Go Module Architecture Patterns for ai8future

Service Chassis, Golden Path, and Library-First Design

ai8future Engineering

2026-01-28

Table of contents

Go Module Architecture Patterns for ai8future

Executive Summary

This report captures the architectural patterns, assessment findings, and strategic roadmap for transforming the ai8future codebase ecosystem from service-first to library-first design. The core insight: **every service should be a thin wrapper around an importable library**.

Two existing codebases already exemplify this standard:

- **pricing_db** (Go) — the gold-standard reference implementation featuring a dual API, lazy singleton, embedded configuration, thread safety, and zero external dependencies.

- **encryptedcol** (Go) — a pure library from inception with functional options, interface-based extensibility, and type-safe generics.

Three additional codebases — serp_svc (70% ready), pdfgen_svc (60%), and markdown_svc (60%) — require structured conversion. Beyond conversion, we propose a **service chassis** (shared infrastructure libraries), a **golden path** workflow (scaffolding + linting), and an **automated converter tool** to scale these patterns across the organization.

---

1. The pricing_db Reference Pattern

`pricing_db` is the gold standard for Go modules that function as both CLI tools and importable libraries. Every new Go module should follow its patterns.

1.1 Dual API Design

Provide **both** convenience functions **and** explicit instance management:

```
// CONVENIENCE API: Package-level functions (lazy singleton)
cost := pricing_db.CalculateCost("gpt-4o", 1000, 500)

// CONTROL API: Explicit instance creation
pricer, err := pricing_db.NewPricer()
if err != nil {
    log.Fatal(err)
}
details := pricer.Calculate("gpt-4o", 1000, 500)
```

Benefits:

- Beginners get a simple, quick-start API
- Advanced users get full lifecycle control
- No global state pollution
- Fully testable and mockable

1.2 Lazy-Initialized Singleton

Thread-safe initialization via `sync.Once` — happens exactly once, only when first used:

```
var (
    defaultInstance *Pricer
    initOnce        sync.Once
    initErr         error
)

func ensureInitialized() {
    initOnce.Do(func() {
        defaultInstance, initErr = NewPricer()
    })
}
```

```
func CalculateCost(model string, input, output int) float64 {
    ensureInitialized()
    if initErr != nil {
        return 0.0
    }
    return defaultInstance.Calculate(model, int64(input),
int64(output)).TotalCost
}
```

1.3 Thread Safety by Default

Protect all shared state with `RWMutex`. Read-heavy operations use `RLock` for concurrency; write operations get exclusive locks:

```
type Pricer struct {
    models map[string]ModelPricing
    mu     sync.RWMutex
}

func (p *Pricer) Calculate(model string, input, output int64) Cost {
    p.mu.RLock()
    defer p.mu.RUnlock()
    pricing, ok := p.models[model]
    if !ok {
        return Cost{Error: "model not found"}
    }
    return Cost{
        TotalCost: float64(input)*pricing.InputCost +
float64(output)*pricing.OutputCost,
    }
}
```

1.4 Embedded Configuration

Use `go:embed` to bundle data files. Provide both default FS and custom FS entry points for testing/extensibility:

```
//go:embed configs/*.json
var ConfigFS embed.FS

func NewPricer() (*Pricer, error) {
    return NewPricerFromFS(ConfigFS, "configs")
}

func NewPricerFromFS(fs fs.FS, dir string) (*Pricer, error) {
    // Load models from filesystem
}
```

1.5 Thin CLI Wrapper

The CLI imports and delegates to the library. **All** logic lives in the library:

CLI Responsibilities (minimal): Parse flags, read input, call library, format output, handle exit codes.

Library Responsibilities (everything else): Business logic, data validation, computation, error handling, type definitions.

1.6 Zero External Dependencies

The `go.mod` has no external dependencies — standard library only. This makes the module trivially importable and produces a single portable binary.

