● JANUARY 2026 SERIES

FROM GO BUILD TO GO RUN

GOLANG 2026 - NIV RAVE

#21

ERROR PHILOSOPHY

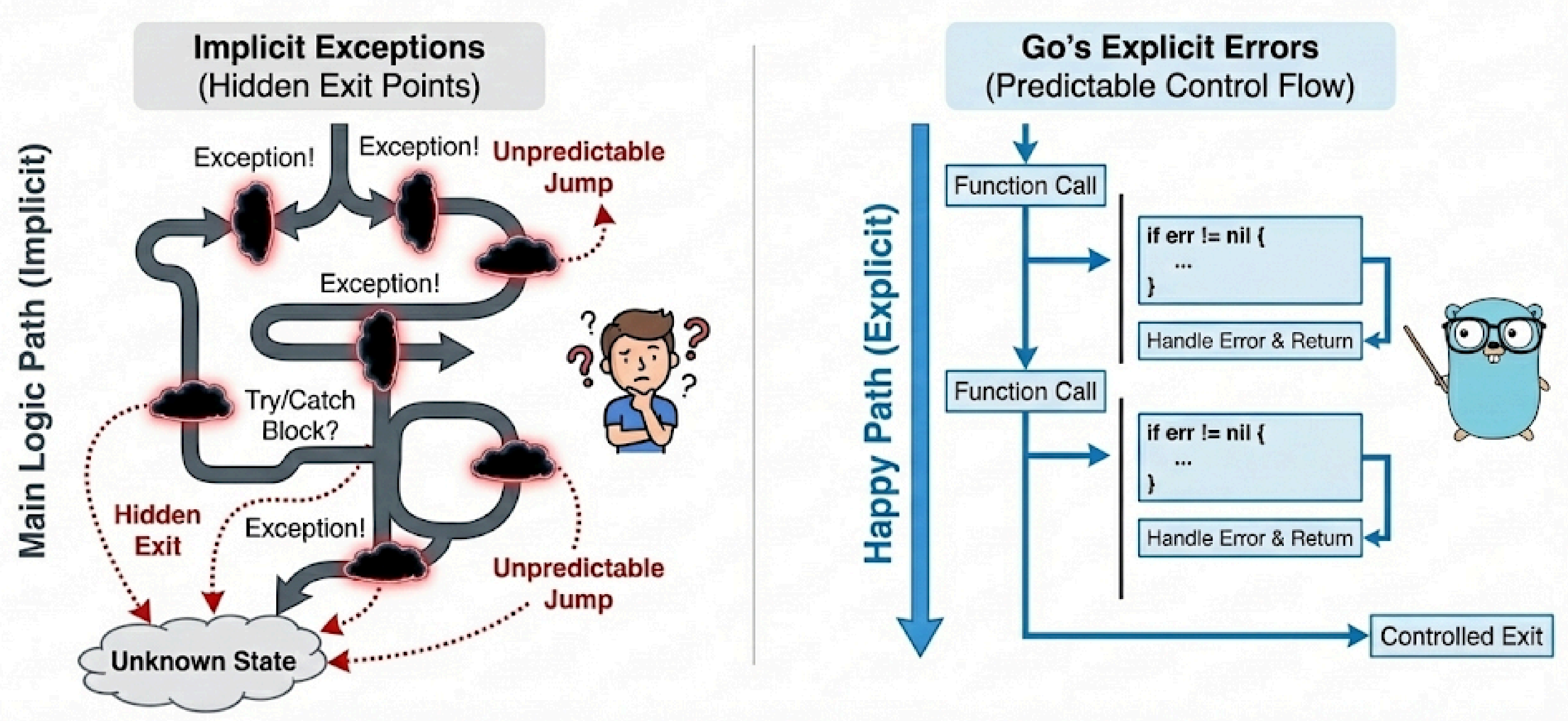
WHY GO DOESN'T HAVE TRY/CATCH, AND WHY THAT'S A FEATURE

Control Flow Clarity

Explicit vs. Implicit

Exceptions create "hidden" exit points in your logic. Go forces you to acknowledge every failure point

When you look at a Go function, you see the "Happy Path" on the left margin and the error handling indented. It makes the control flow predictable and auditable.



