BEYOND ZERO SERIES

Web Development Beyond Zero

AN INTERMEDIATE GUIDE TO

ARCHITECTURE, FRAMEWORKS, AND REAL-WORLD DELIVERY

LANCASTER SOLUTIONS LLC

WEB DEVELOPMENT

BEYOND ZERO

Intermediate Architecture, Framework Strategy, Security, Reliability, and Delivery

Expanded Professional Edition

Architecture • maintainability • testing • security • performance • deployment • guided labs • production capstone

Curtis Wayne Lancaster II

Published by Lancaster Solutions LLC

Beyond Zero Series • Practical technology education for real-world builders

About This Expanded Edition

> **AUDIENCE:** Readers who completed Web Development From Zero or can build a small full-stack application and explain common rendering, backend, database, and deployment choices.

This revised edition moves from comparing architectures to designing and operating one: requirements, boundaries, contracts, data, identity, events, delivery, observability, reliability, security, and framework governance.

Technical baseline

Current web standards, official framework guidance, OWASP security practices, and architecture principles designed to remain useful beyond individual framework releases.

What “intermediate” means in this series

You can already write small working programs or pages. This book focuses on what happens when the work must be changed safely, tested repeatedly, deployed predictably, and understood by someone other than its original author.

The objective is not to memorize every feature. It is to build judgment: choosing boundaries, stating contracts, designing failure behavior, gathering evidence, and balancing complexity against the real needs of a project.

Examples are intentionally compact. The labs, projects, and capstone ask you to supply missing production details such as validation, logging, authorization, migration, accessibility, or rollback. Those omissions are part of the learning design rather than permission to ignore them.

How to use the book

- Read the chapter thesis first and write down what you expect the technique to improve.
- Run or reproduce the example in a small, disposable project before transplanting it into a larger codebase.
- Complete the guided lab with tests and notes. Then attempt the independent challenge without copying the lab structure mechanically.
- Keep an engineering notebook containing assumptions, decisions, failed experiments, measurements, and follow-up work.
- Treat the capstone as a portfolio-quality delivery: source code is only one deliverable alongside evidence, documentation, and an operational plan.

Publication and educational use

Contents and Learning Roadmap

The roadmap is organized into four stages. Each stage introduces decisions that the next stage assumes you can explain and verify.

Stage	Focus	Coverage	Core topics
Part 1	Structure, Models, and Contracts	Chapters 1–4	Requirements, Domain Modeling, and Architecture Drivers, Architecture Decision Records and Evolutionary Design, Modular Monoliths and Service Boundaries, Frontend Application Architecture
Part 2	Boundaries, Data, and Collaboration	Chapters 5–8	Backend Application Architecture, API Styles, Contracts, and Compatibility, Data Architecture, Transactions, and Consistency, Identity, Authorization, Sessions, and Multi-Tenancy
Part 3	Reliability, Scale, and Verification	Chapters 9–12	Events, Queues, Workflows, and Backpressure, Caching, CDN, and Performance Architecture, Testing Strategy and Release Evidence, CI/CD, Infrastructure, and Safe Database Change
Part 4	Delivery, Operations, and Evolution	Chapters 13–16	Containers, Serverless, Cloud, and Platform Tradeoffs, Observability, Reliability, and Incident Response, Security Architecture and Threat Modeling, Framework Selection, Migration, and Technology Governance

Projects, capstone, and references

PART I

Structure, Models, and Contracts

Chapters 1–4

CHAPTER 1
Requirements, Domain Modeling, and Architecture Drivers

> **CHAPTER THESIS:** Intermediate web development begins by turning product goals, risks, data, scale, compliance, and team constraints into architecture drivers before selecting frameworks.

Learning outcomes

- Explain how user journeys shapes boundaries, testability, and failure behavior in a maintainable implementation.
- Apply domain language in a focused implementation and identify the contract it creates for callers.
- Compare an implementation that uses quality attributes with a simpler alternative, then justify the added complexity.
- Evaluate the operational and maintenance consequences of constraints under realistic change and failure.
- Evaluate the architecture Driver Record example, identify assumptions about user journeys and domain language, and justify one viable alternative.
- For “Requirements, Domain Modeling, and Architecture Drivers”, produce evidence using architecture decision records, contract examples, threat models, release evidence, service-level indicators, and incident exercises; state what the evidence proves and what remains uncertain.

The intermediate shift

The intermediate challenge in “Requirements, Domain Modeling, and Architecture Drivers” is not learning one more API; it is deciding where the behavior belongs and what must remain true when the surrounding system changes.

User journeys, Domain language, Quality attributes, and Constraints reinforce one another, but they optimize different risks. The chapter asks you to decide which concern owns each rule, where translation occurs, and how the design behaves when one assumption changes.

Credible evidence for “Requirements, Domain Modeling, and Architecture Drivers” includes architecture decision records, contract examples, threat models, release evidence, service-level indicators, and incident exercises. Capture the setup and expected result so another reader can reproduce the claim instead of trusting a successful demonstration.

User journeys

Architecture exists to support specific tasks such as registration, search, checkout, reporting, or collaboration.

Implementation guidance. Write critical journeys with success, failure, latency, availability, accessibility, and data requirements.

For User journeys, the design payoff is controlled change: callers depend on the contract while the implementation can evolve behind it.

