main() – starting point of your entire Go program

- ctx := context.Background(): creating a background “context” This ctx is like a controller that manages timeouts, cancellations, and dependencies between AWS calls. it’s passed to all AWS functions so they know when to stop if needed.
- cfg, err := config.LoadDefaultConfig(ctx) : “Connect me to AWS with the right credentials.”
- secretClient := secretsmanager.NewFromConfig(cfg) and ecsClient := ecs.NewFromConfig(cfg) : creating two **AWS clients** using the loaded configuration. secretClient - Talks to AWS **Secrets Manager** — used to get PagerDuty key. ecsClient - Talks to AWS **ECS** — used to list and describe ECS tasks.
- Get environment variable: Environment = os.Getenv("ENVIRONMENT"): reads the `ENVIRONMENT` variable from AWS Lambda’s environment.Know which I’m operating in — production, staging, or development. This helps the code decide which secrets or clusters to use.
- Fetch the secret: secretValue, err := getSecret(ctx, secretClient, "visibility-eventing@"+Environment+"_secrets"): The name of your secret → `"visibility-eventing@"+Environment+"_secrets"`. `getSecret()` runs and returns your **PagerDuty key**,
  which is stored in: secretValue

getSecret():

- This function’s job is to **fetch a secret value** (like a password or API key) from **AWS Secrets Manager**. *secretsmanager.Client : AWS client object to connect with the Secrets Manager. secretName: The name of the secret you want to fetch (e.g., `"visibility- eventing@prod_secrets"`).

- Create input for the AWS API call: input :=
  &secretsmanager.GetSecretValueInput{
  SecretId: aws.String(secretName),
  } preparing the **request** that will be sent to AWS Secrets Manager. we tell AWS which secret you want by name. The `SecretId` is just the secret name, converted to AWS format (`aws.String`).
- Example: secretName = "visibility-eventing@prod_secrets". So AWS will look for that specific secret.

✅ Think of it as:

"Hey AWS, I want to fetch the secret named 'visibility-eventing@prod_secrets'."

- `client.GetSecretValue(...)` actually calls AWS and tries to fetch the secret value
- If it succeeds, you now have the secret data stored inside the variable `result`.
- Convert secret string (JSON) into a Go map
- The secret you got from AWS is a **JSON string**, like this:

```
{

 "pagerduty_key": "1234-ABCDE-5678"

}
```

So you **decode (unmarshal)** it into a Go map:

```
secretMap = map[string]string{

   "pagerduty_key": "1234-ABCDE-5678"

}
```

✅ Now, `secretMap` holds your secret values in an easy-to-use Go format.

Get the actual key you need

```
value, exists := secretMap["pagerduty_key"]

if !exists {

   return "", fmt.Errorf("key %q not found in secret",
"pagerduty_key")

}
```

💬 Meaning:

From that JSON, pick the exact key you need: `"pagerduty_key"`.

If it doesn't exist → error.

If it exists → store it in `value`.

- The variable `value` temporarily holds the PagerDuty key.
- Then it's **returned** to the function that called it (`main()`).
- The returned `value` (from `getSecret()`)
  is **stored inside** a new variable named `secretValue`.
- Passed from `main()` → `handler()` → `sendPagerDutyAlert()`
- Here, `secretValue` travels as a function argument into `handler()`
  and later into `sendPagerDutyAlert()`.
- The same `secretValue` variable is placed inside `"routing_key"`,
  and this is what PagerDuty uses to authenticate your alert request.
- `value` = The key when it's just taken out of the vault (Secrets Manager) 🔐
- `secretValue` = The key safely stored in your robot's hand as it walks around 🦾

Continuation : Start the Lambda function

→ This line **starts your AWS Lambda function —**

basically tells AWS:

“Hey, when an event comes in, run this code.”

