Case Management Domain Architecture & Decision Guide with Information Flow Diagrams

The WHY Behind Code Organization

January 2026

Abstract

This guide explains the **architectural principles** and **decision-making framework** for the Case Management domain. Unlike a directory reference, this document answers: *Why is the code organized this way?* and *Where should I add new code?* It includes detailed diagrams showing how information flows from apps through modules to data persistence, and provides decision trees, real examples, and anti-patterns to help developers make the right architectural choices.

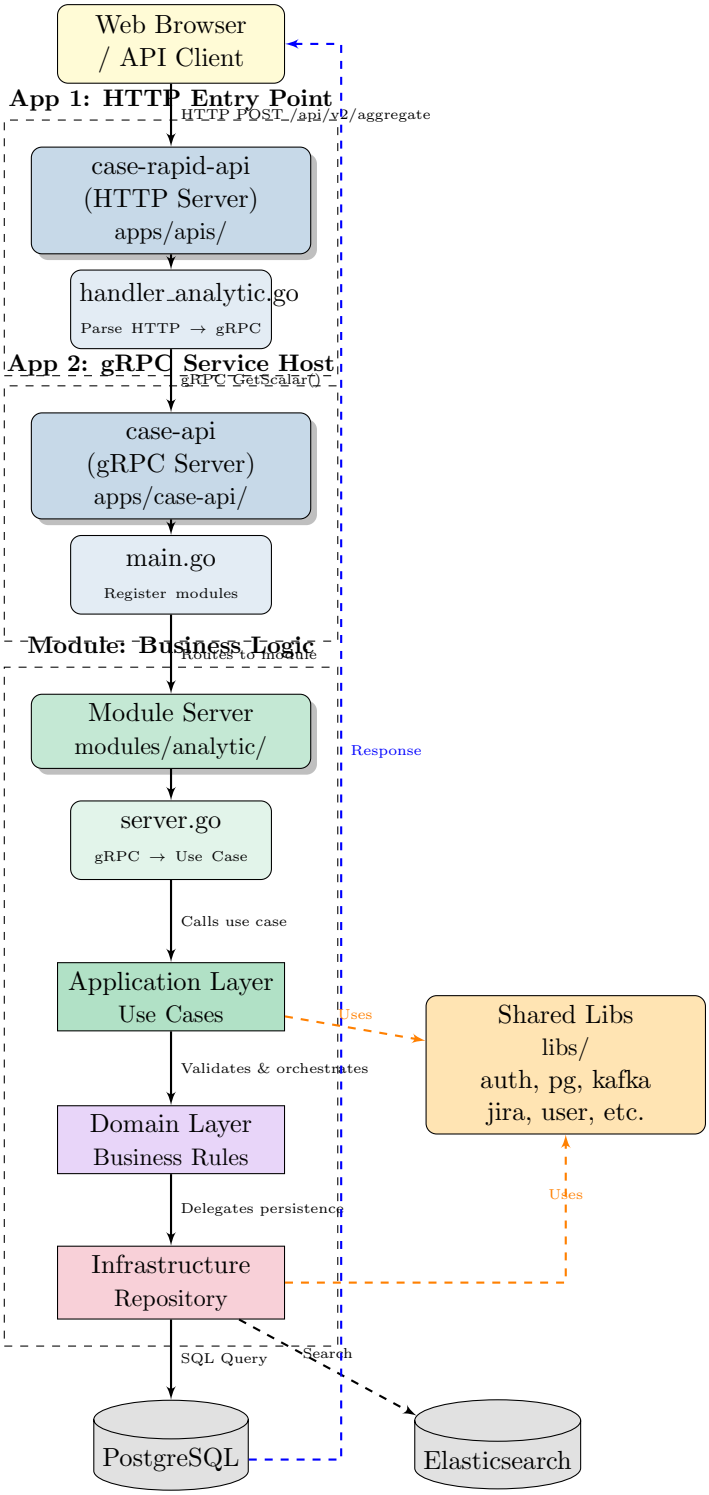
Contents

1 Visual Architecture: Information Flow

This section provides comprehensive diagrams showing how information flows through the Case Management domain architecture.

1.1 Complete Request Flow: HTTP to Database

This diagram shows the complete journey of an HTTP request through all layers of the architecture.



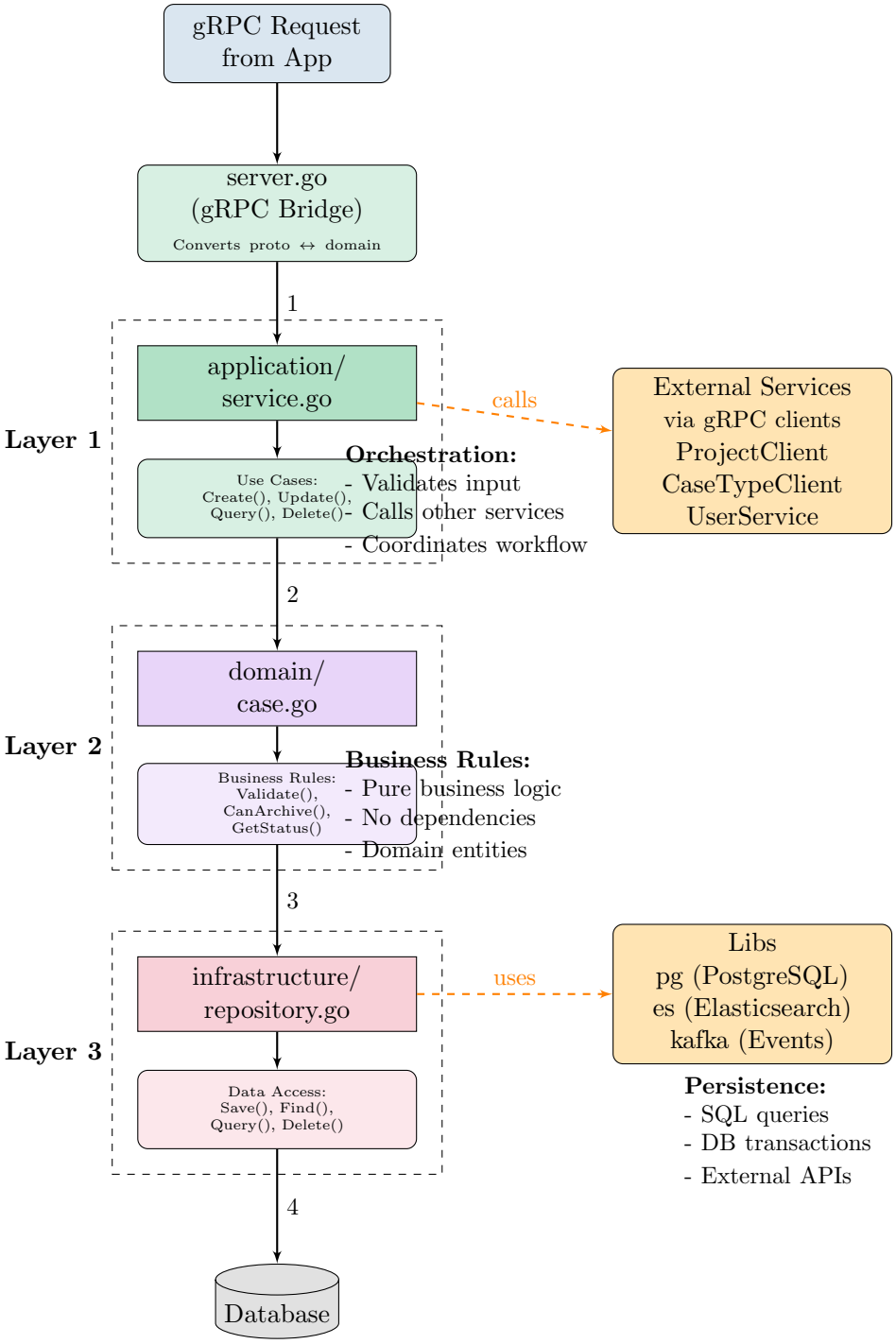
Key Points:

