



TECHNICAL
ENGINEERING
EDITION



CODEIGNITER 4 DEEP DIVE

Clean Architecture & Auditing CI3 Legacy

FROM **LEGACY CODE** TO **PRODUCTION-READY ARCHITECTURE**

LEGACY CI3 CODE

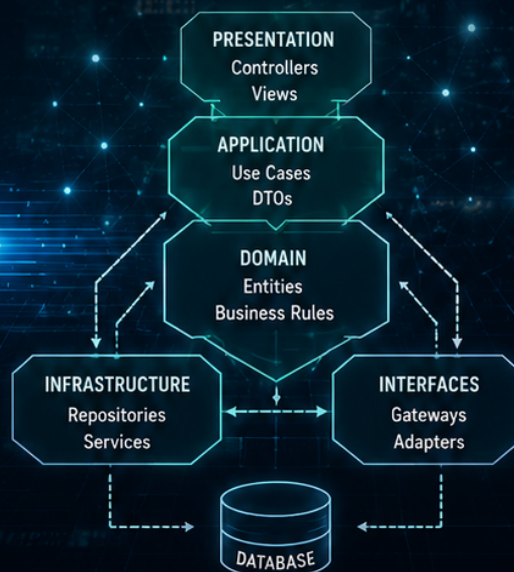
TIGHT COUPLING • SPAGHETTI CODE • HIGH RISK

```
<?php
class User extends CI_Controller {
    public function save() {
        $this->load->model('User_model');
        $this->load->library('form_validation');
        $data = $this->input->post();
        if($this->form_validation->run() == FALSE){
            $this->load->view('user/form');
        } else {
            $this->User_model->insert($data);
            redirect('user/list');
        }
    }
}
```



CI4 CLEAN ARCHITECTURE

LOOSE COUPLING • TESTABLE • SCALABLE • SECURE



AUDIT & ANALYZE
CI3 Legacy Systems



IMPLEMENT
Clean Architecture
in CI4



SECURE
Your Application
by Design



MIGRATE
Step-by-Step
to CI4



PRODUCTION
Ready &
Observability

BY **XECHCYBERSECURITY**

ENGINEER • SECURE • MODERNIZE

BUILT FOR
ENGINEERS
BY ENGINEERS

CODEIGNITER 4 DEEP DIVE

Clean Architecture & Auditing CI3 Legacy

From Legacy Code to Production-Ready Architecture

TECHNICAL ENGINEERING EDITION

BY XECHCYBERSECURITY

ENGINEER • SECURE • MODERNIZE

Copyright & Technical Disclaimer

Copyright © 2026 XechCybersecurity. All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without prior written permission, except brief quotations permitted by applicable copyright law.

CodeIgniter® is a trademark of the CodeIgniter Foundation. Other product and company names belong to their respective owners.

Technical disclaimer: framework APIs, PHP versions, packages, and deployment practices change. Verify the exact versions used by your project against the official documentation before production use.

Security warning: security testing and auditing methodologies described in this book must only be performed on systems for which you have explicit authorization. Production implementation must be tested and validated before release.

Preface

Many developers learn how to build an application. Fewer learn how to keep an application maintainable for a decade. Legacy code, technical debt, undocumented assumptions, and security weaknesses can make a simple change unexpectedly risky.

This book follows the engineering journey from CodeIgniter 4 fundamentals to Clean Architecture, security hardening, CI3 legacy auditing, incremental migration, observability, and a production-oriented enterprise capstone.

The guiding principle is simple: modernization is not a rewrite contest. It is controlled risk reduction. Every architectural change should improve testability, security, maintainability, reliability, or delivery speed.

By XechCybersecurity

How to Use This Book

Read Part I if you need a solid CI4 foundation. Continue to Parts II–IV when you are designing a new system. Use Part V as an audit and migration playbook for CI3. Part VI turns the concepts into a production-oriented capstone.

Recommended learning loop:

- Read the concept.
- Implement the smallest working example.
- Write a test that proves the behavior.
- Review security and failure modes.
- Refactor only after behavior is observable.
- Document the decision and operational impact.

