with 5 slots
56 var sem = make(chan struct{}, 5)
57 sem <- struct{}{} // Acquire
58 // Access to limited resource
59 <-sem // Release
```

*golang.org/x/sync/semaphore

```
61 // Mutex is a private case of Semaphore
62 var sem = make(chan struct{}, 1)
63 sem <- struct{}{} // Acquire
64 // Critical section
65 <-sem // Release
```

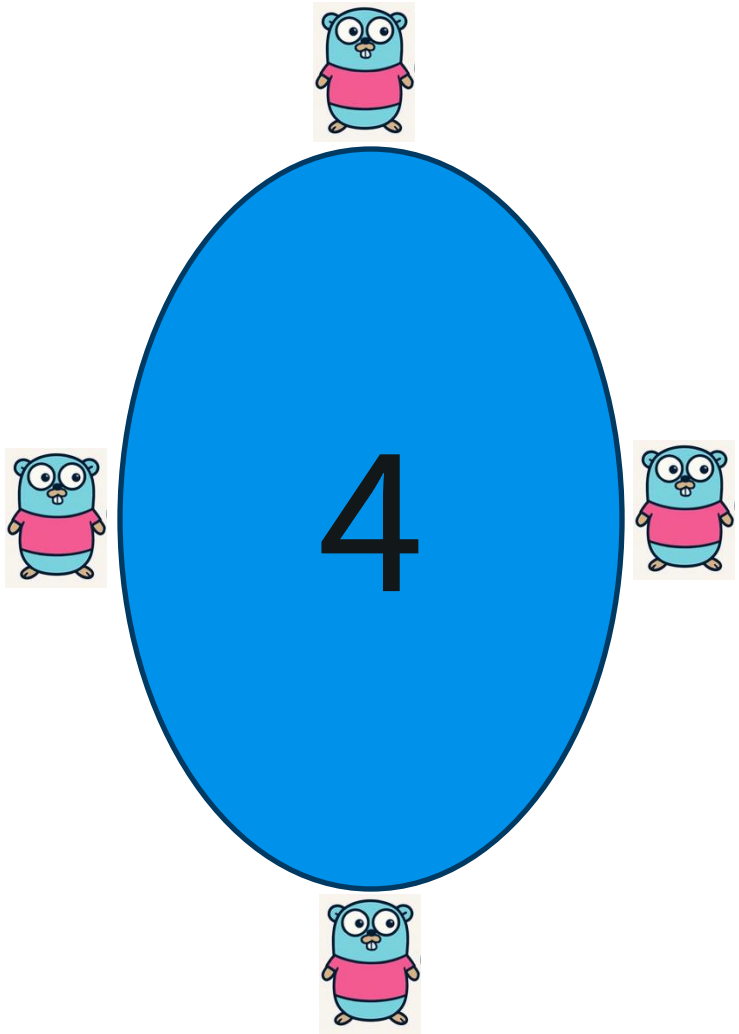
```
61 // Mutex is a private case of Semaphore
62 var sem = make(chan struct{}, 1)
63 sem <- struct{}{} // Acquire
64 // Critical section
65 <-sem // Release
```