When AWS triggers your Lambda, it will:

- Receive an event (`rawEvent`)
- Pass it into your `handler()` function
- Along with the ECS client and the secret key you just fetched

✅ So this wraps your handler logic inside a Lambda execution.

handler()

→ func handler(ctx context.Context, rawEvent json.RawMessage, client *ecs.Client, secretValue string) ([]TaskResult, error): This function (`handler`) is the **main worker** of your AWS Lambda function. It runs **every time** your Lambda is triggered by Splunk or another event source.

`rawEvent`	`json.RawMessage`	The raw event data that Lambda receives (from Splunk).
`client`	`*ecs.Client`	The AWS ECS client used to call ECS APIs.
`secretValue`	`string`	The secret key (like PagerDuty key) fetched from Secrets Manager.

Returns:

- `[]TaskResult` → the list of analyzed ECS task results
- `error` → any problem that happens during processing

→ Define outer structure

```
var outerEvent struct {

     Body string `json:"body"`

}
```

💬 Meaning:

We define a **temporary structure** to capture the outermost part of the JSON event that AWS Lambda receives.

For example, AWS sends data like this:

```
{

 "body": "{\"severity\": \"critical\", \"dimensions\": {\"ServiceName\": \"payment-service\"}}"

}
```

See?

There's a `"body"` key — and inside it is another JSON string.

So here we create a simple struct with just one field, `"body"`,

to extract that part.

→ Decode the outer JSON

```
if err := json.Unmarshal(rawEvent, &outerEvent); err != nil {

     return nil, fmt.Errorf("failed to parse outer event: %w", err)
```

```
}
```

💬 Meaning:

This line takes the raw JSON (from Splunk → Lambda → Go)

and **decodes** it into our `outerEvent` variable.

After this line runs,

`outerEvent.Body` now holds the inner JSON string from Splunk.

Example:

```
outerEvent.Body =
"{\"severity\":
\"critical\", \"status\":
\"failed\", ...}"
```

✅ So now we have the inner JSON string ready for the next step.

→ Decode the inner JSON

```
var splunk SplunkBody

if err := json.Unmarshal([]byte(outerEvent.Body), &splunk); err
!= nil {

     return nil, fmt.Errorf("failed to parse inner body: %w",
err)

}
```

💬 Meaning:

Now we take the **inner** JSON string (inside `"body"`)

and decode it into a Go struct called `SplunkBody`.

We defined this struct earlier to match the structure of Splunk's JSON payload:

```
type SplunkBody struct {
```

```
    Severity string `json:"severity"`

    Status   string `json:"status"`

    ...

}
```

After this,

the variable `splunk` will hold all the Splunk data in Go form.

✅ Example:

```
splunk.Severity = "critical"

splunk.Status = "failed"

splunk.Dimensions.ServiceName = "payment-service"
```

→ Validate Service Name

```
if splunk.Dimensions.ServiceName == "" {

    return nil, fmt.Errorf("missing ServiceName in Splunk event")

}
```

💬 Meaning:

The code checks if the Service Name is missing.

If Splunk didn’t send a service name → we can’t query ECS → stop and return an error.

✅ So this ensures: “We have the service name before we continue.”

→ Prepare “customDetails” (for logging or alerts)

```
customDetails := map[string]interface{}{
```

```
    "ServiceName": splunk.Dimensions.ServiceName,

    "detector":    splunk.Detector,

    "inputs": map[string]interface{}{

         "signal": map[string]interface{}{

              "fragment": splunk.Inputs.Signal.Fragment,

              "key":      splunk.Inputs.Signal.Key,

              "value":    splunk.Inputs.Signal.Value,

         },

    },

    "rule":      splunk.Rule,

    "severity":  splunk.Severity,

    "status":    splunk.Status,

    "timestamp": splunk.Timestamp,

}
```

💬 Meaning:

Here we are **building a data map** that contains
key information about the Splunk event —
this will later be sent to **PagerDuty** (or logged).

✅ Think of it as:

“Gather all important fields from Splunk into a single bundle.”