Table of Contents

PART I — THE CODEIGNITER 4 FOUNDATION

- Chapter 1 — Understanding CodeIgniter 4
- Chapter 2 — Project Structure & Namespaces
- Chapter 3 — HTTP Request Lifecycle

PART II — FROM MVC TO CLEAN ARCHITECTURE

- Chapter 4 — When MVC Is Not Enough
- Chapter 5 — Clean Architecture Principles
- Chapter 6 — Implementing Clean Architecture in CI4

PART III — DESIGN PATTERNS & ENGINEERING

- Chapter 7 — Dependency Injection & Services
- Chapter 8 — Repository Pattern
- Chapter 9 — Service Layer & Use Cases
- Chapter 10 — DTOs, Value Objects & Domain Models

PART IV — DATABASE & SECURITY

- Chapter 11 — Database Architecture
- Chapter 12 — Migrations, Seeders & Data Integrity
- Chapter 13 — Transactions, Concurrency & Failure
- Chapter 14 — Secure CodeIgniter 4
- Chapter 15 — Authentication & Authorization
- Chapter 16 — Secure API Engineering

PART V — AUDITING CI3 LEGACY

- Chapter 17 — CI3 Audit Methodology
- Chapter 18 — Detecting Architectural Smells
- Chapter 19 — CI3 to CI4 Migration
- Chapter 20 — Incremental Modernization
- Chapter 21 — Legacy Database Audit
- Chapter 22 — Legacy Authentication Audit
- Chapter 23 — Legacy File Upload Audit
- Chapter 24 — Legacy Session & CSRF Audit

PART VI — PRODUCTION & CAPSTONE

- Chapter 25 — Observability & Structured Logging
- Chapter 26 — Audit Trail Engineering

Chapter 27 — Testing Strategy

Chapter 28 — CI/CD & Deployment

Chapter 29 — Xech Enterprise System

Chapter 30 — Capstone Domain Design

Chapter 31 — Capstone Application Layer

Chapter 32 — Capstone Infrastructure Layer

Chapter 33 — Capstone Security Hardening

Chapter 34 — Capstone Migration Runbook

Chapter 35 — Production Readiness

Reference Architecture

The book uses the following practical architecture as a reference model:

Layer	Responsibility	Typical CI4 examples
Presentation	HTTP, routing, request/response mapping	Controllers, routes, filters, request DTO mapping
Application	Use-case orchestration	CreateUser, ApproveRequest, ExportAuditLog
Domain	Business rules and invariants	Entities, value objects, domain services
Infrastructure	Technical implementation	CI4 models, repositories, DB, mail, external APIs

The exact directory structure can vary. Architecture should serve the system's boundaries and team workflow rather than become a rigid folder ceremony.

PART I — THE CODEIGNITER 4 FOUNDATION

Engineering objective: understand the trade-offs, implement safely, and leave the system easier to change than you found it.

Chapter 1 — Understanding CodeIgniter 4

CodeIgniter 4 (CI4) is a modern PHP framework designed to keep web application development lightweight while still providing routing, HTTP handling, validation, database access, filters, sessions, testing, CLI tooling, and security facilities.

CI4 is not simply a cosmetic upgrade from CI3. The framework was redesigned around modern PHP practices, namespaces, Composer, PSR-oriented conventions, stronger configuration boundaries, and a more explicit application structure. The correct migration mindset is therefore: understand the target architecture first, then move behavior safely.

At the time this edition was prepared, the official CodeIgniter site lists CI4.7.4 as the current release and PHP 8.1+ as the supported baseline. Always verify the exact framework and PHP version used by your project before implementation.

Engineering rule: pin and document the framework version, PHP version, extensions, database driver, and production runtime. A reproducible environment is part of architecture, not an operational afterthought.

Practical baseline

```
composer create-project codeigniter4/appstarter my-app
cd my-app
php spark serve
```

The official documentation recommends Composer for installation because it makes dependency management and updates easier.

Chapter 2 — Project Structure & Namespaces

A healthy CI4 application separates public entry points, application code, writable runtime data, and framework dependencies. The public directory should be the web server document root; sensitive project files should not be directly exposed.

Namespaces make dependencies explicit. Instead of relying on loosely named classes, define a clear namespace for each architectural layer and import dependencies with use statements. This becomes especially valuable when the application grows beyond conventional MVC.

