ChatGPT

Go Concurrency: A Comprehensive Revision

Concurrency in Go is built on goroutines and channels – lightweight threads and communication pipes managed by the Go runtime [1] [2]. It lets you perform multiple tasks *simultaneously*. Go's model follows the CSP style ("Communicating Sequential Processes"), favoring communication over shared memory. Goroutines and channels make it easy to write concurrent programs in Go [1] [2]. This guide reviews key concepts with code examples and diagrams, ending with common pitfalls and interview questions.

Goroutines and the `go` Keyword

A **goroutine** is a **lightweight thread of execution** [1]. You create one by prefixing a function call with `go`, which starts that function in a new goroutine, running concurrently with the caller [3]. For example:

```
func sayHello(id int) {
    fmt.Printf("Hello from goroutine %d\n", id)
}

func main() {
    // Launch two goroutines
    go sayHello(1)
    go sayHello(2)
    // The main goroutine may exit immediately unless we wait
    time.Sleep(100 * time.Millisecond)
    fmt.Println("Main done")
}
```

In this code, `sayHello(1)` and `sayHello(2)` run concurrently, with the **main goroutine** continuing immediately [3]. Goroutines are **extremely cheap** (a few kilobytes of stack) compared to OS threads [1], so you can create hundreds or thousands of them. The Go scheduler multiplexes goroutines onto OS threads. Each goroutine grows its stack as needed, so there's no fixed limit to goroutines.

Tips: The main function itself is a goroutine, so if `main` returns, the program exits even if other goroutines are running. Use synchronization (like `sync.WaitGroup`) or `time.Sleep` (for quick tests) to keep the main goroutine alive while others run. Goroutines do not propagate panics by default; you may need to use `recover` or error channels to handle errors inside goroutines.

Interview Questions: - **Q:** What is a goroutine and how do you start one?
Answer: A goroutine is a lightweight thread managed by Go's runtime [1]. You start one by using the `go` keyword before a function call (e.g., `go f()`), which executes `f` concurrently with the calling goroutine [3]. - **Q:** How do goroutines differ from OS threads?
Answer: Goroutines are scheduled by Go's runtime (not the OS) and have a small initial stack that grows

dynamically [1]. They're much more lightweight than OS threads, so you can efficiently run many of them. The Go scheduler multiplexes many goroutines onto a smaller number of OS threads. - **Q:** What happens if `main` returns before goroutines complete?
Answer: The program exits and all running goroutines are terminated. To wait for goroutines, use `sync.WaitGroup` or other synchronization methods.

`sync.WaitGroup`: Waiting for Goroutines

The `sync.WaitGroup` type lets the main or coordinating goroutine **wait for a collection of goroutines to finish**. The basic pattern is: 1. Declare a `WaitGroup`: `var wg sync.WaitGroup`.
2. Before launching each goroutine, call `wg.Add(1)` to increment the counter [4].
3. In each goroutine, `defer wg.Done()` so that the counter is decremented when the goroutine completes.
4. After launching, call `wg.Wait()`. This blocks until the counter goes to zero (all goroutines have called `Done()`) [5].

For example:

```
var wg sync.WaitGroup

for i := 1; i <= 3; i++ {
    wg.Add(1)
    go func(n int) {
        defer wg.Done()
        fmt.Printf("Worker %d starting\n", n)
        time.Sleep(time.Second)
        fmt.Printf("Worker %d done\n", n)
    }(i)
}

wg.Wait()
fmt.Println("All workers finished")
```

This ensures the main function waits for all workers. Under the hood, when the counter reaches zero, `Wait()` unblocks [5]. A few important notes: - **Pointer Use:** Always pass a `*sync.WaitGroup` (a pointer). A WaitGroup contains an unexported no-copy field, so you **must not copy it** after first use [6] [7]. For example, do `go worker(&wg)`, not `go worker(wg)`.
- **Add/Done Balance:** Each `Add(1)` must eventually have a corresponding `Done()`. If the counter ever goes negative, the program panics [4]. Make sure to call `Add` **before** starting goroutines to avoid race conditions, and `Done()` exactly once per `Add`.
- **Reuse:** A WaitGroup can be reused for another batch of goroutines only after `Wait()` has returned and the counter is zero [8]. Don't call `Add` after `Wait` if you intend to wait again; instead, declare a new WaitGroup or ensure the first has completed.

