

Cheap Rigor

*On Go, agents, and doing the right
things from commit one*

Redowan Delowar
Software Engineer
Wolt

Context – industrial programming

Peter Bourgon defined it as programming

- in a startup or corporate environment
- within a team where engineers come and go
- on code that outlives any single engineer
- and, serving highly mutable business requirements

How much of this do you actually set up from the get go?

- Config management
- Working test harness
- CI, linters, & formatters
- Circuit breakers, rate limiting, & other resilience patterns
- Structured logging, metrics, & tracing
- Continuous profiling
- A style guide everyone follows

We skip some of the steps because ...

- Team of two - both backend & on-call
- Feature due last sprint
- Never had bandwidth for fuzz testing
- Hitting the ground with a few unit tests
- Bare-minimum observability
- Project might not survive the quarter

What changed?

LLM agents reduced the cost of doing the right thing enough that it now fits inside those constraints.

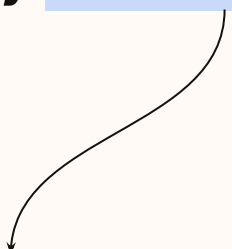
“Shipping worse code with agents is a choice. We can choose to ship code that is better instead.”

– Simon Willison

Code generation is cheap, but validation isn't

The bottleneck moved to **validation & judgement**: what to build, what to keep, what to throw away.

*LLMs often produce working code that is utterly **tasteless***



“Any sin is acceptable provided it is carried off in good **taste**.”

– Evelyn Waugh, A Handful of Dust

So why are we here?

~20 things you could do with your agents to ensure rigor in your output

1. DESIGN → Own the architecture
2. WRITE → Write code in a controlled environment
3. TEST → Prove every change in a tight loop
4. OBSERVE → Instrument before you need it
5. HARDEN → Survive failure
6. ENFORCE → Guard the codebase from entropy

01

Design

Tip 1 – You own the design

“If the AI writes something I do not understand, I stop and study it. I try to recreate it myself. If I cannot reach the same diff, I need to learn more.”

– Mitchell Hashimoto

Tip 1 – You own the design (cont’)

- Collect the PRD
- Write the RFC & think about the problem space
- Spend more time in planning
- Handroll the initial app structure
- Write the types, structs, interfaces, the constructor signature. Let the agent fill in the body.

Tip 2 – Functional core, imperative shell (Gary Bernhardt)

```
// Core: pure, testable, no I/O
func ApplyDiscount(price Money, tier string) (Money, error) {
    // just business logic
}

// Shell: HTTP, DB, side effects
func HandleCheckout(w http.ResponseWriter, r *http.Request) {
    order := decode(r)
    total, err := ApplyDiscount(order.Price, order.Tier)
    save(w, total, err)
}
```

Pure functions for decisions. Side effects at the edges. The core is trivial to test without mocks.

Tip 3 – Define interfaces at the consumer

```
// In your package, not the producer's  
type Uploader interface {  
    PutObject(  
        ctx context.Context,  
        in *s3.PutObjectInput,  
        ...func(*s3.Options)) (*s3.PutObjectOutput, error)  
}
```

The producer returns a concrete type. On the consumer side, you write an interface for the methods you need.

Tip 4 – Experiment with Domain Driven Design (DDD)

- DDD has lots of good ideas - decouple business logic from persistence, make your app testable and extensible
- The cost was always verbosity: value objects, repositories, domain services, aggregate roots
- Let the agent generate the wiring; you keep the architecture

02

Write

Tip 5 – Explicit code over clever code

“The most important thing that you can do when working with an LLM is give it a way to quickly evaluate if it's doing the right thing, and if it isn't, point it in the right direction.”

– Steve Klabnik

LLMs write the verbose version as fast as the clever one. Pick the version that's easier to review. Go is inherently less confusing to the agents.

Tip 6 – You don't need a DI framework

```
func main() {  
    cfg := NewConfig()  
  
    db    := NewDB(cfg.DSN)  
    repo  := NewRepo(db)  
    flags := NewFlagClient(cfg.FlagURL)  
  
    svc := NewService(repo, flags, cfg.APIKey)  
    srv := NewServer(svc, cfg.ListenAddr)  
  
    srv.Run()  
}
```

The call order is the dependency graph. If a constructor signature changes, the compiler points straight at every broken call site.

Tip 7 – Wrap errors at the boundary

```
// In the service layer  
if err != nil {  
    return fmt.Errorf("creating user %q: %w", name, err)  
}
```

```
// In the HTTP handler  
if errors.Is(err, usersvc.ErrDuplicate) {  
    http.Error(w, "user exists", http.StatusConflict)  
    return  
}
```

Wrap with %w when callers inspect the chain. At system boundaries, translate errors. Don't leak sentinels across services.

Tip 8 – Handle context properly from the get-go

- Retrofitting context is painful
- Define your **timeouts**, **graceful shutdown**, and **cancellation behavior** early
- Point the agent at a canonical style guide or blog post for the pattern
- Let it carry out the propagation plumbing across every handler and client call
- Validate aggressively

03

Test

Tip 9 – Red/green TDD

“After being a career-long skeptic of the test-first, maximum test coverage school of software development... I've recently come around to test-first processes as a way to force agents to write only the code that's necessary to solve the problem at hand.”

– Simon Willison, "Introducing Showboat and Rodney"

write failing test ~> implement until green ~> refactor ~> commit

Tip 10 – Fakes over mocks (or you don't need a mocking library)

```
type FakeDB struct{ users map[string]bool }

func NewFakeDB() *FakeDB {
    return &FakeDB{users: map[string]bool{}}
}

func (f *FakeDB) InsertUser(name string) error {
    if f.users[name] {
        return usersvc.ErrDuplicate
    }
    f.users[name] = true
    return nil
}
```

Handwritten fakes over gomock and mockery. Writing fakes can be tedious - let the agent do it.

