

Go^{1.27}

Style Guide

Conventions for writing, structuring, testing and shipping Go 1.27 services, agents and CLIs that stay simple, explicit and easy to read.

```
func (s *Service) MarkPaid(ctx context.Context, id InvoiceID) error {  
    // Clear is better than clever.  
    return fmt.Errorf("mark paid %s: %w", id, err)  
}
```

Inside this guide

01	Purpose and how to read this guide	2
02	Baseline and toolchain	3
03	Project layout	5
04	Naming	7
05	Formatting and linting	9
06	Packages, APIs and interfaces	10
07	Types, structs, generics and zero values	12
08	Error handling	14
09	Functions, methods and control flow	17
10	Context and cancellation	19
11	Concurrency	20
12	Logging and observability	22
13	Configuration, CLIs and startup	24
14	HTTP services	26
15	Data access	28
16	Agents and daemons	30
17	Performance and memory	32
18	Security	33
19	Testing	34
20	Documentation and comments	36
21	Git, commits, releases and review	37
A	Appendix	40

Rule levels

MUST

MUST NOT

SHOULD

SHOULD NOT

MAY

AVOID

01

Purpose and how to read this guide

Every Go project that adopts this guide — services, CLIs, agents and daemons, libraries — is written so that any two projects read as if one person wrote them. It builds on [Effective Go](#), the [Go Code Review Comments](#) and the [Google Go Style Guide](#), and adds decisions those leave open.

The guide is generic. Examples use the module path `example.com/project`, the binary `projectd` and an `invoice` domain; swap in your own.

Rule levels

LEVEL	MEANING
<code>MUST</code> / <code>MUST NOT</code>	Required. CI or review blocks the change.
<code>SHOULD</code> / <code>SHOULD NOT</code>	The default. Deviate only with a short comment explaining why.
<code>MAY</code>	Allowed when it helps; no justification needed.
<code>AVOID</code>	Legal but discouraged; expect a review question.

Principles

- **Clarity first.** The reader's time matters more than the writer's. Clear beats clever; obvious beats short.
- **Simplicity.** Use the least machinery that solves the problem: a function before a type, a type before an interface, the standard library before a dependency.
- **Consistency.** Match the surrounding code, then this guide, then the wider Go community.
- **Let the tools decide.** Anything `gofmt`, `go vet`, `go fix` or `golanci-lint` can check is not debated in review.
- **Errors are values.** Handle them where they happen, add context, and never hide them.
- **Measure before optimizing.** Profile first; keep the fast path readable.

02

Baseline and toolchain

AREA	STANDARD
Language	Go 1.27 (August 2026): generic methods, <code>encoding/json/v2</code> , standard <code>uuid</code> package
Previous release	Go 1.26 remains supported until Go 1.28 ships; libraries MAY target it
Modules	One module per repository by default; <code>go.work</code> only for local multi-module work
Formatting	<code>gofmt</code> (via <code>gofumpt</code>) + <code>goimports</code>
Linting	<code>go vet</code> + <code>golangci-lint</code> v2 with the committed config (Appendix A)
Modernizing	<code>go fix</code> modernizers, run before each release
Vulnerabilities	<code>govulncheck</code> in CI
Tests	<code>go test -race -shuffle=on</code> ; <code>testing/syncstest</code> for time and concurrency
Tasks	<code>Taskfile.yml</code> or a <code>Makefile</code> with standard targets (Appendix B)
Release	GoReleaser for binaries and container images

Rules

- MUST** GO-02.1 declare the language version with a `go 1.27.0` line in `go.mod`, and pin the toolchain with a `toolchain` line so every machine and CI builds with the same compiler.
- MUST** GO-02.2 commit `go.sum`. **MUST NOT** commit a `vendor/` directory unless the build environment has no module proxy.
- MUST** GO-02.3 manage developer tools with `go tool` (the `tool` directive in `go.mod`), not with a `tools.go` file or global installs: `go get -tool github.com/golangci/golangci-lint/v2/cmd/golangci-lint`.
- SHOULD** GO-02.4 upgrade to each new Go release within one minor release, running `go fix ./...` to apply the modernizers.
- MUST NOT** GO-02.5 add a dependency for something the standard library does well (HTTP routing, structured logging, UUIDs, JSON, slices and maps helpers).
- SHOULD** GO-02.6 prefer small, well-maintained dependencies; every new direct dependency is called out in the pull request description.

- **MUST** GO-02.7 build release binaries with `CGO_ENABLED=0` unless the project needs cgo, and with `-trimpath` and version information injected (§21).

03

Project layout

Start flat and add structure only when a package earns it. A small CLI can be a single `main` package; the layout below is for services and agents.

```
project/
  cmd/
    projectd/          # One directory per binary; main.go only wires things up
      main.go
    projectctl/
      main.go
  internal/           # Everything not meant for other modules
    invoice/          # Domain package: types, rules, service
      invoice.go
      invoice_test.go  # Tests live beside the code they test
      service.go
      service_test.go
      store.go          # Interface the domain needs from storage
    postgres/         # Storage implementation (sqlc-generated + wrappers)
      queries/
        invoice_store.go
    httpapi/           # HTTP handlers, routing, middleware
    agent/             # Long-running agent loop (if any)
    config/
    platform/          # Cross-cutting: logging, metrics, shutdown
  migrations/         # Numbered SQL migrations
  api/                # OpenAPI specs, protobuf files
  deploy/             # Dockerfile, systemd units, Helm/compose
  scripts/
  testdata/           # Fixtures shared by tests
  go.mod
  go.sum
  Taskfile.yml
  .golangci.yml
```

Rules

- MUST** GO-03.1 put each binary in `cmd/<name>/`, with `main.go` limited to parsing config, building dependencies and calling `run(ctx)`. All logic lives in `internal/`.
- MUST** GO-03.2 put code in `internal/` unless another module is meant to import it. **MUST NOT** create a `pkg/` directory by default; a public package lives at the module root or under a named directory and is documented as public API.

- **MUST** GO-03.3 keep tests beside the code they test, in `<name>_test.go`. Black-box tests use the `<pkg>_test` package; white-box tests use the same package.
- **MUST** GO-03.4 organise packages by **what they provide** (domain or capability: `invoice`, `postgres`, `httpapi`), not by kind (`models`, `controllers`, `utils`).
- **MUST NOT** GO-03.5 create `util`, `common`, `helpers`, `misc` or `base` packages.
- **MUST NOT** GO-03.6 create import cycles; when two packages need each other, move the shared types into the domain package or define a small interface on the consumer side.
- **SHOULD** GO-03.7 keep test fixtures in a `testdata/` directory next to the test (the go tool ignores it).
- **SHOULD** GO-03.8 keep generated code in its own files ending in `.gen.go` or under a `gen/` directory, with the generator command in a `//go:generate` line.