Interview Questions: - **Q:** How do you use `sync.WaitGroup` to wait for goroutines?
Answer: Use `wg.Add(1)` before each goroutine, `defer wg.Done()` inside each goroutine, and `wg.Wait()` in the main goroutine [9] [5]. This makes the main goroutine block until all workers call `Done()`. - **Q:** Why should you pass a pointer to `sync.WaitGroup` to functions?
Answer: The WaitGroup struct must not be copied once used [7]. Copying its value (e.g. passing by value) can corrupt its state. Therefore, use a `*sync.WaitGroup` (pointer) when sharing it among goroutines [6]. - **Q:** What happens if you call `Add` with a negative number or if the counter doesn't reach zero?
Answer: Calling `Add` so that the counter goes negative causes a runtime panic [4]. Also, if the counter never reaches zero (e.g., missing a `Done`), `Wait` will block forever (deadlock).

Channels: Unbuffered and Buffered

A **channel** is a typed conduit for communication between goroutines. You create a channel with `make(chan T)`, where `T` is the element type. There are two main types:

- **Unbuffered channels:** `make(chan T)` (no second argument). An unbuffered channel has **no capacity**, so sends (`ch <- val`) block until another goroutine is ready to receive (`<-ch`), and vice versa [10]. This synchronizes the sender and receiver. If one tries to send on an unbuffered channel when no goroutine is ready to receive, the sender blocks (potentially causing deadlock) [10]. For example:

```
ch := make(chan int)
go func() {
    fmt.Println(<-ch) // will print 42
}()
ch <- 42 // waits for the receiver to be ready, then sends
```

- **Buffered channels:** `make(chan T, capacity)`. A buffered channel has a fixed capacity. Sends block only when the buffer is full, and receives block only when the buffer is empty. For example, `ch := make(chan int, 2)` creates a buffer of size 2. You can send two values without a receiver:

```
ch := make(chan int, 2)
ch <- 1  // buffer has [1]
ch <- 2  // buffer has [1,2]
// another send would block until a receive occurs
fmt.Println(<-ch) // prints 1, buffer now [2]
ch <- 3  // now buffer [2,3]
```

A buffered channel thus decouples senders and receivers up to the buffer size [11]. Use buffered channels when you want limited asynchronous sending (e.g. to prevent a goroutine from blocking immediately).

Interview Questions: - **Q:** What's the difference between buffered and unbuffered channels? When would you use each?
Answer: An unbuffered channel (`make(chan T)`) requires both sender and receiver to be ready at the

same time, providing synchronization [10]. A buffered channel (`make(chan T, n)`) has capacity `n`; sends succeed without a receiver until the buffer is full [11]. Use unbuffered channels for tight synchronization or when you want the guarantee that the receiver is ready. Use buffered channels to allow a limited queue of values and decouple the sender from the receiver. - **Q:** What happens if you send to an unbuffered channel with no receiver?
Answer: The send will block, possibly causing a deadlock if nothing else receives on that channel [10]. - **Q:** Can you send on a closed channel? Or close a channel twice?
Answer: Sending on a closed channel causes a panic. Closing a closed channel also panics. Always ensure only one goroutine closes a channel (typically the sender) [12] [13].

Closing Channels and Range

A channel can be closed with `close(ch)`, which signals that no more values will be sent on it. Closing is **only done by the sender** (never the receiver) when no further sends will occur [13]. After a channel is closed: - Receivers can still **drain** any remaining values in the buffer. - Once empty, further receives yield the zero value of the channel's element type and a `false` flag.

For example, the Go by Example *closing channels* illustrates this: the sender closes the `jobs` channel when done, and the receiver loop uses the two-value receive form `v, ok := <-ch`. The `ok` is false when the channel is closed and drained [14] [15]. A `for v := range ch` loop will iterate until the channel is closed and all values have been received [16].

```
jobs := make(chan int, 5)
done := make(chan bool)

// Worker goroutine: reads jobs until channel is closed
go func() {
    for {
        j, more := <-jobs
        if more {
            fmt.Println("received job", j)
        } else {
            fmt.Println("no more jobs")
            done <- true
            return
        }
    }
}()

// Send some jobs and then close
for j := 1; j <= 3; j++ {
    jobs <- j
    fmt.Println("sent job", j)
}
close(jobs)  // no more sends
```

```
<-done
fmt.Println("worker done")
```

Output will include “no more jobs” when the channel is closed.

