On-Demand Data Architecture

Event-Driven Systems, Domain Modeling, and Systematic Modernization

A comprehensive framework for building composable, evolvable enterprise data systems

Slide 1: The Challenge

Information Systems Lag Reality

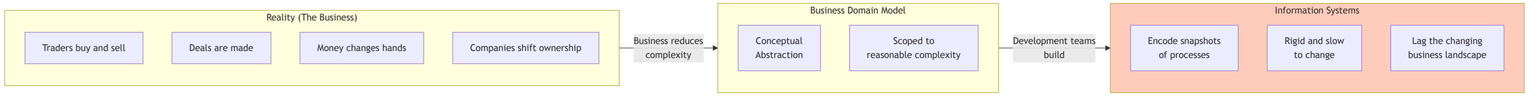


Diagram 0

The problem: Systems traditionally are rigid, slow to change, slow to extend - generally lagging the changing business landscape reflected by reality.

Slide 2: The Domain Landscape

From Reality to Systems

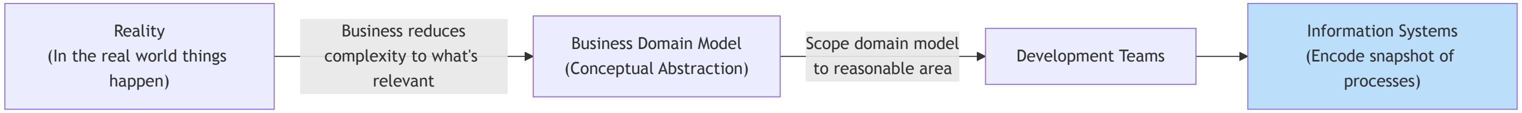

Reality
(In the real world things happen)
Business reduces complexity to what's relevant
Business Domain Model
(Conceptual Abstraction)
Scope domain model to reasonable area
Development Teams
Information Systems
(Encode snapshot of processes)

Diagram 1

Key insight: The broader the domain model, the more abstract it needs to be. We don’t try to model the entire business end-to-end in a single domain.