- **Apps** (blue) handle infrastructure: HTTP parsing, gRPC servers, wiring

- **Modules** (green) contain business logic in 3 layers
- **Libs** (orange) are used across all layers for cross-cutting concerns
- Request flows DOWN, response flows UP (dashed blue line)

1.2 Module Internal Structure: 3-Layer Architecture

This diagram shows how a module is structured internally with its three layers and how they interact.

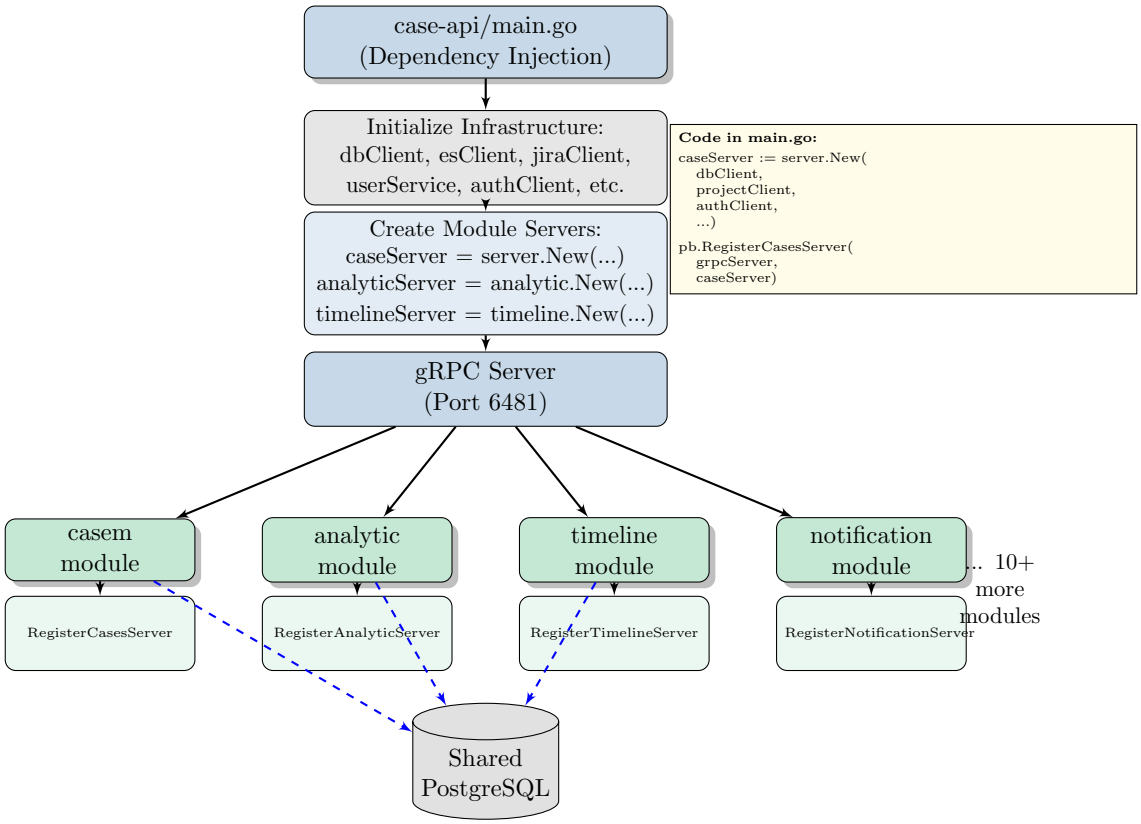


Information Flow:

1. **Application Layer** receives command from server.go
2. **Domain Layer** validates business rules
3. **Infrastructure Layer** persists to database
4. Response flows back up through layers

1.3 App Wiring: How case-api Composes Modules

This diagram shows how the `case-api` app wires multiple modules together into a single gRPC service.



