universidade de aveiro

deti departamento de eletrónica, telecomunicações e informática

Software Components

Software Engineering

2025/2026

Outline

- Transform requirements into software components
- Horizontal vs Vertical Slicing
- Requirements and software components
- Problem-oriented refactoring

User story defined. What's next?

Mapping Requirements to Architecture

- Each user story identifies a behavior or feature.
- That feature must be implemented by one or more software components.

Granularity Alignment

- Simple stories may map directly to a single component.
- Complex stories may require coordination across several components.

Traceability

- Maintaining links between stories and components ensures that:
  - All requirements are covered.
  - Code changes can be traced back to user needs.

What should we do if a user story is too large?

INVEST

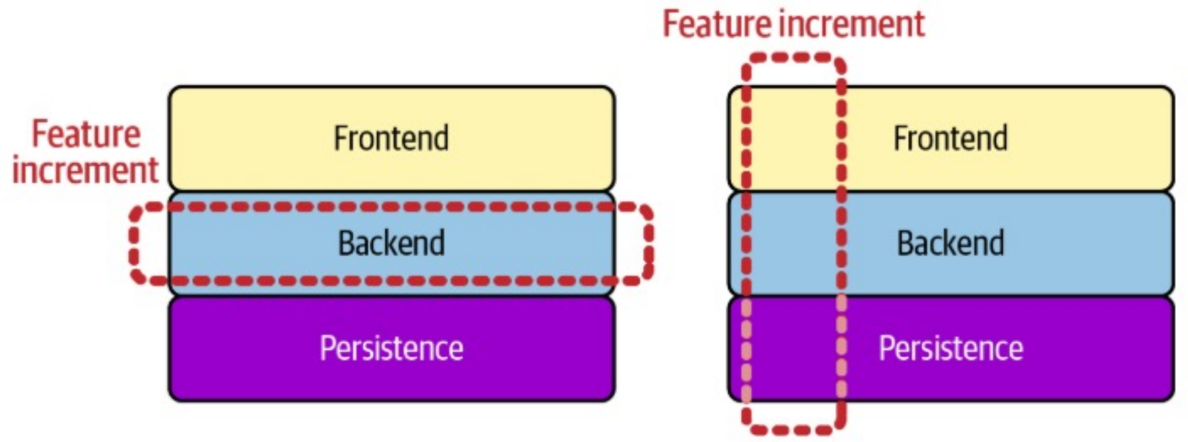
- **Independent**
  - Deliverable in isolation, minimizing dependencies across stories.
- **Negotiable**
  - Story details and priority should remain flexible and open to discussion.
- **Valuable**
  - Each story must provide clear value to the user or customer on its own.
- **Estimable**
  - Story should be clear and small enough to allow reasonable estimation.
- **Small**
  - Kept to a manageable size, typically fitting within a single iteration.
- **Testable**
  - Must add externally verifiable behavior, with acceptance criteria guiding validation.

This is how a user story should look like!

Horizontal vs Vertical Slicing

Horizontal vs Vertical Slicing

- Large initiatives, often called *epics (*)*, are too extensive for a single iteration.
- Subdividing epics into smaller, manageable pieces is essential.
- Benefits:
  - Better visibility of progress
  - Early feedback opportunities
  - Incremental delivery of value



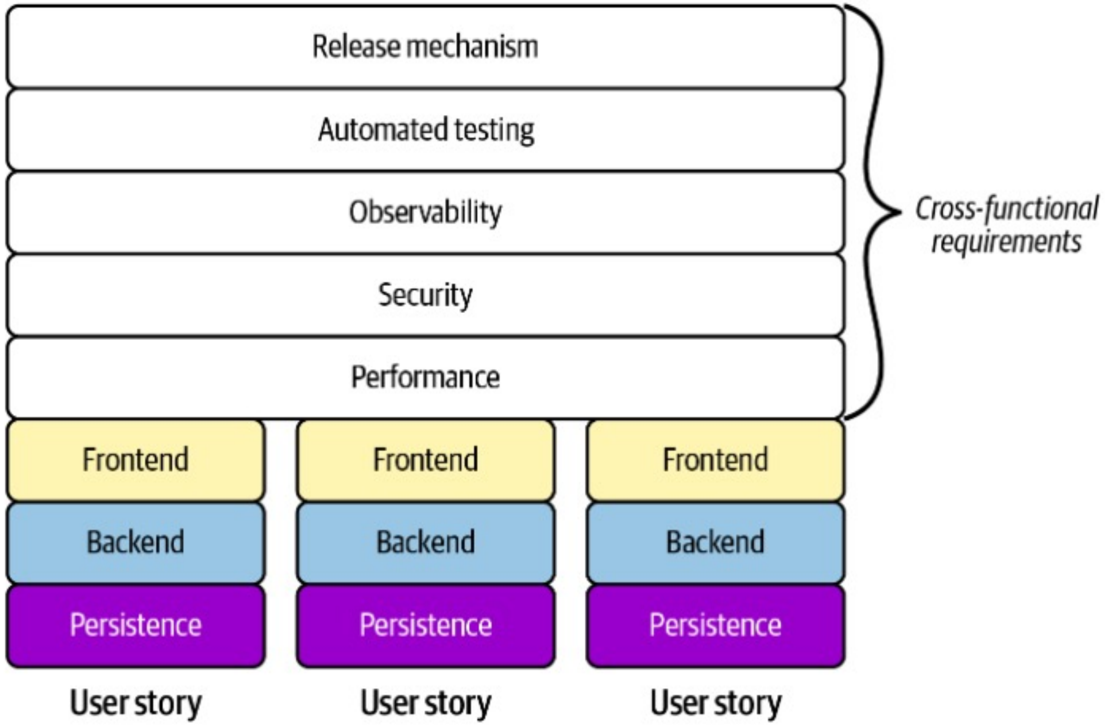
(*) An epic is a large body of work that cannot be delivered within a single iteration.
Typically represents a major feature, capability, or business initiative.
Requires decomposition into smaller user stories or increments.

Horizontal Slicing

- Work is divided along technical layers of the system (e.g., backend, frontend, database).
- Developers focus only on one area of the architecture at a time.
- Leads to siloed work; integration issues may arise later.
- No value can be delivered to the user until all layers are completed.
- The result of horizontal slicing is better described as “tasks,” since they represent technical steps rather than user value.

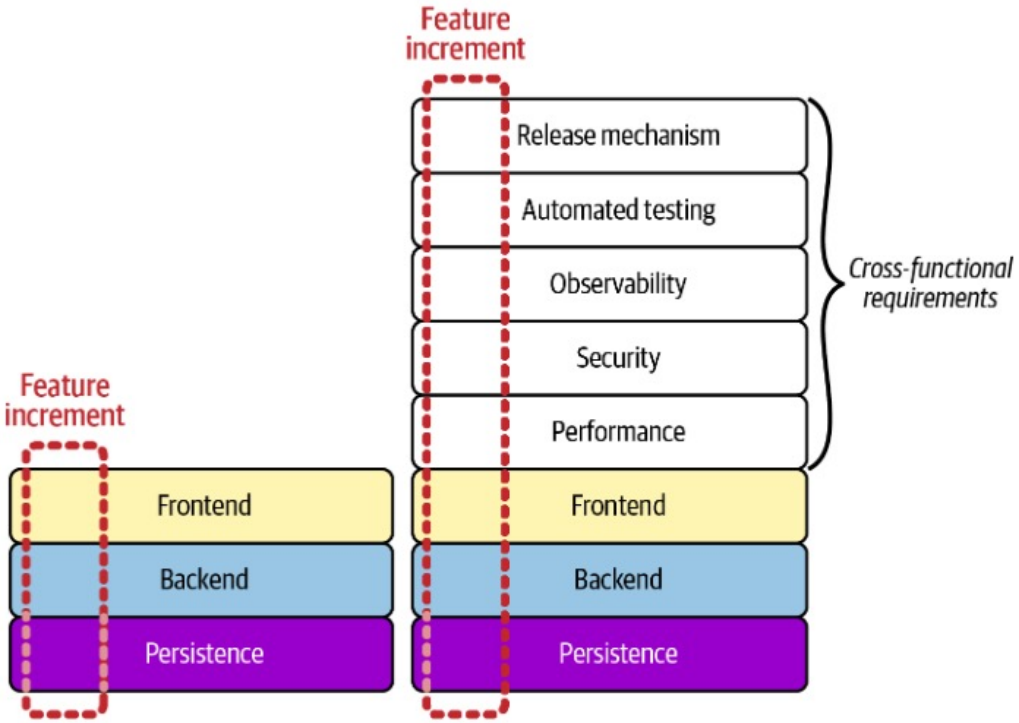
Vertical Slicing (1/2)

- Backlogs must support incremental, frequent deployment.
- Vertical slicing breaks features into small, deployable units.
- Provides finer control over code updates and reduces deployment risks.
- Focuses primarily on functional aspects of a system.