1.7 Gold-Standard Directory Structure

```
pricing_db/
├── go.mod                    # Module definition
├── README.md                 # Usage documentation
├── VERSION                   # Semantic version
├── pricer.go                 # Main Pricer struct + methods
├── types.go                  # All type definitions
├── helpers.go                # Package-level convenience functions
├── embed.go                  # Embedded config filesystem
├── gemini.go                 # Specific calculators
├── cmd/pricing-cli/
│   └── main.go               # Thin CLI wrapper
├── configs/                  # Embedded JSON data files
│   ├── openai.json
│   ├── anthropic.json
│   └── google.json
└── *_test.go                 # Unit + example tests
```

---

2. Codebase Assessment Matrix

Overview

Codebase	Language	Current State	Readiness	Effort	Priority
pricing_db	Go	CLI + Module	100%	N/A (reference)	N/A
encrypted-col	Go	Pure Library	100%	None needed	N/A
serp_svc	Go	Service-First	70%	Medium (2-3 weeks)	High

Codebase	Language	Current State	Readiness	Effort	Priority
pdfgen_svc	Python	Service-Only	60%	Medium (2-3 weeks)	Medium
markdown_svc	TypeScript	Service-Only	60%	Medium (2-3 weeks)	Medium

pricing_db (100% — Reference Implementation)
No changes needed. Exemplary dual API, thread safety, zero dependencies, embedded configuration, thin CLI wrapper, excellent documentation, and example tests. **Use as template for all new Go modules.**

encryptedcol (100% — Pure Library)
Production-ready from day one. Key patterns to replicate elsewhere:

- **Functional options pattern:** `WithKey()`, `WithDefaultKeyID()`, `WithCompressionThreshold()`
- **Interface-based extensibility:** `KeyProvider` interface allows Vault/KMS integration
- **Type-safe generics:** `SealJSON[T]()`, `OpenJSON[T]()`
- **Single package namespace:** Clean imports

```
cipher, err := encryptedcol.New(
    encryptedcol.WithKey("v1", key),
    encryptedcol.WithDefaultKeyID("v1"),
    encryptedcol.WithCompressionThreshold(1024),
)
```

serp_svc (70% — High Priority)
What's already good: Core logic isolated in `internal/service/serper.go`, query builders in `internal/query/dorks.go`, handlers cleanly separated, proper Go modules, VERSION file, good Makefile.

What needs work:

1. Extract `SerperClient` to `pkg/serper/` or separate `serper-go` module
2. Create `SearchClient` interface for mocking and implementation swapping
3. Add optional CLI wrapper

Conversion timeline: ~1 week extract, ~1 week interfaces, ~1 week CLI/tests/docs.

pdfgen_svc (60% — Medium Priority)
Good: Core logic in `src/core/pdfgen.py`, gRPC server just wraps core. **Needs:** Package as proper Python module with `pyproject.toml`, add dual API (convenience + advanced), keep service as separate importer.

markdown_svc (60% — Medium Priority)
Good: Processors in `src/processors/`, clear service boundaries. **Needs:** Package processors as `@ai8future/markdown-processor` npm module, add dual API, export clean TypeScript types.

---

3. Service-to-Module Conversion Patterns

The Four-Step Process

Step 1: Identify Core Logic

Extract: Code with no framework dependencies that performs actual business operations. **Keep in service:** HTTP/gRPC handlers, server initialization, middleware, request/response translation.

Step 2: Extract to Public Package

Option A — `pkg/` within same repo (simpler):

```
// internal/service/client.go → pkg/serper/client.go
package serper

type Client struct { ... }
func (c *Client) Search(ctx context.Context, req *SearchRequest)
(*SearchResponse, error) { ... }
```

Option B — Separate module (better long-term):

```
// github.com/ai8future/serper-go
package serpergo

type Client struct { ... }
```

Step 3: Create Interface Abstraction

```
type SearchClient interface {
    Search(ctx context.Context, req *SearchRequest) (*SearchResponse, error)
    Images(ctx context.Context, req *SearchRequest) (*ImagesResponse, error)
    News(ctx context.Context, req *SearchRequest) (*NewsResponse, error)
}

// Compile-time check
var _ SearchClient = (*SerperClient)(nil)
```