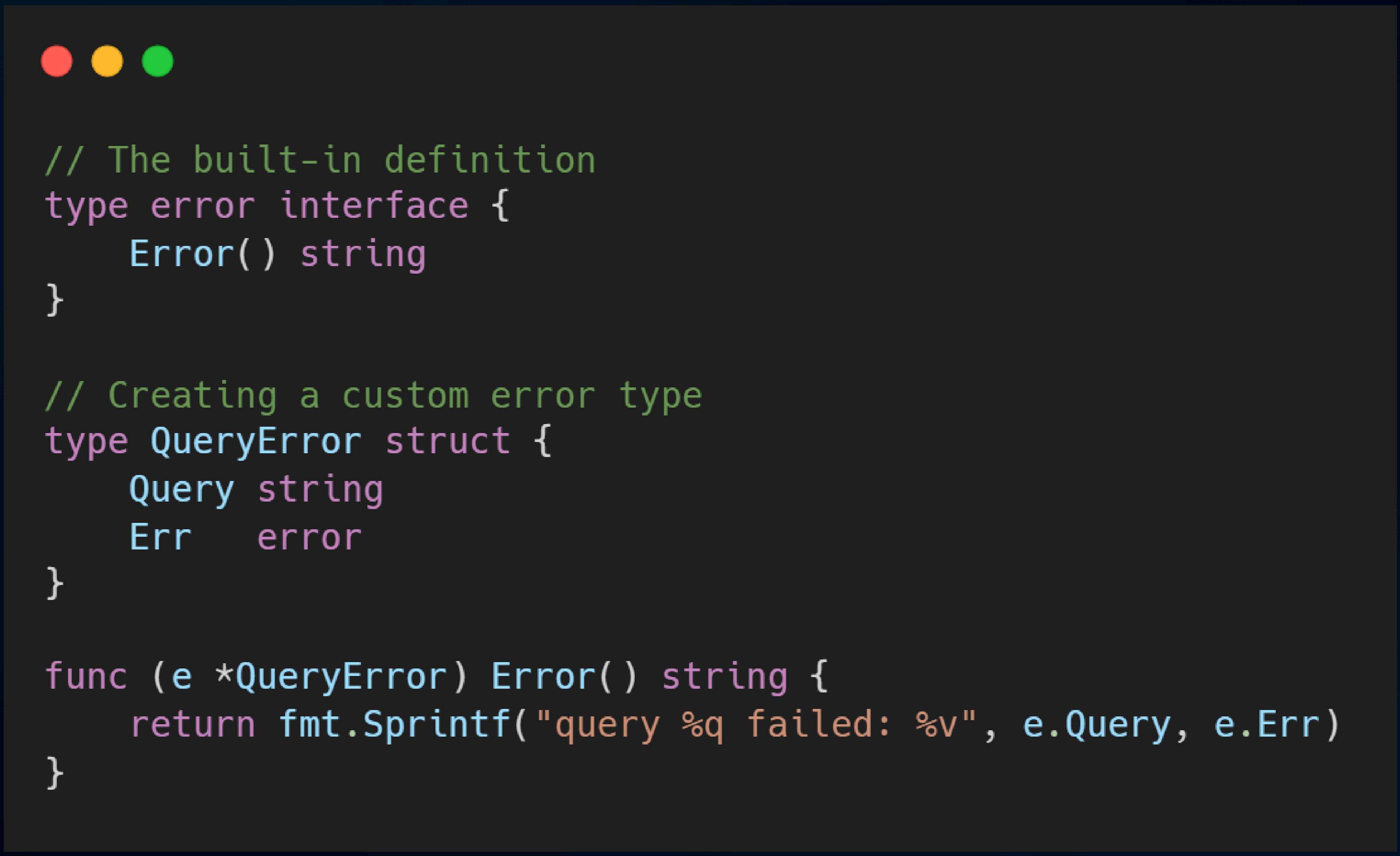
```
if err != nil {
    ...
}
```

```
if err != nil {
    ...
}
```

The Golang error

It's Just an Interface

In Go, an *error* is simply any type that implements a single method: *Error() string*.



```go
// The built-in definition
type error interface {
    Error() string
}

// Creating a custom error type
type QueryError struct {
    Query string
    Err   error
}

func (e *QueryError) Error() string {
    return fmt.Sprintf("query %q failed: %v", e.Query, e.Err)
}
```

Choosing Your Weapon

Sentinel Errors vs. Error Types

Sentinel Errors (Fixed Values)

Sentinel errors are pre-defined variables used for simple state checking. They represent a specific condition that the caller can check for using *errors.Is()*.