04

Naming

THING	CONVENTION	EXAMPLE
Package	Short, lower case, one word, no underscores	<code>invoice</code> , <code>httpapi</code> , <code>postgres</code>
File	Lower case, underscores allowed	<code>invoice_store.go</code> , <code>service_test.go</code>
Exported identifier	MixedCaps	<code>NewService</code> , <code>MaxRetries</code>
Unexported identifier	mixedCaps	<code>parseLine</code> , <code>defaultTimeout</code>
Initialism	All one case	<code>ID</code> , <code>URL</code> , <code>HTTPClient</code> , <code>userID</code> , <code>apiURL</code>
Interface (one method)	Method + <code>er</code>	<code>Reader</code> , <code>Notifier</code>
Receiver	1-2 letters, consistent per type	<code>func (s *Service)</code>
Constructor	<code>New</code> / <code>New<Type></code>	<code>invoice.NewService</code>
Getter	No <code>Get</code> prefix	<code>inv.Total()</code> , not <code>inv.GetTotal()</code>
Error variable	<code>Err</code> prefix	<code>ErrNotFound</code>
Error type	<code>Error</code> suffix	<code>ValidationError</code>
Test	<code>Test<Func>_<case></code> or subtests	<code>TestParse</code> , <code>t.Run("empty input", ...)</code>
Context variable	<code>ctx</code> , first parameter	<code>func Do(ctx context.Context, ...)</code>
Constants	MixedCaps, never UPPER_SNAKE	<code>defaultPort</code>

Rules

- MUST** GO-04.1 avoid stutter: the package name is part of every call site. `invoice.Service`, not `invoice.InvoiceService`; `postgres.NewStore`, not `postgres.NewPostgresStore`.
- MUST** GO-04.2 keep names short in proportion to their scope: `i`, `n`, `r` in a three-line loop; descriptive names for package-level identifiers.

- **MUST NOT** GO-04.3 use `this` or `self` as a receiver name, and **MUST** use the same receiver name in every method of a type.
- **MUST NOT** GO-04.4 shadow package names or built-ins (`len`, `new`, `error`, `url`) with local variables.
- **SHOULD** GO-04.5 name booleans as states or questions: `enabled`, `isLeader`, `hasLines`.
- **SHOULD** GO-04.6 name functions that return a boolean as questions (`IsOverdue`), and functions with side effects as verbs (`Send`, `Close`).
- **AVOID** GO-04.7 naming a package after its pattern (`manager`, `handler`, `service`); name it after what it manages.

05

Formatting and linting

Formatting is never a review topic: the tools decide, CI enforces.

Formatting

- **MUST** GO-05.1 format every file with `gofumpt` (a stricter `gofmt`) and group imports with `goimports -local example.com/project`. CI fails on any diff.
- **MUST** GO-05.2 indent with **tabs**, as `gofmt` does. This is the one place the 2-space rule from other guides does not apply; Go's formatter is not configurable and that is the point.
- **MUST** GO-05.3 group imports in three blocks: standard library, third-party, then this module.
- **SHOULD** GO-05.4 keep lines under about **100** characters. There is no hard limit; break a long call one argument per line with a trailing comma rather than leaving a 200-character line.
- **MUST NOT** GO-05.5 use dot imports outside tests, or blank imports anywhere but `main` and tests, where database drivers are registered.
- **SHOULD** GO-05.6 order a file: package doc, imports, constants, variables, types (each followed by its constructor and methods), then free functions.

Static analysis

- **MUST** GO-05.7 run `go vet ./...` and `golangci-lint run` in CI with the committed `.golangci.yml` (Appendix A). The build fails on any finding.
- **MUST NOT** GO-05.8 silence a linter without a reason: `//nolint:gosec // path is from a constant, not user input`. Blanket `//nolint` is forbidden.
- **MUST** GO-05.9 run `govulncheck ./...` in CI and on a schedule.
- **SHOULD** GO-05.10 run `go fix ./...` (the Go 1.26+ modernizers) when upgrading Go, and commit the result in its own `refactor:` commit.
- **SHOULD** GO-05.11 add `//go:fix inline` to deprecated wrapper functions in public packages so callers can migrate automatically.

One command

- **MUST** GO-05.12 expose `task lint` and `task test` (or `make lint` / `make test`), and run exactly those in CI (Appendix B).

06

Packages, APIs and interfaces

Package design

- **MUST** GO-06.1 give each package one clear purpose that its name states; if the package doc needs "and", consider splitting it.
- **MUST** GO-06.2 keep the exported surface small. Export a name only when another package uses it today.
- **MUST NOT** GO-06.3 use package-level mutable state (globals, `init()` side effects) beyond registering drivers or metrics. Pass dependencies explicitly.
- **AVOID** GO-06.4 `init()` functions; when one is unavoidable, it must not do I/O or depend on flags or environment.
- **SHOULD** GO-06.5 keep a package's dependencies pointing inward: `httpapi` and `postgres` import `invoice`; `invoice` imports neither.

Interfaces

- **MUST** GO-06.6 define interfaces in the package that **uses** them, not the package that implements them. Producers return concrete types.
- **MUST** GO-06.7 keep interfaces small — usually one to three methods — and declare only the methods the consumer calls.
- **MUST NOT** GO-06.8 create an interface for a single implementation "for testing" when a real or in-memory implementation would do. Introduce the interface when the second implementation or a test double is actually needed.
- **SHOULD** GO-06.9 follow "accept interfaces, return structs".
- **SHOULD** GO-06.10 assert that a type satisfies an interface at compile time where the relationship matters:
`var _ invoice.Store = (*Store)(nil).`
- **MUST NOT** GO-06.11 embed an interface in a struct to "implement" it partially; a nil method call panics at runtime.

```

// Package invoice holds invoice rules and the service that applies them.
package invoice

// Store is what the invoice service needs from persistence.
type Store interface {
    Get(ctx context.Context, id InvoiceID) (Invoice, error)
    Save(ctx context.Context, inv Invoice) error
}

// Service applies invoice rules.
type Service struct {
    store Store
    clock func() time.Time
    log    *slog.Logger
}

// NewService returns a Service backed by store.
func NewService(store Store, log *slog.Logger) *Service {
    return &Service{store: store, clock: time.Now, log: log}
}

```

Compatibility

- **MUST** GO-06.12 follow semantic versioning for any module others import, and bump the major version (`/v2` module path) for breaking changes.
- **MUST** GO-06.13 mark replaced APIs with a `// Deprecated:` paragraph naming the replacement, and keep them for at least one minor release.