Key Responsibilities of case-api/main.go:

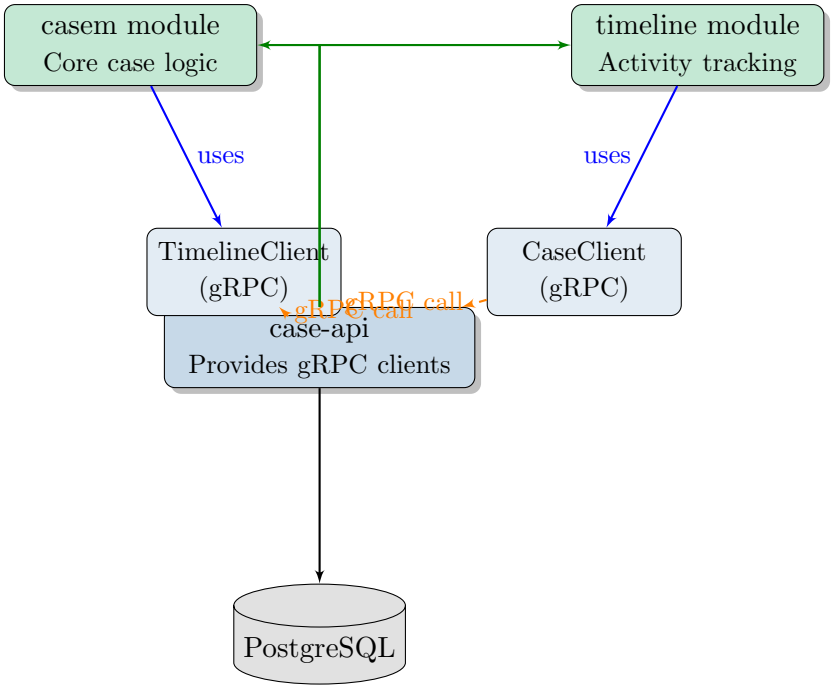
1. Initialize all infrastructure clients (DB, external services)
2. Instantiate module servers with dependencies (Dependency Injection)
3. Register each module with the gRPC server
4. Start gRPC server and listen for requests
5. Handle graceful shutdown

Why this design?

- **Single binary** hosts multiple business domains (modules)
- **Shared infrastructure** (one DB connection pool, one gRPC port)
- **Independent modules** can be developed separately
- **Easy testing** - modules don't know about each other's internals

1.4 Cross-Module Communication

This diagram shows how modules communicate with each other through gRPC clients.



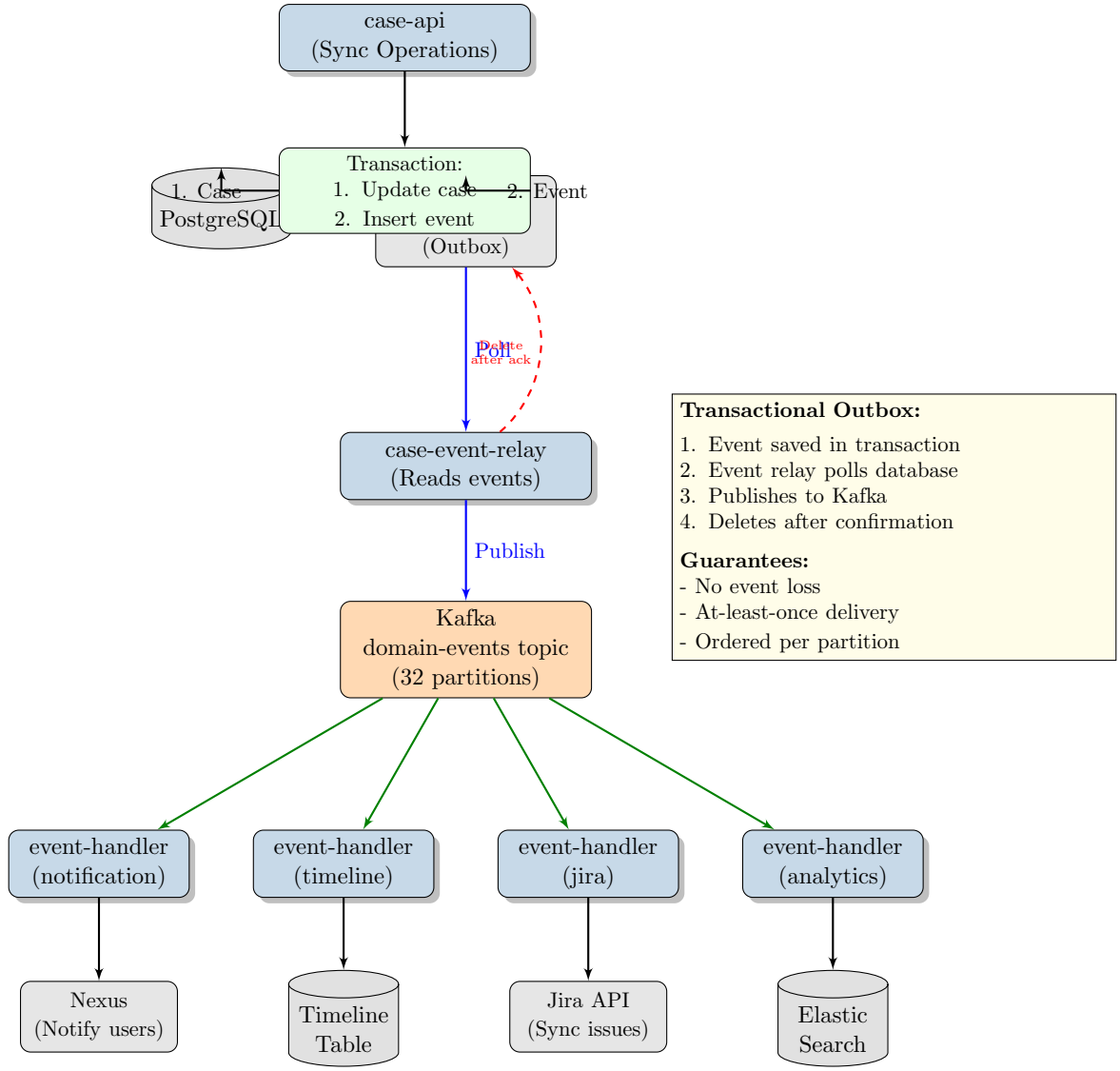
Example Flow: 1. casem module updates case → 2. Needs to add timeline entry → 3. Uses TimelineClient (gRPC) → 4. Client calls case-api → 5. case-api routes to timeline module → 6. timeline module saves to DB. **Benefit:** Clean boundaries, no direct module dependencies.