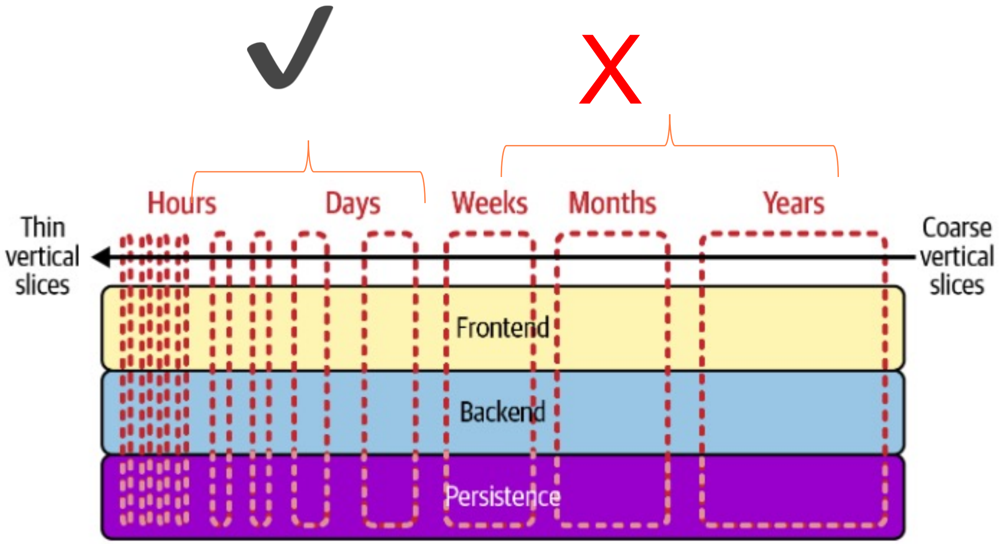
Vertical Slicing (2/2)

- CFRs (Cross Functional Requirements) are not tied to a single feature but may be impacted by any change.
- Every deployment can affect how well the system meets CFRs.
- Key CFRs for safe continuous deployment:
  - Deployability
  - Testability
  - Observability
  - Security
  - Performance
- In traditional setups, CFRs were often postponed, treated as "icing."
- Delaying them increases cost, complexity, and risks neglect by stakeholders.
- In continuous deployment, CFRs must be addressed continuously alongside functional slices.



Example of Vertical Slicing

- Consider a feature:
  - **“User can view purchase history.”**
- Possible increments:
  - Retrieve and store transaction data.
  - Display basic list of past purchases.
  - Add filtering and sorting options.
  - Provide export to file (e.g., PDF, CSV).
- Each increment delivers functional value and can be deployed independently.
  - Slicing epics into smaller increments enables incremental, continuous delivery.
  - Vertical slicing ensures that every increment is functional and deployable.



Case study

Case Study: Groceroo

- Fictional company offering at-home grocery delivery
- Customers order through a web application
- Orders fulfilled by personal shopping assistants
- Stores manage inventory through a partner portal

- **Goal: increase average order size via new feature**

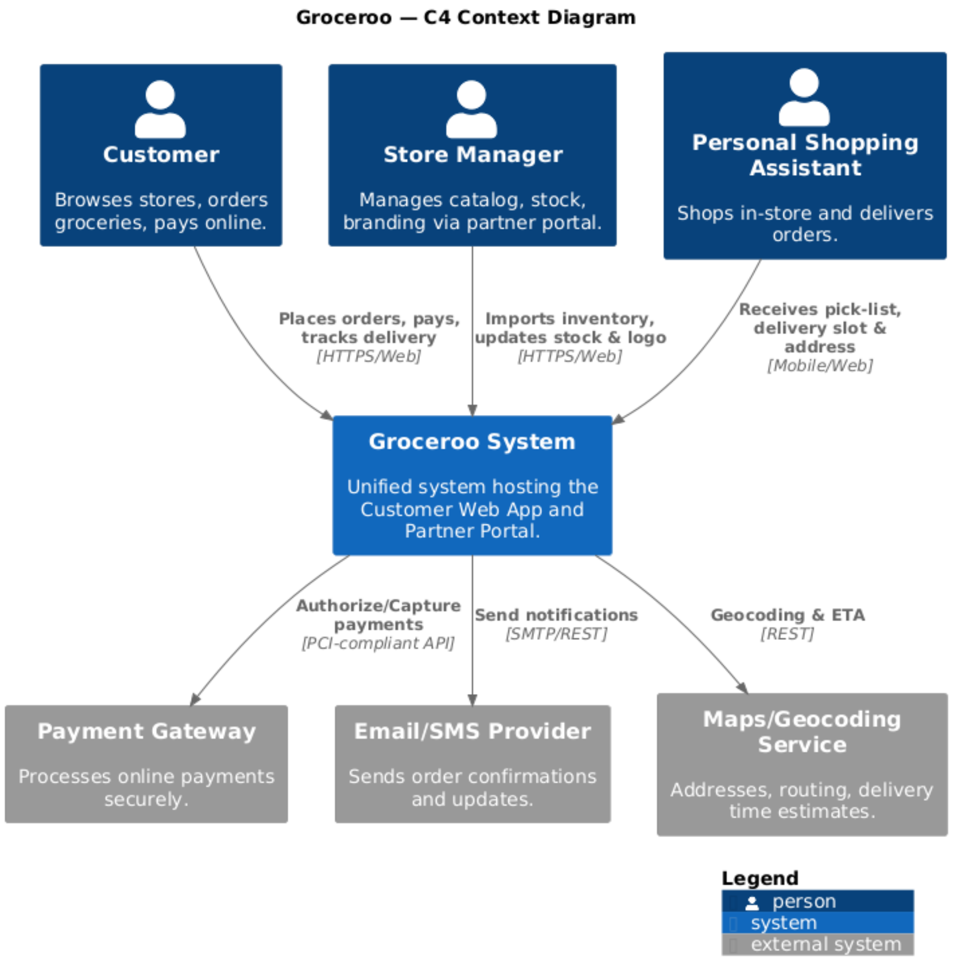
System Overview

- **Customer Web App**
  - Store selection
  - Cart management
  - Online payment & checkout
- **Partner Portal**
  - Inventory import/update
  - Store branding (logo, details)
  - Stock management
- Both modules are part of the **same system**

Case Study: Groceroo

- Fictional company offering at-home grocery delivery
- Customers order through a web application
- Orders fulfilled by personal shopping assistants
- Stores manage inventory through a partner portal

- **Goal: increase average order size via new feature**

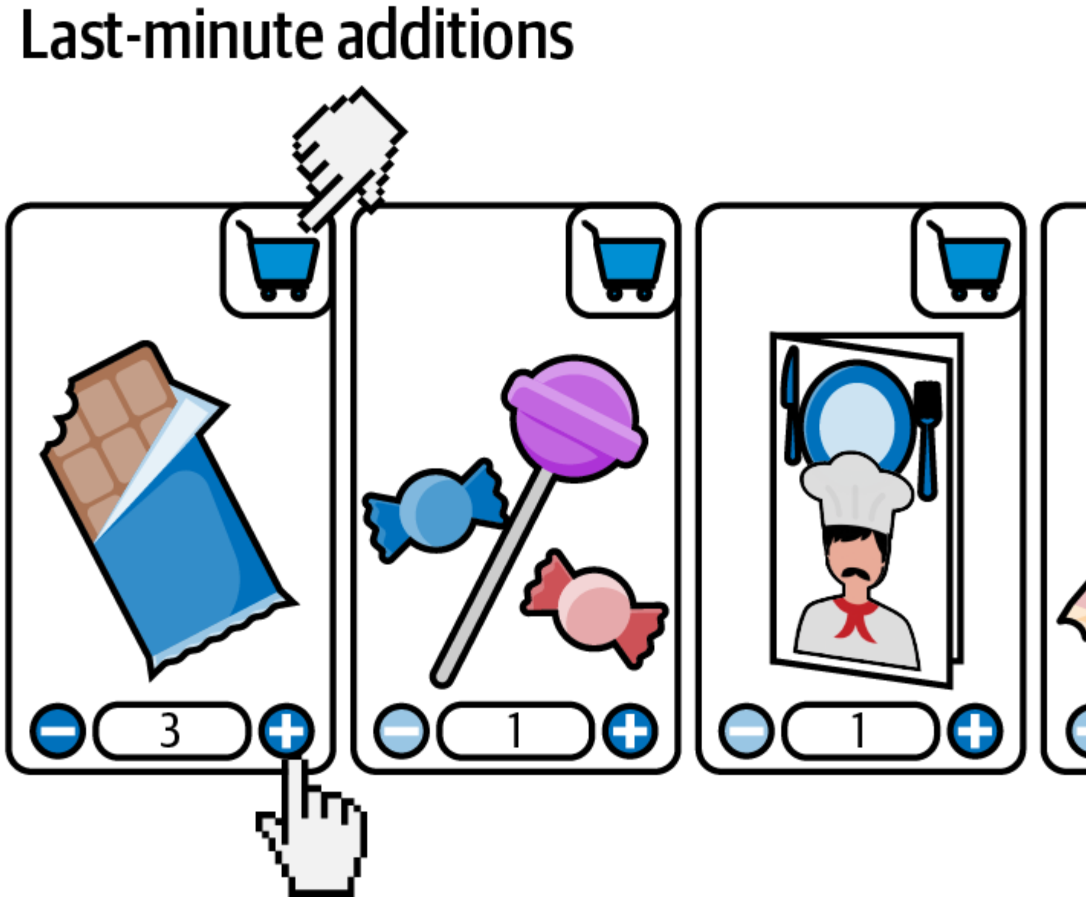

Groceroo — C4 Context Diagram
Customer
Browses stores, orders groceries, pays online.
Store Manager
Manages catalog, stock, branding via partner portal.
Personal Shopping Assistant
Shops in-store and delivers orders.
Places orders, pays, tracks delivery [HTTPS/Web]
Imports inventory, updates stock & logo [HTTPS/Web]
Receives pick-list, delivery slot & address [Mobile/Web]
Groceroo System
Unified system hosting the Customer Web App and Partner Portal.
Authorize/Capture payments [PCI-compliant API]
Send notifications [SMTP/REST]
Geocoding & ETA [REST]
Payment Gateway
Processes online payments securely.
Email/SMS Provider
Sends order confirmations and updates.
Maps/Geocoding Service
Addresses, routing, delivery time estimates.
Legend
person
system
external system

CS: Groceroo | “Last-Minute Items”

- Groceroo wanted to add a "Last-Minute Items" carousel to their checkout.