07

Types, structs, generics and zero values

Structs and zero values

- **SHOULD** GO-07.1 make the zero value useful: a `sync.Mutex`, `bytes.Buffer` or `Config` with sensible defaults needs no constructor. When the zero value cannot be valid, provide `New...` and document that the zero value must not be used.
- **MUST** GO-07.2 use field names in struct literals for types from other packages: `http.Server{Addr: addr}`, never positional.
- **MUST** GO-07.3 use pointer receivers when a method mutates the receiver, when the struct holds a `sync.Mutex` or similar, or when it is large; otherwise value receivers. **MUST NOT** mix receiver kinds on one type without reason.
- **MUST NOT** GO-07.4 copy a struct that contains a mutex, `sync.WaitGroup` or `atomic` value (`go vet` copylocks enforces this).
- **SHOULD** GO-07.5 use `new(expr)` (Go 1.26) for optional pointer fields instead of a temporary variable: `Age: new(yearsSince(born))`.
- **SHOULD** GO-07.6 use typed IDs and units to prevent mix-ups: `type InvoiceID uuid.UUID`, `type Cents int64`.
- **MUST** GO-07.7 store money as integer minor units (`int64` cents) with a currency code, never as `float64`.
- **MUST** GO-07.8 use UUIDv7 for generated identifiers so they sort by creation time, using the standard `uuid` package (Go 1.27) rather than a third-party module.

Enums

- **MUST** GO-07.9 declare enumerations as a named type with `iota` constants, starting from a meaningful zero (`StatusUnknown`) so an unset value is detectable.
- **SHOULD** GO-07.10 generate `String()` with `stringer` and implement `encoding.TextMarshaler` / `TextUnmarshaler` so the value round-trips through JSON and config as text.
- **MUST** GO-07.11 handle every case in a `switch` over an enum and return an error (or panic in truly impossible cases) in `default`; the `exhaustive` linter enforces this.

```

type Status int

const (
    StatusUnknown Status = iota
    StatusDraft
    StatusSent
    StatusPaid
    StatusOverdue
)

//go:generate go tool stringer -type=Status -trimprefix=Status

func (s Status) IsOpen() bool {
    return s == StatusSent || s == StatusOverdue
}

```

Generics

- **SHOULD** GO-0712 reach for generics when the same code would otherwise be written for several types (containers, **slices** / **maps** -style helpers, typed caches, result sets). **AVOID** generics for code that has one caller.
- **MUST** GO-0713 prefer the standard **slices**, **maps**, **cmp** and **iter** packages to hand-written loops or third-party helpers.
- **MAY** GO-0714 use generic methods (Go 1.27) when a method genuinely needs its own type parameter; remember that interface methods cannot be generic, so such a method can never satisfy an interface.
- **SHOULD** GO-0715 keep constraints as loose as the code allows (**any**, **comparable**, **cmp.Ordered**) and name type parameters with one capital letter or a short word (**T**, **K**, **V**, **Item**).
- **SHOULD** GO-0716 return iterators (**iter.Seq[T]**, **iter.Seq2[K, V]**) from collections or paged sources that callers range over, instead of building a full slice.

```

// All yields every invoice for the customer, fetching pages lazily.
func (s *Store) All(ctx context.Context, customer CustomerID) iter.Seq2[Invoice, error] {
    return func(yield func(Invoice, error) bool) {
        for page, err := range s.pages(ctx, customer) {
            if err != nil {
                yield(Invoice{}, err)
                return
            }
            for _, inv := range page {
                if !yield(inv, nil) {
                    return
                }
            }
        }
    }
}

```

08

Error handling

Handling and wrapping

- **MUST** GO-08.1 check every returned error. **MUST NOT** discard one with `_` unless a comment says why it cannot matter.
- **MUST** GO-08.2 handle an error once: either log it **or** return it, never both. The top of the call stack (the handler, the job runner, `main`) logs.
- **MUST** GO-08.3 add context when returning an error from another package, using `%w`: `fmt.Errorf("load invoice %s: %w", id, err)`. Context describes what **this** function was doing, in lower case, with no "failed to" or "error" prefix.
- **MUST** GO-08.4 keep the happy path left-aligned: handle the error and return early; no `else` after a `return`.
- **SHOULD** GO-08.5 use `%v` instead of `%w` when callers must not depend on the underlying error (for example, hiding a driver error behind a domain boundary).
- **MUST NOT** GO-08.6 wrap a pointer to an error value type (`*MyErr` when `MyErr` implements `error`); Go 1.27's vet flags it because `errors.Is` and type checks stop working.

Inspecting errors

- **MUST** GO-08.7 compare errors with `errors.Is` and extract them with `errors.AsType[T]` (Go 1.26), never with `=` on wrapped errors or string matching.
- **SHOULD** GO-08.8 expose **sentinel errors** (`var ErrNotFound = errors.New("invoice: not found")`) for conditions callers branch on, and **error types** when callers need fields.
- **MUST** GO-08.9 translate storage errors into domain errors at the boundary: `postgres` turns `pgx.ErrNoRows` into `invoice.ErrNotFound`.
- **SHOULD** GO-08.10 use `errors.Join` to return several independent errors (validation, closing multiple resources).

```

var ErrNotFound = errors.New("invoice: not found")

type ValidationError struct {
    Field string
    Reason string
}

func (e *ValidationError) Error() string {
    return fmt.Sprintf("invoice: %s %s", e.Field, e.Reason)
}

func (s *Service) MarkPaid(ctx context.Context, id InvoiceID) error {
    inv, err := s.store.Get(ctx, id)
    if err != nil {
        return fmt.Errorf("mark paid %s: %w", id, err)
    }
    if inv.Status == StatusPaid {
        return &ValidationError{Field: "status", Reason: "is already paid"}
    }

    inv.Status = StatusPaid
    inv.PaidAt = s.clock()

    if err := s.store.Save(ctx, inv); err != nil {
        return fmt.Errorf("mark paid %s: save: %w", id, err)
    }
    return nil
}

// In the HTTP layer:
switch err := svc.MarkPaid(ctx, id); {
case err == nil:
    w.WriteHeader(http.StatusNoContent)
case errors.Is(err, invoice.ErrNotFound):
    writeProblem(w, http.StatusNotFound, "invoice not found")
default:
    if verr, ok := errors.AsType[*invoice.ValidationError](err); ok {
        writeProblem(w, http.StatusUnprocessableEntity, verr.Error())
        return
    }
    writeInternalError(w, r, err)
}

```

Panics

- **MUST NOT** GO-08.11 panic for expected failures (bad input, missing rows, network errors). Return an error.
- **MAY** GO-08.12 panic for programmer errors that make continuing unsafe (an impossible state, a broken invariant), and in **Must...** helpers used only at startup with constant input (**regexp.MustCompile**).
- **MUST** GO-08.13 recover panics at goroutine and request boundaries (HTTP middleware, worker loops), log them with the stack, and turn them into errors. **MUST NOT** use **recover** for control flow.