```
76 func (pip *parallelImagePuller) pullImage(ctx context.Context, spec kubecontainer.
   ImageSpec, credentials []credentialprovider.TrackedAuthConfig, pullChan chan<-
   pullResult, podSandboxConfig *runtimeapi.PodSandboxConfig) {
    go func() {
        if pip.tokens != nil {
            pip.tokens <- struct{}{}
            defer func() { <-pip.tokens }()
        }
        startTime := time.Now()
86          // Getting the image size with best effort, ignoring the error.
87          size, _ = pip.imageService.GetImageSize(ctx, spec)
88      }
89      pullChan <- pullResult{
90          imageRef:        imageRef,
91          imageSize:       size,
92          err:             err,
93          pullDuration:    time.Since(startTime),
94          credentialsUsed: creds,
95      }
96  }()
97 }
```

From: kubernetes/pkg/kubelet/images/puller.go

Attribute	Semaphore	Worker pool
Concurrency control	+	+
Goroutine reuse	-	+
Simplicity	+	-

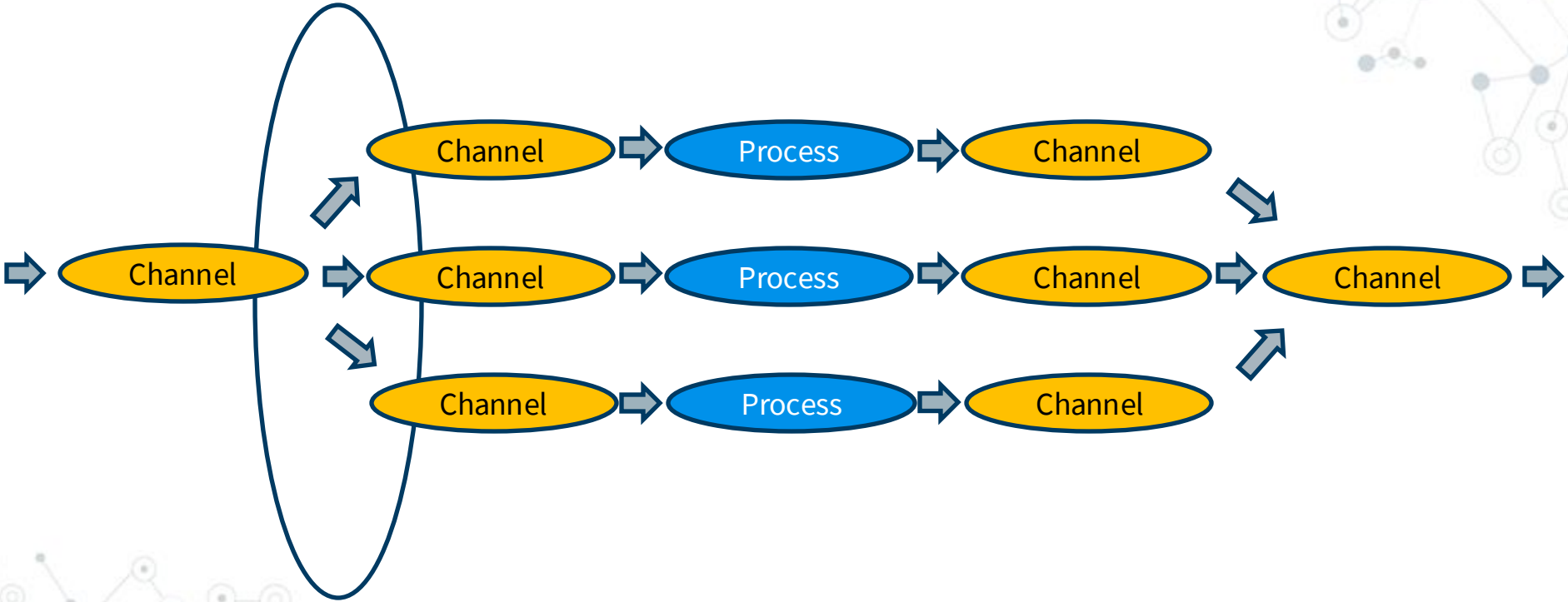
*Use the **Semaphore** pattern for simple concurrency control*


4

#4 Log Distribution System

- Large batches of mixed log entries (Errors, Warnings, Info)
- Each log type requires different handling and destination

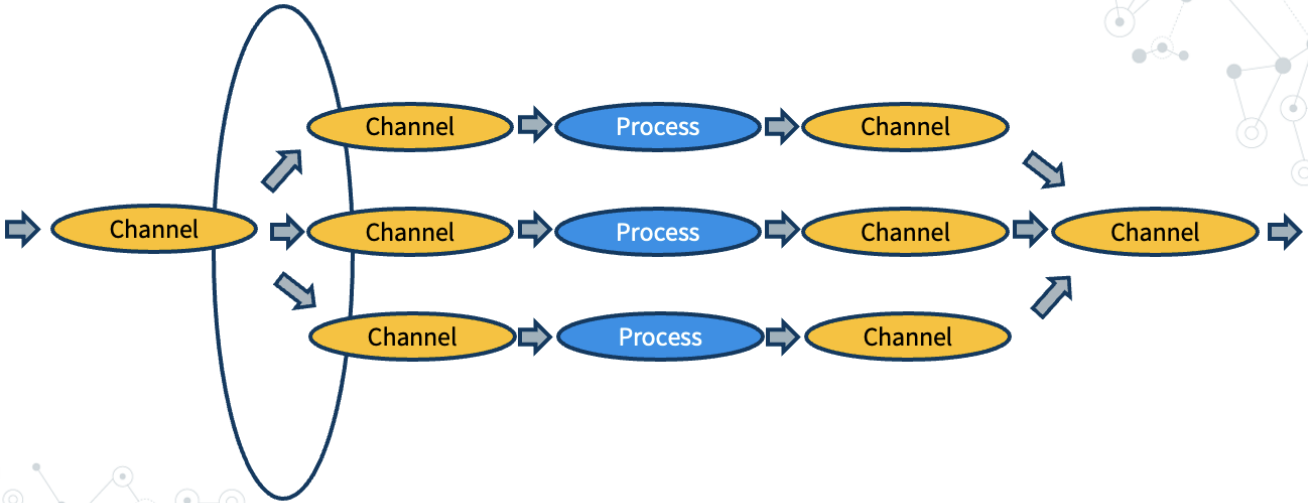
Fan-Out



Fan-Out

Fan-Out

- Independent Processing: Handles errors, warnings, and info logs in separate channels
- Scalability: Allocate more workers to specific channels



```go
func FanOut(source <-chan LogMsg) (<-chan LogMsg, <-chan LogMsg) {
    errors := make(chan LogMsg)
    infos := make(chan LogMsg)
    go func() {
        defer close(errors)
        defer close(infos)
        for msg := range source {
            switch msg.Type {
            case "error":
                errors <- msg
            case "info":
                infos <- msg
            }
        }
    }()
    return errors, infos
}
```