Some key points on closing: - **Never send after close:** After calling `close(ch)`, **any subsequent send** (`ch <- ...`) will panic.
- **Single closer:** Only the sender should close. If multiple senders exist, coordinating close is tricky and can easily cause race conditions [13]. Common practice is to have one goroutine responsible for closing once work is done.
- **Range behavior:** A `for v := range ch { … }` loop automatically stops when the channel is closed and drained [16]. Thus, closing a channel is a clean way to terminate a range loop. Even if the channel still has values in its buffer, those values will be received before the loop ends [16] [17].
- **Signal completion:** Closing a channel is often used as a broadcast to receivers that no more data is coming [18]. For example, a worker can range over a jobs channel and the main goroutine closes it to indicate shutdown.

Interview Questions: - **Q:** When and who should close a channel in Go?
Answer: Only the sender (producer) should close a channel, and only after it is done sending all values [13]. The receiver must not close it. Closing signals to receivers that no more values will be sent [18]. - **Q:** What happens if you close a channel from the wrong side or close it twice?
Answer: Closing a channel from the receiver or closing it more than once causes a panic. It’s a runtime error to close a closed channel. Always ensure exactly one close by the sender. - **Q:** How can you tell if a channel is closed when receiving?
Answer: Use the two-value receive: `v, ok := <-ch`. If `ok` is `false`, the channel is closed and drained. For example, `v, ok := <-ch` yields `(zero, false)` after close [15]. - **Q:** What happens to values already sent to a channel when it’s closed?
Answer: Values already in the channel’s buffer can still be received after closing. Receivers will get those values (in order) and then receive zero values indefinitely [16].

`select` Statement (Timeouts, Default, etc.)

The `select` statement lets a goroutine wait on **multiple channel operations** simultaneously. Each `case` in a `select` must be a send or receive on a channel. The `select` blocks until one of its cases can run, then executes that case (choosing at random if multiple are ready). For example:

```
select {
case msg := <-ch1:
    fmt.Println("received", msg, "from ch1")
case val := <-ch2:
    fmt.Println("received", val, "from ch2")
}
```

This waits until either `ch1` or `ch2` has a value. If `ch1` is ready first, the first case runs, etc.

Timeouts: You can implement timeouts using `select` with `time.After`:

```
c1 := make(chan string, 1)
go func() {
    time.Sleep(2 * time.Second)
    c1 <- "result"
}()

select {
case res := <-c1:
    fmt.Println("got result:", res)
case <-time.After(1 * time.Second):
    fmt.Println("timeout")
}
```

Here, if `<-c1` doesn't happen within 1 second, the `time.After` case fires. As Go by Example notes, "`select` proceeds with the first receive that's ready" [19], so this implements a timeout.

Non-blocking with `default`: You can add a `default` case to avoid blocking:

```
select {
case x := <-msgs:
    fmt.Println("got message", x)
default:
    fmt.Println("no message ready")
}
```

With `default`, if no channels are ready, the `default` case runs immediately, making the select non-blocking [20]. Similarly, a non-blocking send can be done:

```
select {
case msgs <- "hello":
    fmt.Println("sent message")
default:
    fmt.Println("message not sent")
}
```

Advanced usage: `select` is very powerful – you can listen on many channels at once, use it in loops, and combine it with `time.After` for timeouts or with a `default` clause for non-blocking behavior [19] [20]. If multiple cases are ready, one is chosen at random. Without a `default`, `select` blocks until a case can proceed.

Interview Questions: - **Q:** How do you use `select` to implement a timeout on channel operations? *Answer:* Include a case `<-time.After(duration)` in the `select`. For example, `select { case v := <-ch: … ; case <-time.After(1*time.Second): fmt.Println("timeout") }`. The `time.After` channel sends a value after the timeout, so that case triggers if the other channel is not ready [19]. - **Q:** What does a `default` case do in a `select`? *Answer:* A `default` case makes the `select` non-blocking. If no other channel case is ready, the `default` case runs immediately [20]. It's often used for polling or fallback behavior (e.g., skip if no message). - **Q:** Can `select` listen on the same channel multiple times? Or multiple operations in one case? *Answer:* Each case in `select` must be a single channel operation (send or receive). You cannot combine multiple operations in one case. Listening on the same channel in multiple cases is allowed, but they are treated independently and one will be selected if it's ready. - **Q:** What happens if none of the `select` cases are ready and there is no `default`? *Answer:* The `select` blocks and waits until at least one case can proceed. The current goroutine sleeps until a channel operation is possible.

Fan-Out and Fan-In Concurrency Patterns

Fan-out and **fan-in** are common patterns to parallelize work. *Fan-out* means launching multiple goroutines to do similar work in parallel, typically all reading from the same input channel. *Fan-in* means merging multiple channels of results into a single channel, so the results can be processed together.