- **MUST NOT** GO-08.14 call `log.Fatal` or `os.Exit` outside `main`; they skip deferred cleanup.

09

Functions, methods and control flow

Signatures

- MUST** GO-09.1 put `ctx context.Context` first and `error` last.
- SHOULD** GO-09.2 keep functions under about 60 lines and four parameters. When parameters grow, group them in a struct (`Params` , `Options`) or use functional options.
- MUST NOT** GO-09.3 use named results except to document two results of the same type or when a deferred function must set the error. **MUST NOT** use naked `return` in functions longer than a few lines.
- MUST NOT** GO-09.4 use boolean flag parameters that change behaviour (`Send(inv, true)`); use two functions or an options struct.
- SHOULD** GO-09.5 return a concrete value and an error, not a pointer to signal "maybe": `(Invoice, error)` with `ErrNotFound`, rather than `(*Invoice, error)` with `nil, nil`.
- MUST NOT** GO-09.6 return `nil, nil`.

Options

- SHOULD** GO-09.7 use an options struct for constructors with several optional settings; it is easy to read, document and extend.
- MAY** GO-09.8 use functional options (`WithTimeout(d)`) for public APIs where defaults matter and the set of options will grow.

```
type ClientOptions struct {
    Timeout    time.Duration // Default 10s.
    MaxRetries int           // Default 3; 0 disables retries.
    UserAgent  string
}

func NewClient(baseUrl string, opts ClientOptions) *Client {
    if opts.Timeout == 0 {
        opts.Timeout = 10 * time.Second
    }
    // ...
}
```

Control flow

- MUST** GO-09.9 return early to keep the main path at the lowest indentation.

- **SHOULD** GO-09.10 use `switch` with no condition instead of long `if / else if` chains.
- **SHOULD** GO-09.11 range over integers (`for i := range n`) and over functions (`for v := range seq`) instead of three-clause loops where it reads better.
- **MUST** GO-09.12 use `defer` to release resources right after acquiring them, and check the error from `Close` on writers: `defer func() { err = errors.Join(err, f.Close()) }()`.
- **MUST NOT** GO-09.13 `defer` inside a loop that runs many times; move the body into a function.
- **SHOULD** GO-09.14 declare variables close to first use, and prefer `:=` except when the zero value is the point (`var buf bytes.Buffer`).
- **AVOID** GO-09.15 `goto` and labelled `break / continue` except to leave a nested loop or `select` .

10

Context and cancellation

- **MUST** GO-10.1 pass `context.Context` as the first parameter, named `ctx`, to every function that does I/O, blocks or calls something that does.
- **MUST NOT** GO-10.2 store a context in a struct field. The only exception is a type that represents one request's lifetime and documents it.
- **MUST NOT** GO-10.3 pass `nil` as a context. Use `context.TODO()` while migrating, and `context.Background()` only in `main`, tests and top-level goroutines.
- **MUST** GO-10.4 derive the root context from signals in `main` with `signal.NotifyContext(ctx, os.Interrupt, syscall.SIGTERM)`; since Go 1.26 its cancel cause names the signal.
- **MUST** GO-10.5 set deadlines on every outbound call: `context.WithTimeout` per call, or a client-level timeout. A call with no deadline is a bug waiting for a slow dependency.
- **SHOULD** GO-10.6 use `context.WithCancelCause` / `WithTimeoutCause` and read `context.Cause(ctx)` so logs say **why** work stopped.
- **MUST** GO-10.7 call the `cancel` function returned by `WithCancel` / `WithTimeout`, normally with `defer cancel()`.
- **SHOULD** GO-10.8 use `context.WithoutCancel(ctx)` for work that must finish after the request ends (audit writes), and `context.AfterFunc` instead of a goroutine waiting on `ctx.Done()`.
- **MAY** GO-10.9 carry request-scoped values (request ID, authenticated principal, trace span) in a context with unexported key types and typed accessors. **MUST NOT** use context values for optional parameters or dependencies.

```
type ctxKey struct{}

func WithPrincipal(ctx context.Context, p Principal) context.Context {
    return context.WithValue(ctx, ctxKey{}, p)
}

func PrincipalFrom(ctx context.Context) (Principal, bool) {
    p, ok := ctx.Value(ctxKey{}).(Principal)
    return p, ok
}
```

11

Concurrency

Goroutines

- **MUST** GO-11.1 know how every goroutine ends before starting it: it stops when its context is cancelled, its input channel closes, or its work completes. A goroutine with no exit is a leak.
- **MUST** GO-11.2 wait for goroutines you start before returning from the function or shutting down: `sync.WaitGroup.Go` (Go 1.25+) or `errgroup.Group`.
- **SHOULD** GO-11.3 use `golang.org/x/sync/errgroup` with `errgroup.WithContext` for a set of tasks where the first error cancels the rest, and `g.SetLimit(n)` to bound parallelism.
- **MUST NOT** GO-11.4 start unbounded goroutines per item (one per row, per request to a dependency); bound them with a limit or a worker pool.
- **MUST** GO-11.5 recover panics in long-lived goroutines that are not covered by a request middleware, and report them.
- **SHOULD** GO-11.6 check for leaks in tests with the `goroutineleak` profile (generally available in Go 1.27) or `go.uber.org/goleak`.

Channels vs mutexes

- **SHOULD** GO-11.7 use channels to pass ownership of data or signal events between goroutines, and a `sync.Mutex` to protect shared state. Pick the one that makes the code simpler.
- **MUST** GO-11.8 close a channel only from the sending side, and only once; document which goroutine owns it.
- **SHOULD** GO-11.9 keep channels unbuffered or size 1 unless a measured reason calls for more; a large buffer usually hides a back-pressure problem.
- **MUST** GO-11.10 use `select` with `ctx.Done()` on every blocking send or receive that can outlive the caller.
- **MUST** GO-11.11 keep the mutex next to the fields it guards, declared just above them, and keep critical sections short with no I/O inside.
- **SHOULD** GO-11.12 use `atomic.Int64`, `atomic.Bool`, `atomic.Pointer[T]` (the typed atomics) instead of the `atomic.AddInt64` functions; `go fix` migrates old code.
- **SHOULD** GO-11.13 use `sync.OnceValue` / `OnceValues` for lazy initialization.
- **MUST** GO-11.14 run every test suite with `-race`; a race report is a bug, never flaky.

```

func (s *Syncer) SyncAll(ctx context.Context, sites []Site) error {
    g, ctx := errgroup.WithContext(ctx)
    g.SetLimit(8)

    for _, site := range sites {
        g.Go(func() error {
            if err := s.syncSite(ctx, site); err != nil {
                return fmt.Errorf("sync site %s: %w", site.ID, err)
            }
            return nil
        })
    }
    return g.Wait()
}

type Cache struct {
    mu      sync.RWMutex
    entries map[string]Entry // guarded by mu
}