Step 4: Service Uses Library

The handler becomes a pure translation layer. Business logic is testable without HTTP. The library is reusable in CLI tools, batch jobs, and other services.

```go
func main() {
    client := serper.NewClient(os.Getenv("SERPER_API_KEY"),
        serper.WithTimeout(10*time.Second),
    )
    searchHandler := handler.NewHTTPHandler(client)
    http.HandleFunc("/search", searchHandler.Search)
    http.ListenAndServe(":8080", nil)
}
```

---

4. Layered Module Dependencies

For complex domains (crawler + proxy + storage), use layered modules with clear dependency boundaries.

```
┌─────────────────────────────────────┐
│   Your Application                  │  ← Uses crawler
├─────────────────────────────────────┤
│       crawler module                │  ← Core orchestration
├──────────────┬──────────────────────┤
│  proxypool   │   seaweedfs          │  ← Infrastructure modules
│   module     │    module            │
└──────────────┴──────────────────────┘
```

Hard Dependencies vs Interfaces

Use Hard Dependency	Use Interface
Internal services communicating	Public library meant for reuse
You control both modules	Users might want different implementations
No need to swap implementations	Testing requires mocking
Performance-critical path	Flexibility more important than speed
Prototype/MVP phase	Production-ready library

Hard dependency — simpler code, no abstraction overhead, but tightly coupled. Good for internal modules where you control both sides.

Interface-based — more flexible, testable with mocks, swappable implementations. Good for public libraries and modules with external dependencies.

```go
// Interface approach — crawler doesn't know about SeaweedFS
type Storage interface {
    Store(key string, data []byte) error
    Retrieve(key string) ([]byte, error)
```

```
}

type Crawler struct {
    storage Storage
}

// Application wires concrete implementations
func main() {
    storage := seaweedfs.NewClient("localhost:9333")
    c := crawler.NewCrawler(storage)
}
```

---

5. Future Module Candidates

Gemini Wrapper (Fit Score: 9/10)
Encapsulates: Authentication, retry logic, rate limiting, error translation, API versioning, token counting (integrates with pricing_db), cost tracking, streaming.

```
// Simple API
result, err := client.Generate(ctx, "Explain quantum computing")

// Advanced API
resp, err := client.GenerateWithOptions(ctx, &GenerateRequest{...})
```

Priority: High | **Effort:** 2-3 weeks

Crawl + Proxy Manager (Fit Score: 8/10)
Recommendation: Split into two modules for maximum reusability.

- **proxypool** — Proxy rotation, health checking, provider management
- **crawler** — Orchestration using `Storage` and `ProxyPool` interfaces

Priority: Medium | **Effort:** 3-4 weeks total

SeaweedFS Storage (Fit Score: 9/10)
Provides: Connection pooling, retry logic, circuit breaker, health checks, master/volume server management. Implements a standard `Store` interface, allowing future swap to S3.

Priority: High | **Effort:** 2 weeks

Priority Matrix

Module	Business Value	Reusability	Complexity	Priority
Gemini Wrapper	High	Medium	Medium	High
SeaweedFS Storage	High	High	Low	High

Module	Business Value	Reusability	Complexity	Priority
Proxy Pool	Medium	High	Medium	Medium
Crawler	Medium	Medium	High	Medium

---

6. Service Chassis Design

The Problem

Without standardization across 10+ codebases: 10 different ways to handle configuration, logging, health checks, Dockerfiles, and onboarding friction every time you touch a different repo.

Observed Patterns

Aspect	Current Pattern
Languages	Go (primary), Python, TypeScript
Protocols	gRPC + HTTP dual-protocol
Observability	Prometheus metrics, structured JSON logging
Reliability	Circuit breakers, graceful shutdown, health checks
Containers	Multi-stage Dockerfiles
Orchestration	Kubernetes manifests
API Contracts	Protocol Buffers with buf tooling