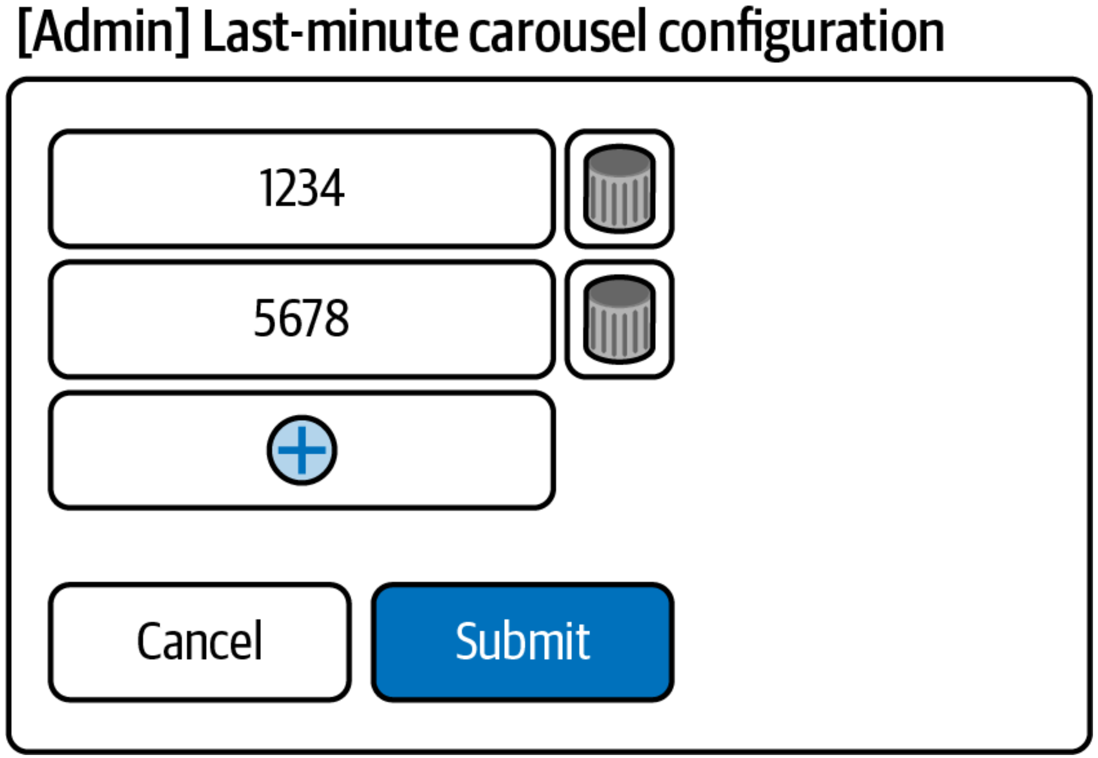
CS: Groceroo | “Last-Minute Items”

- Requirements - User Interface
  - Goal: increase average order size via an on-store carousel
  - UI: scrollable product cards with image, price, qty selector, Add to Cart
  - Scope: per-store ranking and ordering
  - Ranking: by # of times added to cart by other users



CS: Groceroo | “Last-Minute Items”

- Requirements - Admin
  - New section in Shop Manager Portal
  - Purpose: stores curate products shown just before checkout
  - Provides a simple form to edit a list of product IDs (SKUs)
  - Items configured here will appear in the checkout page for upselling
  - Supports store-specific customization and flexibility



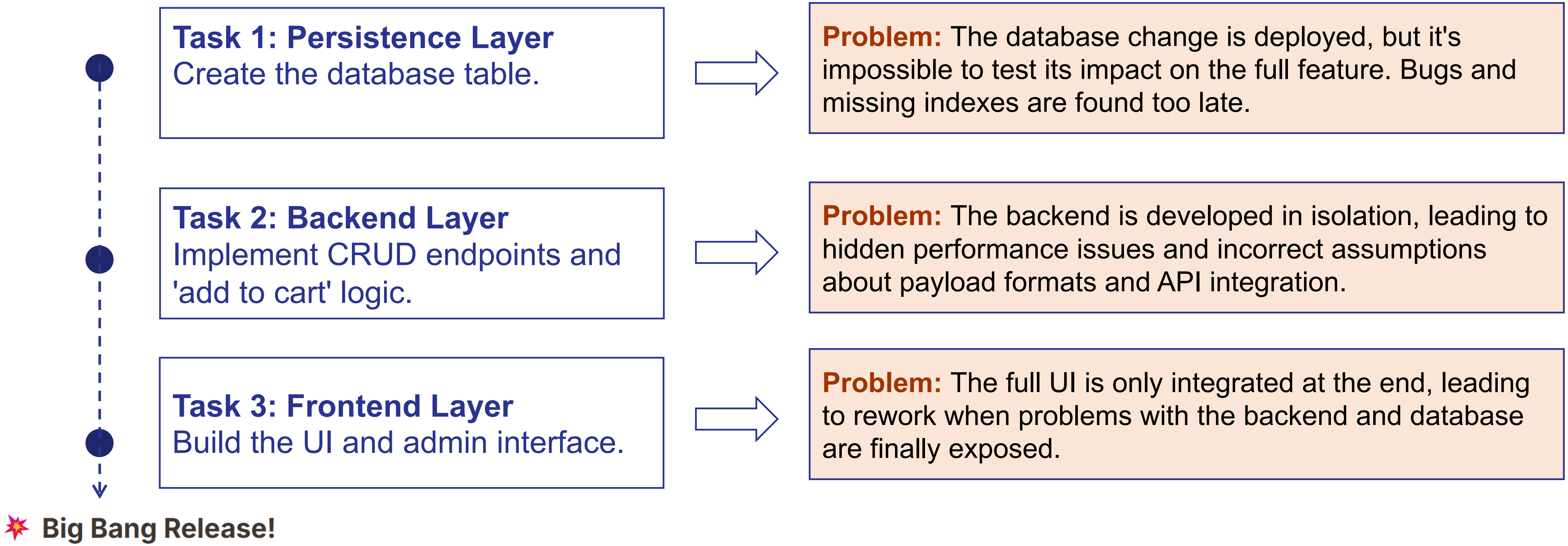
[Admin] Last-minute carousel configuration

1234

5678

CS: Groceroo | “Last-Minute Items”

Implementing with Horizontal Slicing



Task 1: Persistence Layer
Create the database table.

Problem: The database change is deployed, but it's impossible to test its impact on the full feature. Bugs and missing indexes are found too late.

Task 2: Backend Layer
Implement CRUD endpoints and 'add to cart' logic.

Problem: The backend is developed in isolation, leading to hidden performance issues and incorrect assumptions about payload formats and API integration.

Task 3: Frontend Layer
Build the UI and admin interface.

Problem: The full UI is only integrated at the end, leading to rework when problems with the backend and database are finally exposed.

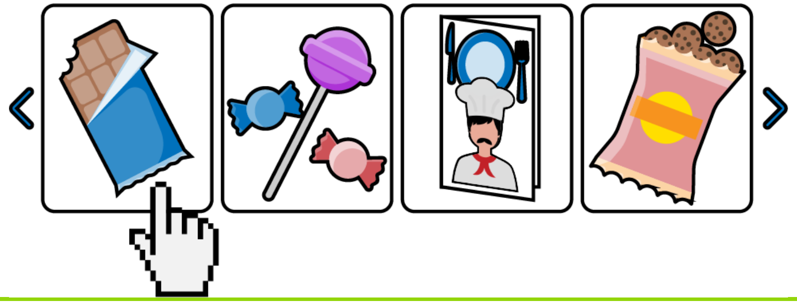
Big Bang Release!