```

12

Logging and observability

Logging with slog

- MUST** GO-12.1 log with `log/slog`. **MUST NOT** use `log.Printf`, `fmt.Println` or a third-party logger in new code.
- MUST** GO-12.2 inject a `*slog.Logger` into types that log; **AVOID** `slog.Default()` outside `main`.
- MUST** GO-12.3 log structured attributes, not formatted strings: `log.InfoContext(ctx, "invoice sent", "invoice_id", id, "attempt", n)`. Keys are `snake_case` and stable.
- SHOULD** GO-12.4 use the `...Context` methods so handlers can add request IDs and trace IDs from the context.
- MUST** GO-12.5 use levels consistently: `Debug` for developer detail, `Info` for business events and lifecycle, `Warn` for degraded but handled conditions, `Error` for failures someone must look at.
- MUST** GO-12.6 emit JSON (`slog.NewJSONHandler`) in production and text locally. Use `slog.NewMultiHandler` (Go 1.26) to send to more than one destination.
- MUST NOT** GO-12.7 log secrets, tokens, passwords or personal data; implement `slog.LogValuer` on sensitive types to redact them.
- SHOULD** GO-12.8 log an error once, at the boundary, with the full wrapped message under the `err` key.

```
type APIKey string

func (APIKey) LogValue() slog.Value { return slog.StringValue("[redacted]") }

log := slog.New(slog.NewJSONHandler(os.Stdout, &slog.HandlerOptions{Level: cfg.LogLevel}))
log = log.With("service", "projectd", "version", version.Version)
```

Metrics and tracing

- SHOULD** GO-12.9 instrument services with OpenTelemetry (traces + metrics) or Prometheus metrics; pick one per project and use it everywhere.
- MUST** GO-12.10 export the RED metrics for every server (rate, errors, duration) and the USE metrics for every worker pool or queue (utilization, saturation, errors).
- MUST** GO-12.11 keep metric label values bounded: no IDs, emails or URLs with parameters as labels.
- SHOULD** GO-12.12 expose `net/http/pprof` on a separate, internal-only listener, including the `goroutineleak` profile.

- **MUST** GO-12.13 expose `/healthz` (process is alive) and `/readyz` (dependencies are reachable) on services.

13

Configuration, CLIs and startup

Configuration

- **MUST** GO-13.1 load configuration once in `main` into a typed `Config` struct and pass the parts each component needs. **MUST NOT** call `os.Getenv` deep in the code.
- **MUST** GO-13.2 validate configuration at startup and exit with a clear message listing every problem (use `errors.Join`).
- **SHOULD** GO-13.3 read configuration from environment variables (twelve-factor) with an optional file for local development; precedence is flags > environment > file > defaults.
- **MUST** GO-13.4 give every setting a safe default or make it required; document both in a `Config` doc comment or `config.example.toml`.
- **MUST NOT** GO-13.5 commit secrets. Read them from the environment, a mounted file or a secret manager, and keep them in typed values that redact themselves in logs.
- **SHOULD** GO-13.6 use `time.Duration` and typed sizes in config (`30s` , `512MiB`), not bare integers with units in the name.

CLIs

- **MUST** GO-13.7 use the standard `flag` package for small tools, and `spf13/cobra` (or `urfave/cli`) only when there are subcommands.
- **MUST** GO-13.8 print usage to stderr, results to stdout, and exit with `0` on success, `1` on failure and `2` on bad usage.
- **SHOULD** GO-13.9 support `--json` output for any command whose output another tool may read.
- **MUST** GO-13.10 include a `version` command or `--version` flag that prints the version, commit and build date injected at build time.

main and run

- **MUST** GO-13.11 keep `main` tiny: it calls `run` , prints the error and sets the exit code. `run` takes its inputs (args, env, stdout) as parameters so it is testable.

```

func main() {
    ctx, stop := signal.NotifyContext(context.Background(), os.Interrupt, syscall.SIGTERM)
    defer stop()

    if err := run(ctx, os.Args[1:], os.Getenv, os.Stdout); err != nil {
        fmt.Fprintf(os.Stderr, "projectd: %v\n", err)
        os.Exit(1)
    }
}

func run(ctx context.Context, args []string, getenv func(string) string, stdout io.Writer) error {
    cfg, err := config.Load(args, getenv)
    if err != nil {
        return fmt.Errorf("load config: %w", err)
    }

    log := platform.NewLogger(stdout, cfg.Log)

    db, err := postgres.Open(ctx, cfg.Database)
    if err != nil {
        return fmt.Errorf("open database: %w", err)
    }
    defer db.Close()

    svc := invoice.NewService(postgres.NewInvoiceStore(db), log)
    srv := httpapi.NewServer(cfg.HTTP, svc, log)

    return platform.Serve(ctx, srv, log)
}

```

14

HTTP services

Routing and handlers

- **MUST** GO-14.1 route with the standard `http.ServeMux` patterns (method + path + wildcards: `"GET /invoices/{id}"`) and read wildcards with `r.PathValue("id")`. **MUST NOT** add a router dependency for routing alone.
- **MUST** GO-14.2 register all routes in one `routes()` function per server so the whole API is visible in one place.
- **MUST** GO-14.3 keep handlers thin: decode and validate input, call one service method, encode the response. Business rules live in the domain package.
- **SHOULD** GO-14.4 write handlers as methods on a server struct that holds dependencies, or as functions returning `http.Handler` that close over them. **MUST NOT** reach for globals.
- **SHOULD** GO-14.5 use a small adapter so handlers return an error and a single place maps errors to responses.

Middleware

- **MUST** GO-14.6 write middleware as `func(http.Handler) http.Handler` and compose it in `routes()` in a visible order: recover → request ID → logging → metrics → auth → handler.
- **MUST** GO-14.7 protect browser-facing state-changing routes with `http.CrossOriginProtection` (Go 1.25+) or an equivalent CSRF defence.