Why gRPC Clients Between Modules?

- **Loose coupling** - Modules don't directly depend on each other
- **Contract-first** - Communication via proto definitions
- **Versioning** - Can version APIs independently
- **Testing** - Easy to mock gRPC clients

1.5 Event-Driven Architecture: Async Processing

This diagram shows how events flow from case-api through Kafka to event handlers.



Why Async Event Processing?

- **Decoupling** - case-api doesn't wait for downstream processing
- **Scalability** - Each handler can scale independently
- **Reliability** - Transactional outbox prevents event loss
- **Extensibility** - Add new handlers without changing case-api

2 The Core Question: Why Three Directories?

The case management domain organizes code into three primary categories:

- **apps/** - Deployable services (12 apps)
- **modules/** - Business logic domains (22 modules)
- **libs/** - Cross-cutting utilities (46 libraries)

2.1 The Fundamental Principle

Separation of Concerns by Responsibility, Not by Technology
This is not MVC (Model-View-Controller). This is:

1. **Apps** = Execution entry points + infrastructure wiring
2. **Modules** = Business domains + feature boundaries
3. **Libs** = Reusable utilities + external integrations

2.2 Why This Matters

2.2.1 Benefit 1: Independent Deployment

Apps can be deployed independently. Example:

- Deploy `case-event-relay` without touching `case-api`
- Scale `case-event-handler` (notification flavor) independently
- Roll back `case-indexer` without affecting other services

2.2.2 Benefit 2: Clear Ownership

Each module owns its business logic:

- `modules/notification` team owns notification rules
- `modules/analytic` team owns analytics queries
- Changes don't leak across boundaries

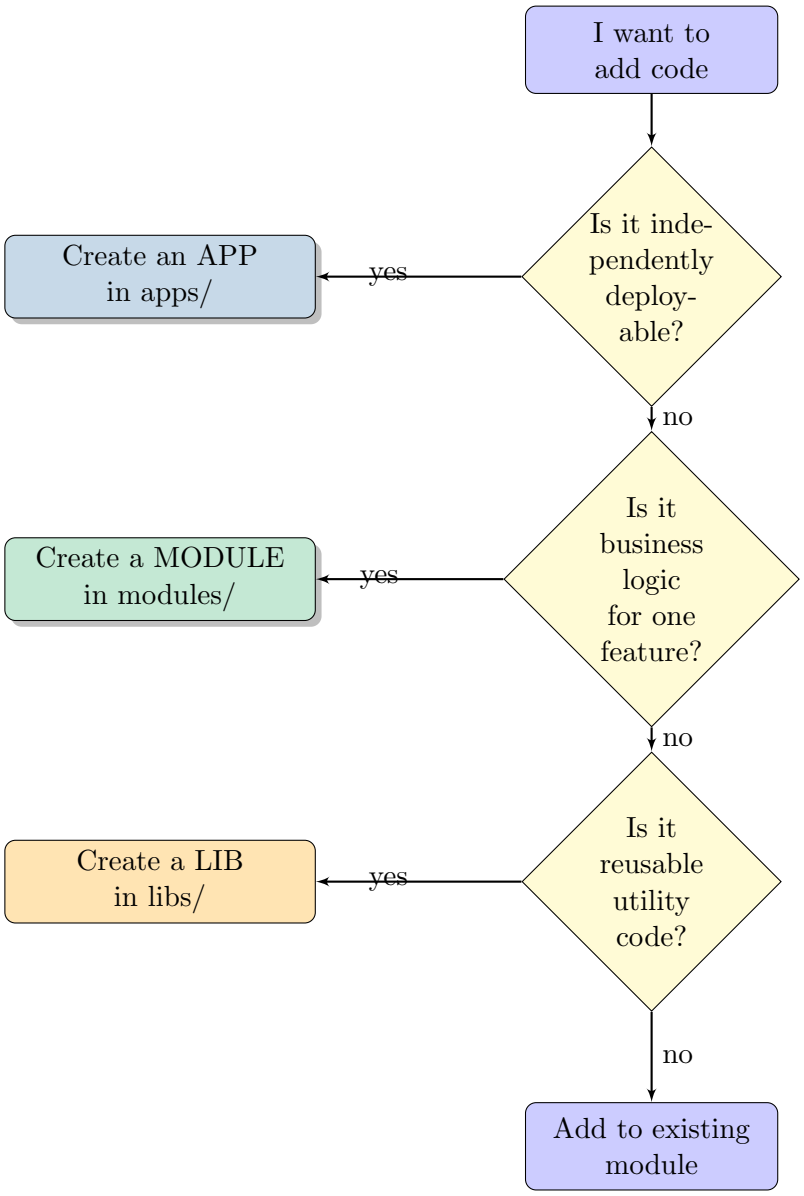
2.2.3 Benefit 3: Dependency Control

Bazel visibility rules enforce clean dependencies:

- Modules cannot depend on apps
- Libs cannot depend on modules
- Prevents circular dependencies and "spaghetti code"

3 Decision Tree: Where Does My Code Go?

3.1 The Primary Decision Flow



3.2 Decision 1: Is it Independently Deployable?

3.2.1 YES → Create an APP

Examples of Apps:

- `case-api` - gRPC server hosting multiple modules
- `case-event-handler` - Kafka consumer processing events
- `case-event-relay` - Event relay from database to Kafka
- `case-indexer` - Elasticsearch indexing service
- `case-intake` - Case creation entry point

Characteristics of an App:

1. Has a `main.go` file (entry point)

2. Can be deployed as a Kubernetes pod
3. Wires together modules, libs, and external services
4. Contains infrastructure setup (gRPC server, HTTP server, Kafka consumer)
5. Has its own Kubernetes deployment YAML in `config/k8s/`

Real Example - case-event-relay:

```
// apps/case-event-relay/main.go
func main() {
    // Infrastructure setup
    db := initDB()
    kafkaProducer := initKafkaProducer()

    // Wire components
    lockCtrl := NewLockController(db)
    reader := NewReader(db)
    producer := NewProducer(kafkaProducer)
    acknowledger := NewAcknowledger(db)

    // Start goroutines
    go lockCtrl.Run(ctx)
    go processor.Run(ctx)
    go acknowledger.Run(ctx)

    // Health check server
    http.ListenAndServe(":8080", nil)
}
```