Architecture: Scaffolding + Shared Libraries

```
github.com/ai8future/
├── chassis-go/              # Go shared infrastructure
│   ├── config/              # Env-based config with validation
│   ├── server/              # HTTP + gRPC lifecycle
│   ├── middleware/           # Logging, metrics, recovery
│   ├── health/              # Composable health checks
│   ├── lifecycle/           # Graceful shutdown orchestration
│   ├── errors/              # HTTP ↔ gRPC status mapping
│   └── client/              # Retry + circuit breaker helpers
├── chassis-py/              # Python equivalent
├── chassis-ts/              # TypeScript equivalent
├── templates/               # Scaffolding templates
│   ├── go-service/
│   ├── go-module/
│   ├── python-service/
│   └── typescript-service/
├── proto/                   # Shared proto definitions
└── tools/
    ├── ai8-init/            # Project scaffolding CLI
```

```
    ├── ai8-convert/         # Migrate existing to standards
    └── ai8-lint/            # Check compliance
```

Standardized Across All Services

Directory structure — Same layout for Go, Python, and TypeScript projects.

Makefile targets — `build`, `test`, `lint`, `proto`, `docker`, `run`, `clients`, `clean`.

Configuration — Environment variables with standard names: `SERVICE_NAME`, `LOG_LEVEL`, `LOG_FORMAT`, `HTTP_PORT`, `GRPC_PORT`, `METRICS_PORT`.

Health endpoints — `GET /health` (liveness), `GET /ready` (readiness), `GET /metrics` (Prometheus).

Logging — Structured JSON with `time`, `level`, `msg`, `method`, `path`, `status`, `duration_ms`, `request_id`.

Error responses — Consistent shape: `{ "error": { "code": "...", "message": "...", "details": {...} } }`.

What Stays Flexible

Business logic, config keys, log messages, readiness check logic, error codes, build steps, specific dependencies, and proto message definitions.

Escape Hatches

Per-project `.ai8.yaml` overrides with documented reasons. `ai8-lint` allows these deviations when reason is provided.

---

7. Golden Path Workflow

The golden path is the officially supported, friction-free way to accomplish common tasks.

The Six Steps

1. CREATE — `ai8-init go myservice --grpc --http` generates a fully wired project from templates with chassis imported, Makefile, Dockerfile, Kubernetes manifests, proto definitions, README, VERSION, and CHANGELOG.

2. DEVELOP — Write business logic in `internal/core/`. The chassis handles config, logging, metrics, and health automatically. `make run` for hot reload, `make test` for testing.

3. VALIDATE — `ai8-lint .` checks structure, required files, config patterns, and outputs a compliance score. Runs in CI, blocks merge below threshold.

4. BUILD — `make docker` produces a multi-stage image tagged with VERSION + git SHA. Typical Go service image: ~15MB.

5. DEPLOY — `kubectl apply -f k8s/` with pre-configured health probes, resource limits, and ServiceMonitor for Prometheus auto-discovery.

6. OPERATE — Import standard Grafana dashboards. Pre-configured alerts for error rate > 5%, P99 latency > 1s, and service unavailability.

Before vs After

Before Golden Path	After Golden Path
Copy-paste health checks from other repos	Import `chassis.Health()`
Debug graceful shutdown issues	It just works
Different logging formats per service	Consistent JSON everywhere
Forgot to add metrics	Metrics are automatic
Each Dockerfile is slightly different	One proven pattern
"How does config work in this repo?"	Same everywhere
Onboarding takes days	Productive in hours

---

8. Automated Converter Tool

Feasibility Analysis

Aspect	Feasibility	Notes
Boilerplate generation	100%	Easy to template
File reorganization	90%	AST parsing reliable
Import updates	95%	gofmt handles most
Interface extraction	70%	Needs human review
Core logic separation	50%	Needs guidance, context
API design decisions	0%	Human judgment required

Four-Phase Design

Phase 1: Analyze — Detect language, map structure, identify violations, generate compliance report.

Phase 2: Plan — Propose file moves and code extractions, flag decisions needing human input, generate `MIGRATION.md`.

Phase 3: Apply Safe Changes — Create missing directories, add boilerplate files, move files per approved plan, update imports.

Phase 4: Guided Refactoring — AI-assisted extraction of core logic, interface creation, pattern wrapping, and test generation — all with confirmation prompts.

