ENGINEERING STANDARDS LIBRARY / SOFTWARE ENGINEERING

Go Engineering, Concurrency, and Service Architecture Standard

Simple service boundaries, goroutine lifecycle discipline, backpressure, cancellation, observability, and operable Go systems.

PERMANENT DOCUMENT ID

MYTHOS-SWE-0003

PUBLICATION

Candidate Standard
Version 1.0.0-candidate

ASSURANCE PROFILE

Standard / Advanced
High Assurance

PERMANENT PUBLICATION IDENTITY

Go Engineering, Concurrency, and Service Architecture Standard

DOCUMENT ID
MYTHOS-SWE-0003

VERSION
1.0.0-candidate

STATUS
Candidate Standard

PUBLISHER
Mythos Systems

PUBLICATION DATE
2026-07-24

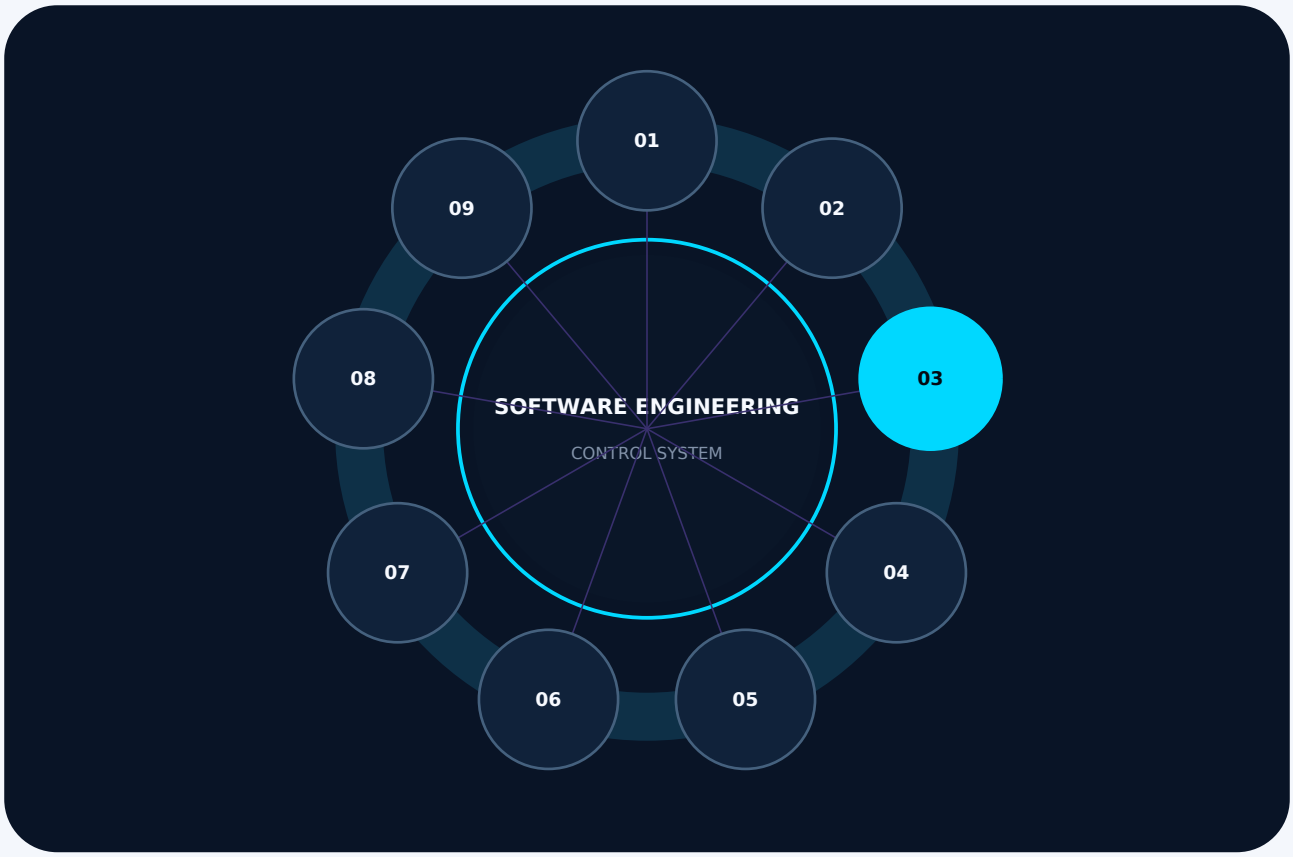
SCHEDULED REVIEW
2027-01-24

15	6	4	1
ATOMIC CRITERIA	ASSURANCE VIEWS	IMPLEMENTATION PROFILES	PERMANENT IDENTITY

Publication doctrine. This publication establishes durable guidance or requirements for its defined scope. Interpretation must preserve the permanent document identity, recorded evidence, controlled exceptions, and formal revision history.

DOMAIN CONTROL SYSTEM

MYTHOS-SWE-0003 inside the software engineering standard constellation



Connected assurance	Architecture, implementation, operations, evidence, and lifecycle are evaluated as one controlled system.
Specialization rule	Companion standards may add precision, but they cannot silently weaken this publication's requirements.

EXECUTIVE POSITION

Go Engineering, Concurrency, and Service Architecture Standard

The deployment of Go services in production has failed to live up to the language's promise of simple, reliable concurrency.

Evidence-bearing delivery path

Assurance maturity spectrum

CONTENTS

Publication navigation

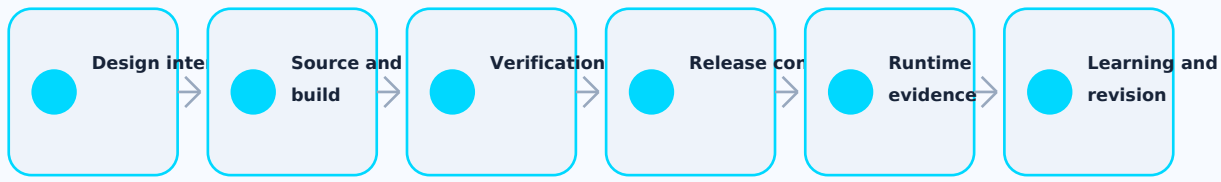
CONTROL TOPOLOGY

Assurance model and evaluation surface