What: Global variables, usually created with *errors.New* or *fmt.Errorf*.

When to use: When the caller only needs to know if a specific error happened (the "What"), not the technical details behind it.

Best for: *ErrNotFound*, *ErrPermissionDenied*, *ErrUnauthenticated*.

Custom Error Types (Structs)

Custom error types are structs that implement the *Error()* interface. They allow you to attach dynamic metadata to the error.

What: Structs that hold specific fields and implement *Error() string*.

When to use: When the caller needs to know the "Why" or "Where" to take programmatic action (e.g., retrying after a specific duration).

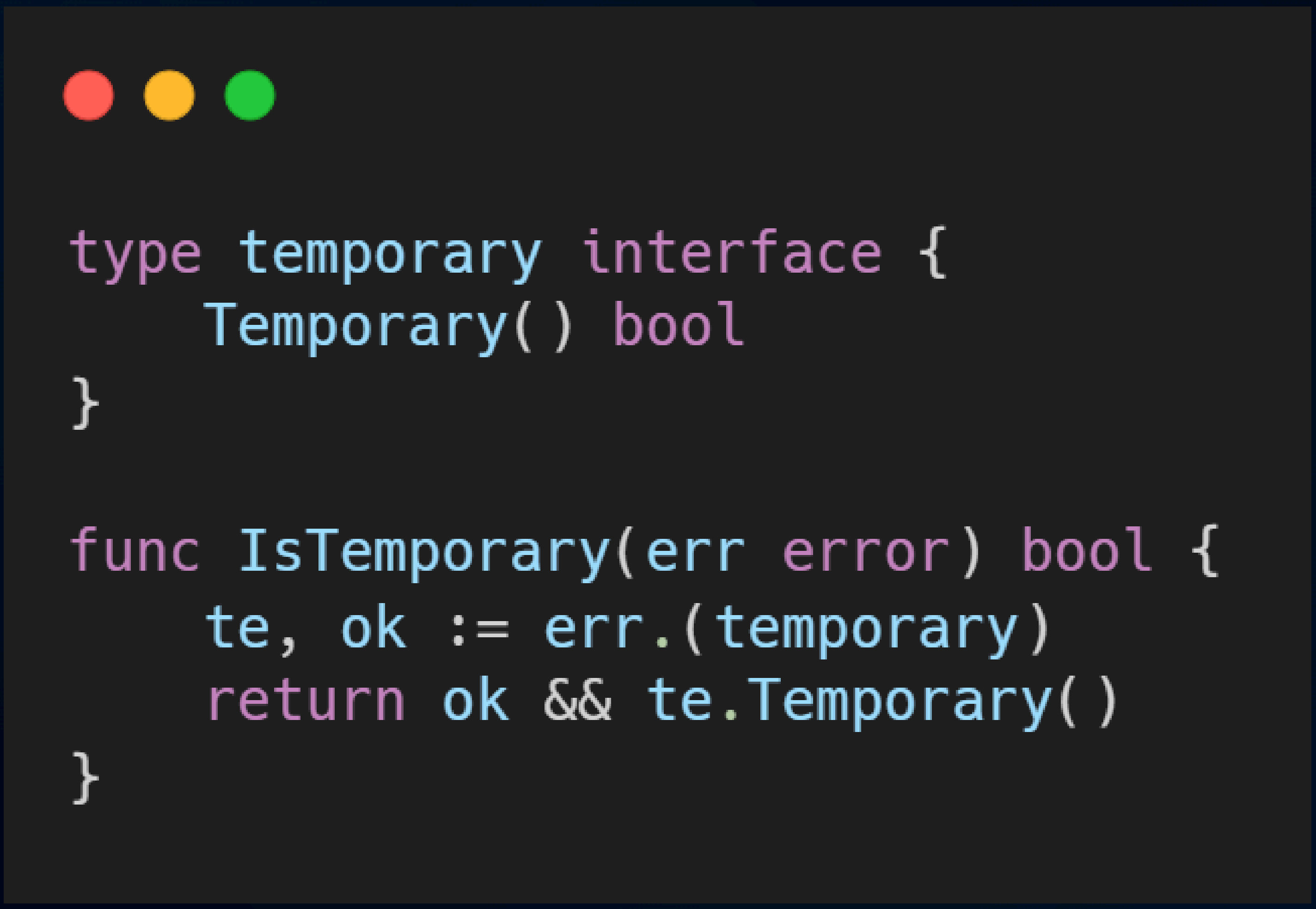
Best for: *TimeoutError* (with duration), *ValidationError* (with field names), *DatabaseError* (with query codes).