When NOT to create an app:

- You just want to add a new gRPC endpoint → Add to existing module
- You want to add a new event handler → Add flavor to case-event-handler
- You want to add business logic → Create/update a module

3.3 Decision 2: Is it Business Logic for One Feature?

3.3.1 YES → Create or Update a MODULE

Examples of Modules:

- `casem` - Core case management (CRUD, queries, state)
- `notification` - Notification rules and delivery
- `analytic` - Analytics queries with group-by
- `timeline` - Case activity tracking
- `automation` - Automation rules and execution

Module Structure (Clean Architecture):

```
modules/myfeature/
|-- domain/              # Business rules, entities, value objects
|   |-- myfeature.go     # Core domain model
|   |-- validation.go    # Business rule validation
|   |-- errors.go        # Domain-specific errors
|
|-- application/         # Use cases, orchestration, commands
```

```
|   |-- service.go       # Application service
|   |-- create.go        # Create use case
|   |-- update.go        # Update use case
|   |-- query.go         # Query use case
|
|-- infrastructure/      # Data persistence, external integrations
|   |-- repository.go    # Database repository
|   |-- elasticsearch.go # Search indexing
|
|-- server.go            # gRPC service implementation
|-- BUILD.bazel          # Bazel build configuration
```

3.3.2 The Three-Layer Pattern

Layer 1: Domain (Business Rules)

This is where business logic lives. Pure Go, no dependencies on frameworks.

```
// modules/notificationrule/domain/notification_rule.go
type NotificationRule struct {
    ID         string
    OrgID      int64
    ProjectID string
    Query      string    // Filter expression
    Recipients []string
    Enabled    bool
}

// Business rule validation
func (r *NotificationRule) Validate() error {
    if r.Query == "" {
        return errors.New("query␣cannot␣be␣empty")
    }
    if len(r.Recipients) == 0 {
        return errors.New("must␣have␣at␣least␣one␣recipient")
    }
    return nil
}
```

Layer 2: Application (Use Cases)

Orchestrates domain logic and external dependencies.

```
// modules/notificationrule/application/service.go
type Service struct {
    repo Repository
    userClient user.Client
}

func (s *Service) CreateRule(ctx context.Context,
    orgID int64, projectID, query string,
    recipients []string) (*domain.NotificationRule, error) {

    // Validate recipients exist
    for _, r := range recipients {
        if _, err := s.userClient.GetUser(ctx, r); err != nil {
            return nil, fmt.Errorf("invalid␣recipient␣%s", r)
        }
    }

    // Create domain object
    rule := &domain.NotificationRule{
```

```
        ID: uuid.New().String(),
        OrgID: orgID,
        ProjectID: projectID,
        Query: query,
        Recipients: recipients,
        Enabled: true,
    }

    // Validate business rules
    if err := rule.Validate(); err != nil {
        return nil, err
    }

    // Persist
    if err := s.repo.Save(ctx, rule); err != nil {
        return nil, err
    }

    return rule, nil
}
```

Layer 3: Infrastructure (Data Access)
Handles persistence and external integrations.

```
// modules/notificationrule/infrastructure/repository.go
type Repository struct {
    db *sql.DB
}

func (r *Repository) Save(ctx context.Context,
    rule *domain.NotificationRule) error {

    query := `INSERT␣INTO␣notification_rules
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣(id,␣org_id,␣project_id,␣query,␣recipients,␣enabled)
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣VALUES␣($1,␣$2,␣$3,␣$4,␣$5,␣$6)`

    _, err := r.db.ExecContext(ctx, query,
        rule.ID, rule.OrgID, rule.ProjectID,
        rule.Query, pq.Array(rule.Recipients), rule.Enabled)

    return err
}
```

Bridge: server.go (gRPC Service)
Connects gRPC requests to application layer.

```
// modules/notificationrule/server.go
type Server struct {
    service *application.Service
}

func (s *Server) Create(ctx context.Context,
    req *notificationrulepb.CreateRequest)
    (*notificationrulepb.CreateResponse, error) {

    // Call application layer
    rule, err := s.service.CreateRule(ctx,
        req.OrgId, req.ProjectId, req.Query, req.Recipients)
    if err != nil {
        return nil, status.Error(codes.Internal, err.Error())
    }
```

```
    // Convert domain model to proto
    return &notificationrulepb.CreateResponse{
        Rule: toProto(rule),
    }, nil
}
```

3.3.3 When to Create a New Module vs Update Existing

Create a NEW module if:

1. It's a new business capability (e.g., "SLA tracking")
2. It has its own database tables
3. It has distinct domain concepts and rules
4. Different teams would own it

Update EXISTING module if:

1. Adding fields to existing entity (e.g., add "due_date" to Case)
2. Extending existing feature (e.g., add new analytic group-by dimension)
3. Adding query/command to existing domain

3.4 Decision 3: Is it Reusable Utility Code?

3.4.1 YES → Create a LIB

Examples of Libs:

- `pg` - PostgreSQL client utilities
- `kafka` - Kafka producer/consumer helpers
- `auth` - Authorization/RBAC client
- `jira` - Jira API client
- `ddclient` - Datadog service clients (notebooks, incidents)

Characteristics of a Lib:

1. Used by multiple modules or apps
2. No business logic (only technical utilities)
3. Broad visibility across domains
4. Depends only on other libs (never on modules)

Real Example - libs/jira:

```
// libs/jira/client.go
type Client struct {
    httpClient *http.Client
    baseURL    string
    apiToken   string
}

func (c *Client) CreateIssue(ctx context.Context,
    issue *Issue) (*Issue, error) {
```

```
    // Generic Jira API interaction
    // No case management business logic

    resp, err := c.httpClient.Post(
        c.baseURL + "/rest/api/3/issue",
        "application/json",
        marshalIssue(issue))

    // ...
    return unmarshalIssue(resp), nil
}
```

When NOT to create a lib:

- Code is specific to one module → Keep it in that module
- It contains business rules → Should be in domain/ layer
- It's only used by one app → Keep it in that app