CS: Groceroo | “Last-Minute Items”

Implementing with Vertical Slicing

- **User Story 1: Simple Carousel (MVP)**
  - Display a basic carousel of last-minute items (no frills)
  - Exclude: add-to-cart, quantity selector, ranking logic, admin form
  - Items hardcoded for one/two stores (with marketing input)
  - Users click item → detail page → add to cart

Last-minute additions



Summary

As a user

I want to see a selection of last-minute possible purchases before I complete my order

So that I can quickly add anything I might have forgotten

Acceptance criteria

Given I have items in the basket; **And** the current shop has last-minute purchases configured;

When I go to checkout; **Then** I should see a carousel with some last-minute purchases for the current shop; **Given** I have items in the basket;

And the current shop does not have any last-minute purchases configured

When I go to checkout; **Then** I should see the normal checkout page without any extra carousel

CS: Groceroo | “Last-Minute Items”

Implementing with Vertical Slicing

- **User Story 1: Simple Carousel (MVP)**
  - Display a basic carousel of last-minute items (no frills)
  - Exclude: add-to-cart, quantity selector, ranking logic, admin form
  - Items hardcoded for one/two stores (with marketing input)
  - Users click item → detail page → add to cart

Last-minute additions

Summary

As a user

I want to see a selection of last-minute possible purchases before I complete my order

So that I can quickly add anything I might have forgotten

Acceptance criteria

Given I have items in the basket; **And** the current shop has last-minute purchases configured;

When I go to checkout; **Then** I should see a carousel with some last-minute purchases for the current shop;

Given I have items in the basket; **And** the current shop does not have any last-minute purchases configured

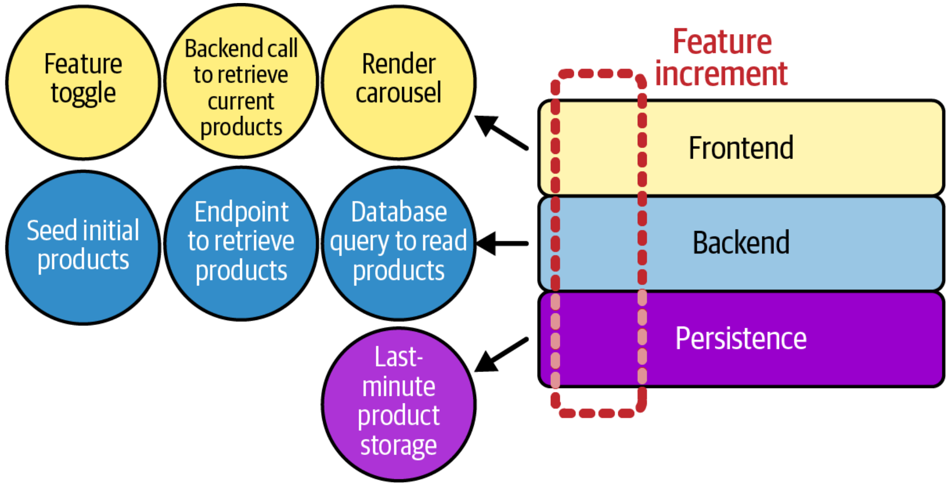
When I go to checkout; **Then** I should see the normal checkout page without any extra carousel

CS: Groceroo | “Last-Minute Items”

Implementing with Vertical Slicing

User Story 1: Simple Carousel (MVP)

- Display a basic carousel of last-minute items (no frills)
- Exclude: add-to-cart, quantity selector, ranking logic, admin form
- Items hardcoded for one/two stores (with marketing input)
- Users click item → detail page → add to cart

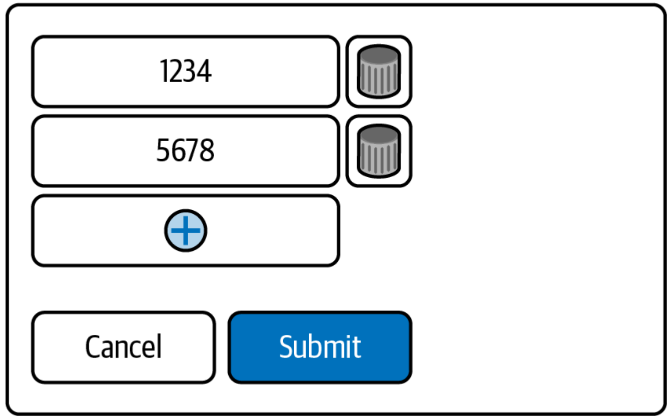

Feature toggle
Backend call to retrieve current products
Render carousel
Seed initial products
Endpoint to retrieve products
Database query to read products
Last-minute product storage
Feature increment
Frontend
Backend
Persistence

CS: Groceroo | “Last-Minute Items”

Implementing with Vertical Slicing

- **User Story 2: Configurable Carousel**
  - Move beyond hardcoded items
  - Database changes not sustainable long-term
  - Provide admin interface for partners
  - Partners self-manage their “last-minute items”

[Admin] Last-minute carousel configuration



Summary

As a store manager

I want to configure the last-minute items for my store

So that I can update them by myself, without having to wait for Groceroo’s team

Acceptance criteria

Given I am logged in to the admin portal **When** I view my store’s dashboard

Then I should see an extra section called “last-minute items” **And** it should contain a form where I can add and remove products

Given I am logged in to the admin portal; **And** I have added products to the last-minute items list; **When** I submit the last-minute items form; **Then** I should see those products in the checkout page for my store

Given I am logged in to the admin portal; **And** I have removed products from the last-minute items list;

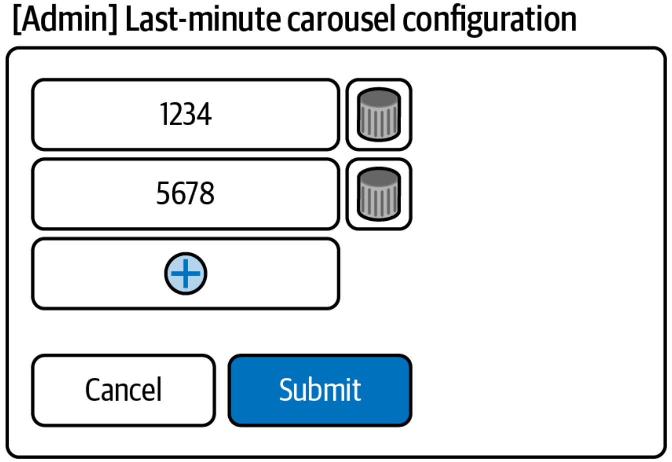
When I submit the last-minute items form;

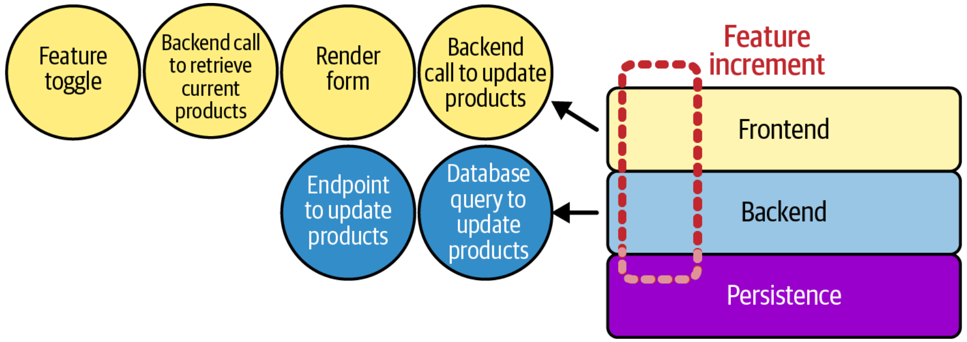
Then I should not see those products anymore in the checkout page for my store

CS: Groceroo | “Last-Minute Items”

Implementing with Vertical Slicing

- **User Story 2: Configurable Carousel**
  - Move beyond hardcoded items
  - Database changes not sustainable long-term
  - Provide admin interface for partners
  - Partners self-manage their “last-minute items”



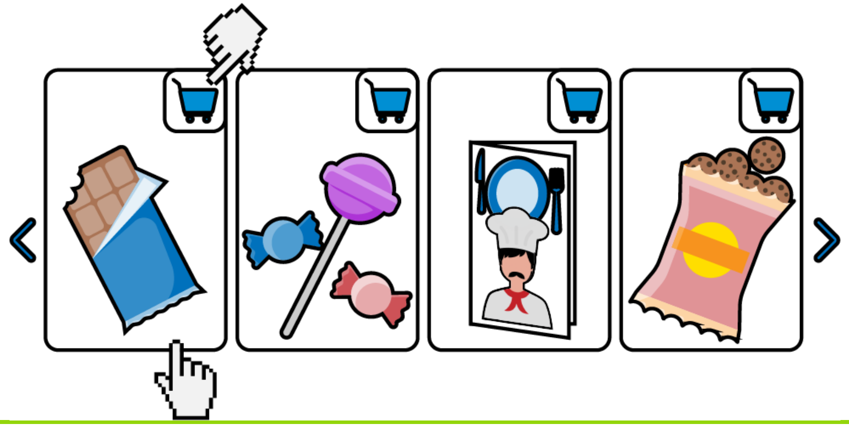

Feature toggle
Backend call to retrieve current products
Render form
Backend call to update products
Endpoint to update products
Database query to update products
Feature increment
Frontend
Backend
Persistence

CS: Groceroo | “Last-Minute Items”

Implementing with Vertical Slicing

- **User Story 3: One-Click Add to Basket**
  - Improve carousel usability & checkout flow
  - Current flow: item → detail page → add to cart (inefficient). Risk: users abandon checkout
  - New feature: “Add to basket” button directly in carousel & Use data to order items by relevance

Last-minute additions



Summary

As a user

I want an “Add to cart” button on the last-minute items

So that I can place an order more quickly, avoiding going back and forth between checkout and article detail pages