Requests and responses

- **MUST** GO-14.8 bound request bodies with `http.MaxBytesReader` and reject unknown JSON fields on write endpoints.
- **SHOULD** GO-14.9 use `encoding/json/v2` (Go 1.27) in new code: it rejects invalid UTF-8 and duplicate keys by default and unmarshals much faster. Existing `encoding/json` code keeps working and is now backed by v2.
- **MUST** GO-14.10 return errors as RFC 9457 problem details (`application/problem+json`) with the right status code; never leak internal error text on 5xx.
- **MUST** GO-14.11 set `Content-Type` on every response and write the status before the body.

```

func (s *Server) routes() http.Handler {
    mux := http.NewServeMux()
    mux.Handle("GET /invoices/{id}", s.handle(s.getInvoice))
    mux.Handle("POST /invoices/{id}/pay", s.handle(s.payInvoice))
    mux.HandleFunc("GET /healthz", healthz)

    var h http.Handler = mux
    h = s.auth(h)
    h = s.metrics(h)
    h = s.logRequests(h)
    h = requestID(h)
    h = recoverPanics(h, s.log)
    return h
}

func (s *Server) getInvoice(w http.ResponseWriter, r *http.Request) error {
    id, err := invoice.ParseID(r.PathValue("id"))
    if err != nil {
        return badRequest("id must be a UUID")
    }
    inv, err := s.invoices.Get(r.Context(), id)
    if err != nil {
        return err // mapped to 404/500 in s.handle
    }
    return writeJSON(w, http.StatusOK, toInvoiceResponse(inv))
}

```

Servers and clients

- **MUST** GO-14.12 set `ReadHeaderTimeout`, `ReadTimeout`, `WriteTimeout` and `IdleTimeout` on every `http.Server`. **MUST NOT** use `http.ListenAndServe` with the default server in production.
- **MUST** GO-14.13 shut down gracefully: on context cancellation call `srv.Shutdown` with a deadline, then close dependencies.
- **MUST** GO-14.14 reuse one `http.Client` per dependency with `Timeout` set. **MUST NOT** use `http.DefaultClient` or `http.Get` for outbound calls.
- **MUST** GO-14.15 close every response body (`defer resp.Body.Close()`), and check the status code before decoding.
- **SHOULD** GO-14.16 retry only idempotent requests, with exponential backoff and jitter, honouring `Retry-After`.

15

Data access

- **MUST** GO-15.1 use PostgreSQL through `jackc/pgx/v5` (`pgxpool`) for new services; `database/sql` is fine for other databases or libraries that must stay driver-neutral.
- **SHOULD** GO-15.2 write SQL by hand and generate type-safe Go with **sqlc**. **AVOID** ORMs; a query builder (`squirrel`) is allowed only for genuinely dynamic filters.
- **MUST** GO-15.3 pass every value as a bind parameter (`$1`). **MUST NOT** build SQL with `fmt.Sprintf` or string concatenation on input.
- **MUST** GO-15.4 pass `ctx` to every query so cancellation and deadlines reach the database.
- **MUST** GO-15.5 keep SQL types out of the domain: the `postgres` package maps rows to `invoice.Invoice` and storage errors to domain errors.
- **MUST** GO-15.6 run multi-statement writes in a transaction through one helper that commits on success, rolls back on error or panic, and retries serialization failures.
- **MUST** GO-15.7 use UUIDv7 primary keys generated in Go (or with `uuidv7()` in PostgreSQL 18), stored as the native `uuid` type.
- **MUST** GO-15.8 close `rows` and check `rows.Err()`; prefer pgx's `pgx.CollectRows` helpers, which do both.
- **MUST** GO-15.9 configure pool limits (`MaxConns` , `MaxConnLifetime` , `MaxConnIdleTime`) explicitly.
- **MUST** GO-15.10 manage schema with numbered SQL migrations (`goose` or `golang-migrate`) committed in `migrations/`; each runs in a transaction and is never edited after it ships.
- **SHOULD** GO-15.11 test stores against a real PostgreSQL started by `testcontainers-go` , not mocks.

```

func (s *Store) WithTx(ctx context.Context, fn func(q *db.Queries) error) (err error) {
    tx, err := s.pool.BeginTx(ctx, pgx.TxOptions{})
    if err != nil {
        return fmt.Errorf("begin: %w", err)
    }
    defer func() {
        if err != nil {
            err = errors.Join(err, tx.Rollback(ctx))
        }
    }()

    if err := fn(s.queries.WithTx(tx)); err != nil {
        return err
    }

    return tx.Commit(ctx)
}

```

16

Agents and daemons

Many Go programs here are long-running agents: they sit on a host, poll hardware or the network, and report to a control plane. They run unattended for months, so they get extra rules.

Lifecycle

- **MUST** GO-16.1 run every component as `Run(ctx) error` that returns when `ctx` is cancelled, and start them together with an `errgroup` so one failure stops the rest cleanly.
- **MUST** GO-16.2 handle `SIGTERM` and `SIGINT` by cancelling the root context, then give in-flight work a bounded grace period (for example 15 seconds) before exiting.
- **SHOULD** GO-16.3 reload configuration on `SIGHUP` without restarting, when the agent supports it.
- **MUST** GO-16.4 run under a supervisor (systemd with `Restart=on-failure`, a container runtime, or a Windows service) and **MUST NOT** daemonize itself.
- **SHOULD** GO-16.5 notify systemd of readiness and watchdog pings (`sd_notify`) when deployed with `Type=notify`.

Loops, retries and the control plane

- **MUST** GO-16.6 drive periodic work with `time.Ticker` (stopped with `defer t.Stop()`) inside a `select` on `ctx.Done()`.
- **MUST** GO-16.7 retry failed connections with capped exponential backoff and jitter; **MUST NOT** spin in a tight retry loop.
- **MUST** GO-16.8 buffer reports locally (bounded queue or on-disk spool) while the control plane is unreachable, and drop the oldest data with a counter when the buffer fills.
- **MUST** GO-16.9 authenticate to the control plane with mTLS or a per-agent token, and pin the control plane's CA.
- **MUST** GO-16.10 keep resource use bounded and visible: cap goroutines, buffers and file handles, and report memory, goroutine count and queue depth as metrics.

Updates and identity

- **MUST** GO-16.11 verify the signature (or checksum from a signed manifest) of any self-update before swapping binaries, and keep the previous binary for rollback.