```go
func FanOut(source <-chan LogMsg) (<-chan LogMsg, <-chan LogMsg) {
    errors := make(chan LogMsg)
    infos := make(chan LogMsg)
    go func() {
        defer close(errors)
        defer close(infos)
        for msg := range source {
            switch msg.Type {
            case "error":
                errors <- msg
            case "info":
                infos <- msg
            }
        }
    }()
    return errors, infos
}
```

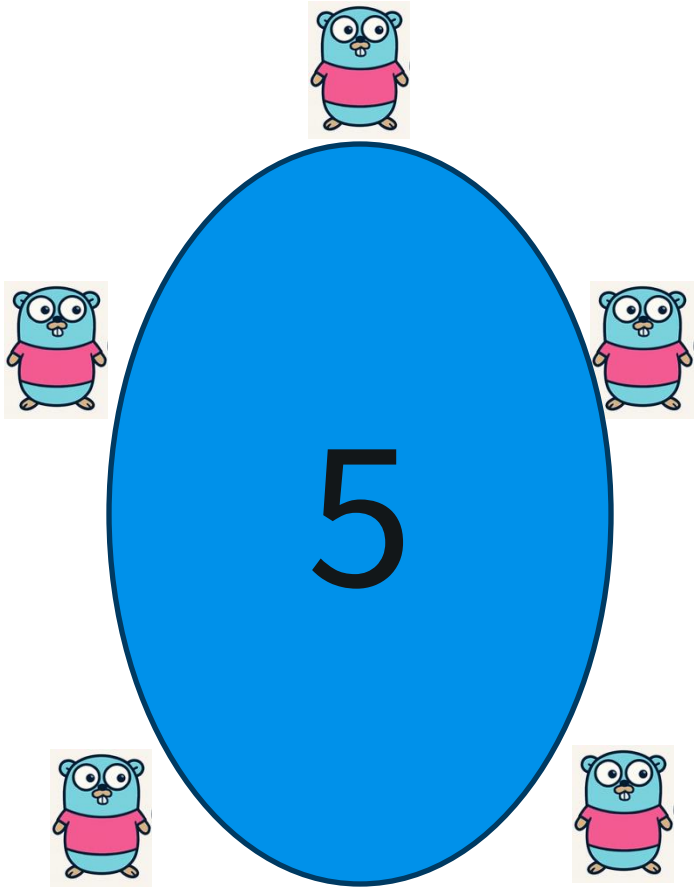
```go
func FanOut(source <-chan LogMsg) (<-chan LogMsg, <-chan LogMsg) {
    errors := make(chan LogMsg)
    infos := make(chan LogMsg)
    go func() {
        defer close(errors)
        defer close(infos)
        for msg := range source {
            switch msg.Type {
            case "error":
                errors <- msg
            case "info":
                infos <- msg
            }
        }
    }()
    return errors, infos
}
```

```go
func FanOut(source <-chan LogMsg) (<-chan LogMsg, <-chan LogMsg) {
    errors := make(chan LogMsg)
    infos := make(chan LogMsg)
    go func() {
        defer close(errors)
        defer close(infos)
        for msg := range source {
            switch msg.Type {
            case "error":
                errors <- msg
            case "info":
                infos <- msg
            }
        }
    }()
    return errors, infos
}
```

```go
func FanOut(source <-chan LogMsg) (<-chan LogMsg, <-chan LogMsg) {
    errors := make(chan LogMsg)
    infos := make(chan LogMsg)
    go func() {
        defer close(errors)
        defer close(infos)
        for msg := range source {
            switch msg.Type {
            case "error":
                errors <- msg
            case "info":
                infos <- msg
            }
        }
    }()
    return errors, infos
}
```

```go
func FanOut(source <-chan LogMsg) (<-chan LogMsg, <-chan LogMsg) {
    errors := make(chan LogMsg)
    infos := make(chan LogMsg)
    go func() {
        defer close(errors)
        defer close(infos)
        for msg := range source {
            switch msg.Type {
            case "error":
                errors <- msg
            case "info":
                infos <- msg
            }
        }
    }()
    return errors, infos
}
```

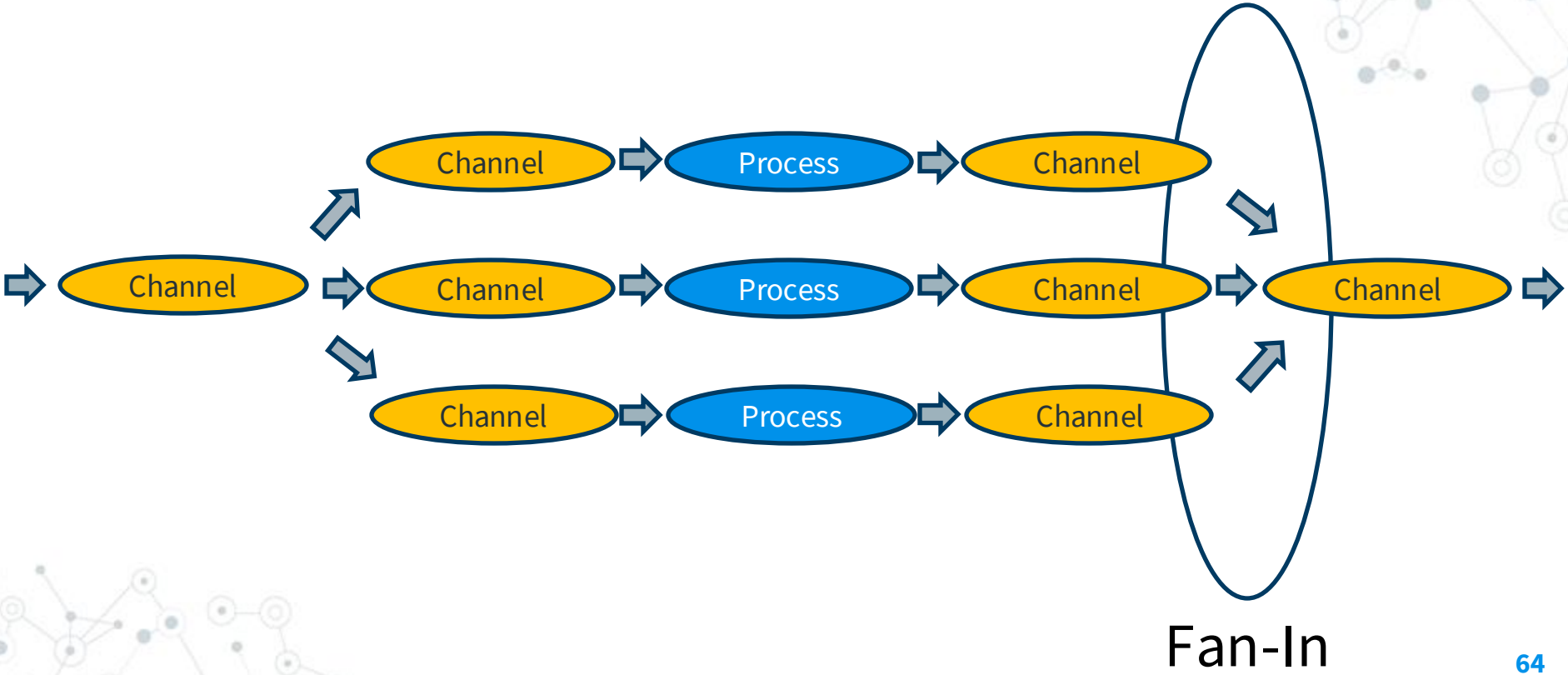
*Use the **Fan-Out** pattern for independent processing and scalable handling of channel output*


5

#5 Data Pipeline Integration

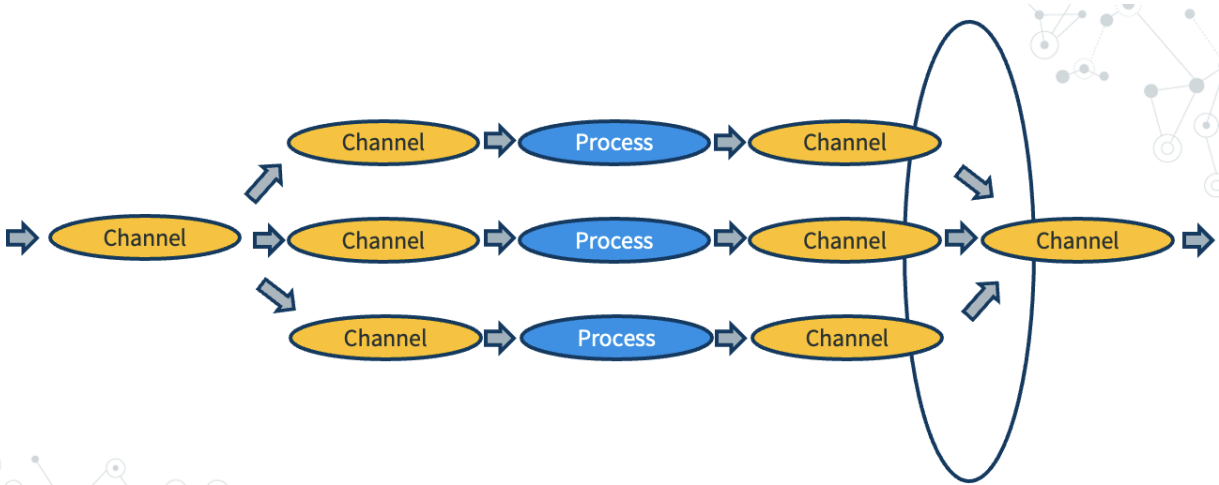
- Nightly data pipeline needs to merge data from multiple databases
- Sources: MySQL, PostgreSQL, MongoDB
- Need to transform and load into a data lake

Fan-In


Channel
Channel
Process
Channel
Channel
Process
Channel
Channel
Channel
Process
Channel
Fan-In

Fan-In

- Job Results: Aggregating results from multiple worker goroutines after they complete their tasks.
- Combining Data: Merging processed data pieces back into a unified stream.



Fan-In