Acceptance criteria

Given I have items in the basket

When I go to checkout

Then I should see the last-minute items have an “Add to cart” button on them

Given I am viewing the last-minute items carousel

When I click the “Add to cart” button

Then the product should be added to my cart

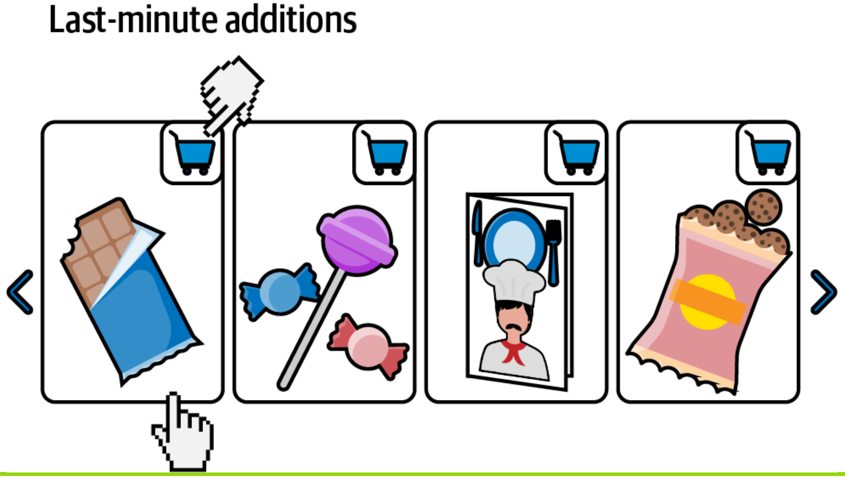
And I remain on the checkout page

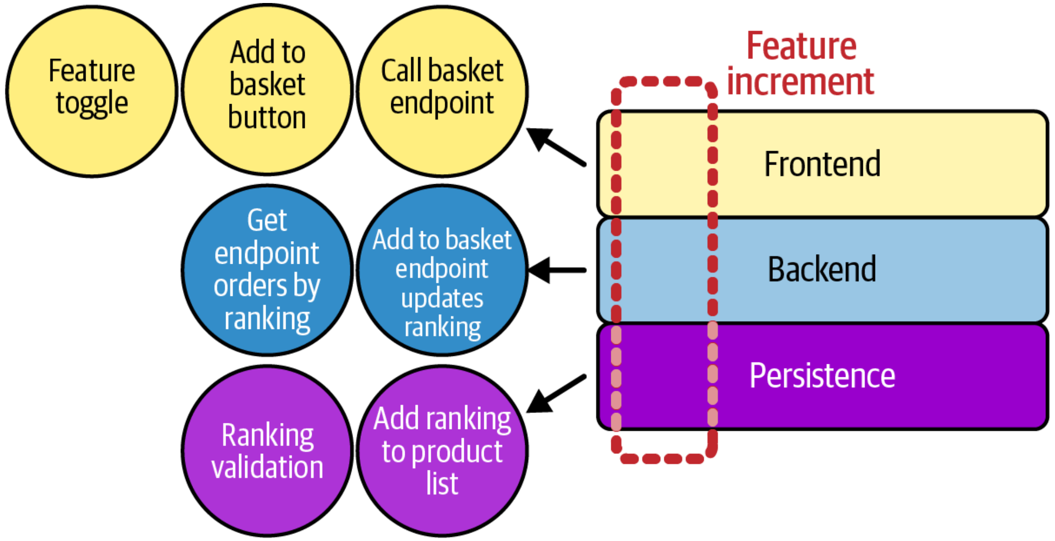
CS: Groceroo | “Last-Minute Items”

Implementing with Vertical Slicing

User Story 3: One-Click Add to Basket

- Improve carousel usability & checkout flow
- Current flow: item → detail page → add to cart (inefficient). Risk: users abandon checkout
- New feature: “Add to basket” button directly in carousel & Use data to order items by relevance

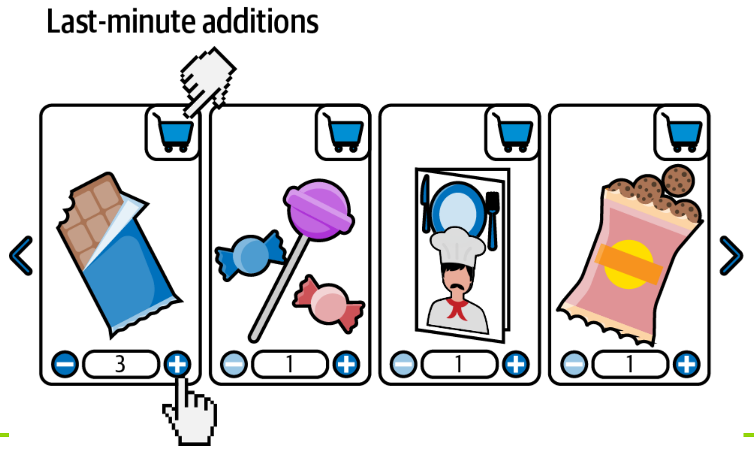




CS: Groceroo | “Last-Minute Items”

Implementing with Vertical Slicing

- **User Story 4: Quantity Selection in Carousel**
  - Adds “UX sugar” for better convenience
  - Users choose quantity before adding to basket
  - Not essential (can adjust during checkout)
  - Potential to boost average order size



Summary

As a user

I want to be able to add more than one of a last-minute item at once

So that I can place an order more quickly without clicking on the basket button multiple times

Acceptance criteria

Given I have items in the basket; **When** I go to checkout; **Then** I should see that the last-minute items have a quantity selector; **And** the selector allows me to modify the number in it (between 1 and 20)

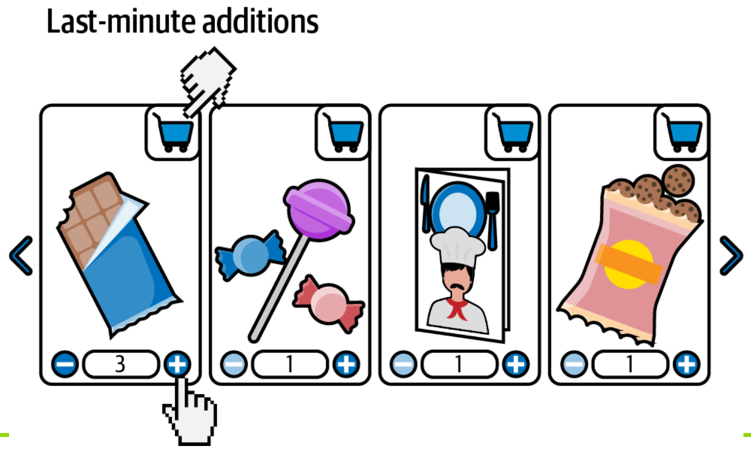
Given I am viewing the last-minute items carousel; **And** I have not interacted with the quantity selector; **When** I click the “Add to cart” button; **Then** one unit of the product should be added to my cart

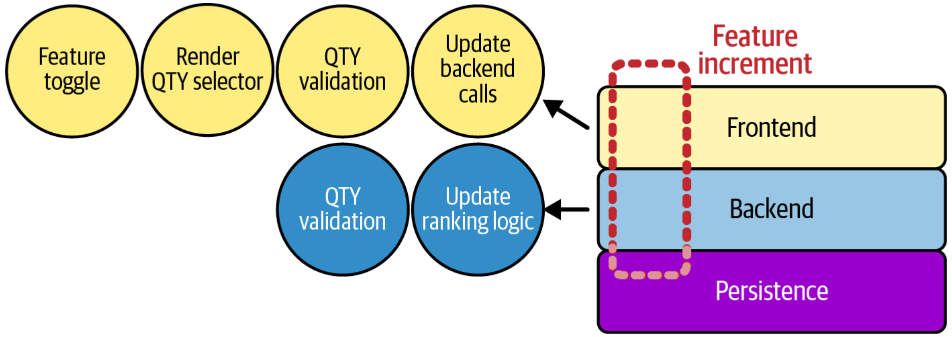
Given I am viewing the last-minute items carousel; **And** I have modified the quantity in the quantity selector; **When** I click the “Add to cart” button; **Then** the number of units added to cart matches the ones in the selector; **And** the selector resets to 1

CS: Groceroo | “Last-Minute Items”

Implementing with Vertical Slicing

- **User Story 4: Quantity Selection in Carousel**
  - Adds “UX sugar” for better convenience
  - Users choose quantity before adding to basket
  - Not essential (can adjust during checkout)
  - Potential to boost average order size





To think about..

- ➢ New user story – draw a plan to implement with horizontal & vertical slicing
  - ○ We have a new user story that you should decouple and present an architecture change.
  - ○ **User Story Stock-Aware Carousel**
    - Integrate with store inventory system
    - Show only items in stock
    - Avoid customer frustration at checkout

Impact on Software architecture

User story - if architecture changed?

- **The goal of software architecture is to minimize the human resources required to build and maintain the required system.**

- **In business, we want to reduce the cost/LOC.**

- What is NOT an architecture?
  - Libraries
  - Frameworks
  - MVC
  - Tools . . .

Architecture

- Software architecture is the shape of a system created by its builders.

- Defined by:
  - Division into components
  - Arrangement of those components
  - Communication mechanisms between components

- A good architecture must support:
  - The use cases and operation of the system.
  - The maintenance of the system.
  - The development of the system.
  - The deployment of the system.