Slide 3: Bounded Contexts

Managing Complexity Through Separation

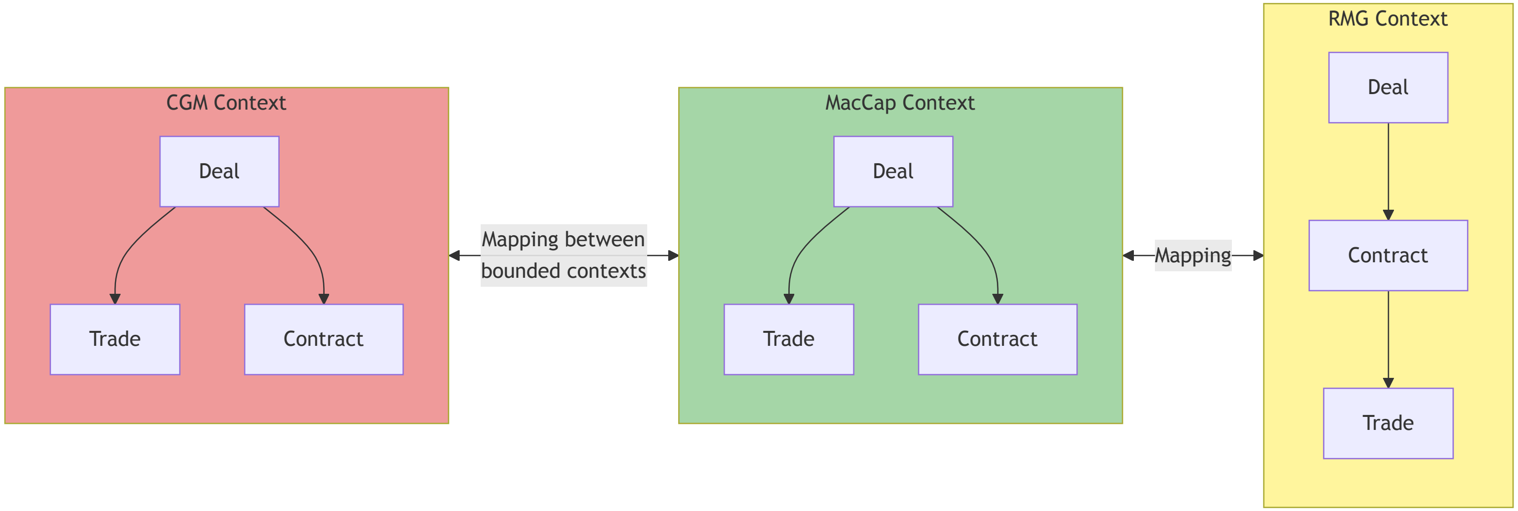


Diagram 2

Bounded Context principle: - An organization is complex - Single model = excessively challenging, costly, hard to maintain - Separate into areas with **consistent language, operations, and data model** - Each context may use the same term but with subtly different definitions

Slide 4: Corporate Anti-Patterns

What We’re Fighting Against

Anti-Pattern	Description
Monolithic Big Ball of Mud	Everything coupled together, no clear boundaries
File Transfers	Custom point-to-point integrations
Manual Processes	Humans bridging gaps in integrations
Output-Only Focus	Ignoring the transform, focusing only on output data
Attribute-Level Thinking	Dealing with data at attribute level instead of Entity level

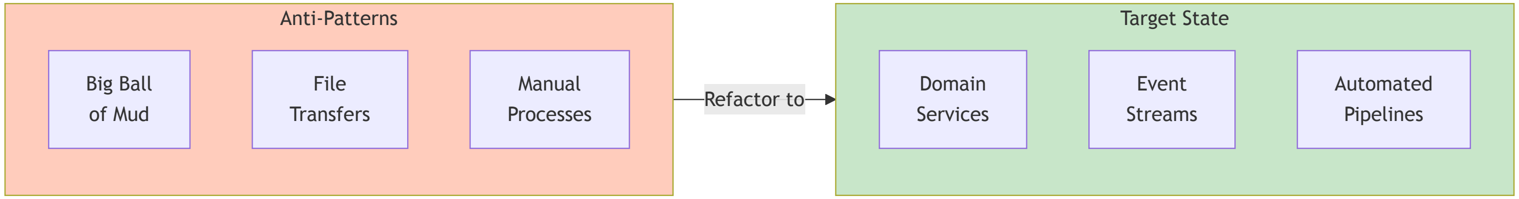
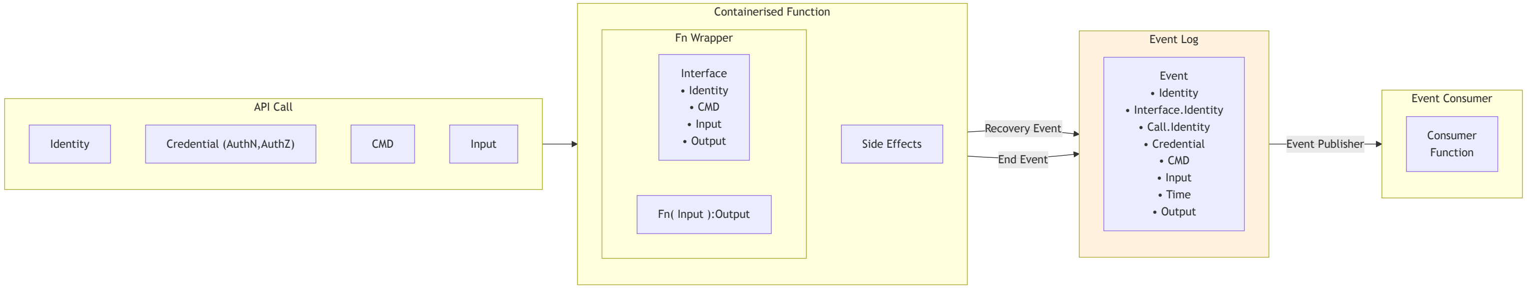


Diagram 3

Slide 5: Event-Driven Architecture

Systems Composable Through Events



API Call

Identity

Credential (AuthN,AuthZ)

CMD

Input

Containerised Function

Fn Wrapper

Interface
- Identity
- CMD
- Input
- Output

Fn(Input):Output

Side Effects

Recovery Event

End Event

Event Log

Event
- Identity
- Interface.Identity
- Call.Identity
- Credential
- CMD
- Input
- Time
- Output

Event Publisher

Event Consumer

Consumer Function

Diagram 4

Identity structure: - GUID - Credential (AuthN, AuthZ) - Creation Time UTC

Slide 6: What Does a Good Event Look Like?