```
func fanIn(sources ...<-chan string) <-chan string {
    var wg sync.WaitGroup
    out := make(chan string)
    wg.Add(len(sources))
    for _, c := range sources {
        go func(c <-chan string) {
            defer wg.Done()
            for n := range c {
                out <- n
            }
        }(c)
    }
```

```go
func fanIn(sources ...<-chan string) <-chan string {
    var wg sync.WaitGroup
    out := make(chan string)
    wg.Add(len(sources))
    for _, c := range sources {
        go func(c <-chan string) {
            defer wg.Done()
            for n := range c {
                out <- n
            }
        }(c)
    }
```

```
func fanIn(sources ...<-chan string) <-chan string {
    var wg sync.WaitGroup
    out := make(chan string)
    wg.Add(len(sources))
    for _, c := range sources {
        go func(c <-chan string) {
            defer wg.Done()
            for n := range c {
                out <- n
            }
        }(c)
    }
```

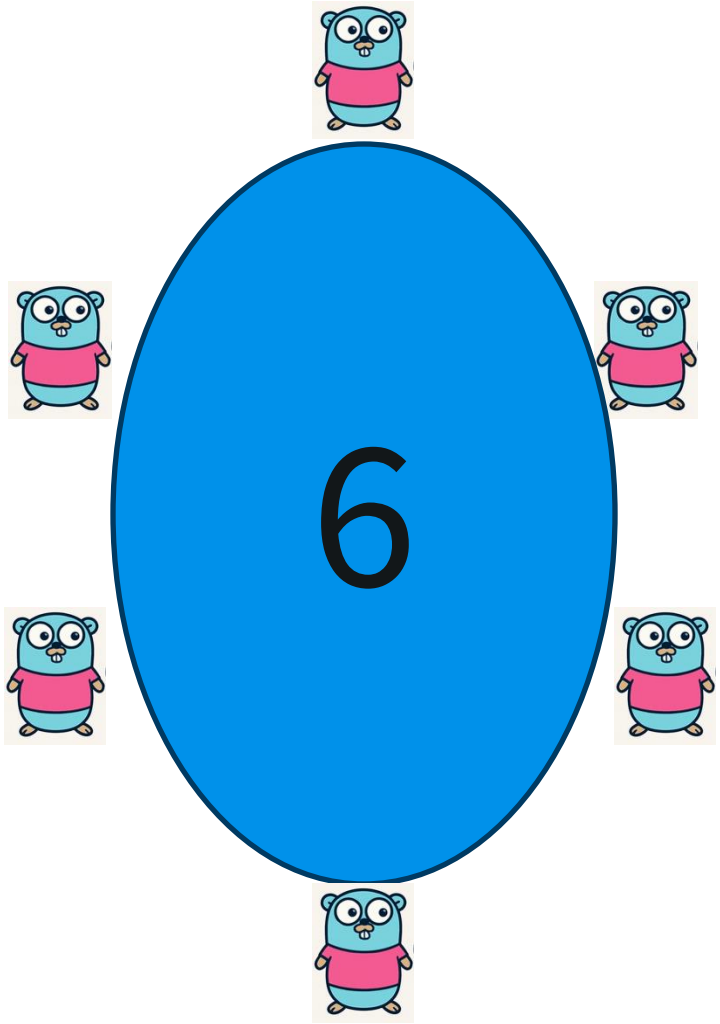
```go
func fanIn(sources ...<-chan string) <-chan string {
    var wg sync.WaitGroup
    out := make(chan string)
    wg.Add(len(sources))
    for _, c := range sources {
        go func(c <-chan string) {
            defer wg.Done()
            for n := range c {
                out <- n
            }
        }(c)
    }
```