SOLID

SOLID Principles in Software Engineering: 1) Guiding rules for designing maintainable and extensible software 2) Widely applied in object-oriented programming 3) Promote modularity, reusability, and robustness.

- **SRP: The Single Responsibility Principle.**
  - Each software module has one, and only one, reason to change.
- **OCP: The Open-Closed Principle.**
  - For software systems to be easy to change, they must be designed to allow the behavior of those system to be changed by adding new code, rather than changing existing code.
- **LSP: The Liskov Substitution Principle.**
  - To build software systems from interchangeable parts, those parts must adhere to a contract that allows those parts to be substituted one for another.
- **ISP: The Interface Segregation Principle.**
  - This principle advises software designers to avoid depending on things that they don’t use.
- **DIP: The Dependency Inversion Principle.**
  - The code that implements high-level policy should not depend on the code that implements low-level details. Rather, details should depend on policies.

COMPONENT PRINCIPLES

A component is a grouping of related functionality behind a nice clean interface, which resides inside an execution environment like an application.

REP: The Reuse/Release Equivalence Principle.

- The granule of reuse is the granule of release. This means that the classes and modules that are formed into a component must belong to a cohesive group.

CCP: The Common Closure Principle.

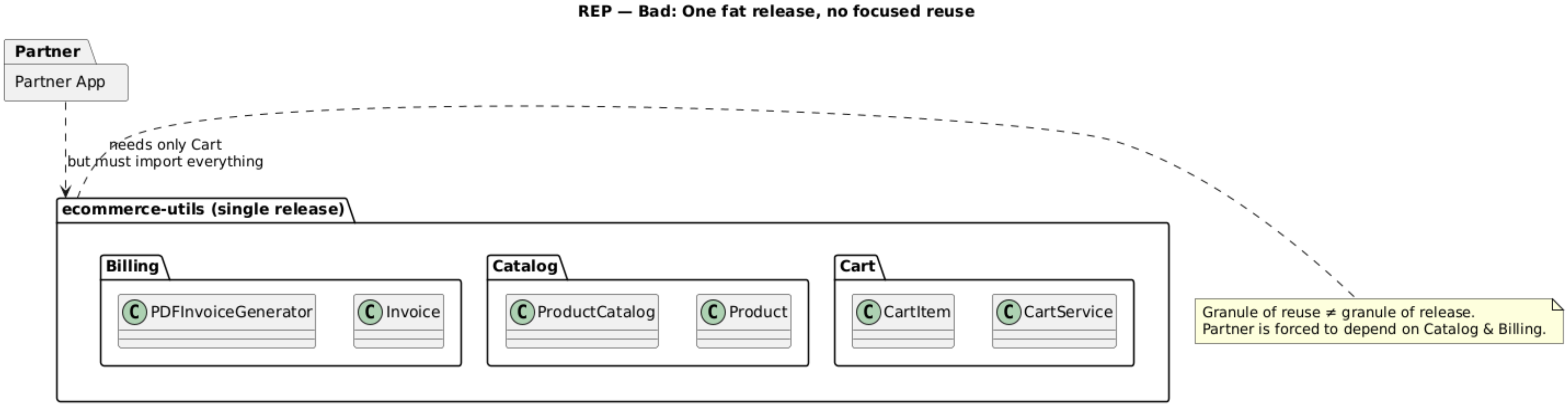
- Gather into components those classes that change for the same reasons and at the same times. Separate into different components those classes that change at different times and for different reasons.

CRP: The Common Reuse Principle.

- Don’t force users of a component to depend on things they don’t need.

COMPONENT PRINCIPLES

- **REP: The Reuse/Release Equivalence Principle.**

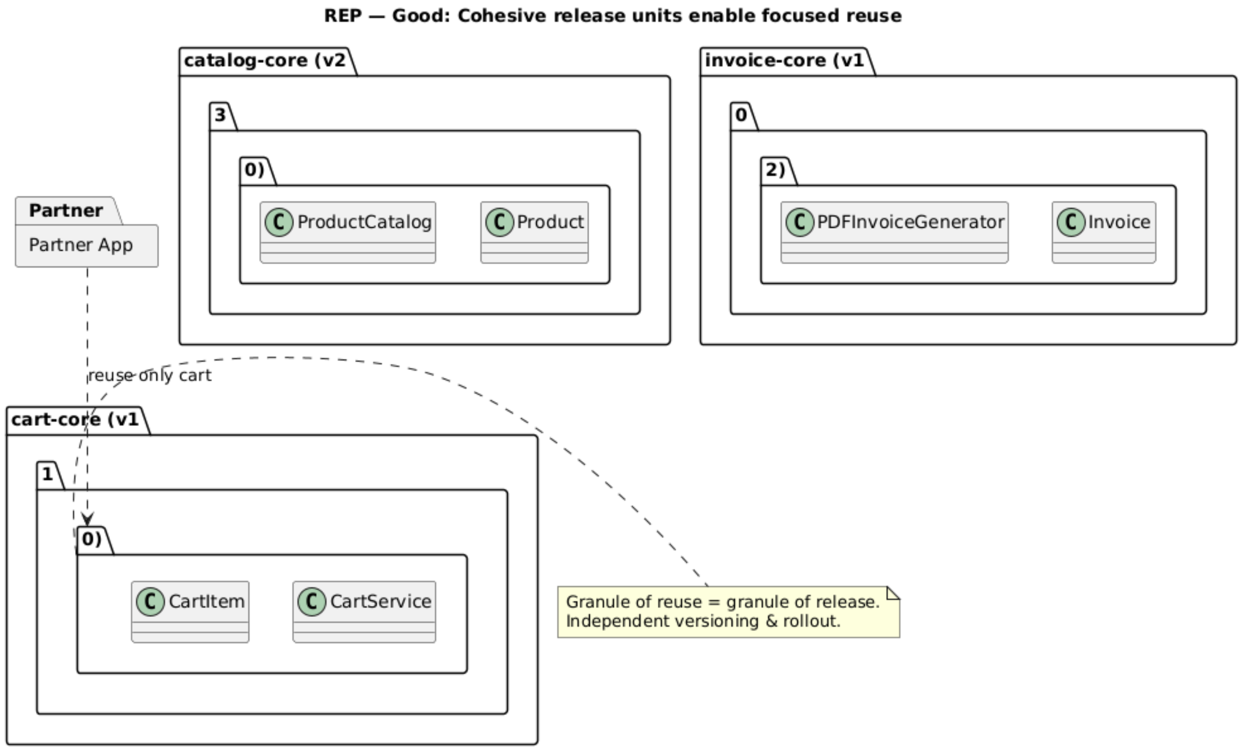


REP — Bad: One fat release, no focused reuse

Granule of reuse ≠ granule of release.
Partner is forced to depend on Catalog & Billing.

COMPONENT PRINCIPLES

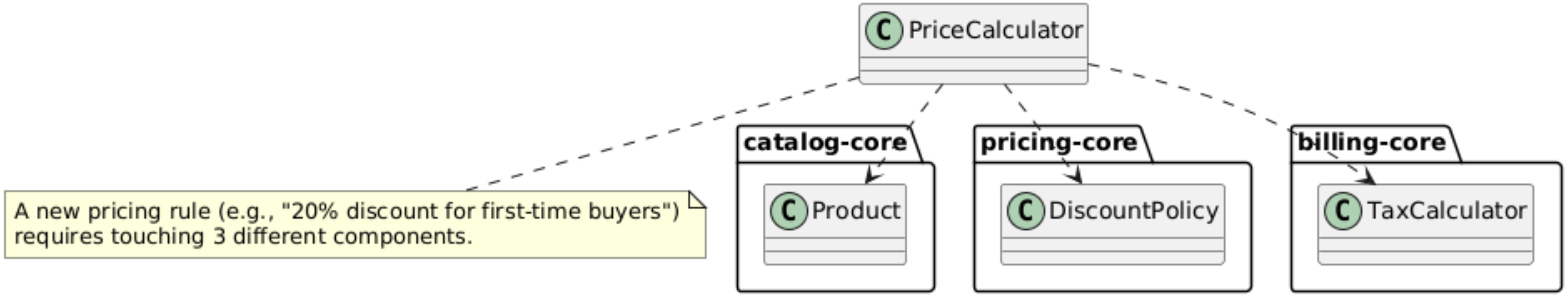
- **REP: The Reuse/Release Equivalence Principle.**