Recommended enterprise-oriented structure:

```
app/  
  Domain/  
  Application/  
  Infrastructure/  
  Controllers/  
  Filters/  
  Config/  
  Database/  
public/  
writable/  
tests/
```

Chapter 3 — HTTP Request Lifecycle

A request should be understood as a pipeline rather than a single controller call. The web server forwards the request to the public front controller; CI4 boots configuration and services, resolves routing, executes applicable filters, dispatches the controller, and produces an HTTP response.

Use the lifecycle to place responsibilities correctly. Authentication and access checks belong in filters or authorization services; request normalization belongs at the boundary; business rules belong in use cases/domain services; persistence belongs behind repository interfaces where Clean Architecture is applied.

Debugging becomes easier when every stage has a known responsibility. Log correlation IDs, request method/path, actor identity when appropriate, and outcome without logging secrets or sensitive payloads.

PART II — FROM MVC TO CLEAN ARCHITECTURE

Engineering objective: understand the trade-offs, implement safely, and leave the system easier to change than you found it.

Chapter 4 — When MVC Is Not Enough

MVC is a useful delivery pattern, but a controller can become a dumping ground. A typical legacy controller may validate input, query several tables, calculate business rules, send notifications, write audit records, and decide authorization—all inside one method.

Architectural smell indicators include fat controllers, models that contain unrelated business workflows, duplicated authorization logic, static calls everywhere, direct database access from views, and tests that require a complete application stack for simple business rules.

The goal is not to reject MVC. Keep MVC at the delivery boundary while moving business decisions into application and domain components.

Chapter 5 — Clean Architecture Principles

Clean Architecture organizes software around business rules and dependency direction. A practical C4 implementation can use four conceptual layers: Presentation, Application, Domain, and Infrastructure.

Presentation receives HTTP input and maps it to application commands. Application coordinates use cases. Domain contains business concepts and rules. Infrastructure implements persistence, external APIs, mail, queues, and other technical details.

The Dependency Rule is the key idea: inner business layers should not be forced to know the details of outer infrastructure. Interfaces can be owned by the inner layer and implemented by the outer layer.

Do not create layers merely for decoration. Each layer should answer a distinct question: how did the request arrive, what use case is being executed, what business rule applies, and how is data or an external capability provided?

Chapter 6 — Implementing Clean Architecture in CI4

A practical directory layout can be organized under app/ with Domain, Application, Infrastructure, and Presentation boundaries. CI4 controllers remain thin. Dependency construction can be centralized in factories or service configuration.

Example flow: UserController receives an HTTP request → validates DTO input → calls CreateUserUseCase → use case checks domain rules → repository interface persists the aggregate → controller maps the result to an HTTP response.

The architecture should remain incremental. You can introduce one use case and one repository at a time rather than rewriting the whole application.

```
interface UserRepository
{
    public function findById(UserId $id): ?User;
    public function save(User $user): void;
}

final class CreateUser
{
    public function __construct(
        private UserRepository $users
    ) {}
}
```

PART III — DESIGN PATTERNS & ENGINEERING

Engineering objective: understand the trade-offs, implement safely, and leave the system easier to change than you found it.

Chapter 7 — Dependency Injection & Services

Dependency Injection means a component receives what it needs instead of constructing every dependency internally. This reduces coupling and makes tests easier.

A service container is useful for wiring interfaces to implementations, but avoid turning it into a hidden global dependency. Prefer explicit constructor dependencies for business-critical components.

Bad: a use case calls a static database helper directly. Better: the use case receives a UserRepository interface and is unaware of the SQL implementation.

Chapter 8 — Repository Pattern

A repository abstracts persistence for a business concept. It should expose meaningful operations such as `findById`, `findByEmail`, `save`, or `deactivate` rather than leaking query-builder details into the domain.

Repositories are not mandatory for every CRUD screen. Use them when the domain has meaningful persistence behavior, when multiple data sources exist, or when isolation from infrastructure materially improves testing and maintainability.

A repository interface belongs near the consumer. The concrete CI4 implementation belongs in Infrastructure.

Repository rule of thumb

Expose business-oriented operations, not query-builder vocabulary. If a caller needs to know that a record lives in a table called `tbl_customer_legacy`, the abstraction has already leaked.