```go
func fanIn(sources ...<-chan string) <-chan string {
    var wg sync.WaitGroup
    out := make(chan string)
    wg.Add(len(sources))
    for _, c := range sources {
        go func(c <-chan string) {
            defer wg.Done()
            for n := range c {
                out <- n
            }
        }(c)
    }
```

```
    go func() {
        wg.Wait()
        close(out)
    }()
    return out
}
```

```
    go func() {
        wg.Wait()
        close(out)
    }()
    return out
}
```

*Use **Fan-In** to merge data or collect results*

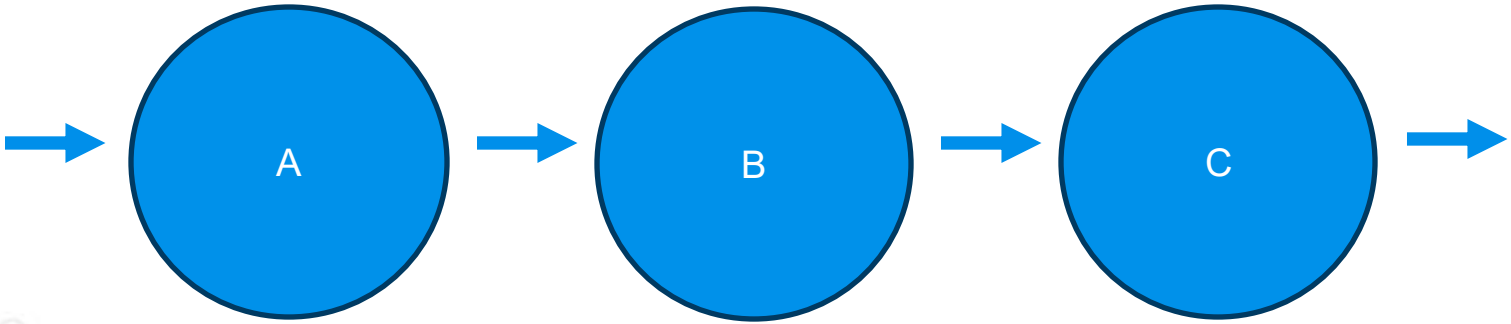

6

Problem 6

- Network packets need sequential processing stages for deep packet inspection:
  - Capture raw packets
  - Decode packet headers (TCP/IP/UDP)
  - Protocol classification (HTTP/DNS/TLS)
  - Payload inspection
  - Threat detection
  - Log/Store suspicious packets

Pipeline pattern

This design pattern is used to process data in stages, where each stage performs a specific operation and passes the result to the next stage



```
44  func PipelineExample() {
45      lines := make(chan string)
46      wordCounts := make(chan int)
47      var wg sync.WaitGroup
48
49      wg.Add(1)
50
51      go readLines("example.txt", lines)
52      go countWords(lines, wordCounts)
53      go printCounts(wordCounts, &wg)
54
55      wg.Wait()
56  }
```

```
44  func PipelineExample() {
45      lines := make(chan string)
46      wordCounts := make(chan int)
47      var wg sync.WaitGroup
48
49      wg.Add(1)
50
51      go readLines("example.txt", lines)
52      go countWords(lines, wordCounts)
53      go printCounts(wordCounts, &wg)
54
55      wg.Wait()
56  }
```

```
44  func PipelineExample() {
45      lines := make(chan string)
46      wordCounts := make(chan int)
47      var wg sync.WaitGroup
48
49      wg.Add(1)
50
51      go readLines("example.txt", lines)
52      go countWords(lines, wordCounts)
53      go printCounts(wordCounts, &wg)
54
55      wg.Wait()
56  }
```

```
44  func PipelineExample() {
45      lines := make(chan string)
46      wordCounts := make(chan int)
47      var wg sync.WaitGroup
48
49      wg.Add(1)
50
51      go readLines("example.txt", lines)
52      go countWords(lines, wordCounts)
53      go printCounts(wordCounts, &wg)
54
55      wg.Wait()
56  }
```

```
44  func PipelineExample() {
45      lines := make(chan string)
46      wordCounts := make(chan int)
47      var wg sync.WaitGroup
48
49      wg.Add(1)
50
51      go readLines("example.txt", lines)
52      go countWords(lines, wordCounts)
53      go printCounts(wordCounts, &wg)
54
55      wg.Wait()
56  }
```

```go
// Stage 1: Read lines from a file
func readLines(filename string, out chan<- string) {
    file, err := os.Open(filename)
    if err != nil {
        fmt.Println("Error opening file:", err)
        close(out)
        return
    }
    defer file.Close()

    scanner := bufio.NewScanner(file)
    for scanner.Scan() {
        out <- scanner.Text()
    }
    close(out)
}
```