Print those details (for logs)

```
customDetailsJSON, err := json.MarshalIndent(customDetails, "",
"  ")

if err != nil {

     fmt.Printf("Failed to marshal customDetails: %v\n", err)

} else {

     fmt.Printf("Custom Details:\n%s\n",
string(customDetailsJSON))

}
```

💬 Meaning:

Convert that `customDetails` map into a nice formatted JSON

and print it out (again for CloudWatch Logs).

So you can easily see in logs:

```
{

 "ServiceName": "payment-service",

 "severity": "critical",

 "status": "failed",

 "timestamp": "2025-10-21T10:30:00Z"

}
```

✅ It’s just for debugging and visibility.

Two Apis:

1️⃣ For `ListTasks`

Here's the code again 👇

```
listOut, err := client.ListTasks(ctx, &ecs.ListTasksInput{

   Cluster:       aws.String(Environment),

   ServiceName:   aws.String(splunk.Dimensions.ServiceName),

   DesiredStatus: ecsTypes.DesiredStatusStopped,

   MaxResults:    aws.Int32(10),

                                        })
```

🧠 Explanation:

- `client.ListTasks(...)` calls AWS ECS API.
- ECS sends back a **response object** (like a JSON with data inside it).
- That response is **stored in the variable** 👉 `listOut`.

✅ So:

`listOut` = the variable that stores the output of `ListTasks()`

💡 What `listOut` contains:

It's of type:

```
*ecs.ListTasksOutput
```

And inside it, there's an important field:

```
listOut.TaskArns
```

That holds the **list of stopped ECS task IDs (ARNs)**.

Example:

```
listOut.TaskArns = [

 "arn:aws:ecs:task1",

 "arn:aws:ecs:task2",

 "arn:aws:ecs:task3"

]
```

✅ **So:**

`listOut` → complete response

`listOut.TaskArns` → the actual list of task IDs

2️⃣ For `DescribeTasks`

Here's the next code:

```
descOut, err := client.DescribeTasks(ctx,
&ecs.DescribeTasksInput{

   Cluster: aws.String(Environment),

   Tasks:   listOut.TaskArns,

})
```

🧠 Explanation:

- This calls ECS again, asking for details about the tasks.
- ECS sends back a **detailed response**.

- That response is **stored in the variable** 👉 `descOut`.

✅ So:

`descOut` = the variable that stores the output of `DescribeTasks()`

💡 What `descOut` contains:

It's of type:

```
*ecs.DescribeTasksOutput
```

And inside it, you'll find:

```
descOut.Tasks
```

That's a slice (list) of task objects,

each with multiple details like:

- `TaskArn`
- `StoppedReason`
- `LastStatus`
- etc.

Example:

```
descOut.Tasks = [

 {

   TaskArn: "arn:aws:ecs:task1",

   StoppedReason: "Scaling down service",

 },

 {

   TaskArn: "arn:aws:ecs:task2",
```

```
    StoppedReason: "Maintenance update",
  }
]
```

✅ **So:**

`descOut` → full ECS task detail response

`descOut.Tasks` → actual list of detailed task info

🧾 Summary Table

Operation	Function	Stored in Variable	Important Field	What It Holds
1️⃣ List stopped tasks	`ListTasks()`	`listOut`	`listOut.TaskArns`	List of stopped ECS task IDs
2️⃣ Describe each task	`DescribeTasks()`	`descOut`	`descOut.Tasks`	Detailed info for each stopped task