Opaque Errors

Opaque Errors: Flexible APIs

Sometimes you don't want the caller to depend on a specific type, but you want them to know a **behavior**.

The Concept: Define an interface for the behavior (e.g., *IsTemporary()*) rather than the type.



```go
type temporary interface {
    Temporary() bool
}

func IsTemporary(err error) bool {
    te, ok := err.(temporary)
    return ok && te.Temporary()
}
```

This decouples your packages. The caller doesn't need to import your internal error types to handle them correctly.

The "Stack" Problem

Where are my Stack Traces?

In languages like Java or Python, an exception is a heavy object containing a full stack trace. In Go, an *error* is just a string-generator. This is great for the CPU, but a nightmare for the developer trying to find a "nil pointer" in 50,000 lines of code.

The Solution Space:

The "Context" Approach: Instead of a stack trace, manually wrap the error at every layer with fmt.Errorf("layer name: %w", err).

The "External" Approach: Use a library like pkg/errors or uber-go/zap to capture a snapshot of the stack only when the error is first created.

Performance: Capturing a stack trace is an expensive operation - it requires the runtime to "walk" the stack. Never use stack traces for expected errors (like *UserNotFound*). Only use them for unexpected, "this-should-never-happen" scenarios.

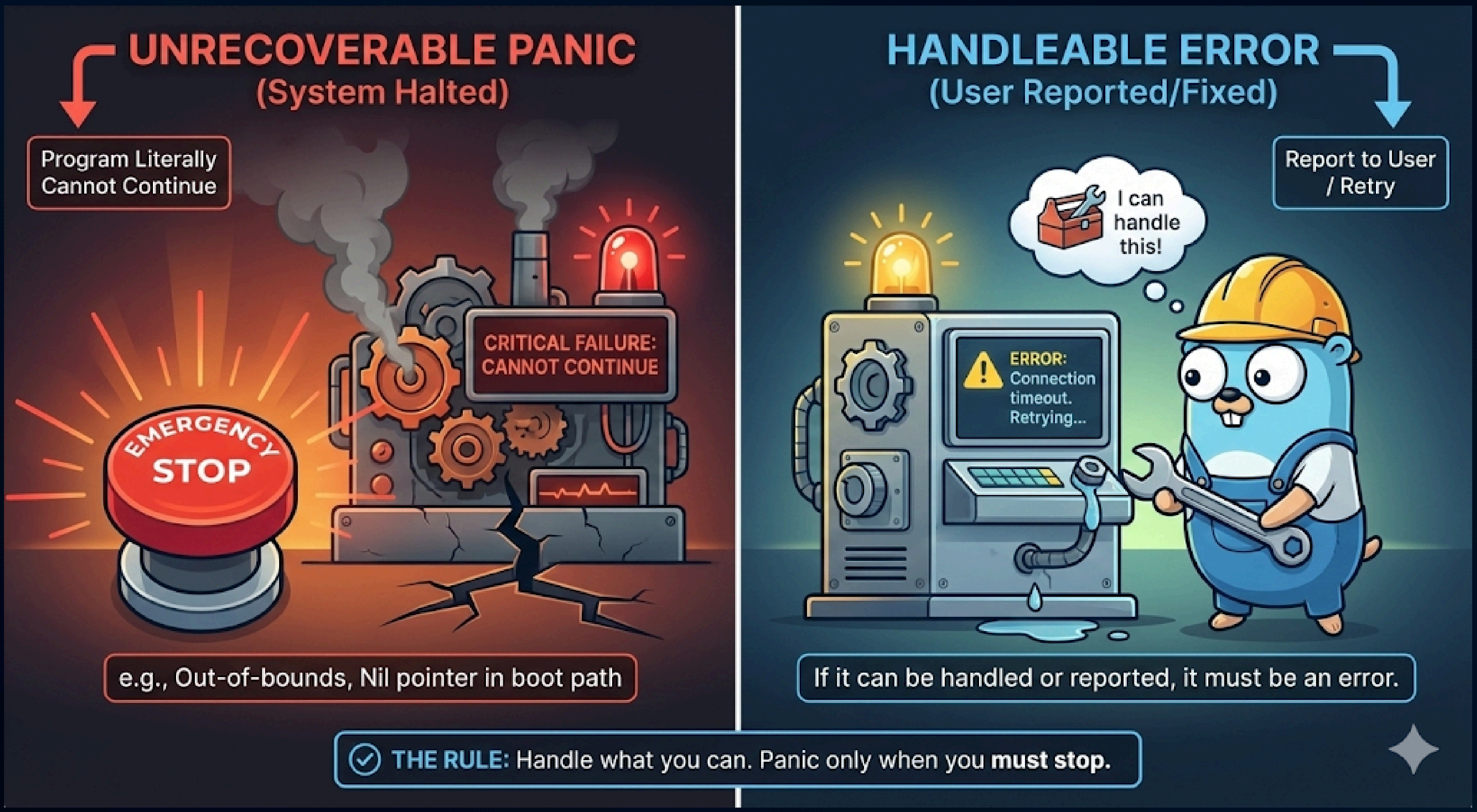
The Log Rule: Don't print the stack trace to the end-user. Use a structured logger (like *slog*) to attach the stack trace to your internal telemetry, while sending a clean *request_id* to the client.

Modern Go: With the rise of OpenTelemetry, the "Stack Problem" is often solved at the tracing layer (spans) rather than inside the Go error variable itself.