Chapter 9 — Service Layer & Use Cases

A use case represents an application capability: RegisterUser, ApproveInvoice, ChangePassword, or MigrateLegacyCustomer. Each use case should have a clear input, a clear output, and explicit failure conditions.

Keep orchestration in the application layer. Keep invariant rules in domain objects where practical. Avoid a generic 'GodService' that contains unrelated operations.

Use cases are also ideal audit boundaries because they represent meaningful business actions rather than low-level SQL events.

Chapter 10 — DTOs, Value Objects & Domain Models

DTOs carry data across boundaries without pretending to be domain objects. Value Objects represent concepts with validation and semantics, such as EmailAddress, Money, UserId, or LegacyCustomerCode.

Domain entities should protect their invariants. Do not allow every controller to mutate critical state arbitrarily.

A useful heuristic: if a value has a rule that should always be true, give that rule a home close to the value.

Practical baseline

```
composer create-project codeigniter4/appstarter my-app
cd my-app
php spark serve
```

The official documentation recommends Composer for installation because it makes dependency management and updates easier.

PART IV — DATABASE & SECURITY

Engineering objective: understand the trade-offs, implement safely, and leave the system easier to change than you found it.

Chapter 11 — Database Architecture

Separate database concerns from business decisions. CI4 models and query builders are excellent infrastructure tools, but a domain service should not need to know table names.

Use transactions around business operations that must succeed or fail atomically. Define indexes according to access patterns, enforce uniqueness in the database, and treat constraints as a second line of defense—not a replacement for application validation.

Document schema ownership and migration strategy. Production databases should never depend on an undocumented sequence of manual SQL edits.

Practical baseline

```
composer create-project codeigniter4/appstarter my-app
cd my-app
php spark serve
```

The official documentation recommends Composer for installation because it makes dependency management and updates easier.

Chapter 12 — Migrations, Seeders & Data Integrity

CI4 migrations provide a structured way to version database schema changes. Each migration represents a repeatable deployment step, while seeders can establish controlled reference or development data.

Treat migrations as code: review them, test them, and understand rollback implications. Destructive changes should be staged carefully, especially when old application versions may still run during deployment.

Never put production secrets or real personal data into development seeders.

Practical baseline

```
composer create-project codeigniter4/appstarter my-app
cd my-app
php spark serve
```

The official documentation recommends Composer for installation because it makes dependency management and updates easier.

Chapter 13 — Transactions, Concurrency & Failure

A transaction protects a unit of database work, but it does not automatically solve every concurrency problem. Decide which invariants must be enforced by unique constraints, row-level locking, optimistic versioning, or application-level coordination.

Always define failure behavior. If an order is created but the audit record fails, is the operation rolled back? If an external API call fails after the database commits, should the system retry asynchronously? These are architectural decisions, not merely error-handling details.

Practical baseline

```
composer create-project codeigniter4/appstarter my-app
cd my-app
php spark serve
```

The official documentation recommends Composer for installation because it makes dependency management and updates easier.

Chapter 14 — Secure CodeIgniter 4

Security should be designed into request boundaries, authorization, data validation, output encoding, session management, configuration, dependency management, logging, and deployment.

CI4 provides security facilities including validation, CSRF protection, filters, escaping, secure headers, query bindings/query builder, encryption services, and integration options such as CodeIgniter Shield. These mechanisms are controls, not guarantees: the application still needs correct authorization and threat modeling.

Never trust request parameters because they passed validation. Validation answers 'is this input acceptable?' Authorization answers 'is this actor allowed to perform this action?' Both are required.

Do not log passwords, session tokens, API keys, authorization headers, or full sensitive payloads.

Practical baseline

```
composer create-project codeigniter4/appstarter my-app
cd my-app
php spark serve
```

The official documentation recommends Composer for installation because it makes dependency management and updates easier.

Chapter 15 — Authentication & Authorization

Authentication identifies an actor. Authorization determines what that actor may do. Keep the distinction explicit.

Use centralized authorization policies or gates rather than repeating role checks in every controller. For high-value actions, authorize against the resource and business context, not merely a role string.

Prefer established authentication tooling and current framework guidance rather than implementing password storage, session security, or token rotation from scratch.

