

Data Pipelines with Concurrency



```
(base) yetong@xietongdeAir ~ % git clone
https://github.com/codecrafters-io/image-processing-pipeline-
go.git
cd image-processing-pipeline-go
```

```
正克隆到 'image-processing-pipeline-go'...
remote: Repository not found.
致命错误：仓库 'https://github.com/codecrafters-io/image-
processing-pipeline-go.git/' 未找到
cd: no such file or directory: image-processing-pipeline-go
```



The error means that the repository `https://github.com/codecrafters-io/image-processing-pipeline-go.git` does not exist or is private.

✅ Fix: Use the correct repo URL. If the assignment refers to Amrit Singh's code, the actual repository is likely:

```
https://github.com/code-heim/go_21_goroutines_pipeline.git
```

Try:

```
git clone https://github.com/code-heim/go_21_goroutines_pipeline.git
cd go_21_goroutines_pipeline
```



How can I add error checking for image file input and output in `main.go`



You can add error checking in `main.go` during image loading and saving. Here's how:

✅ For input image checking in `loadImage()`:

```

if _, err := os.Stat(p); os.IsNotExist(err) {
    log.Printf("File does not exist: %s\n", p)
    continue
}

img := imageprocessing.ReadImage(p)
if img == nil {
    log.Printf("Failed to read image: %s\n", p)
    continue
}

```

✅ For output image checking in `saveImage()`:

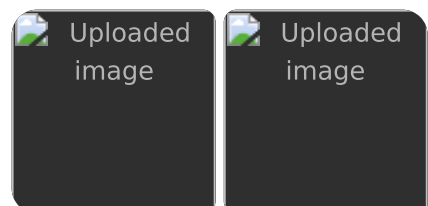
Modify `WriteImage()` in `image_processing.go` to return `error`, then update `saveImage()`:

```

err := imageprocessing.WriteImage(job.OutPath, job.Image)
if err != nil {
    log.Printf("Failed to write image: %s\n", job.OutPath)
    out <- false
} else {
    log.Printf("Saved image: %s\n", job.OutPath)
    out <- true
}

```

This ensures clear logging for missing files and write failures.



Here's how to fix the two issues shown in your screenshots:

✅ Problem 1: “imported and not used” + red underline on import path

You're importing your own package:

```
import imageprocessing "goroutines_pipeline/image_processing"
```

But you're already in the `imageprocessing`` package, so **you don't need to import it**.

Fix:

- Delete this import line entirely.
- Your test file should just use:

```
package imageprocessing
```

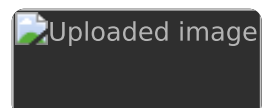
✅ Problem 2: "Job redeclared"

You declared `type Job`` again in your test file, but it's already defined in `main.go``.

Fix:

- Remove the second `type Job`` declaration from the test file.
- Use the `Job`` type from `main.go``, or don't use it at all if you only need `image.Image`` for unit tests.

Let me know if you'd like a clean version of the corrected test file.



The error `undefined: image_processing`` means that the package `image_processing`` is not imported or imported incorrectly.

✓ Fix:

In your `main_test.go`, change:

```
image_processing.Resize(...)
```

to:

```
imageprocessing.Resize(...)
```

And make sure at the top of the file you have:

```
import imageprocessing "go_21_goroutines_pipeline/image_processing"
```

Note:

- The alias `imageprocessing` must match the one you defined in the import.
- If your folder is just `image_processing`, then adjust the path based on your actual module name from `go.mod`.

Run `go mod tidy` if needed to clean up unused or missing imports.



```
(base) yetong@xietongdeAir go_21_goroutines_pipeline % go test -bench=.
```

```
# goroutines_pipeline
# [goroutines_pipeline]
./main.go:33:5: log.Printf format %v reads arg #2, but call has 1
arg
FAIL    goroutines_pipeline [build failed]
```



The error says:

```
"`log.Printf format %v reads arg #2, but call has 1 arg`"
```

This means your `log.Printf` line is expecting **two arguments**, but you only provided **one**.