The standard is assessed as an evidence-bearing system. Capability scores do not replace mandatory gates, implementation proof, or operational validation.

Capability family	Evaluation nodes
Goroutine Lifecycle and Concurrency (Weight: 25%)	2
Error Handling and Observability (Weight: 25%)	2
Context Discipline (Weight: 15%)	1
Interface and Package Architecture (Weight: 15%)	2
Race Conditions and Safety (Weight: 10%)	1
Operational Lifecycle (Weight: 10%)	1
Suite Composition	2
Additional Evaluation Dimension	4

Lifecycle control loop



EVALUATION INDEX

Criteria and evidence map

#	Capability family	Criterion
1	Goroutine Lifecycle and Concurrency (Weight: 25%)	Bounded Concurrency
2	Goroutine Lifecycle and Concurrency (Weight: 25%)	Goroutine Lifecycle Tracking
3	Error Handling and Observability (Weight: 25%)	Error Wrapping
4	Error Handling and Observability (Weight: 25%)	Structured Logging
5	Context Discipline (Weight: 15%)	Context Propagation
6	Interface and Package Architecture (Weight: 15%)	Interface Segregation
7	Interface and Package Architecture (Weight: 15%)	Package Structure
8	Race Conditions and Safety (Weight: 10%)	Race Detector
9	Operational Lifecycle (Weight: 10%)	Graceful Shutdown
10	Suite Composition	M-GO-S-Lifecycle
11	Suite Composition	M-GO-S-Errors
12	Additional Evaluation Dimension	Go Dimensions
13	Additional Evaluation Dimension	The Go Doctrine
14	Additional Evaluation Dimension	Scoring Model
15	Additional Evaluation Dimension	Suite Composition

NORMATIVE PUBLICATION

Requirements, evaluation methodology, and implementation guidance

The following body preserves the substantive source standard while normalizing publication status, identity, navigation, and visual presentation.

Executive Mandate

The deployment of Go services in production has failed to live up to the language's promise of simple, reliable concurrency.