```go
// Stage 1: Read lines from a file
func readLines(filename string, out chan<- string) {
    file, err := os.Open(filename)
    if err != nil {
        fmt.Println("Error opening file:", err)
        close(out)
        return
    }
    defer file.Close()

    scanner := bufio.NewScanner(file)
    for scanner.Scan() {
        out <- scanner.Text()
    }
    close(out)
}
```

```go
// Stage 1: Read lines from a file
func readLines(filename string, out chan<- string) {
    file, err := os.Open(filename)
    if err != nil {
        fmt.Println("Error opening file:", err)
        close(out)
        return
    }
    defer file.Close()

    scanner := bufio.NewScanner(file)
    for scanner.Scan() {
        out <- scanner.Text()
    }
    close(out)
}
```

```go
// Stage 1: Read lines from a file
func readLines(filename string, out chan<- string) {
    file, err := os.Open(filename)
    if err != nil {
        fmt.Println("Error opening file:", err)
        close(out)
        return
    }
    defer file.Close()

    scanner := bufio.NewScanner(file)
    for scanner.Scan() {
        out <- scanner.Text()
    }
    close(out)
}
```

```go
// Stage 1: Read lines from a file
func readLines(filename string, out chan<- string) {
    file, err := os.Open(filename)
    if err != nil {
        fmt.Println("Error opening file:", err)
        close(out)
        return
    }
    defer file.Close()

    scanner := bufio.NewScanner(file)
    for scanner.Scan() {
        out <- scanner.Text()
    }
    close(out)
}
```

```
28  // Stage 2: Count words in each line
29  func countWords(in <-chan string, out chan<- int) {
30      for line := range in {
31          out <- len(strings.Fields(line))
32      }
33      close(out)
34  }
```

```
28  // Stage 2: Count words in each line
29  func countWords(in <-chan string, out chan<- int) {
30      for line := range in {
31          out <- len(strings.Fields(line))
32      }
33      close(out)
34  }
```

```
28  // Stage 2: Count words in each line
29  func countWords(in <-chan string, out chan<- int) {
30      for line := range in {
31          out <- len(strings.Fields(line))
32      }
33      close(out)
34  }
```

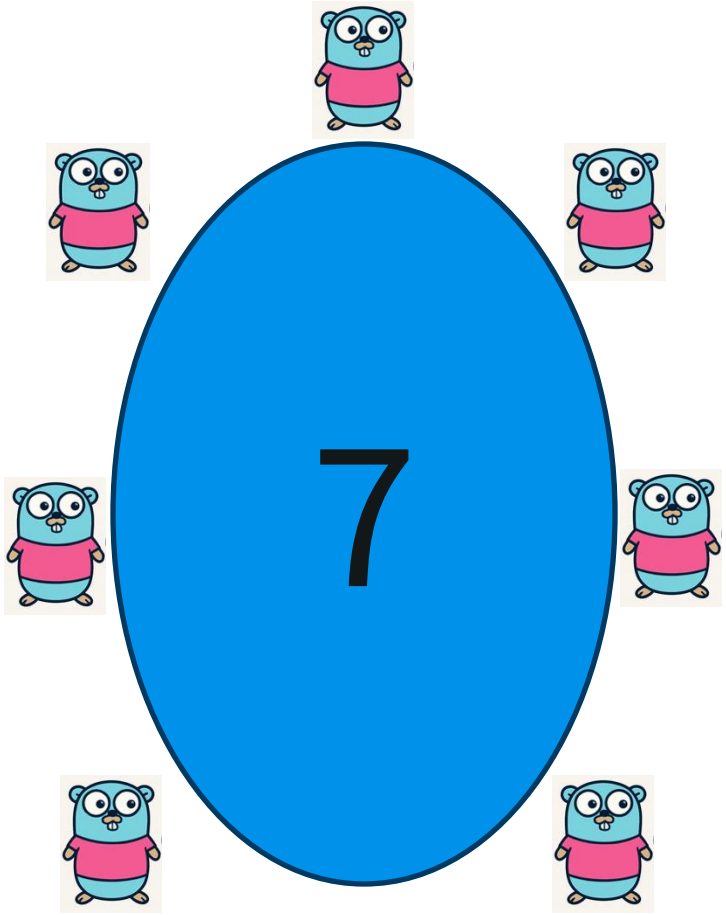
```
36  // Stage 3: Print word counts
37  func printCounts(in <-chan int, wg *sync.WaitGroup) {
38      defer wg.Done()
39      for count := range in {
40          fmt.Println("Word count:", count)
41      }
42  }
```

```
36  // Stage 3: Print word counts
37  func printCounts(in <-chan int, wg *sync.WaitGroup) {
38      defer wg.Done()
39      for count := range in {
40          fmt.Println("Word count:", count)
41      }
42  }
```

```
36  // Stage 3: Print word counts
37  func printCounts(in <-chan int, wg *sync.WaitGroup) {
38      defer wg.Done()
39      for count := range in {
40          fmt.Println("Word count:", count)
41      }
42  }
```

```
44  func PipelineExample() {
45      lines := make(chan string)
46      wordCounts := make(chan int)
47      var wg sync.WaitGroup
48
49      wg.Add(1)
50
51      go readLines("example.txt", lines)
52      go countWords(lines, wordCounts)
53      go printCounts(wordCounts, &wg)
54
55      wg.Wait()
56  }
```

*Use **Pipeline** pattern when you have streaming data and multi-step logic workflow*


7

#7 Graceful Shutdown

- Need to stop a goroutine externally (e.g., for graceful shutdown).