- **Fan-out:** Multiple goroutines read from the same channel until it's closed. For example, you might have a channel of jobs and several worker goroutines all doing `for job := range jobs { … }`. This distributes work concurrently among the workers [21].
- **Fan-in:** Take multiple input channels and merge them into one output channel. A typical merge function starts a goroutine for each input channel that reads values and forwards them to an output channel, and then closes the output once all inputs are drained [22].

The official Go blog explains: "Multiple functions can read from the same channel until that channel is closed; this is called *fan-out*... A function can read from multiple inputs and proceed until all are closed by multiplexing the input channels onto a single channel... this is called *fan-in*" [21]. In practice, a `sync.WaitGroup` is often used to implement fan-in (to know when all inputs are done) as shown in the merge example of [43].

Example (fan-in): Merging two channels of integers:

```
func merge(cs ...<-chan int) <-chan int {
    var wg sync.WaitGroup
    out := make(chan int)
    output := func(c <-chan int) {
        defer wg.Done()
        for n := range c {
            out <- n
        }
    }
    wg.Add(len(cs))
```

```
    for _, c := range cs {
        go output(c)
    }
    go func() {
        wg.Wait()
        close(out)
    }()
    return out
}

// Usage:
c1 := generate(2, 3, 5) // <-chan int from earlier example
c2 := generate(7, 11)
for v := range merge(c1, c2) {
    fmt.Println(v) // values from both c1 and c2
}
```

In this way, values from multiple sources flow into one channel, which is closed when all senders are done [22].

Interview Questions: - **Q:** What is *fan-out* in Go concurrency?
Answer: Fan-out is when you have multiple goroutines reading from the same channel to perform work in parallel [21]. It spreads tasks across goroutines. For example, several workers all doing `for job := range jobs { … }`. - **Q:** What is *fan-in* and how do you implement it?
Answer: Fan-in merges multiple channels into one. You start a goroutine for each input channel that reads and forwards values into a single output channel, then close the output when all inputs are done [22]. Often a `sync.WaitGroup` is used to detect when all input goroutines have finished before closing. - **Q:** Why might you use fan-out and fan-in together?
Answer: You can distribute (fan-out) tasks among parallel workers and then collect (fan-in) all their results back into one channel. This maximizes CPU use and keeps code organized.

Pipeline Pattern (Channel Chaining)

A **pipeline** is a series of stages connected by channels, where each stage is a group of goroutines that perform a transformation on the data. Each stage reads from an inbound channel, processes values, and sends to an outbound channel [23]. The first stage (generator) only has an outbound channel, and the last stage (consumer) only has an inbound channel.

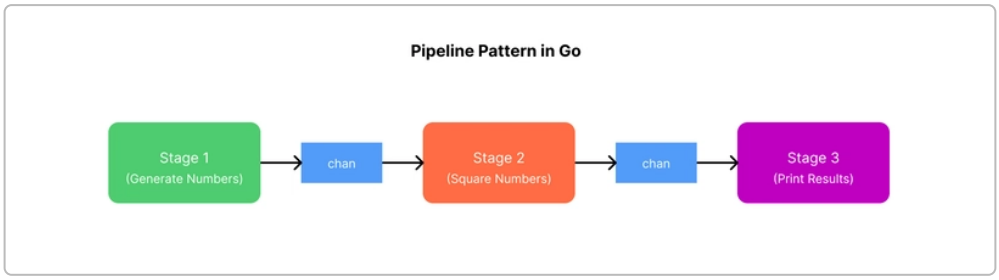

Figure: A simple 3-stage pipeline in Go (generate, square, print).

For example, suppose we want to process numbers by generating them, squaring them, then printing them. We can write:

```
// Stage 1: generator
func generate(nums ...int) <-chan int {
    out := make(chan int)
    go func() {
        for _, n := range nums {
            out <- n
        }
        close(out)
    }()
    return out
}

// Stage 2: square numbers
func square(in <-chan int) <-chan int {
    out := make(chan int)
    go func() {
        for n := range in {
            out <- n * n
        }
        close(out)
    }()
    return out
}

// Stage 3: print results
func printer(in <-chan int) {
    for n := range in {
        fmt.Println(n)
    }
}

func main() {
    // Connect the stages: generate -> square -> printer
    nums := generate(2, 3, 4)
    squares := square(nums)
    printer(squares) // prints 4, 9, 16
}
```

Each stage here closes its output channel after sending all values, allowing the next stage's range loop to terminate [16] [24]. By chaining stages, we can build complex processing flows. As the Go blog notes, in a

pipeline “stages close their outbound channels when all send operations are done, and keep receiving values until inbound channels are closed,” enabling clean shutdown of the pipeline [25].