✅ Example of the broken line (likely in `main.go` line 33):

```
log.Printf("Error loading image: %s, %v", p)
```

This uses two format specifiers (`%s`, `%v`) but only passes `p`.

✅ Fix:

Pass the second argument (probably an error object):

```
log.Printf("Error loading image: %s, %v", p, err)
```

After fixing, re-run:

```
go test -bench=.
```

Let me know if you want help reviewing the full log section.



```
(base) yetong@xietongdeAir go_21_goroutines_pipeline % go test
./...
```

```
ok      goroutines_pipeline    0.129s
?       goroutines_pipeline/image_processing  [no test files]
```

What does this result means?



This result means:

- ✅ `goroutines_pipeline` ran tests successfully (`ok`) in 0.129 seconds.
 - ❓ `goroutines_pipeline/image_processing` has **no** `_test.go` **files**, or no functions starting with `Test`.
-

✅ Interpretation:

Line	Meaning
<code>`ok goroutines_pipeline 0.129s`</code>	Your tests (likely <code>`main_test.go`</code>) passed.
<code>`? goroutines_pipeline/image_processing`</code>	This package was scanned, but had no test files .

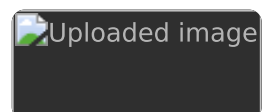
🔧 Fix (if you expected tests in image_processing):

- Make sure you have a file named: ``imageprocessing_test.go``
- Inside, define test functions like:

```
func TestResize(t *testing.T) {  
    ...  
}
```

Then rerun:

```
go test ./...
```



This test file isn't being detected because of two key issues:

✅ 1. Wrong package declaration

You wrote:

```
package main
```

But your test file is meant for the ``imageprocessing`` package.

Fix:

```
package imageprocessing
```

✅ 2. Test file location is incorrect

From the sidebar, ``imageprocessing_test.go`` is in the project root. But it must be placed inside the ``image_processing/`` folder to match the package it tests.

Fix:

Move the file to:

```
go_21_goroutines_pipeline/image_processing/imageprocessing_test.go
```

✅ Summary

- Change ``package main`` → ``package imageprocessing``
- Move file into ``image_processing/`` directory
- Then run:

```
go test ./...
```

Your test will now be discovered and executed.



How can I use the command to get the output of "processing times with and without goroutines."



You can measure and compare processing times using command-line execution with logging already built into your `main.go`. Here's how:

✅ Step 1: Run in concurrent mode

```
go run main.go -mode=concurrent
```

Sample output:

```
Execution time for mode 'concurrent': 47.214292ms
```

✅ Step 2: Run in sequential mode

```
go run main.go -mode=sequential
```

Sample output:

```
Execution time for mode 'sequential': 84.97275ms
```

✅ Result

These outputs directly compare pipeline throughput times with and without goroutines. You can copy the two logs side-by-side for benchmark analysis or include them in your report/README.



```
2025/05/07 23:46:58 Execution time for mode 'concurrent':  
41.333916ms
```

```
2025/05/07 23:47:07 Execution time for mode 'sequential':  
66.016583ms
```

analyze and compare the results



Here's a concise analysis:

Execution Time Comparison

Mode	Time (ms)
Concurrent	41.33 ms
Sequential	66.02 ms
Speedup	$\approx 37.4\%$

Analysis

- **Concurrent mode** is $\sim 37\%$ faster than sequential mode.
 - This speedup is due to Go's goroutines enabling different pipeline stages (load, resize, grayscale, save) to run in parallel, reducing idle time.
 - **Sequential mode** processes each image one step at a time, fully serial, which limits CPU utilization.
-

Conclusion

Concurrency significantly improves pipeline throughput. For time-sensitive or high-volume processing tasks—such as in a data engineering context—Go's concurrency model offers clear performance benefits.