4 Real-World Scenarios: Where Does Code Go?

4.1 Scenario 1: Add New Notification Type (Slack)

Question: I want to add Slack notifications alongside email. Where does this go?
Analysis:

- Is it independently deployable? NO (uses existing case-event-handler)
- Is it business logic? YES (notification delivery is a feature)
- Is it utility code? NO (specific to notification domain)

Decision: Update the `modules/notificationrule` module
Changes:

1. Add Slack client to `libs/slack/` (if not exists)
2. Update `modules/notificationrule/domain/` to support Slack recipients
3. Update `modules/notificationrule/application/` to route to Slack
4. Update `apps/case-event-handler/handlers/notification/` to invoke Slack delivery

File locations:

```
libs/slack/                                  # NEW LIB
|-- client.go                                # Slack API client
|-- message.go                               # Message formatting

modules/notificationrule/domain/
|-- recipient.go                             # ADD: SlackRecipient type

modules/notificationrule/application/
|-- service.go                               # ADD: Slack delivery logic

apps/case-event-handler/handlers/notification/
|-- processor/notification_processor.go      # UPDATE: Add Slack path
```

4.2 Scenario 2: Add New Analytics Dimension (Group by Assignee)

Question: Users want to group cases by assignee. Where does this go?
Analysis:

- Is it independently deployable? NO
- Is it business logic? YES (analytics is a feature)
- Is it new module or existing? EXISTING (analytics already exists)

Decision: Update `modules/analytic` module
Changes:

1. Add "assignee" to group-by enum in `proto/analyticpb/`
2. Update `modules/analytic/groupby/suggestions.go` for assignee values
3. Update `modules/analytic/server.go` to handle assignee dimension
4. Add database query logic in `modules/analytic/infrastructure/`

File locations:

```
proto/analyticpb/
|-- analytic.proto                          # UPDATE: Add ASSIGNEE enum

modules/analytic/groupby/
|-- suggestions.go                          # UPDATE: Add assignee suggestions

modules/analytic/
|-- server.go                               # UPDATE: Handle assignee dimension

modules/analytic/infrastructure/
|-- repository.go                           # UPDATE: Add SQL for assignee
```

4.3 Scenario 3: Add ServiceNow Integration

Question: We need to sync cases with ServiceNow tickets. Where does this go?
Analysis:

- Is it independently deployable? NO (reuses case-event-handler)
- Is it business logic? BOTH (integration + sync rules)
- ServiceNow client - utility code? YES
- Sync logic - business logic? YES

Decision: Multi-part implementation
Changes:

1. Create `libs/servicenow/` for API client
2. Create or update `modules/synchronisation/` for sync logic
3. Add flavor to `apps/case-event-handler/handlers/snow-sync/`

File locations:

```
libs/servicenow/                          # NEW LIB
|-- client.go                             # ServiceNow REST API
|-- ticket.go                             # Ticket CRUD
|-- mapping.go                            # Field mapping utilities

modules/synchronisation/                  # NEW or UPDATE MODULE
|-- domain/
|   |-- sync_config.go                    # Sync configuration domain
|   |-- field_mapping.go                  # Field mapping rules
|-- application/
|   |-- sync_service.go                   # Sync orchestration
|-- infrastructure/
|   |-- repository.go                     # Persist sync state

apps/case-event-handler/handlers/snow-sync/  # NEW HANDLER
|-- handler.go                            # EventHandler impl
|-- sync_processor.go                     # Sync logic
```

4.4 Scenario 4: Add Real-Time Case Search via WebSocket

Question: We want to add a WebSocket API for real-time case updates. Where?
Analysis:

- Is it independently deployable? YES (new protocol, separate scaling)
- Is it business logic? NO (transport layer)
- Needs to access case data? YES (via modules)

Decision: Create NEW app `apps/case-websocket-api/`
Changes:

1. Create `apps/case-websocket-api/` with WebSocket server
2. Consume from Kafka `domain-events` topic for real-time updates
3. Call `modules/casem/` gRPC service for initial data load

File locations:

```
apps/case-websocket-api/                  # NEW APP
|-- main.go                               # WebSocket server entry
|-- websocket/
|   |-- server.go                         # WebSocket upgrade handler
|   |-- connection.go                     # Connection management
|   |-- subscription.go                   # Subscribe to case updates
|-- kafka/
|   |-- consumer.go                       # Kafka event consumer
|-- config/
|   |-- k8s/                              # Kubernetes deployment
```

4.5 Scenario 5: Add Case Archiving Logic

Question: Cases should be automatically archived after 90 days. Where?
Analysis:

- Is it independently deployable? YES (background job, separate schedule)
- Is it business logic? YES (archiving rules)

- Needs scheduled execution? YES (cron-like)

Decision: Create NEW app `apps/case-archiver/`
Changes:

1. Create `apps/case-archiver/` as cron job app
2. Use `modules/casem/` domain/application for archiving logic
3. Update `modules/casem/application/` to add ArchiveCases method

File locations:

```
modules/casem/domain/
|-- case.go                                # UPDATE: Add CanArchive() method

modules/casem/application/
|-- archive_service.go                     # NEW: Archiving logic

apps/case-archiver/                        # NEW APP (Kubernetes CronJob)
|-- main.go                                # Cron job entry point
|-- archiver.go                            # Queries old cases, archives
|-- config/
|   |-- k8s/
|       |-- cronjob.yaml                   # Runs daily at 2am
```

5 Module Design Patterns

5.1 Pattern 1: The Event Handler Plugin System

The `case-event-handler` app uses a "flavor" pattern for pluggable handlers.

Why?

- One binary, many deployment configurations
- Easy to add new event processing logic
- Each handler can scale independently

Implementation:

```
// apps/case-event-handler/commonhandler/handler.go
type EventHandler interface {
    HandleEvent(ctx context.Context, event *Event) error
}

// apps/case-event-handler/main.go
const (
    notificationFlavor = "notification"
    analyticFlavor     = "analytic"
    jiraflavor         = "jira"
    // ... 10+ flavors
)

func newProcessor(cfg *config.File) (EventHandler, error) {
    flavor := cfg.GetFlavor()
    switch flavor {
    case notificationFlavor:
        return notification.NewHandler(deps...), nil
    case analyticFlavor:
        return analytic.NewHandler(deps...), nil
```

```
    case jiraFlavor:
        return jira.NewHandler(deps...), nil
    }
}
```

When to add a new handler:

1. Create `apps/case-event-handler/handlers/myhandler/`
2. Implement `EventHandler` interface
3. Add constructor to switch statement in `main.go`
4. Deploy with flavor config: `DD_FLAVOR=myhandler`

5.2 Pattern 2: The Repository Interface

Modules define repository interfaces in application layer.