Practical baseline

```
composer create-project codeigniter4/appstarter my-app  
cd my-app  
php spark serve
```

The official documentation recommends Composer for installation because it makes dependency management and updates easier.

Chapter 16 — Secure API Engineering

API endpoints should define authentication, authorization, input schema, output schema, rate limiting, error semantics, and audit requirements.

Use stable error responses that do not reveal stack traces or database details. Validate JSON payloads with strict rules, enforce size limits, and avoid mass assignment by explicitly mapping request fields into DTOs.

Treat outbound HTTP calls as security-sensitive. Validate destination URLs where user input influences them, use timeouts, restrict redirects where appropriate, and defend against SSRF.

Practical baseline

```
composer create-project codeigniter4/appstarter my-app  
cd my-app  
php spark serve
```

The official documentation recommends Composer for installation because it makes dependency management and updates easier.

PART V — AUDITING CI3 LEGACY

Engineering objective: understand the trade-offs, implement safely, and leave the system easier to change than you found it.

Chapter 17 — CI3 Audit Methodology

Legacy auditing begins with inventory, not refactoring. Record PHP and CI versions, modules, routes, controllers, models, database dependencies, scheduled jobs, external integrations, authentication paths, file uploads, and deployment procedures.

Classify findings by impact: security, correctness, reliability, maintainability, performance, and operational risk. A legacy system can be ugly yet stable; do not create risk by rewriting working behavior without a reason.

Create an evidence-backed modernization backlog. Each item should have an observed problem, impact, proposed remediation, owner, and verification method.

Practical baseline

```
composer create-project codeigniter4/appstarter my-app
cd my-app
php spark serve
```

The official documentation recommends Composer for installation because it makes dependency management and updates easier.

Chapter 18 — Detecting Architectural Smells

Look for fat controllers, SQL in views, duplicated queries, business logic in model callbacks, direct superglobal access, unvalidated POST data, hard-coded credentials, mixed HTML/SQL/business logic, global state, and duplicated authorization checks.

Measure before changing. Search the codebase for repeated patterns and identify high-centrality modules—components that many workflows depend on. These are migration hotspots because they offer high leverage but also high risk.

Practical baseline

```
composer create-project codeigniter4/appstarter my-app
cd my-app
php spark serve
```

The official documentation recommends Composer for installation because it makes dependency management and updates easier.

Chapter 19 — CI3 to CI4 Migration

CI3 and CI4 differ in application structure, namespaces, configuration, routing, request handling, validation, database APIs, and security practices. A mechanical search-and-replace migration is therefore unsafe.

Use a staged approach: inventory → characterization tests → isolate legacy boundaries → establish CI4 shell → migrate one vertical slice → validate behavior → observe in production → repeat.

Where possible, preserve business behavior first and improve architecture second. This makes failures easier to diagnose.

Practical baseline

```
composer create-project codeigniter4/appstarter my-app  
cd my-app  
php spark serve
```

The official documentation recommends Composer for installation because it makes dependency management and updates easier.

Chapter 20 — Incremental Modernization

The Strangler pattern replaces a legacy capability gradually. New routes or use cases are implemented in the modern system while old paths continue to operate until traffic is migrated.

Use anti-corruption boundaries when CI3 data models do not match the modern domain. Translate legacy records into modern DTOs or domain representations instead of spreading legacy terminology throughout the new code.

Set explicit exit criteria for each migration slice: tests green, security review complete, observability available, rollback understood, and legacy dependency reduced.

```
app/  
  Domain/  
  Application/  
  Infrastructure/  
  Controllers/  
  Filters/  
  Config/  
  Database/  
public/  
writable/  
tests/
```

Chapter 21 — Legacy Database Audit

Map tables to business concepts. Identify primary keys, foreign keys, nullability, duplicate records, orphaned rows, implicit enums, date formats, and columns whose names no longer match their meaning.

Do not 'clean' data silently. Profile first, define remediation rules, run migrations in a controlled environment, and preserve an auditable record of transformations.

```
app/  
  Domain/  
  Application/  
  Infrastructure/  
  Controllers/  
  Filters/  
  Config/  
  Database/  
public/  
writable/  
tests/
```

Chapter 22 — Legacy Authentication Audit

Authentication is a high-risk migration boundary. Inventory password hashing, session storage, remember-me behavior, logout semantics, password reset flows, account lockout, MFA, role mapping, and privileged accounts.