While Go's goroutines and channels make concurrent code easy to write, they make it exceptionally difficult to reason about lifecycle, resource exhaustion, and error propagation. Organizations deploy Go services that silently leak goroutines, swallow context cancellations, and return untraceable `error` interfaces up the stack, turning distributed tracing into a black box.

This standard exists because:

1. Goroutine Leaks are Silent Killers: A goroutine that never terminates holds onto memory, stack space, and referenced closures. In a long-running service, a leak of 1 goroutine per request will exhaust memory and crash the process. Goroutine lifecycle MUST be as rigorously managed as memory in C.
2. `if err != nil { return err }` is Anti-Observability: Returning bare errors destroys the stack trace and context. Production debugging requires structured, wrapped errors with domain context, correlation IDs, and classification (retryable vs. fatal).
3. Context Cancellation is Not Optional: Ignoring `ctx.Done()` in I/O-bound goroutines creates zombie processes that consume resources even after the client has disconnected. All blocking operations MUST respect context.
4. Unbounded Concurrency is a DoS Vector: Spawning a goroutine per request without a semaphore or bounded channel is equivalent to disabling backpressure. The system WILL crash under load.

This document establishes the Mythos Go Engineering Standard (MGES), a comprehensive, evidence-based methodology for evaluating Go codebases against concurrency safety, error discipline, and service architecture.

Scope and Applicability

In Scope

This standard covers the evaluation of:

1. Goroutine lifecycle management and leak prevention
1. Structured error handling and wrapping (`fmt.Errorf`, custom types)
1. `context.Context` propagation and cancellation

1. Bounded concurrency (semaphores, worker pools, bounded channels)
1. Interface design and package architecture
1. Race condition prevention and `-race` enforcement
1. Memory management and GC tuning for low-latency services
1. Service initialization and graceful shutdown

Out of Scope

1. General domain-driven design (covered in MYTHOS-SWE-0001)
1. Rust memory safety (covered in MYTHOS-SWE-0002)
1. Polyglot IPC boundaries (covered in MYTHOS-SWE-0005)

Audience

1. Go engineers and systems programmers
1. Principal architects selecting languages for control planes
1. Security teams evaluating concurrency safety
1. Platform engineering teams building Go service scaffolding
1. Standards bodies seeking a reference Go methodology

Definitions and Taxonomy

Go Dimensions

Dimension	Definition
Goroutine Leak	A goroutine that blocks forever or runs indefinitely without a termination condition, preventing the garbage collector from reclaiming its stack and referenced memory.
Context Propagation	The practice of passing `context.Context` as the first parameter to all functions performing I/O or long-running work, enabling cancellation and deadlines.
Bounded Concurrency	Limiting the number of concurrent goroutines using semaphores (`chan struct{}`), worker pools, or `errgroup.Group` with `SetLimit()`.
Error Wrapping	The practice of adding context to an error using `fmt.Errorf("doing X: %w", err)` or custom error types, preserving the original error chain for `errors.Is` and `errors.As`.

The Go Doctrine

> A goroutine without a lifecycle is a leak. An error without context is a lie. A context without cancellation is a zombie. Simplicity is not an excuse for negligence.

Evaluation Methodology

Scoring Model

Each capability is scored from 0 to 5.

Score	Meaning
0	Fails basic task; goroutine leaks, bare errors, unbounded concurrency
1	Basic context usage but lacks error wrapping or lifecycle management
2	Functional but lacks bounded concurrency or structured errors
3	Solid production performance; bounded, wrapped, context-aware
4	Strong performance; structured error classification, leak detection
5	Best-in-class; sets the standard for deterministic, observable Go services

Weighting

Category	Weight	Rationale
Goroutine Lifecycle & Concurrency	25%	Goroutine leaks are the #1 operational failure in Go
Error Handling & Observability	25%	Bare errors make debugging impossible
Context Discipline	15%	Ignored contexts cause resource exhaustion
Interface & Package Architecture	15%	Circular dependencies and "util" packages kill modularity
Race Conditions & Safety	10%	Map concurrent writes cause panics
Operational Lifecycle	10%	Graceful shutdown is mandatory

Total: 100%

A codebase MUST score at least 3.0 in Goroutine Lifecycle and Error Handling to be considered for production use.