Why?

- Decouples business logic from data access
- Enables testing with mock repositories
- Infrastructure can change without affecting application

Implementation:

```
// modules/casem/application/repository.go (interface)
type Repository interface {
    Save(ctx context.Context, case *domain.Case) error
    FindByID(ctx context.Context, id string) (*domain.Case, error)
    Query(ctx context.Context, filter *Filter) ([]*domain.Case, error)
}

// modules/casem/infrastructure/repository.go (implementation)
type PostgresRepository struct {
    db *sql.DB
}

func (r *PostgresRepository) Save(ctx context.Context,
    c *domain.Case) error {
    // PostgreSQL-specific implementation
}
```

5.3 Pattern 3: Proto-Module Mapping

Each module has a corresponding proto package.

Why?

- Proto defines the public API contract
- Module implements the contract
- Versioning is handled at proto level

Mapping:

```
proto/analyticpb/
|-- analytic.proto                          # Service definition

modules/analytic/
|-- server.go                               # Implements AnalyticsServer
|-- domain/                                 # Business logic
|-- application/                            # Use cases
```

Pattern:

1. Define service in proto: `service Analytics { ... }`
2. Generate Go code: `analyticpb.AnalyticsServer` interface
3. Implement in module: `modules/analytic/server.go`
4. Register in app: `analyticpb.RegisterAnalyticsServer(grpcServer, analyticServer)`

6 Common Anti-Patterns (What NOT to Do)

6.1 Anti-Pattern 1: Business Logic in Apps

Wrong:

```
// apps/case-api/main.go - DON'T DO THIS
func (s *server) CreateCase(ctx, req) (*Case, error) {
    // Validation logic
    if req.Title == "" {
        return nil, errors.New("title␣required")
    }

    // Business rule
    if req.Priority == "P1" && req.Assignee == "" {
        return nil, errors.New("P1␣cases␣must␣have␣assignee")
    }

    // Database write
    _, err := s.db.Exec("INSERT␣INTO␣cases␣...")
    return &Case{...}, err
}
```

Why it's wrong:

- Business rules buried in app code
- Cannot reuse validation elsewhere
- Hard to test (requires running full app)
- No domain model

Correct:

```
// modules/casem/domain/case.go
type Case struct { ... }
func (c *Case) Validate() error { /* business rules */ }

// modules/casem/application/service.go
func (s *Service) CreateCase(...) (*domain.Case, error) {
    case := &domain.Case{...}
    if err := case.Validate(); err != nil { return nil, err }
    return s.repo.Save(ctx, case)
```

```
}

// modules/casem/server.go
func (s *Server) CreateCase(ctx, req) (*Response, error) {
    case, err := s.service.CreateCase(...)  // delegate
    return toProto(case), err
}

// apps/case-api/main.go
func main() {
    caseServer := casem.NewServer(caseService)
    casempb.RegisterCasesServer(grpcServer, caseServer)
}
```

6.2 Anti-Pattern 2: Modules Depending on Apps

Wrong:

```
modules/timeline/
|-- server.go
|   import "domains/case_management/apps/case-api/client"
```

Why it's wrong:

- Circular dependency (app depends on module, module depends on app)
- Cannot reuse module in other apps
- Bazel visibility rules will reject this

Correct:

```
modules/timeline/application/
|-- service.go
    // Define interface for external dependency
    type CaseClient interface {
        GetCase(ctx, id) (*Case, error)
    }

apps/case-api/main.go
    // App provides implementation
    timelineService := timeline.NewService(caseAPIClient)
```

6.3 Anti-Pattern 3: Domain Logic in Libs

Wrong:

```
// libs/caseutil/validator.go - DON'T DO THIS
func ValidateCasePriority(priority string, assignee string) error {
    // Business rule: P1 cases must have assignee
    if priority == "P1" && assignee == "" {
        return errors.New("P1 requires assignee")
    }
    return nil
}
```

Why it's wrong:

- Business rules belong in domain layer, not libs
- Libs are for technical utilities, not business logic

- Hard to discover (who looks in libs for business rules?)

Correct:

```
// modules/casem/domain/case.go
type Case struct {
    Priority string
    Assignee string
}

func (c *Case) Validate() error {
    if c.Priority == "P1" && c.Assignee == "" {
        return errors.New("P1 requires assignee")
    }
    return nil
}
```

6.4 Anti-Pattern 4: Creating Apps for Everything

Wrong:

```
apps/case-notification-sender/     # Separate app for notifications
apps/case-jira-syncer/              # Separate app for Jira
apps/case-snow-syncer/              # Separate app for ServiceNow
apps/case-analytics-processor/      # Separate app for analytics
```

Why it's wrong:

- Duplicates infrastructure (12 apps instead of 1)
- More Kubernetes resources
- Harder to manage deployments
- Violates DRY (each app has similar boilerplate)

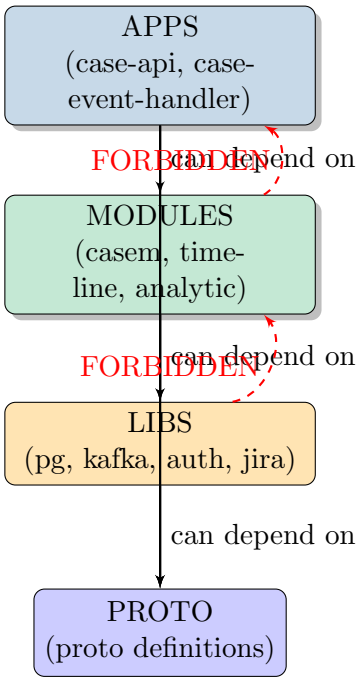
Correct:

```
apps/case-event-handler/           # ONE app
|-- handlers/
    |-- notification/              # Flavor: notification
    |-- jira/                      # Flavor: jira
    |-- snow/                      # Flavor: snow
    |-- analytic/                  # Flavor: analytic

# Deploy with different configs:
# Pod 1: DD_FLAVOR=notification (3 replicas)
# Pod 2: DD_FLAVOR=jira (1 replica)
# Pod 3: DD_FLAVOR=analytic (2 replicas)
```