Event Design Principles

Every event must have:

1. **Captured and available** to authorized subscribers
2. **Security token** under which it was authorized
3. **Signature of parent event** (lineage)
4. **Adaptive to schema changes** over time

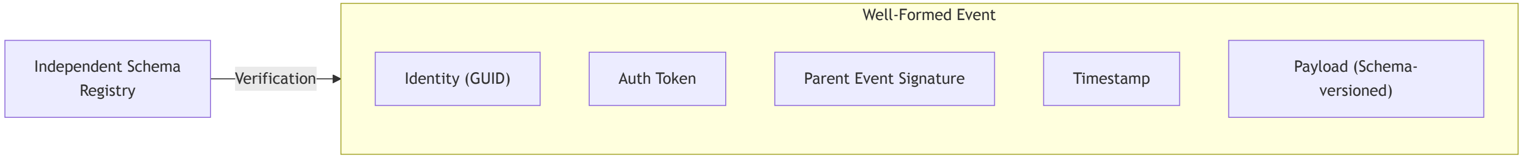
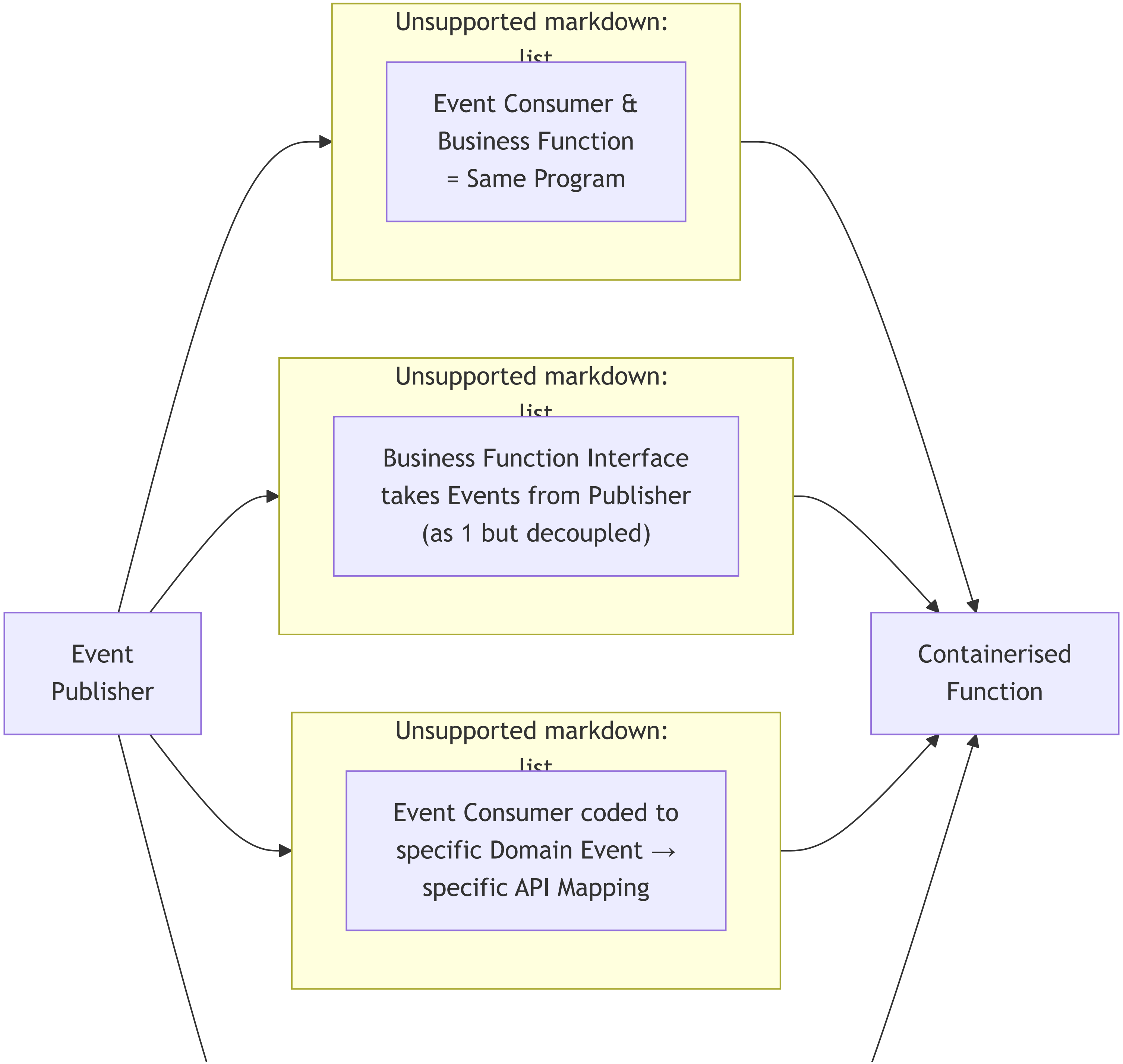