- **MUST** GO-16.12 generate a stable agent ID (UUIDv7) on first start and persist it; never derive identity from hostname or IP alone.
- **SHOULD** GO-16.13 report version, commit, OS, architecture and uptime with every heartbeat.

```
func (a *Agent) Run(ctx context.Context) error {
    ticker := time.NewTicker(a.interval)
    defer ticker.Stop()

    for {
        select {
        case <-ctx.Done():
            return a.flush(context.WithoutCancel(ctx))
        case <-ticker.C:
            if err := a.collectAndReport(ctx); err != nil {
                a.log.WarnContext(ctx, "report failed", "err", err, "queued", a.queue.Len())
            }
        }
    }
}
```

17

Performance and memory

- **MUST** GO-17.1 measure before optimizing: a benchmark (`b.Loop()`) or a profile (`pprof` CPU, heap, mutex, block) must show the problem, and the pull request includes before and after numbers from `benchstat` .
- **SHOULD** GO-17.2 build release binaries with profile-guided optimization (`default.pgo` from a production CPU profile) for hot services.
- **SHOULD** GO-17.3 preallocate slices and maps when the size is known: `make([]Line, 0, len(rows))` .
- **SHOULD** GO-17.4 use `strings.Builder` / `bytes.Buffer` for building strings in loops, and `strconv` over `fmt.Sprintf` on hot paths.
- **SHOULD** GO-17.5 stream large payloads (`io.Reader` , `json.NewDecoder` , iterators) instead of loading them fully into memory.
- **MAY** GO-17.6 use `sync.Pool` for frequently allocated, short-lived buffers; **MUST** reset objects before returning them and never pool objects that hold references to request data.
- **SHOULD** GO-17.7 set `GOMEMLIMIT` in containers (around 90% of the memory limit) and leave `GOGC` at its default unless a profile says otherwise. Go 1.26+'s Green Tea collector already lowers GC cost.
- **MUST NOT** GO-17.8 use `unsafe` or `reflect` for speed without a benchmark, a comment explaining the invariant, and a test that guards it.
- **AVOID** GO-17.9 premature concurrency: a goroutine per item is often slower than a simple loop for small, CPU-cheap work.
- **SHOULD** GO-17.10 keep struct fields ordered for readability first; reorder for alignment only in types allocated millions of times, with a comment.

18

Security

- **MUST** GO-18.1 run `govulncheck` in CI and fix reachable vulnerabilities before release; keep dependencies current with Renovate or Dependabot.
- **MUST** GO-18.2 validate all input at the boundary (lengths, ranges, formats, enums) before it reaches the domain.
- **MUST** GO-18.3 use bind parameters for SQL (§15) and `exec.CommandContext` with an argument list — never a shell string — for subprocesses.
- **MUST** GO-18.4 open user-influenced paths through `os.Root` (Go 1.24+) or `filepath.IsLocal` checks to prevent path traversal.
- **MUST** GO-18.5 use `crypto/rand` for tokens, keys and IDs; `math/rand/v2` is for simulations and jitter only.
- **MUST** GO-18.6 compare secrets with `crypto/subtle.ConstantTimeCompare` or `hmac.Equal`.
- **MUST** GO-18.7 hash passwords with `argon2id` or `bcrypt` (`golang.org/x/crypto`), never with a fast hash.
- **MUST** GO-18.8 use TLS 1.3 by default with the standard library's defaults (post-quantum hybrid key exchange is on by default since Go 1.26). **MUST NOT** set `InsecureSkipVerify` outside tests.
- **MUST** GO-18.9 use mTLS for service-to-service and agent-to-control-plane traffic on untrusted networks.
- **MUST** GO-18.10 set HTTP security headers on browser-facing services and protect state-changing routes from cross-origin requests (§14).
- **MUST NOT** GO-18.11 log or return secrets, stack traces or internal errors to clients.
- **SHOULD** GO-18.12 run services as a non-root user in a minimal image (`distroless/static` or `scratch`) with a read-only root filesystem.
- **SHOULD** GO-18.13 enable FIPS 140-3 mode (`GOFIPS140`) only when a customer requires it, and test the build with it.

19

Testing

Structure

- **MUST** GO-19.1 keep tests beside the code in `<name>_test.go`, and shared fixtures in `testdata/`.
- **MUST** GO-19.2 use the standard `testing` package. **MAY** use `go-cmp` (`cmp.Diff`) for comparing structs; **AVOID** assertion frameworks that hide the failing value.
- **MUST** GO-19.3 write table-driven tests with named subtests (`t.Run(tc.name, ...)`) when a function has more than two interesting cases.
- **MUST** GO-19.4 write failure messages as `Func(input) = got, want want : t.Errorf("Parse(%q) = %v, want %v", in, got, want)`.
- **MUST** GO-19.5 call `t.Helper()` in test helpers, use `t.Cleanup` for teardown, `t.TempDir()` for files and `t.Context()` (Go 1.24+) for contexts.
- **SHOULD** GO-19.6 mark independent tests `t.Parallel()`.
- **MUST** GO-19.7 run the suite with `go test -race -shuffle=on ./...` in CI.

Time, concurrency and I/O

- **MUST** GO-19.8 test time-dependent and concurrent code with `testing/synctest`: inside `synctest.Test`, time is virtual and `synctest.Wait` / `synctest.Sleep` (Go 1.27) make goroutine scheduling deterministic. **MUST NOT** use real `time.Sleep` in tests.
- **SHOULD** GO-19.9 use `httptest.NewServer` (or `httptest.NewTestServer` with `synctest`, Go 1.27) for HTTP clients, and `httptest.NewRecorder` for handlers.
- **SHOULD** GO-19.10 test stores against real PostgreSQL via `testcontainers-go`, gated by a build tag or `-short` so unit runs stay fast.
- **SHOULD** GO-19.11 prefer hand-written fakes (an in-memory `Store`) over generated mocks.

Fuzzing, benchmarks and golden files

- **SHOULD** GO-19.12 add fuzz tests (`func FuzzParse(f *testing.F)`) for parsers, decoders and anything that reads untrusted input, and commit the crashers they find to `testdata/fuzz`.
- **SHOULD** GO-19.13 write benchmarks with `for b.Loop() { ... }` and compare runs with `benchstat`.
- **MAY** GO-19.14 use golden files for large outputs, regenerated with a `-update` flag; save debug output in `t.ArtifactDir()` (Go 1.26).

```

func TestParseAmount(t *testing.T) {
    tests := []struct {
        name    string
        in      string
        want    Cents
        wantErr bool
    }{
        {name: "whole", in: "12", want: 1200},
        {name: "cents", in: "12.34", want: 1234},
        {name: "negative", in: "-1", wantErr: true},
        {name: "empty", in: "", wantErr: true},
    }
    for _, tc := range tests {
        t.Run(tc.name, func(t *testing.T) {
            t.Parallel()
            got, err := ParseAmount(tc.in)
            if (err != nil) != tc.wantErr {
                t.Fatalf("ParseAmount(%q) error = %v, wantErr %v", tc.in, err, tc.wantErr)
            }
            if got != tc.want {
                t.Errorf("ParseAmount(%q) = %d, want %d", tc.in, got, tc.want)
            }
        })
    }
}

func TestAgentReportsEveryInterval(t *testing.T) {
    synctest.Test(t, func(t *testing.T) {
        sink := &fakeSink{}
        a := NewAgent(sink, 30*time.Second)
        ctx, cancel := context.WithCancel(t.Context())
        go a.Run(ctx)

        synctest.Sleep(95 * time.Second)
        if got := sink.Count(); got != 3 {
            t.Errorf("reports after 95s = %d, want 3", got)
        }
        cancel()
        synctest.Wait()
    })
}