Interview Questions: - **Q:** What is a pipeline in Go and why is it useful?
Answer: A pipeline is a sequence of stages, each a function/goroutine reading from one channel and sending to another [23]. It’s useful for building streaming data processes and leveraging concurrency: each stage can work in parallel, and the flow is easy to compose. - **Q:** How do you properly close channels in a pipeline?
Answer: Each stage should close its **outbound** channel after it’s done sending all values [25]. The next stage reads until that channel is closed (typically via a `for range`). This ensures no goroutine is left waiting indefinitely. - **Q:** Can pipeline stages run concurrently on multiple values?
Answer: Yes, each stage is usually implemented with its own goroutine (or goroutines), so while one stage is processing an item, others can process previous or next items. This provides throughput and parallelism. - **Q:** What happens if a downstream stage needs to exit early (e.g., on error)?
Answer: You should design for cancellation. For example, you might close remaining input channels or use a `context.Context` to signal all stages to stop. Otherwise, goroutines may leak by hanging on send/receive.

Practical Examples

Below are some example exercises illustrating these concepts.

- **Ping-Pong (Two-way communication):** One goroutine can send a “ping” to another, which responds with “pong.” For example, using two channels:

```
func pinger(ping <-chan string, pong chan<- string) {
    for msg := range ping {
        fmt.Println("pinger received", msg)
        pong <- "pong"
    }
}
func ponger(ping chan<- string, pong <-chan string) {
    for msg := range pong {
        fmt.Println("ponger received", msg)
        ping <- "ping"
    }
}
func main() {
    pingChan := make(chan string)
    pongChan := make(chan string)
    go pinger(pingChan, pongChan)
    go ponger(pingChan, pongChan)
    pingChan <- "ping" // start the game
    time.Sleep(time.Second * 2) // let them exchange messages
    close(pingChan)
```

```
    close(pongChan)
}
```

Here, `pinger` waits for “ping” and replies “pong”, and vice versa. Channels coordinate the ping-pong exchange. Note: in a real program you’d include logic to exit the loop.

- **Squaring Pipeline:** (Uses the pipeline pattern above.) Given a list of numbers, generate them, square each, then collect or print. The code in the pipeline section shows how to do this with chained channels.

- **Doubling Pipeline:** Similar to squaring, but each stage doubles values. For instance, you could have a generator and a `double(in <-chan int) <-chan int` stage:

```
func double(in <-chan int) <-chan int {
    out := make(chan int)
    go func() {
        for n := range in {
            out <- n * 2
        }
        close(out)
    }()
    return out
}
// Usage:
nums := generate(1, 2, 3)
doubled := double(nums)
for v := range doubled {
    fmt.Println(v) // 2, 4, 6
}
```

- **Prime Checking (Concurrent Sieve):** You can use goroutines and channels to generate primes. One simple approach: a generator sends numbers into a channel, and one goroutine filters primes. For example, sending primes back to another channel and using `sync.WaitGroup` to wait. An outline:

```
func generate(limit int, out chan<- int) {
    defer close(out)
    for i := 2; i <= limit; i++ {
        out <- i
    }
}
func sieve(in <-chan int, out chan<- int, wg *sync.WaitGroup) {
    defer wg.Done()
    for num := range in {
        prime := true
        for i := 2; i*i <= num; i++ {
```

```
            if num%i == 0 {
                prime = false
                break
            }
        }
        if prime {
            out <- num
        }
    }
}
func main() {
    nums := make(chan int)
    primes := make(chan int)
    go generate(50, nums)
    var wg sync.WaitGroup
    wg.Add(1)
    go func() {
        defer close(primes)
        sieve(nums, primes, &wg)
    }()
    go func() {
        wg.Wait()
    }()
    for p := range primes {
        fmt.Println("Prime", p)
    }
}
```

This is a simple example; more advanced prime sieve pipelines exist (e.g. Sieve of Eratosthenes with channel fan-out).