The Code (for reference)

```
for _, t := range descOut.Tasks {

    // Check for context cancelation inside the loop
```

```
    select {
    case <-ctx.Done():
        return nil, fmt.Errorf("operation canceled during loop: %w", ctx.Err())
    default:
    }
    taskArn := aws.ToString(t.TaskArn)
    reason := strings.ToLower(aws.ToString(t.StoppedReason))
    isMaint := isMaintenance(reason)
    isScale := isScaling(reason)
    results = append(results, TaskResult{
        TaskArn:       taskArn,
        StoppedReason: reason,
        IsMaintenance: isMaint,
        IsScaling:     isScale,
    })
    if !isMaint && !isScale {
        fmt.Printf(" pager duty sent: ")
```

```
          err := sendPagerDutyAlert(ctx, splunk.Severity,
splunk.Detector, splunk, taskArn, secretValue)

          if err != nil {

               fmt.Printf(" Failed to send PagerDuty alert:
%v\n", err)

          } else {

               fmt.Printf("PagerDuty alert sent for task %s\n",
taskArn)

          }

     }

}

return results, nil
```

💡 Big Picture Before We Begin

At this stage, your code already:

✅ Got a list of stopped tasks (`listOut.TaskArns`)

✅ Described them and stored details in `descOut.Tasks`

Now, it will go through **each task one by one** inside this loop

and decide:

- Is it maintenance?

- Is it scaling?
- Or is it a real error that needs an alert?

🧠 Step-by-Step Explanation (Simple Flow)

🔹 Step 1️⃣ — Loop start

```
for _, t := range descOut.Tasks {
```

💬 Meaning:

"Go through **each task** (t) inside the list of stopped tasks (descOut.Tasks)."

🧩 Example:

If `descOut.Tasks` has 3 tasks →

This loop runs **3 times**, once for each task.

🔹 Step 2️⃣ — Check for timeout (safety check)

```
select {

case <-ctx.Done():

    return nil, fmt.Errorf("operation canceled during loop:
%w", ctx.Err())

default:

}
```

💬 Meaning:

“If Lambda has been running too long or was stopped, exit the loop safely.”

This is just a **safety guard** to avoid timeouts.

If everything is fine, it continues to the next steps.

🔹 Step 3️⃣ — Extract the task’s unique ID

```
taskArn := aws.ToString(t.TaskArn)
```

💬 Meaning:

Get the task’s unique ECS ID (ARN) and store it in `taskArn`.

Example:

```
taskArn = "arn:aws:ecs:region:account:task/12345"
```

🔹 Step 4️⃣ — Get the reason for stopping

```
reason := strings.ToLower(aws.ToString(t.StoppedReason))
```

💬 Meaning:

Take the “StoppedReason” from ECS,

convert it to lowercase (for easier comparison),

and store it in `reason`.

Example:

```
StoppedReason = "Task stopped due to Maintenance Update"

reason = "task stopped due to maintenance update"
```

🔹 Step 5️⃣ — Check if it’s maintenance or scaling

```
isMaint := isMaintenance(reason)

isScale := isScaling(reason)
```

💬 Meaning:

Call helper functions to see if the stop reason contains

the words “maintenance” or “scaling”.

Example:

reason	isMaint	isScale
"maintenance update"	✅ true	❌ false
"scaling down"	❌ false	✅ true
"container crashed"	❌ false	❌ false

🔹 Step 6️⃣ — Store results

```
results = append(results, TaskResult{

    TaskArn:       taskArn,

    StoppedReason: reason,

    IsMaintenance: isMaint,
```

```
    IsScaling:     isScale,
})
```

💬 Meaning:

Create a new record (TaskResult) for this task

and add it to the `results` list.

So by the end of the loop,

`results` will look like this:

```
[
{TaskArn: "task1", IsMaintenance: true,  IsScaling: false},
{TaskArn: "task2", IsMaintenance: false, IsScaling: true},
{TaskArn: "task3", IsMaintenance: false, IsScaling: false}
]
```

🔹 Step 7️⃣ — If it's not maintenance or scaling, send alert

```
if !isMaint && !isScale {
    fmt.Printf("pager duty sent: ")
    err := sendPagerDutyAlert(ctx, splunk.Severity,
splunk.Detector, splunk, taskArn, secretValue)
    ...
```

}

💬 Meaning:

If both checks are false (so it's neither maintenance nor scaling),

that means it's a **real error** — so send an alert to PagerDuty 🚨.