Diagram 5

Data pipelines need to be adaptive to changes in data over time - independent source for schema verification consumed by integration endpoints.

Slide 7: Event Consumer Use Cases

Four Patterns for Event Consumption


Unsupported markdown: list
Event Consumer & Business Function = Same Program
Unsupported markdown: list
Business Function Interface takes Events from Publisher (as 1 but decoupled)
Unsupported markdown: list
Event Consumer coded to specific Domain Event → specific API Mapping
Event Publisher
Containerised Function

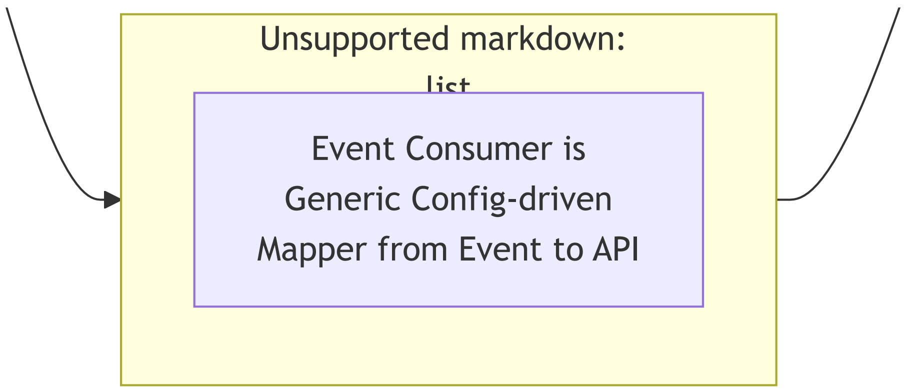

Unsupported markdown:
list
Event Consumer is
Generic Config-driven
Mapper from Event to API