Panic is not an Error

Stop using Panic for Flow Control

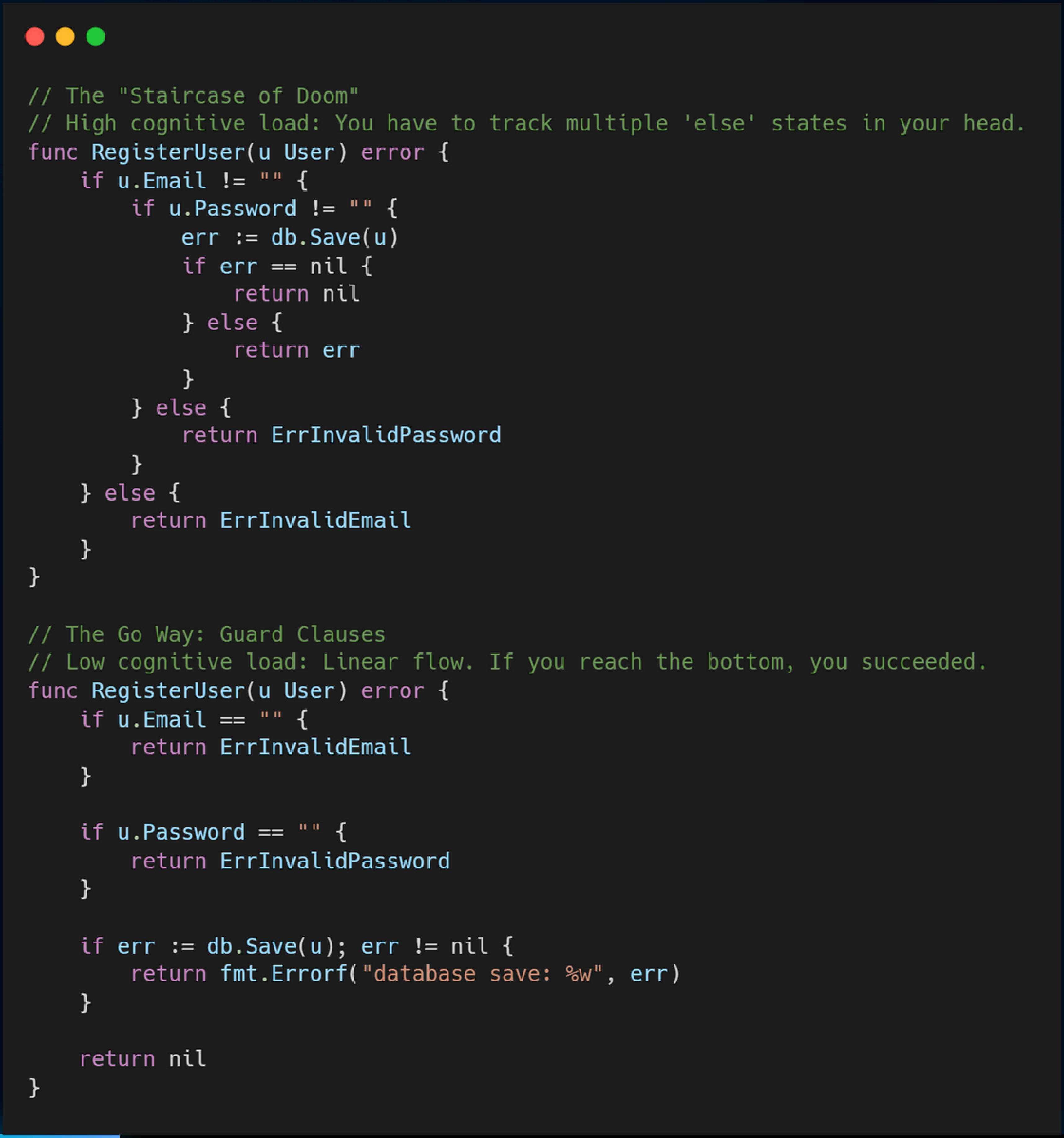
panic is for unrecoverable system states (e.g., out-of-bounds array access, nil pointer in a critical boot path).
The Rule: If it can be handled or reported to a user, it must be an *error*. If the program literally cannot continue, only then *panic*.


UNRECOVERABLE PANIC
(System Halted)
Program Literally Cannot Continue
CRITICAL FAILURE: CANNOT CONTINUE
EMERGENCY STOP
e.g., Out-of-bounds, Nil pointer in boot path
HANDLEABLE ERROR
(User Reported/Fixed)
Report to User / Retry
I can handle this!
ERROR: Connection timeout. Retrying...
If it can be handled or reported, it must be an error.
THE RULE: Handle what you can. Panic only when you must stop.

The "Happy Path"

Avoid the Staircase of Doom, go with the Go way

Deeply nested code is a "code smell" that hides logic in a maze of braces. In Go, we use Guard Clauses to handle errors early and keep the "Happy Path" (the core logic) aligned to the left margin.



```go
// The "Staircase of Doom"
// High cognitive load: You have to track multiple 'else' states in your head.
func RegisterUser(u User) error {
    if u.Email != "" {
        if u.Password != "" {
            err := db.Save(u)
            if err == nil {
                return nil
            } else {
                return err
            }
        } else {
            return ErrInvalidPassword
        }
    } else {
        return ErrInvalidEmail
    }
}

// The Go Way: Guard Clauses
// Low cognitive load: Linear flow. If you reach the bottom, you succeeded.
func RegisterUser(u User) error {
    if u.Email == "" {
        return ErrInvalidEmail
    }

    if u.Password == "" {
        return ErrInvalidPassword
    }

    if err := db.Save(u); err != nil {
        return fmt.Errorf("database save: %w", err)
    }

    return nil
}
```

Errors are not Exceptions!

Recap:

- Errors are values you can inspect and pass.
- Use interfaces to define error behavior.
- Keep the "Happy Path" on the left.

Do you prefer Go's explicit checks, or do you miss the "magic" of try/catch?