Never migrate plaintext passwords. If legacy hashes are weak or incompatible, use a controlled reauthentication strategy or a safe hash-upgrade mechanism supported by the authentication design.

```
app/  
  Domain/  
  Application/  
  Infrastructure/  
  Controllers/  
  Filters/  
  Config/  
  Database/  
public/  
writable/  
tests/
```

Chapter 23 — Legacy File Upload Audit

Review upload size limits, MIME validation, extension validation, storage location, filename generation, permissions, malware scanning strategy, and download authorization.

Do not place user-controlled uploads in a directly executable web directory. Validate both metadata and content where appropriate, generate server-side filenames, and use authorization checks before serving private files.

```
app/  
  Domain/  
  Application/  
  Infrastructure/  
  Controllers/  
  Filters/  
  Config/  
  Database/  
public/  
writable/  
tests/
```

Chapter 24 — Legacy Session & CSRF Audit

Map where sessions are created, what data they contain, when IDs rotate, and how logout invalidates them. Verify cookie security attributes and deployment assumptions.

Review every state-changing browser request for CSRF protection. CI4 provides CSRF protection as a filter; configuration and route behavior must still be tested for the actual application.

```
app/  
  Domain/  
  Application/  
  Infrastructure/  
  Controllers/  
  Filters/  
  Config/  
  Database/  
public/  
writable/  
tests/
```

PART VI — PRODUCTION & CAPSTONE

Engineering objective: understand the trade-offs, implement safely, and leave the system easier to change than you found it.

Chapter 25 — Observability & Structured Logging

Production readiness requires visibility. Record application events in structured form with timestamps, severity, correlation IDs, and useful business identifiers that are safe to log.

Separate operational logs from audit logs. An operational log explains what the system did internally; an audit trail explains who performed a business action, what object was affected, when it happened, and the outcome.

```
app/  
  Domain/  
  Application/  
  Infrastructure/  
  Controllers/  
  Filters/  
  Config/  
  Database/  
public/  
writable/  
tests/
```

Chapter 26 — Audit Trail Engineering

An audit record should answer: actor, action, target, timestamp, outcome, source/context, and relevant change summary. Avoid storing unnecessary sensitive values.

Audit records should be append-oriented and access-controlled. If compliance or forensic requirements exist, define retention, immutability, export, and monitoring requirements explicitly.

```
app/  
  Domain/  
  Application/  
  Infrastructure/  
  Controllers/  
  Filters/  
  Config/  
  Database/  
public/  
writable/  
tests/
```

Chapter 27 — Testing Strategy

Use a test pyramid: domain/unit tests for business rules, application tests for use cases, integration tests for repositories and external adapters, and HTTP/feature tests for critical user journeys.

Characterization tests are especially important during legacy migration. They capture existing behavior before refactoring. Not every legacy behavior deserves preservation, so pair tests with business and security review.

```
app/  
  Domain/  
  Application/  
  Infrastructure/  
  Controllers/  
  Filters/  
  Config/  
  Database/  
public/  
writable/  
tests/
```

Chapter 28 — CI/CD & Deployment

A production deployment should be repeatable. Build dependencies deterministically, validate configuration, run migrations intentionally, execute automated tests, collect artifacts, and define rollback or forward-fix procedures.

Keep secrets outside source control. Use environment-specific configuration, least-privilege service accounts, and protected deployment credentials.

```
app/  
  Domain/  
  Application/  
  Infrastructure/  
  Controllers/  
  Filters/  
  Config/  
  Database/  
public/  
writable/  
tests/
```

Chapter 29 — Xech Enterprise System

The capstone is the Xech Enterprise Management System: a realistic management application rather than a toy CRUD project.

Core goals: RBAC, secure headers, audit logging, clean separation of Domain/Application/Infrastructure, repository interfaces, DTOs, value objects, and a simulated CI3-to-CI4 migration path.