COMPONENT PRINCIPLES

- **CCP: The Common Closure Principle.**

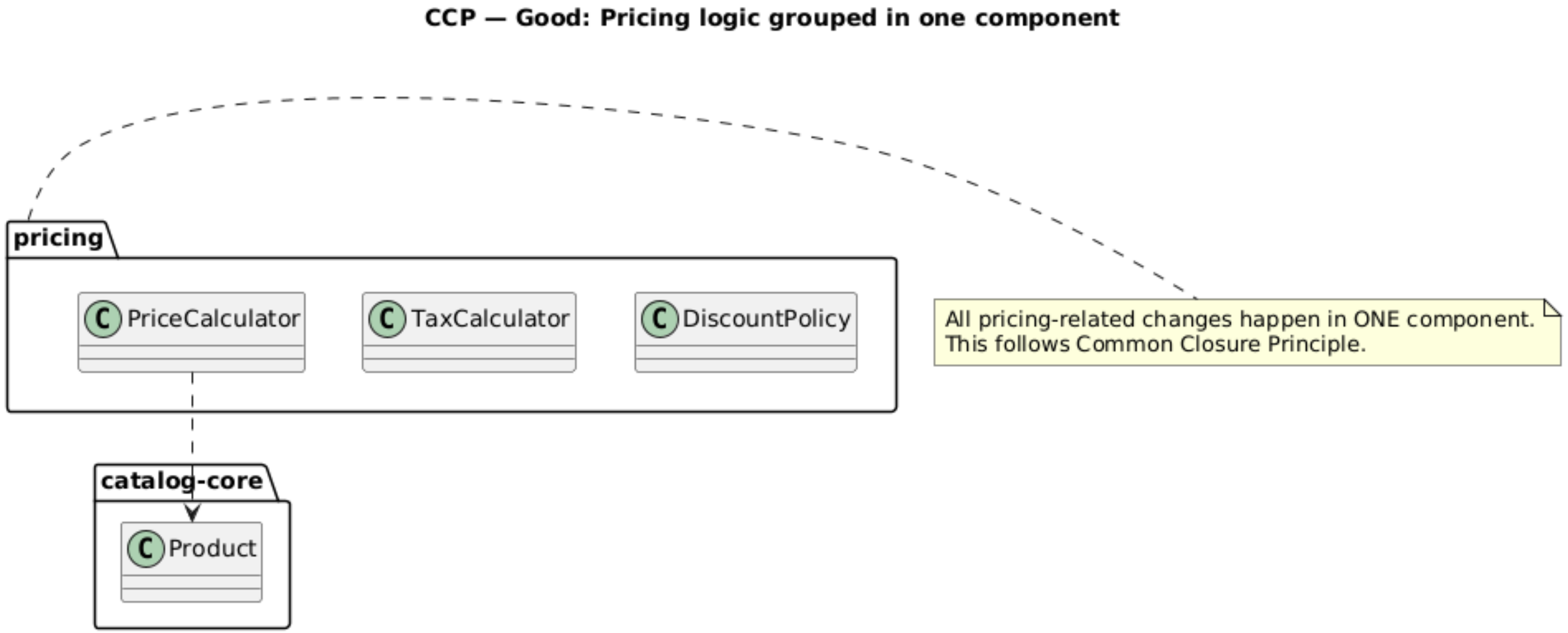
CCP — Bad: Pricing logic scattered



A new pricing rule (e.g., "20% discount for first-time buyers") requires touching 3 different components.

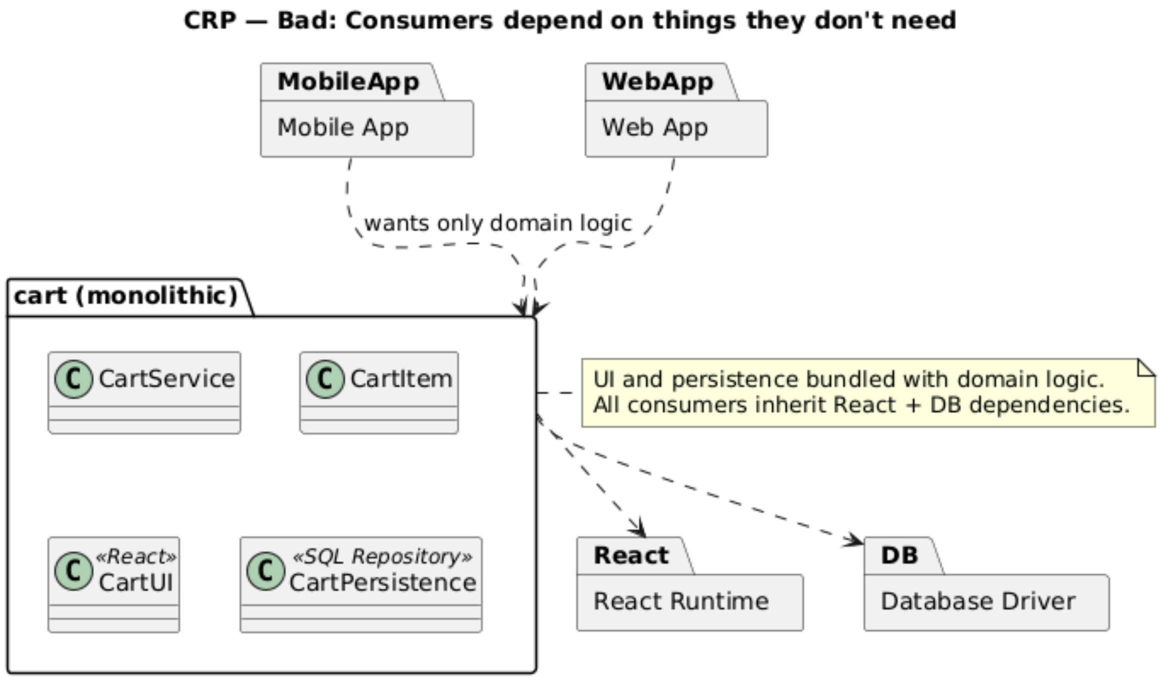
COMPONENT PRINCIPLES

- **CCP: The Common Closure Principle.**


CCP — Good: Pricing logic grouped in one component
pricing
C PriceCalculator
C TaxCalculator
C DiscountPolicy
All pricing-related changes happen in ONE component.
This follows Common Closure Principle.
catalog-core
C Product

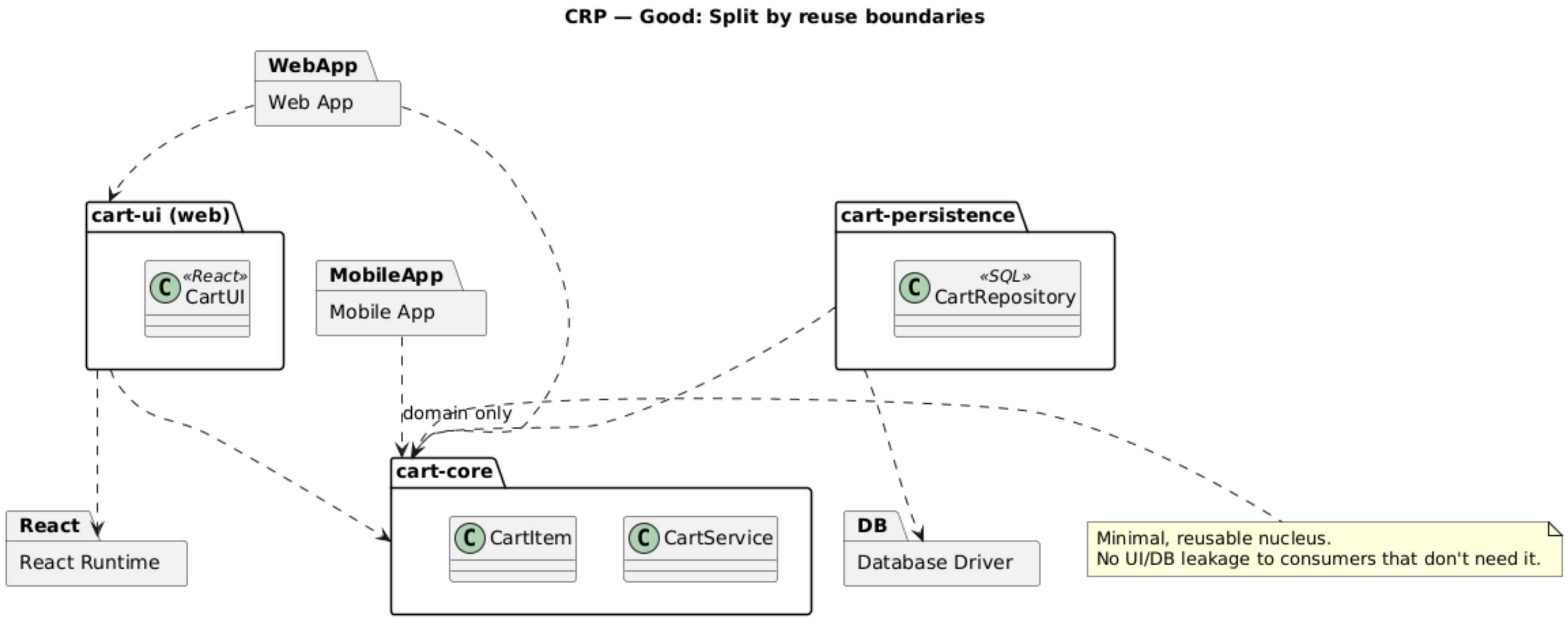
COMPONENT PRINCIPLES

- **CRP: The Common Reuse Principle.**


CRP — Bad: Consumers depend on things they don't need
MobileApp
Mobile App
WebApp
Web App
wants only domain logic
cart (monolithic)
C CartService
C CartItem
C «React» CartUI
C «SQL Repository» CartPersistence
UI and persistence bundled with domain logic.
All consumers inherit React + DB dependencies.
React
React Runtime
DB
Database Driver

COMPONENT PRINCIPLES

➢ **CRP: The Common Reuse Principle.**



Common Pitfalls

- Architecture not tied to real user stories
- UI coupled directly to database models
- Over-engineering beyond user needs
- Ignoring maintainability and scalability early