When to Build

Scenario	Recommendation
3-5 more codebases	Standards doc + Claude Code manually
10+ codebases	Worth building analyzer + scaffolding
20+ or external contributors	Full toolchain pays off

---

9. Anti-Patterns to Avoid

1. Leaking Framework Dependencies
Library code must not require specific frameworks (gRPC, HTTP, etc.). Use interfaces or accept `*http.Client` with nil defaults.

2. Global State
Avoid `var globalConfig Config` + `Init()`. Instead, use struct-based instances: `New(cfg Config)` returning `*Client`. This allows multiple instances with different configs and eliminates test interference.

3. Hardcoded Configuration
Never hardcode connection strings or endpoints. Use functional options with sane defaults:

```
db, _ := mylib.Connect(
    mylib.WithDSN("prod-db:5432/mydb"),
    mylib.WithTimeout(30 * time.Second),
)
```

4. Mixing Service and Library Code
HTTP/gRPC handling must never live in core business logic. The library exposes `ProcessData(input Input) (Output, error)`. The handler package wraps it for HTTP/gRPC contexts.

5. String Errors Without Context
Always wrap errors with context using `fmt.Errorf("load config from %s: %w", path, err)`. Consider typed errors (`ConfigError`, `ValidationError`) for public APIs.

6. Ignoring context.Context
Every function that does I/O must accept `context.Context` as its first parameter. This enables cancellation, timeout, and request tracing.

7. Non-deterministic Initialization
Never use `init()` with side effects or panics. Use lazy initialization with `sync.Once` that captures errors and returns them on use.

---

10. Library-Readiness Checklist

Essential Requirements

- ☐ No `main()` in library code — CLI in `cmd/`, library at root or `pkg/`
- ☐ No framework dependencies in core logic
- ☐ Thread-safe by default (RWMutex on shared state)
- ☐ Functional options or builder pattern for configuration
- ☐ Interfaces for external dependencies
- ☐ Package-level convenience functions + explicit instances (dual API)
- ☐ Proper error types with `%w` wrapping
- ☐ Documentation with examples and README quickstart
- ☐ Minimal external dependencies
- ☐ Embedded data where appropriate (`go:embed`)

Code Quality

- ☐ `context.Context` as first parameter for I/O functions
- ☐ Sensible zero values
- ☐ No global mutable state
- ☐ Clear separation of concerns (core vs adapters)

Testing

- ☐ Unit tests for core logic
- ☐ Example tests (appear in godoc)
- ☐ Benchmarks for performance-critical code

Documentation

- ☐ README with quickstart
- ☐ Godoc on all exported symbols
- ☐ CHANGELOG.md with semantic versioning
- ☐ VERSION file

Distribution

- ☐ Proper `go.mod` with correct module path
- ☐ Tagged releases (v1.2.3 format)
- ☐ CI/CD with automated tests and releases

Advanced (Nice to Have)

- ☐ CLI wrapper (thin, over library)
- ☐ Client libraries (Go, Python, TypeScript)

- ☐ Metrics/observability hooks (optional, not forced)
- ☐ Graceful degradation for missing optional dependencies

Implementation Roadmap

Phase 1: Extract chassis-go (1-2 weeks)
Pull common infrastructure code from serp_svc into `chassis-go`. Health, metrics, logging, graceful shutdown. Validate by using in one new service.

Phase 2: Create Templates (1 week)
Build `go-service` template using `chassis-go`. Implement `ai8-init` as scaffolding CLI (can start as a shell script).

Phase 3: Document the Path (Ongoing)
Write `STANDARDS.md` with conventions. Maintain decision log for architectural choices. Document "when to deviate" guidelines.

Phase 4: Module Extraction (2-3 weeks each)
Start with serp_svc (70% ready, highest priority). Follow with pdfgen_svc and markdown_svc. New modules: Gemini wrapper, SeaweedFS storage.

Phase 5: Tooling Polish (As needed)
Build `ai8-lint` for CI enforcement. Build `ai8-convert` for legacy migration. Create standard dashboards and alert templates.

Generated from architectural analysis session, January 28, 2026.