Evaluation Categories

Goroutine Lifecycle and Concurrency (Weight: 25%)

Bounded Concurrency

Are all goroutine spawns bounded and backpressured?

Score	Criteria
0	`go func()` spawned per request without limits
1	Bounded channels used but semaphores not
2	`errgroup.Group` used for fan-out, but unbounded
3	`errgroup.Group` with `SetLimit()` or semaphore for all fan-out
4	Bounded concurrency + graceful degradation when limits hit

Score	Criteria
5	Perfect discipline: formally verified concurrency limits, automated leak detection in CI (`goleak`), zero unbounded spawns

Goroutine Lifecycle Tracking

Can the system prove no goroutines are leaked?

Score	Criteria
0	No tracking; rely on monitoring memory growth
1	Manual code review for leaks
2	`go.uber.org/goleak` used in tests
3	All goroutines have explicit `select { case <-ctx.Done(): ... }` exit conditions
4	Runtime metrics (`runtime.NumGoroutine`) monitored and alerted
5	Perfect tracking: CI-enforced `goleak`, formally verified exit conditions, zero `go func()` without context

Error Handling and Observability (Weight: 25%)

Error Wrapping

Are errors wrapped with domain context before returning?

Score	Criteria
0	Bare `return err` everywhere
1	`fmt.Errorf("message: %v", err)` (loses wrapping)
2	`fmt.Errorf("message: %w", err)` (preserves chain)
3	Custom error types with domain context and `Unwrap()`
4	Error classification (Retryable, Permanent, NotFound) using `errors.Is`/`errors.As`
5	Perfect handling: structured error envelopes, classification, redaction, and zero bare `return err` outside standard library

Structured Logging

Are errors logged with structured context (trace ID, tenant, request ID)?

Score	Criteria
0	`fmt.Println` or `log.Printf`
1	`logrus` or `zap` but unstructured messages
2	Structured logging (`slog`/`zap`) with key-value pairs
3	Correlation IDs and trace IDs injected into context and logged
4	Errors logged with structured stack traces and domain state
5	Perfect observability: structured logging, redaction of secrets, `slog` context propagation, and zero unstructured error logs

Context Discipline (Weight: 15%)

Context Propagation

Do all I/O and blocking operations accept and respect `context.Context`?

Score	Criteria
0	Functions perform I/O without context
1	Context created at function start, not propagated from caller
2	Context passed but ignored in blocking operations
3	All I/O operations accept `ctx` and pass it down
4	All blocking `select` statements include `case <-ctx.Done()`
5	Perfect discipline: no I/O without context, all goroutines respect cancellation, enforced by linter

Interface and Package Architecture (Weight: 15%)

Interface Segregation

Are interfaces small, defined by the consumer, and not "dumb" data bags?

Score	Criteria
0	God interfaces with 20+ methods
1	Interfaces defined at the provider, not consumer
2	Consumer-defined interfaces (Go style) but still large
3	Single-method interfaces (`io.Reader`, `io.Writer` style) prevalent
4	Interfaces used to decouple from external dependencies (databases, APIs)
5	Perfect architecture: consumer-defined, single-responsibility interfaces, zero provider-defined interfaces outside `internal`

Package Structure

Is the package structure modular and free of circular dependencies?

Score	Criteria
0	Everything in `package main` or a single `utils` package
1	Packages organized by layer (controllers, services, models)
2	Packages organized by domain/bounded context
3	Use of `internal` to prevent external dependencies on implementation
4	Clean dependency graph; no circular dependencies; `cmd/` separate from `internal/`
5	Perfect structure: formally verified dependency graph, domain-centric, zero `util` or `helpers` packages

Race Conditions and Safety (Weight: 10%)

Race Detector

Is the Go race detector (`-race`) mandatory in CI and tests?

Score	Criteria
0	`-race` never used
1	`-race` used locally sometimes
2	`-race` used in unit tests only
3	`-race` mandatory in CI for all tests
4	`-race` used in integration tests and load tests
5	Perfect enforcement: `-race` mandatory in all CI stages, zero map concurrent writes, mutexes validated

Operational Lifecycle (Weight: 10%)

Graceful Shutdown

Does the service handle `SIGTERM` and drain connections?