Diagram 6

Slide 8: Use Case - Serverless Application

Event Flow Through Functions

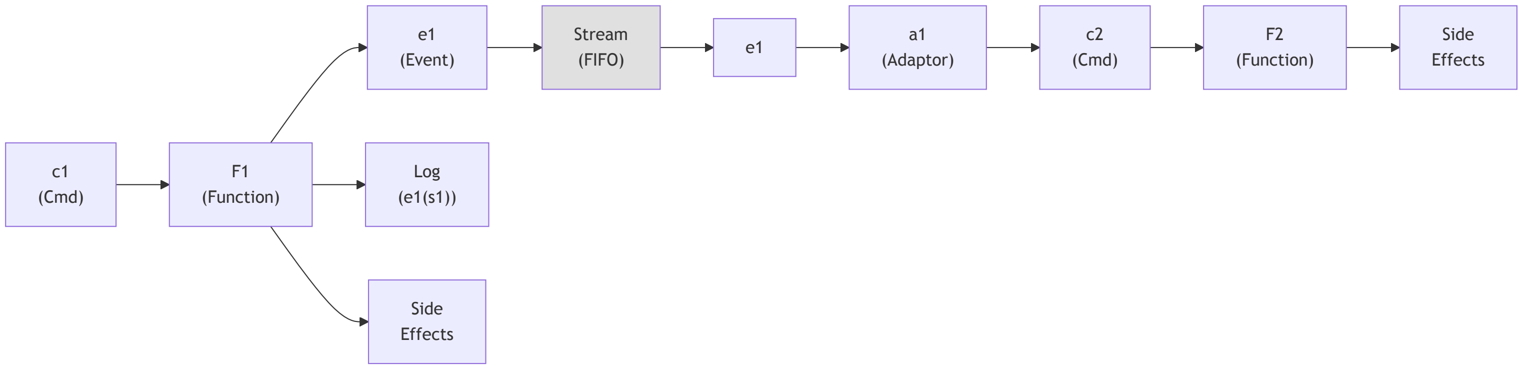


Diagram 7

Key pattern: - `State ++ Event -> Current State` - Current State refers to the accumulated state from events - Reliance on ODBC or equivalent for side effects