Tip 11 – Throw the kitchen sink at test quality

- table-driven tests for exhaustive case coverage
- golden files for snapshot testing
- subtests with `t.Run()` for organization and selective execution
- `t.Parallel()` everywhere to catch shared-state bugs early
- `-race` on every test run
- `go test -fuzz` for boundary cases you and the LLM both missed
- integration tests behind build tags so CI stays fast

Tip 12 – Read what you merge

“I still treat every line as my responsibility, judged as if I wrote it myself. AI doesn't change that.”

- Armin Ronacher

04

Observe

Tip 13 – log/slog from line one

```
logger := slog.New(slog.NewJSONHandler(os.Stdout, nil))  
slog.SetDefault(logger)
```

```
slog.Info("server starting", "addr", addr, "version", version)
```

`JSONHandler` in production, `TextHandler` in development.
Standard library since Go 1.21.

Tip 14 – Traces, metrics, and profile

- OTEL spans on every inbound and outbound call - the wiring is mechanical, let the agent do it
- Propagate trace context through HTTP, gRPC, and message queues
- Expose pprof (`import _ "net/http/pprof"`) - three lines for CPU, heap, goroutine, and mutex profiles
- Wire up Prometheus metrics or OTEL metrics at service boundaries
- You decide what to observe; the agent carries out the plumbing

Tip 14 – Flight recorder (Go 1.25)

```
fr := trace.NewFlightRecorder(trace.FlightRecorderConfig{
    MinAge: 10 * time.Second,
})
if err := fr.Start(); err != nil {
    log.Fatal(err)
}
defer fr.Stop()

// On anomaly
f, _ := os.Create("snapshot.trace")
fr.WriteTo(f)
f.Close()
```

Always-on circular trace buffer. Snapshot on anomaly - makes chasing tail latency quite fun.

05

Harden

Tip 15 – Retry, circuit breaker, & rate limiting

// 1. Define Policies in failsafe-go

```
retryPolicy := retrypolicy.Builder[any]().  
    WithBackoff(1*time.Second, 30*time.Second).  
    WithMaxRetries(5).  
    Build()
```

```
breaker := circuitbreaker.Builder[any]().  
    WithFailureThreshold(3).  
    WithDelay(10 * time.Second).  
    Build()
```

```
limiter := ratelimiter.SmoothBuilder[any](100, time.Second).  
    WithMaxWaitTime(500 * time.Millisecond).  
    Build()
```

Tip 16 – Prevent cache stampede with singleflight

```
var g singleflight.Group // x/sync

func GetUser(ctx context.Context, id string) (*User, error) {
    v, err, _ := g.Do(id, func() (any, error) {
        return db.QueryUser(ctx, id)
    })
    return v.(*User), err
}
```

When a cache key expires, one goroutine fetches. The rest wait and share the result. No stampede.

Tip 17 – Controlled concurrency

```
var g singleflight.Group // x/sync

func GetUser(ctx context.Context, id string) (*User, error) {
    v, err, _ := g.Do(id, func() (any, error) {
        return db.QueryUser(ctx, id)
    })
    return v.(*User), err
}
```

When a cache key expires, one goroutine fetches. The rest wait and share the result. No stampede.

06

Enforce

Tip 18 – Static checks in every loop

- `go vet`: catches shadow variables, unreachable code, wrong `printf` verbs
- `staticcheck`: nil-pointer paths, deprecated API usage, unused results from error-returning functions
- `golanci-lint`: umbrella runner

Write the configs with an agent & then ask it to run the checks on every iteration alongside the unit tests.

Tip 19 – Point the agent at a styleguide

Write a skill or AGENT.md rule that checks conformity with a good style guide – Google, Uber, or your own.

Then tell the agent to follow it rigorously when it generates & reviews code.

Tip 20 – Adopt new Go idioms as they ship

Manual loop checking → `slices.Contains(roles, r)`
membership

`errors.As(err, &target)` → `errors.AsType[T](err)` (Go 1.25)

hand-rolled HTTP mux → Go 1.22+ mux

Use JetBrains' Modern Go Guidelines skill + Go 1.26's `go fix`

Tip 21 – The refactoring loop

“Evaluate the result in a Pull Request. If it's good, land it. If it's almost there, prompt it and tell it what to do differently. If it's bad, throw it away.”

– Simon Willison, "Agentic Engineering Patterns"

How much of this do you actually set up?

“The LLM didn't replace the rigor. It made the rigorous version feasible.”

– Oxide and Friends, "Engineering Rigor in the LLM Age"

All of it.

References

[Simon Willison — “Agentic Engineering Patterns”](#)

[Simon Willison — “Introducing Showboat and Rodney”](#)

[Mitchell Hashimoto — AI-Assisted Development \(SpecStory\)](#)

[Gary Bernhardt — “Functional Core, Imperative Shell”](#)

[Steve Klabnik — “The most important thing when working with LLMs”](#)

[Armin Ronacher — “90%”](#)

[Oxide and Friends — “Engineering Rigor in the LLM Age”](#)

[Peter Bourgon — “Go for Industrial Programming”](#)

[JetBrains — “Write Modern Go Code with Junie and Claude Code”](#)

[Google Go Style Guide](#)

[Uber Go Style Guide](#)

[Go Blog — “Flight Recorder in Go 1.25”](#)

[OpenTelemetry Go Instrumentation](#)

[Grafana Pyroscope](#)

[failsafe-go](#)

[golang.org/x/sync/singleflight](#)

Thank You!