Domain language

A shared vocabulary reduces translation errors between stakeholders, UI, code, and database.

Implementation guidance. Model entities, values, events, commands, and policies using the terms people use to operate the business.

Operationally, Domain language must define failure ownership, retained context, retry safety, and the signal shown to users or operators.

Quality attributes

Performance, security, reliability, maintainability, portability, and cost can conflict.

Implementation guidance. Rank them per journey and attach measurable scenarios rather than saying the system must be fast and secure.

Verify Quality attributes through its narrowest public contract, then add adversarial cases that exercise malformed input and dependency failure.

Constraints

Existing platforms, skills, contracts, budgets, laws, and deadlines narrow choices.

Implementation guidance. Record hard constraints separately from preferences and revisit assumptions when evidence changes.

For maintenance, Constraints should communicate ownership through names, types, modules, and documentation rather than institutional memory.

Worked pattern

ARCHITECTURE DRIVER RECORD

```
Journey: Create a support request
Availability: 99.9% monthly target
Latency: 95% of accepted requests acknowledged within 800 ms
Security: authenticated tenant user; attachment scanning required
Consistency: request and audit record commit together
```

```
Recovery: no more than 15 minutes of accepted data loss
Accessibility: keyboard and screen-reader completion required
```

Walkthrough

1. The record connects technical choices to a user outcome.
2. Targets are measurable enough to test or monitor.
3. The consistency requirement affects transaction and messaging design.

Design decisions to notice

- User journeys: What responsibility is being made explicit, and which caller or layer should remain unaware of its implementation details?
- Domain language: What responsibility is being made explicit, and which caller or layer should remain unaware of its implementation details?
- Quality attributes: What responsibility is being made explicit, and which caller or layer should remain unaware of its implementation details?
- Constraints: What responsibility is being made explicit, and which caller or layer should remain unaware of its implementation details?
- In the architecture Driver Record, which decisions express the policy described in “Requirements, Domain Modeling, and Architecture Drivers”, and which are replaceable mechanisms behind the boundary?
- For “Requirements, Domain Modeling, and Architecture Drivers”, which guarantees are supplied by web protocols, application platforms, data systems, and delivery infrastructure, and which still require explicit validation, cleanup, monitoring, or documentation?

Failure modes and diagnostic thinking

- Choosing React, microservices, or Kubernetes before defining the product problem.
- Treating every page as equally critical.
- Using vague requirements that cannot become tests or alerts.
- Ignoring the team's ability to operate the selected architecture.

> **DIAGNOSTIC EXERCISE:** Reproduce this risk in the smallest useful example: Choosing React, microservices, or Kubernetes before defining the product problem. Identify the first observable symptom, the earliest detection boundary, one preventive control, and one operational signal that would distinguish recurrence from a different failure.

Guided lab

> **OBJECTIVE:** Create architecture drivers for registration, search, payment or submission, reporting, and administration.

Phase 1: Clarify the contract

Turn the objective into acceptance criteria: Create architecture drivers for registration, search, payment or submission, reporting, and administration.

Name trusted and untrusted inputs, explicit non-goals, and the result a reviewer can observe.

Phase 2: Create a baseline

Capture a baseline for “Requirements, Domain Modeling, and Architecture Drivers” using architecture decision records, contract examples, threat models, release evidence, service-level indicators, and incident exercises. Preserve the command, fixture, environment, or screenshot needed to repeat it.

Phase 3: Implement the smallest safe change

Implement the smallest coherent change that advances the objective. Keep user journeys behind a named boundary and verify each increment before adding the next.

Phase 4: Verify normal and hostile conditions

Exercise the normal case plus malformed input, boundary values, and a realistic failure involving domain language. Include cancellation, timeout, rollback, fallback, or recovery where the chapter makes it relevant.

Phase 5: Review and communicate

Review the evidence with the objective in view. Record the chosen design, the rejected alternative, remaining risk, and the signal that would justify revisiting quality attributes.

Independent challenge

Model the domain and identify aggregate and transaction boundaries.

> **CONSTRAINT:** Complete the independent challenge with a different service boundary from the worked pattern. Explain how the change affects failure behavior, testing evidence, and maintenance cost.

Stretch challenge

Prioritize conflicting quality attributes and document accepted tradeoffs.

Chapter review

1. What problem does user journeys solve, and what new complexity can it introduce when overused?
2. What problem does domain language solve, and what new complexity can it introduce when overused?
3. What problem does quality attributes solve, and what new complexity can it introduce when overused?
4. What problem does constraints solve, and what new complexity can it introduce when overused?

5. Which evidence would demonstrate that user journeys remains correct under failure, and which part still requires representative measurement or human review?
6. Under what project constraints would a simpler alternative to constraints be the more responsible choice?

Definition of done

- The objective is demonstrated for the normal path and failure cases relevant to user journeys.
- The contract around domain language is explicit, smaller than its implementation, and exercised by evidence.
- The verification does not depend on hidden secrets, machine-specific paths, or uncontrolled state while testing quality attributes.
- A reviewer can trace the decision about constraints from requirement to implementation and evidence.
- Known limitations, operational signals, and follow-up work for “Requirements, Domain Modeling, and Architecture Drivers” are recorded with an owner or review trigger.