Score	Criteria
0	Immediate exit on `SIGTERM`; requests dropped
1	`os.Signal` caught but no drain logic
2	HTTP server `Shutdown()` with a timeout
3	Graceful shutdown of all goroutines using `context.WithTimeout`
4	Shutdown sequence respects in-flight requests and completes DB transactions
5	Perfect shutdown: deterministic drain, formally verified in-flight completion, zero dropped requests

The Mythos Go Engineering Suite (M-GO-S)

Traditional benchmarks measure p99 latency. The M-GO-S evaluates goroutine leaks, error context, and concurrency bounds.

Suite Composition

M-GO-S-Lifecycle

1. Goroutine Leak: 100+ tests running `go.uber.org/goleak` after every test suite.
1. Context Cancel: 50+ tests cancelling context mid-request, verifying no goroutines hang.

M-GO-S-Errors

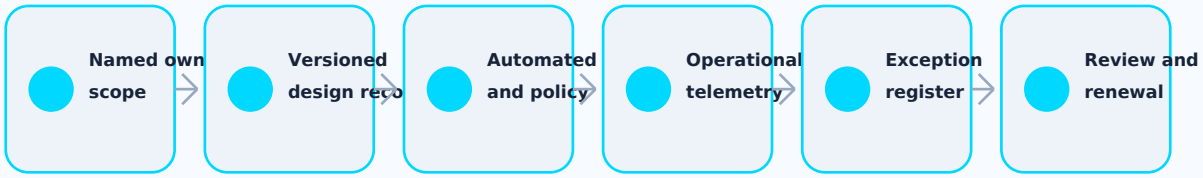
1. Bare Error Scan: 100+ tests scanning for `return err` without wrapping.
1. Structured Log Check: 50+ tests verifying errors are logged with trace IDs.

Execution Protocol

ASSURANCE AND ADOPTION

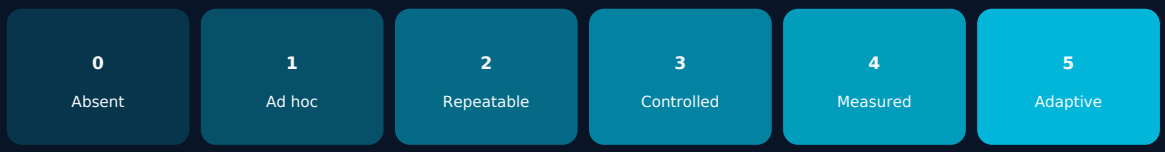
Implementation evidence and review gates

Required assurance chain



A passing score is not sufficient without current evidence. High-assurance adoption requires traceable design decisions, reproducible builds, independently reviewable tests, controlled secrets, runtime observability, failure exercises, and a dated exception register.

Assurance maturity spectrum



Current authority register

Authority	Version / snapshot	Applicability	Review by
Rust Project: The Rust Programming Language and Rust Reference	Rolling official documentation; book assumes Rust 1.90.0 or later and Rust 2024 Edition	High	2026-10-22
Go Project: The Go Programming Language Specification and Effective Go	Rolling official documentation snapshot	High	2026-10-22
Python Software Foundation: Python Language Reference, Packaging, Typing, and PEP Index	Rolling official documentation snapshot	High	2026-10-22
Tauri Programme: Tauri 2 Documentation	Tauri 2 rolling documentation snapshot	High	2026-10-22
Svelte Project: Svelte Documentation	Svelte 5 rolling documentation snapshot	High	2026-10-22
TLA+ Foundation: TLA+ Language and Tools Documentation	Rolling documentation snapshot	High	2027-01-24
OpenAPI Initiative: OpenAPI Specification	3.2.0	High	2027-01-24
OpenTelemetry Project: OpenTelemetry Documentation and Specification	Rolling specification snapshot; profiles public alpha and declarative configuration stable in 2026	High	2026-10-22

Research limitation. Official documentation establishes published interfaces and declared behavior. It does not prove the performance, security, interoperability, cost, or operational suitability of a specific implementation. Those claims require controlled testing and retained evidence.