```

20

Documentation and comments

- **MUST** GO-20.1 give every package a package comment (`// Package invoice ...`) in one file, usually `doc.go` for larger packages.
- **MUST** GO-20.2 document every exported identifier with a full sentence that starts with its name: `// NewService returns a Service backed by store.`
- **SHOULD** GO-20.3 document what is not obvious from the signature: concurrency safety, ownership of returned slices, units, zero-value behaviour and the errors a caller can expect.
- **MUST** GO-20.4 write comments that explain **why**, not what the next line does.
- **MUST** GO-20.5 use the doc-comment syntax `go doc` renders: `[Service.MarkPaid]` links, `# Headings` , indented code blocks and lists.
- **SHOULD** GO-20.6 add runnable `Example` functions for public APIs; they are compiled and checked by `go test` .
- **MUST** GO-20.7 format TODOs as `// TODO(owner): what and why` , with an issue link for anything longer than a day.
- **MUST** GO-20.8 mark deprecated APIs with a `// Deprecated: use X instead.` paragraph.
- **MUST** GO-20.9 keep a `README.md` per repository with: what it is, how to run it locally (one command), how to test, configuration and how it is deployed.
- **SHOULD** GO-20.10 record significant design decisions as short ADRs in `docs/adr/` .

Git, commits, releases and review

Commits

- **MUST** GO-21.1 follow **Conventional Commits**: `type(scope): summary` in the imperative, lower case, no trailing period, under 72 characters.
- Types: `feat`, `fix`, `refactor`, `perf`, `test`, `docs`, `build`, `ci`, `chore`, `style`, `revert`. Scope is the package or area: `fix(agent): back off when the control plane is down`.
- **MUST** GO-21.2 mark breaking changes with `!` and a **BREAKING CHANGE:** footer.
- **SHOULD** GO-21.3 explain the *why* in the body when it is not obvious from the summary.
- **MUST** GO-21.4 keep every commit building and passing tests; squash fix-ups before merging.

Branches and pull requests

- **MUST** GO-21.5 branch from `main` with a short typed name: `feat/agent-spool`, `fix/invoice-rounding`.
- **SHOULD** GO-21.6 keep pull requests under about 400 changed lines, excluding generated code.
- **MUST** GO-21.7 pass `task lint` and `task test` before review, and keep `go.mod` tidy (`go mod tidy` produces no diff).
- **MUST** GO-21.8 include tests and migrations in the same pull request as the change they support.

Releases

- **MUST** GO-21.9 tag releases with semantic versions (`v1.4.0`) and generate the changelog from Conventional Commits.
- **MUST** GO-21.10 inject version information at build time: `-ldflags "-X example.com/project/internal/version.Version={{.Version}} -X ...Commit={{.Commit}}"`, and fall back to `debug.ReadBuildInfo()` for `go install` builds.
- **SHOULD** GO-21.11 build with GoReleaser: reproducible (`-trimpath`, `CGO_ENABLED=0`), multi-platform, checksummed and signed, with container images and an SBOM.

Review checklist

- Is every error checked, wrapped with context and handled once?
- Does every blocking call take a context with a deadline?
- Does every goroutine have a clear owner and exit path, and is it waited for?

- Are interfaces small and defined by the consumer?
- Is the exported surface as small as it can be, and documented?
- Are inputs validated at the boundary and SQL parameterised?
- Are there table tests for the new behaviour, run with `-race`?

A

Appendix

A. `.golangci.yml`

```
version: "2"

run:
  timeout: 5m

linters:
  default: standard
  enable:
    - bodyclose
    - contextcheck
    - copyloopvar
    - errorlint
    - exhaustive
    - gocritic
    - gosec
    - misspell
    - nilerr
    - noctx
    - nolintlint
    - perfsprint
    - revive
    - rowserrcheck
    - sloglint
    - sqlclosecheck
    - unconvert
    - unparam
    - usestdlibvars
    - wrapcheck
  settings:
    nolintlint:
      require-explanation: true
      require-specific: true
    sloglint:
      kv-only: true
      context: scope
      key-naming-case: snake
    exhaustive:
      default-signifies-exhaustive: true

formatters:
  enable:
    - gofumpt
    - goimports
  settings:
    goimports:
```

```
local-prefixes:  
  - example.com/project
```

B. Taskfile.yml

```
version: "3"  
  
tasks:  
  fmt:  
    cmds:  
      - go tool golangci-lint fmt  
  lint:  
    cmds:  
      - go vet ./...  
      - go tool golangci-lint run  
      - go tool govulncheck ./...  
  test:  
    cmds:  
      - go test -race -shuffle=on -count=1 ./...  
  test:integration:  
    cmds:  
      - go test -race -tags=integration ./...  
  generate:  
    cmds:  
      - go generate ./...  
      - go tool sqlc generate  
  modernize:  
    cmds:  
      - go fix ./...  
  build:  
    cmds:  
      - CGO_ENABLED=0 go build -trimpath -o bin/ ./cmd/...
```

C. go.mod header

```
module example.com/project  
  
go 1.27.0  
  
toolchain go1.27.0  
  
tool (  
  github.com/golangci/golangci-lint/v2/cmd/golangci-lint  
  golang.org/x/tools/cmd/stringer  
  golang.org/x/vuln/cmd/govulncheck  
  github.com/sqlc-dev/sqlc/cmd/sqlc  
)
```

D. New package checklist

- ☐ Name is short, lower case, says what it provides, and does not stutter at call sites.
- ☐ Package comment and doc comments on every exported identifier.
- ☐ Dependencies passed in through a constructor; no globals or `init()` I/O.
- ☐ Interfaces it needs are declared here and kept small.
- ☐ Every I/O function takes `ctx` first and returns `error` last.
- ☐ Errors wrapped with context; sentinel or typed errors for conditions callers branch on.
- ☐ Tests beside the code, table-driven, passing with `-race`.

E. Sources

- [Go 1.27 release notes](#)
- [Go 1.26 release notes](#)
- [Effective Go](#)
- [Go Code Review Comments](#)
- [Google Go Style Guide](#)