7 Dependency Rules & Bazel Visibility

7.1 The Dependency Hierarchy



7.2 Enforcing Rules with Bazel Visibility

Example: Module Visibility

```
# modules/casem/server/BUILD.bazel
go_library(
    name = "go_default_library",
    visibility = [
        "//domains/case_management/apps/case-api:__subpackages__",
        "//domains/case_management/modules/timeline:__subpackages__",
        "//domains/case_management/modules/link:__subpackages__",
        # Explicitly list which modules can depend on this
    ],
)
```

Example: Lib Visibility

```
# libs/auth/BUILD.bazel
go_library(
    name = "go_default_library",
    visibility = [
        "//domains/case_management:__subpackages__",
        "//domains/change_management:__subpackages__",
        "//domains/incident_management:__subpackages__",
        # Available to all domains
    ],
)
```

8 Testing Strategy by Layer

8.1 Testing Apps

Focus: Integration testing - does wiring work?

```
// apps/case-api/main_test.go
func TestServerStartup(t *testing.T) {
    // Test that all modules register correctly
    server := setupServer(t)

    // Call a gRPC method
    resp, err := server.CreateCase(ctx, &casempb.CreateRequest{...})
    assert.NoError(t, err)
    assert.NotEmpty(t, resp.Case.Id)
}
```

8.2 Testing Modules

Domain Layer: Pure unit tests

```
// modules/casem/domain/case_test.go
func TestCase_Validate_RequiresTitle(t *testing.T) {
    c := &Case{Title: ""}
    err := c.Validate()
    assert.Error(t, err)
    assert.Contains(t, err.Error(), "title")
}
```

Application Layer: Test with mock repositories

```
// modules/casem/application/service_test.go
func TestService_CreateCase(t *testing.T) {
    mockRepo := &MockRepository{}
    service := NewService(mockRepo, ...)

    case, err := service.CreateCase(ctx, ...)
    assert.NoError(t, err)
    assert.True(t, mockRepo.SaveWasCalled)
}
```

Infrastructure Layer: Integration tests with real database

```
// modules/casem/infrastructure/repository_test.go
func TestRepository_Save(t *testing.T) {
    db := testcontainers.SetupPostgres(t)
    repo := NewRepository(db)

    err := repo.Save(ctx, &domain.Case{...})
    assert.NoError(t, err)

    // Verify in database
    var count int
    db.QueryRow("SELECT␣COUNT(*)␣FROM␣cases").Scan(&count)
    assert.Equal(t, 1, count)
}
```

8.3 Testing Libs

Focus: Unit tests for utility functions

```
// libs/parser/query_parser_test.go
func TestParseQuery(t *testing.T) {
    query := "status:open␣AND␣priority:P1"
    ast, err := ParseQuery(query)
    assert.NoError(t, err)
```

```
    assert.Equal(t, "AND", ast.Operator)
}
```

9 Evolution & Refactoring Patterns

9.1 Pattern: Extract Module from App

Scenario: App has grown complex, needs refactoring

Before:

```
apps/case-api/
|-- main.go
|-- handlers.go          # All business logic here
|-- database.go          # All data access here
```

After:

```
modules/casem/
|-- domain/case.go       # Extracted domain model
|-- application/service.go
|-- infrastructure/repository.go
|-- server.go

apps/case-api/
|-- main.go              # Wires modules together
```

Steps:

1. Identify business logic in app
2. Create module with domain/application/infrastructure
3. Move business logic to domain layer
4. Move orchestration to application layer
5. Move data access to infrastructure layer
6. Create server.go for gRPC
7. Update app to use module

9.2 Pattern: Extract Lib from Module

Scenario: Code is being duplicated across modules

Before:

```
modules/casem/utils/json.go              # JSON utils
modules/timeline/utils/json.go           # Duplicate JSON utils
modules/notification/utils/json.go       # Duplicate JSON utils
```

After:

```
libs/json/
|-- marshal.go           # Shared JSON utilities
|-- unmarshal.go
```

When to extract:

- Code duplicated in 3+ places
- No domain-specific logic
- Pure utility function

10 Quick Reference: Decision Cheat Sheet

I want to...	Where it goes
Add a new gRPC service	Update existing module or create new module with `server.go`
Add a new HTTP endpoint	Update `apps/apis/case-rapid-api/http/`
Add event processing logic	Add handler to `apps/case-event-handler/handlers/`
Add business validation	Module's `domain/` layer
Add use case/command	Module's `application/` layer
Add database query	Module's `infrastructure/` layer
Add external API client	Create or update `libs/servicename/`
Add Kafka consumer	Create flavor in `case-event-handler` or new app if complex
Add scheduled job	Create new app as Kubernetes CronJob
Add authentication logic	Use existing `libs/auth/`
Add shared utility	Create or update `libs/utilname/`
Add proto definition	Create `proto/featurepb/` matching module

11 Conclusion

The case management domain architecture follows clear principles:

1. **Apps** = Independently deployable services
2. **Modules** = Business features with clean architecture
3. **Libs** = Reusable utilities with no business logic

When adding code, always ask:

- Is it independently deployable? → App
- Is it business logic for a feature? → Module
- Is it reusable utility? → Lib
- Is it extending existing feature? → Update existing module

Follow the three-layer pattern within modules:

- `domain/` - Business rules and entities
- `application/` - Use cases and orchestration
- `infrastructure/` - Data access and integrations

Enforce clean dependencies via Bazel visibility rules to prevent architectural erosion.

The diagrams in this guide show how information flows through the system, from HTTP requests through apps, modules, and their internal layers, all the way to data persistence and external integrations.

This architectural guide reflects the case management domain as of January 2026. For implementation details, refer to the source code at:
`domains/case_management/`