Best Practices

- Start with stories that define corresponding use cases
- Apply SOLID and component principles
- Keep frameworks & details at the outer layers
- Validate alignment with business goals regularly
- Document mappings between stories and architecture

Problem-oriented refactoring

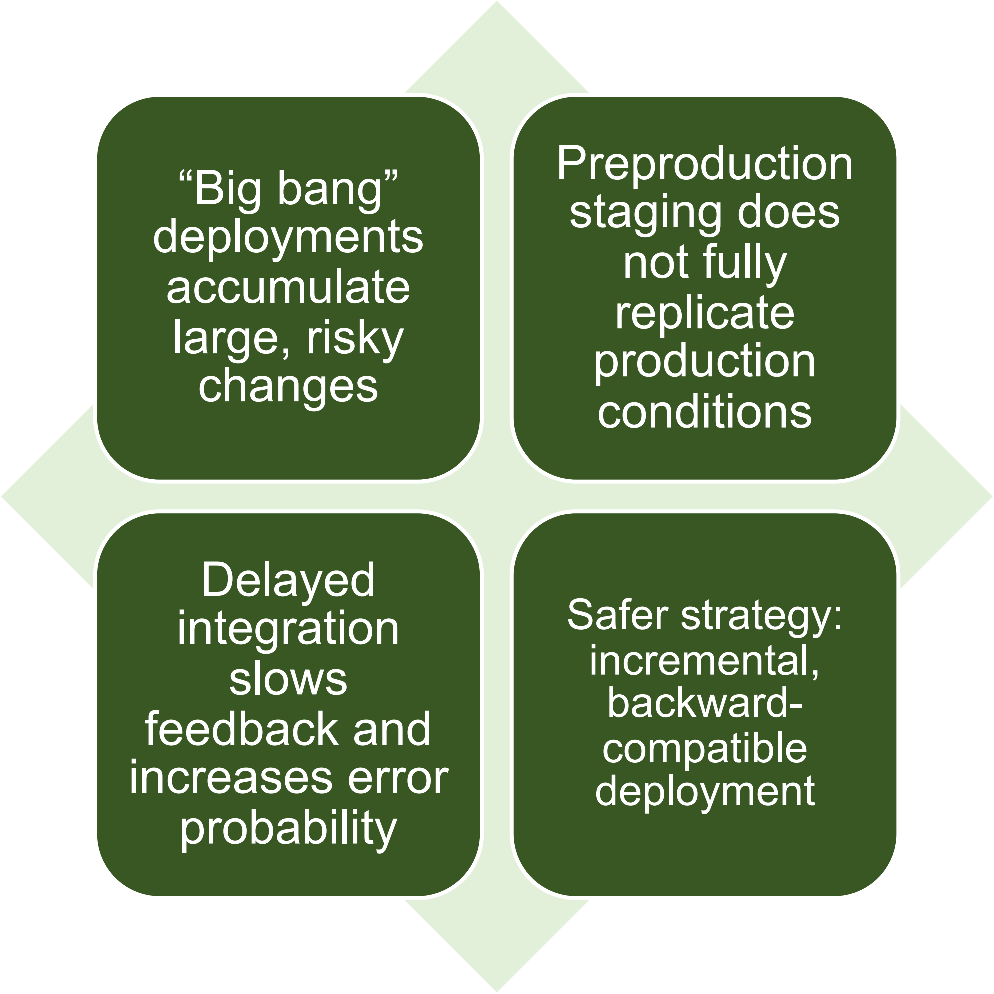
Refactoring Live Features

- Continuous deployment requires **delivering small, safe, incremental changes**
- **Feature toggles** help introduce **new functionality gradually**
- When modifying live features, backward compatibility becomes essential
- **Goal: rearchitect systems without disrupting user experience**

The Nature of Refactoring

- Refactoring: improving internal structure without altering external behavior
- Most tasks involve local, low-risk modifications
- ✓ Larger refactorings affect multiple components or domains
- Goal: maintain service continuity while evolving architecture

But be aware of..



“Big bang” deployments accumulate large, risky changes

Preproduction staging does not fully replicate production conditions

Delayed integration slows feedback and increases error probability

Safer strategy: incremental, backward-compatible deployment

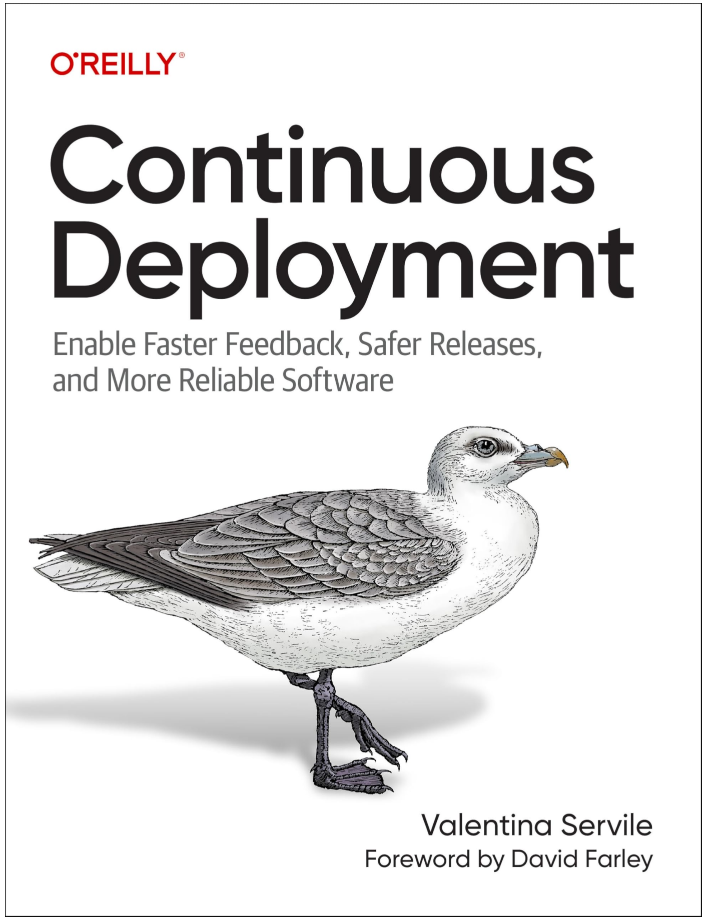
Practical Guide

Practical Guide

➢ In your project:

- Make sure that you have ready to be implemented user-stories;
- Move the tasks to the board to the correct column and assigned to members, in a vertical slicing oriented-process;
- Bootstrap your software components (e.g. Sprint Boot, React, Apache Kafka, Docker materials..)
- Start implementing some user stories and make sure that they followed always in separate "Pull Requests".

Bibliography



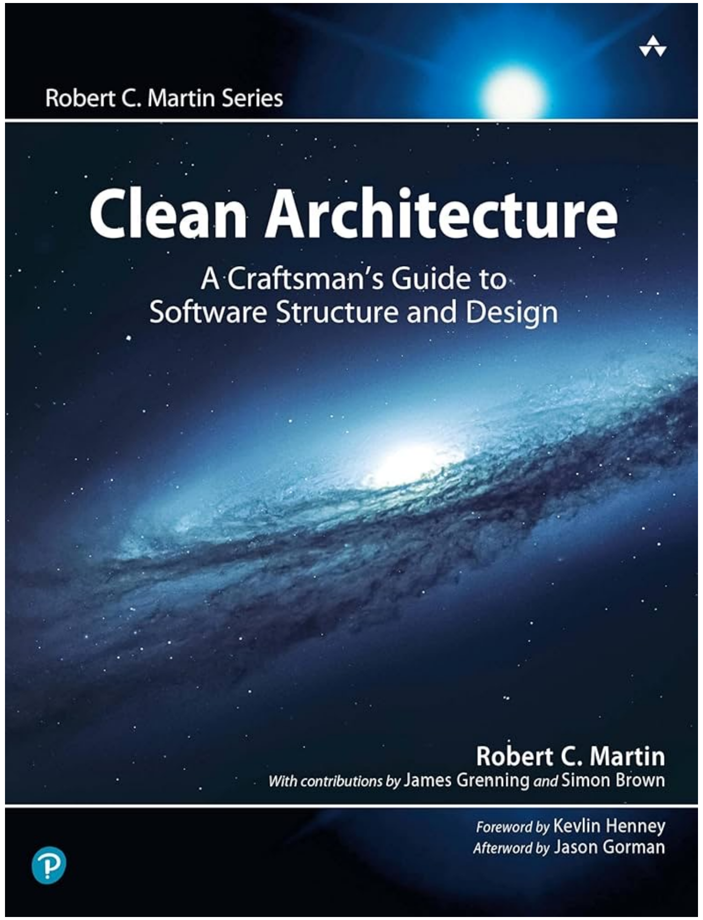
O'REILLY®

Continuous Deployment

Enable Faster Feedback, Safer Releases, and More Reliable Software

Valentina Servile

Foreword by David Farley



Robert C. Martin Series

Clean Architecture

A Craftsman's Guide to Software Structure and Design

Robert C. Martin

With contributions by James Grenning *and* Simon Brown

Foreword by Kevlin Henney

Afterword by Jason Gorman