```
go func() {
    for {
        select {
        case msg := <-ch:
            fmt.Println("Received:", msg)
        case <-done:
            fmt.Println("Shutting down...")
            return
        }
    }
}()
ch <- "Hello"
close(done)
```

```
go func() {
    for {
        select {
        case msg := <-ch:
            fmt.Println("Received:", msg)
        case <-done:
            fmt.Println("Shutting down...")
            return
        }
    }
}()
ch <- "Hello"
close(done)
```

```
go func() {
    for {
        select {
        case msg := <-ch:
            fmt.Println("Received:", msg)
        case <-done:
            fmt.Println("Shutting down...")
            return
        }
    }
}()
ch <- "Hello"
close(done)
```

```
go func() {
    for {
        select {
        case msg := <-ch:
            fmt.Println("Received:", msg)
        case <-done:
            fmt.Println("Shutting down...")
            return
        }
    }
}()
ch <- "Hello"
close(done)
```

```
go func() {
    for {
        select {
        case msg := <-ch:
            fmt.Println("Received:", msg)
        case <-done:
            fmt.Println("Shutting down...")
            return
        }
    }
}()
ch <- "Hello"
close(done)
```

```
go func() {
    for {
        select {
        case msg := <-ch:
            fmt.Println("Received:", msg)
        case <-done:
            fmt.Println("Shutting down...")
            return
        }
    }
}()
ch <- "Hello"
close(done)
```

```
go func() {
    for {
        select {
        case msg := <-ch:
            fmt.Println("Received:", msg)
        case <-done:
            fmt.Println("Shutting down...")
            return
        }
    }
}()
ch <- "Hello"
close(done)
```

```
go func() {
    for {
        select {
        case msg := <-ch:
            fmt.Println("Received:", msg)
        case <-done:
            fmt.Println("Shutting down...")
            return
        }
    }
}()
ch <- "Hello"
close(done)
```

Context

The Go context package provides a standard mechanism for managing cancellation signals, deadlines, and request-scoped values across goroutines and API boundaries.

```go
ctx, cancel := context.WithCancel(context.Background())
go func() {
    for {
        select {
        case msg := <-ch:
            fmt.Println("Received:", msg)
        case <-ctx.Done():
            fmt.Println("Shutting down...")
            return
        }
    }
}()
ch <- "Hello"
cancel()
```

```
ctx, cancel := context.WithCancel(context.Background())
go func() {
    for {
        select {
        case msg := <-ch:
            fmt.Println("Received:", msg)
        case <-ctx.Done():
            fmt.Println("Shutting down...")
            return
        }
    }
}()
ch <- "Hello"
cancel()
```

```
ctx, cancel := context.WithCancel(context.Background())
go func() {
    for {
        select {
        case msg := <-ch:
            fmt.Println("Received:", msg)
        case <-ctx.Done():
            fmt.Println("Shutting down...")
            return
        }
    }
}()
ch <- "Hello"
cancel()
```

```
ctx, cancel := context.WithCancel(context.Background())
go func() {
    for {
        select {
        case msg := <-ch:
            fmt.Println("Received:", msg)
        case <-ctx.Done():
            fmt.Println("Shutting down...")
            return
        }
    }
}()
ch <- "Hello"
cancel()
```

```go
ctx, cancel := context.WithCancel(context.Background())
go func() {
    for {
        select {
        case msg := <-ch:
            fmt.Println("Received:", msg)
        case <-ctx.Done():
            fmt.Println("Shutting down...")
            return
        }
    }
}()
ch <- "Hello"
cancel()
```

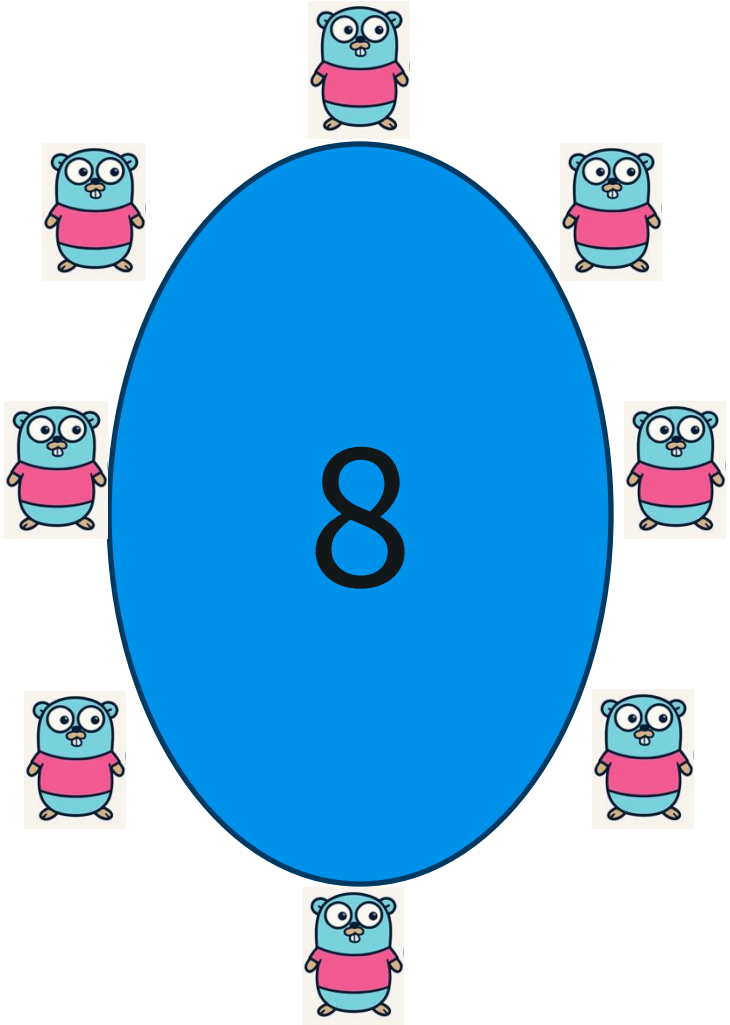
```go
ctx, cancel := context.WithCancel(context.Background())
go func() {
    for {
        select {
        case msg := <-ch:
            fmt.Println("Received:", msg)
        case <-ctx.Done():
            fmt.Println("Shutting down...")
            return
        }
    }
}()
ch <- "Hello"
cancel()
```