Common Debugging Strategies

Even with powerful primitives, concurrency can introduce subtle bugs. Here are some tips:

- **Deadlocks:** The Go runtime panics if all goroutines are blocked (e.g., all are waiting on channel operations). A “fatal error: all goroutines are asleep – deadlock!” means no goroutine can proceed. Common causes:
- Sending on an unbuffered channel with no receiver.
- A `for range` on a channel that is never closed.
- The main goroutine exiting early.
- Fix these by ensuring every send has a corresponding receiver, closing channels, or using `WaitGroup`.
- **Goroutine leaks:** This happens when goroutines get stuck (e.g., waiting on a channel that never receives) and are never cleaned up. Use buffered channels or timeouts to avoid hanging sends, and

make sure loops can exit. Tools like `pprof` (goroutine dump) or `go tool trace` can help find stuck goroutines.

- **Channel misuse:** Typical mistakes include sending on closed channels (panic), forgetting to close (range loops hang), or using wrong channel direction. Always test channel logic, and consider using buffered channels to prevent accidental deadlocks (but only if appropriate).
- **Race conditions:** Shared memory access (even prints or counters) from multiple goroutines without sync can cause races. Use the `-race` flag to detect these. Prefer channels and synchronization primitives (`sync.Mutex`, etc.) over unsynchronized shared variables.
- **Timeouts and Cancellation:** For network calls or blocking operations, use `select` with `time.After` or a `context.Context` with cancelation to avoid goroutines hanging forever.

In general, to debug: add print/log statements, use the `runtime` stack dump (send SIGQUIT or use `runtime.Stack`), or leverage the Go 1.10+ `sync/errgroup` and `context` for structured concurrency.

Further Interview Questions

- **Concurrency vs Parallelism:** *Q:* Explain the difference. *A:* Concurrency is structuring a program to handle multiple tasks at once (time-slicing or interleaving), while parallelism is doing multiple tasks literally at the same time (requires multiple CPU cores). Go's goroutines provide concurrency; if GOMAXPROCS>1, they can also achieve parallelism.
- **GOMAXPROCS:** *Q:* What does GOMAXPROCS do? *A:* It sets the maximum number of OS threads that can execute Go code simultaneously (i.e., CPU cores). By default, it's the number of CPU cores. It controls parallelism but not the number of goroutines.
- **Buffered Channel of Size 1:** *Q:* Why might a buffer of size 1 be special? *A:* A buffered channel of size 1 can act like a lightweight lock or semaphore. At most 1 value can be queued, so sends block only if another send hasn't been received yet. It can prevent a goroutine from blocking immediately while still ensuring synchronization.
- **Context for Cancellation:** *Q:* How do you cancel goroutines? *A:* The standard way is using `context.Context`. Pass a `Context` to goroutines and have them select on `ctx.Done()` to exit early. This propagates cancellation signals cleanly through concurrent flows.
- **Deadlock Example:** *Q:* What code leads to "all goroutines are asleep – deadlock"? *A:* For example:

  ```
  ch := make(chan int)
  ch <- 1 // deadlock: no goroutine is receiving from ch
  ```

  Here the single goroutine (main) tries to send on an unbuffered channel with no receiver, causing an immediate deadlock panic.

This completes our concurrency revision. Use these patterns and tools carefully, and always watch out for blocking operations. Happy coding!

Sources: Go official docs and examples [1] [9] [10] [18] [16] [19] [20] [23]; Go by Example and community posts provide additional examples.

---

[1] [3] Go by Example: Goroutines
https://gobyexample.com/goroutines

[2] Compute Prime Numbers Using Concurrency in Go
https://www.tutorialspoint.com/golang-program-to-compute-all-prime-numbers-up-to-a-given-number-using-concurrency

[4] [5] [7] [8] [9] sync package - sync - Go Packages
https://pkg.go.dev/sync

[6] WaitGroups in Golang – Mohit Khare
https://www.mohitkhare.com/blog/waitgroups-in-golang/

[10] [11] Unbuffered vs Buffered channels in go | by Chethan | Medium
https://medium.com/@chethan13/unbuffered-vs-buffered-channels-in-go-83b1a0956e46

[12] [13] How to Gracefully Close Channels -Go 101
https://go101.org/article/channel-closing.html

[14] [15] [18] Go by Example: Closing Channels
https://gobyexample.com/closing-channels

[16] [17] Go by Example: Range over Channels
https://gobyexample.com/range-over-channels

[19] Go by Example: Timeouts
https://gobyexample.com/timeouts

[20] Go by Example: Non-Blocking Channel Operations
https://gobyexample.com/non-blocking-channel-operations

[21] [22] [23] [24] [25] Go Concurrency Patterns: Pipelines and cancellation - The Go Programming Language
https://go.dev/blog/pipelines