Suggested modules: authentication, users, roles, departments, assets, requests/tickets, approvals, audit logs, reports, and system settings. Every module should expose a clear use-case boundary.

```
app/  
  Domain/  
  Application/  
  Infrastructure/  
  Controllers/  
  Filters/  
  Config/  
  Database/  
public/  
writable/  
tests/
```

Chapter 30 — Capstone Domain Design

Start with business language. Example aggregates include User, Asset, Ticket, Approval, and AuditEntry. Define invariants before designing controllers.

Example invariant: a ticket cannot transition from CLOSED directly to IN_PROGRESS unless an explicit reopen use case exists. The rule belongs to the domain behavior, not a JavaScript button or controller branch.

Chapter 31 — Capstone Application Layer

Create use cases such as `CreateTicket`, `AssignTicket`, `CloseTicket`, `ApproveRequest`, and `ExportAuditLog`. Each use case should validate permissions through an authorization boundary and coordinate repositories and domain rules.

Return explicit result objects or domain exceptions that the presentation layer can translate into HTTP responses. Do not expose infrastructure exceptions directly to users.

Chapter 32 — Capstone Infrastructure Layer

Implement repository interfaces using C14 models/query builder in Infrastructure. Keep SQL mapping details outside the domain.

External integrations such as email, object storage, notification gateways, or legacy databases should be adapters. This keeps the application layer stable when vendors or protocols change.

Chapter 33 — Capstone Security Hardening

Threat-model privileged operations, file uploads, authentication, exports, search endpoints, and administrative functions. Apply CSRF where applicable, strict validation, output encoding, secure headers, least privilege, rate controls, and safe error handling.

Security testing should be performed only against systems for which explicit authorization exists. The book's methodology is intended for defensive engineering and authorized assessment.

Chapter 34 — Capstone Migration Runbook

Phase 1: freeze uncontrolled schema changes and inventory CI3. Phase 2: add characterization tests. Phase 3: create CI4 foundation and observability. Phase 4: migrate one low-risk vertical slice. Phase 5: migrate authentication and high-value workflows with extra controls. Phase 6: retire legacy endpoints and dependencies.

At every phase maintain rollback notes, database backups, monitoring, and acceptance criteria. A migration is successful only when users can complete their work safely and the organization can operate the system afterward.

Migration gate checklist

- Characterization tests pass
- Security review completed
- Database migration verified
- Observability available
- Rollback/forward-fix documented
- Business owner acceptance obtained

Chapter 35 — Production Readiness

Before release, verify application configuration, HTTPS, database connectivity, migrations, cache/session behavior, file permissions, logging, monitoring, backups, restore tests, dependency status, security controls, and operational ownership.

Run a final readiness review across four dimensions: correctness, security, reliability, and maintainability. The goal is not a perfect codebase. The goal is a system whose risks are known, controlled, observable, and supportable.

Production readiness gates

- Correctness
- Security
- Reliability
- Observability
- Backup and restore
- Operational ownership
- Dependency patch plan

Appendix A — CI3 Audit Worksheet

Use this worksheet when assessing an inherited CI3 application. Record facts before recommendations.

- Runtime: PHP version, web server, database, extensions.
- Framework: CI3 version, custom core overrides, third-party libraries.
- Routes: public, authenticated, privileged, API.
- Authentication: hashes, sessions, reset, logout, MFA.
- Data: tables, constraints, sensitive columns, orphaned records.
- Files: upload paths, download authorization, permissions.
- Operations: cron, queues, backups, deployment, monitoring.
- Security: CSRF, XSS, injection, access control, SSRF, secrets.
- Business criticality: revenue, customer, operational, compliance impact.

Appendix B — Engineering Principles

1. Make boundaries explicit.
2. Prefer small use cases over giant services.
3. Keep business rules independent of frameworks where practical.
4. Validate at boundaries and enforce invariants in the domain.
5. Use database constraints as durable integrity controls.
6. Treat authorization as a first-class business rule.
7. Log for operations and audit for accountability.
8. Migrate incrementally and measure behavior.
9. Never trade production safety for architectural purity.
10. Document decisions that future maintainers will otherwise have to rediscover.

Official Technical References

CodeIgniter 4 User Guide — installation, application structure, controllers, databases, validation, security, testing, CLI, and deployment.

CodeIgniter 4 official release information — use the current release and upgrade notes applicable to your project.

OWASP Top 10 and OWASP application-security guidance — use as a threat-modeling and verification reference.

END OF BOOK