```go
ctx, cancel := context.WithCancel(context.Background())
go func() {
    for {
        select {
        case msg := <-ch:
            fmt.Println("Received:", msg)
        case <-ctx.Done():
            fmt.Println("Shutting down...")
            return
        }
    }
}()
ch <- "Hello"
cancel()
```

```go
ctx, cancel := context.WithCancel(context.Background())
go func() {
    for {
        select {
        case msg := <-ch:
            fmt.Println("Received:", msg)
        case <-ctx.Done():
            fmt.Println("Shutting down...")
            return
        }
    }
}()
ch <- "Hello"
cancel()
```

```go
ctx, cancel := context.WithCancel(context.Background())
go func() {
    for {
        select {
        case msg := <-ch:
            fmt.Println("Received:", msg)
        case <-ctx.Done():
            fmt.Println("Shutting down...")
            return
        }
    }
}()
ch <- "Hello"
cancel()
```

*Use **context** to cancel goroutines from outside their scope*


8

#8 API Timeout Management

- Service calls to external services/APIs
- Service may become unresponsive
- Application hangs indefinitely
- User experience degrades

Timeout

```
16      select {
17      case msg := <-ch:
18          fmt.Println(msg)
19      case <-time.After(2 * time.Second):
20          fmt.Println("Timeout occurred")
21      }
22  }
```

```
16      select {
17      case msg := <-ch:
18          fmt.Println(msg)
19      case <-time.After(2 * time.Second):
20          fmt.Println("Timeout occurred")
21      }
22  }
```

Context

The Go context package provides a standard mechanism for managing cancellation signals, deadlines, and request-scoped values across goroutines and API boundaries.

```go
func ContextTimeoutExample() {
    ch := make(chan string)
    timeout := 2 * time.Second
    ctx, cancel := context.WithTimeout(context.Background(), timeout)
    defer cancel()
    select {
    case msg := <-ch:
        fmt.Println(msg)
    case <-ctx.Done():
        if ctx.Err() == context.DeadlineExceeded {
            fmt.Println("Operation timed out")
        }
    }
}
```

```go
func ContextTimeoutExample() {
    ch := make(chan string)
    timeout := 2 * time.Second
    ctx, cancel := context.WithTimeout(context.Background(), timeout)
    defer cancel()
    select {
    case msg := <-ch:
        fmt.Println(msg)
    case <-ctx.Done():
        if ctx.Err() == context.DeadlineExceeded {
            fmt.Println("Operation timed out")
        }
    }
}
```

```go
func ContextTimeoutExample() {
    ch := make(chan string)
    timeout := 2 * time.Second
    ctx, cancel := context.WithTimeout(context.Background(), timeout)
    defer cancel()
    select {
    case msg := <-ch:
        fmt.Println(msg)
    case <-ctx.Done():
        if ctx.Err() == context.DeadlineExceeded {
            fmt.Println("Operation timed out")
        }
    }
}
```

```go
func ContextTimeoutExample() {
    ch := make(chan string)
    timeout := 2 * time.Second
    ctx, cancel := context.WithTimeout(context.Background(), timeout)
    defer cancel()
    select {
    case msg := <-ch:
        fmt.Println(msg)
    case <-ctx.Done():
        if ctx.Err() == context.DeadlineExceeded {
            fmt.Println("Operation timed out")
        }
    }
}
```

```go
func ContextTimeoutExample() {
    ch := make(chan string)
    timeout := 2 * time.Second
    ctx, cancel := context.WithTimeout(context.Background(), timeout)
    defer cancel()
    select {
    case msg := <-ch:
        fmt.Println(msg)
    case <-ctx.Done():
        if ctx.Err() == context.DeadlineExceeded {
            fmt.Println("Operation timed out")
        }
    }
}
```

```go
func ContextTimeoutExample() {
    ch := make(chan string)
    timeout := 2 * time.Second
    ctx, cancel := context.WithTimeout(context.Background(), timeout)
    defer cancel()
    select {
    case msg := <-ch:
        fmt.Println(msg)
    case <-ctx.Done():
        if ctx.Err() == context.DeadlineExceeded {
            fmt.Println("Operation timed out")
        }
    }
}
```

```
Operation timed out
```

```
rs.log.Debug("Sanitizer - plugin: calling", "filename", req.Filename, "contentLength", len(req.
Content))
```

```
rsp, err := rc.Sanitize(ctx, grpcReq)
if err != nil {
    if errors.Is(ctx.Err(), context.DeadlineExceeded) {
        rs.log.Info("Sanitizer - plugin: time out")
        return nil, ErrTimeout
    }

    return nil, err
}
```

```
    return nil, fmt.Errorf("sanitizer - plugin: failed to sanitize: %s", rsp.Error)
}
```

From: grafana/pkg/services/rendering/svgSanitizer.go

*Prefer **context** over time.After for implementing timeouts*

Summary

- Parallelism vs concurrency
- Channel is not a queue
- Worker pool
- Semaphore
- Fan-Out
- Fan-In
- Pipeline pattern
- Cancel
- Timeout

Concurrency is a powerful tool - use it wisely

Thanks!

Any questions?

You can find me at:

https://www.linkedin.com/in/gili-kamma/

gili.kamma@gmail.com