This tells your "robot":

"This task didn't stop for normal reasons — alert the engineers!"

🔹 Step 8️⃣ — End of loop

When the loop finishes checking all tasks,

it returns the full `results` list:

```
return results, nil
```

💬 Meaning:

"Here is the complete list of all stopped ECS tasks and what I found about them."

After `handler()` returns, the `results` are captured by AWS Lambda through `lambda.Start()`, converted into a JSON response, and sent back to the AWS service (like Splunk or EventBridge) that triggered your function.

Then the Lambda finishes and waits for the next event.

Step 1 — You finish writing your Go code

That means your Go files (like `main.go`, `constants.go`) are complete and saved in your folder.

Example:

```
/ecsmonitor

├── main.go

├── constants.go

├── go.mod

├── go.sum
```

✅ Your code is ready, but Go doesn't understand it yet as something it can run —

it must be **compiled** (converted into a binary/executable).

⚙️ Step 2 — Build (Compile) the Go program

Now you turn your `.go` files into one **executable file**.

You do this using your Makefile or a Go command:

Option 1 (simple command):

```
GOOS=linux GOARCH=amd64 go build -o bootstrap
```

Option 2 (using Makefile):

```
make install
```

🧩 What happens here:

1. Go reads all your `.go` files
2. Checks for syntax errors
3. Combines everything

4. Compiles into a single file named `bootstrap`

📦 **bootstrap** = your final compiled Go program (binary file)

💡 For AWS Lambda, this "bootstrap" file is what actually runs — not the `.go` code directly.

📦 Step 3 — Package your Go binary for deployment

Now, AWS Lambda can only accept **.zip files** as uploads.

So, you compress your bootstrap file:

```
zip deployment.zip bootstrap
```

✅ Now you have a file called **deployment.zip**

This is the one you'll upload to AWS.

☁️ Step 4 — Upload to AWS (Deploy)

Now, you move your `deployment.zip` to AWS so Lambda can use it.

2 ways to do this:

Option 1: Using AWS CLI (Command)

```
aws s3 cp deployment.zip s3://your-bucket-name/artifacts/
```

→ This uploads it to your S3 bucket.

Then, you go to AWS Lambda console:

- Choose your Lambda function

- Click “Upload from S3”
- Select your `deployment.zip`

✅ Lambda now runs your Go binary whenever it’s triggered.

🚀 Step 5 — Test your Lambda function

Once Lambda has your Go binary:

1. You can **manually test** by sending a JSON event.
2. Or connect it to **CloudWatch events** or **Splunk** to trigger automatically.

Example test event:

```
{

 "body": "{\"severity\": \"critical\", \"description\": \"Task stopped\"}"

}
```

Lambda will:

- Parse this event
- Check ECS tasks
- Send PagerDuty alerts if needed

✅ You’ll see the logs in **CloudWatch Logs**.

🔍 Step 6 — Check logs and monitor

Every `fmt.Printf()` in your Go code goes to **AWS CloudWatch Logs**.

You can open:

➡️ AWS Console → CloudWatch → Logs → Your Lambda’s Log Group

Here you'll see:

- Your print statements (`fmt.Printf`)
- Any errors
- Alerts sent to PagerDuty

🧪 Step 7 — Test locally (optional)

Before deploying, you can also test your Go Lambda locally using this:

```
go run main.go
```

But note:

- Some AWS calls (like `ecs.ListTasks`) will only work if your local machine has AWS credentials configured (`~/.aws/credentials`).

🧰 Step 8 — Version control (optional but important)

Before or after deployment, always:

```
git add .

git commit -m "Final working version"

git push origin main
```

✅ This keeps your project safe on GitHub.

🧹 Step 9 — Clean up

After deployment or testing, you can clean everything with:

```
make clean
```

This deletes the old binary and zip files.