Slide 9: Event-Driven Scoring Service Example

Real-World Architecture

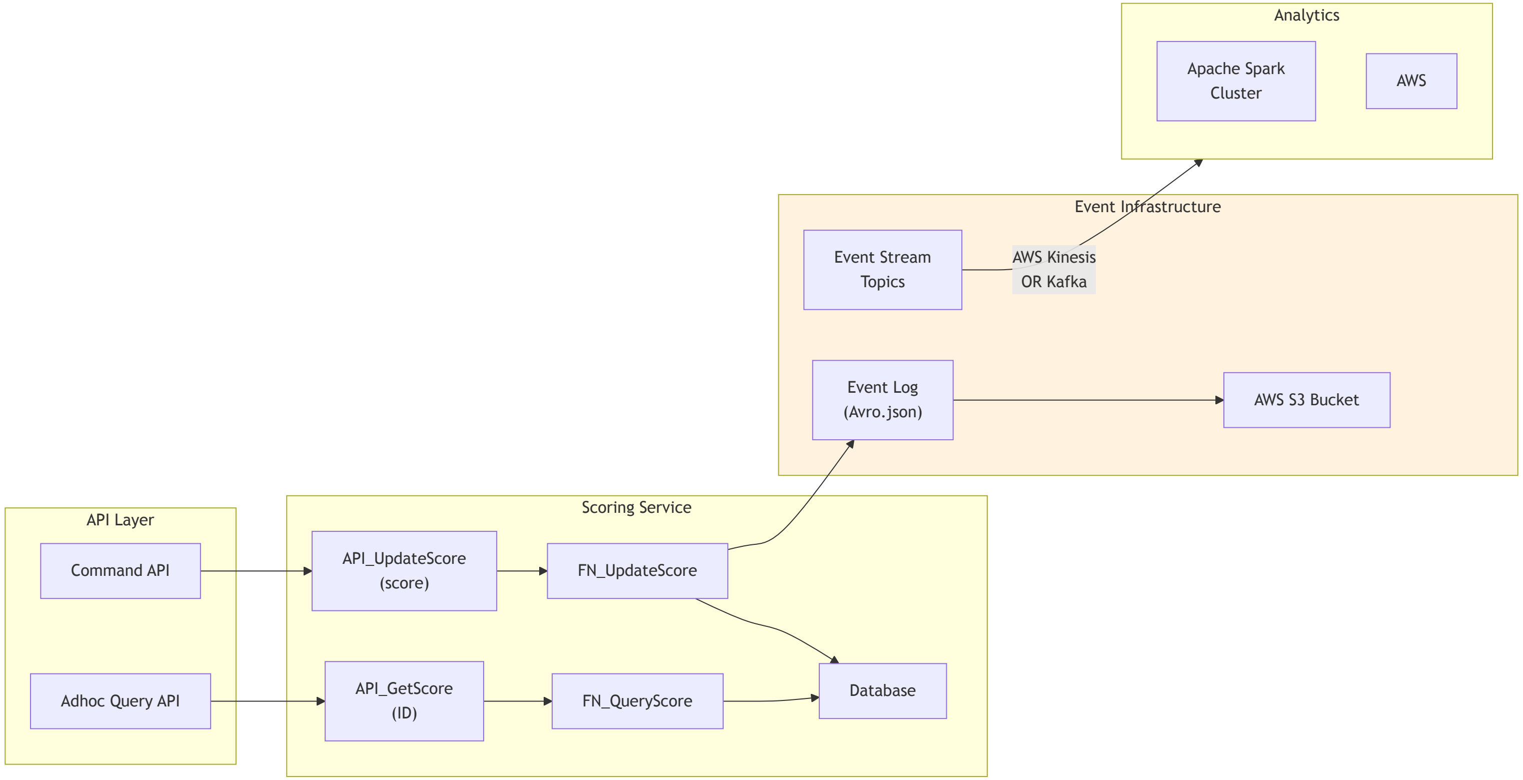


Diagram 8

Slide 10: On-Demand Resource Pipeline (Data Pipes V3)

Complete Architecture

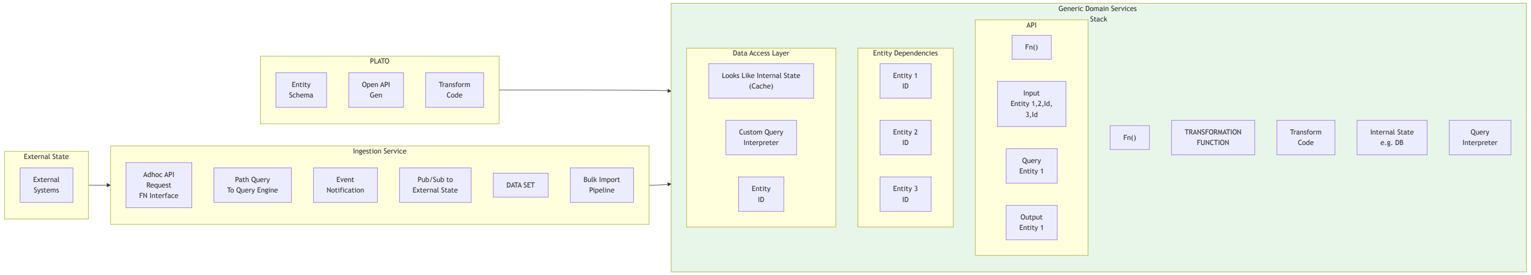


Diagram 9

Slide 11: On-Demand Function Modules

Serverless Execution Architecture

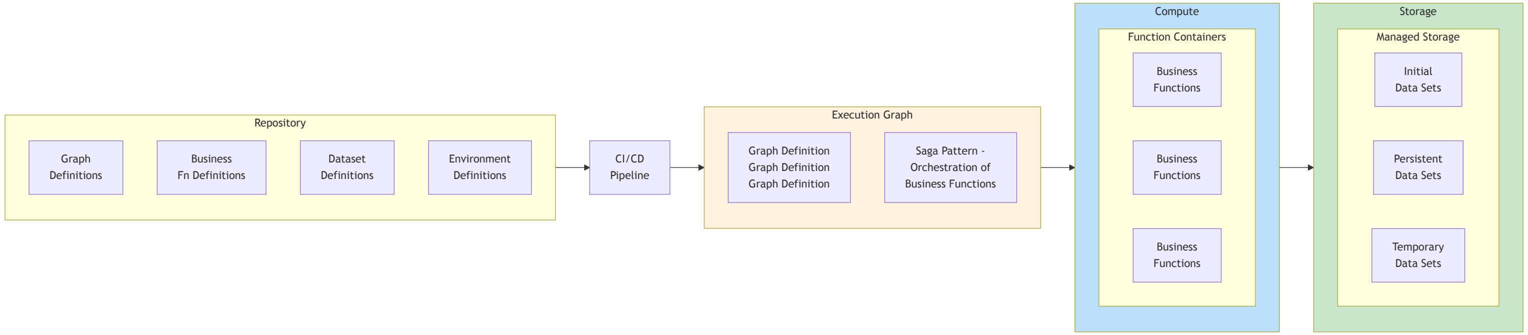


Diagram 10

Slide 12: Spark Invoker Use Case

Workflow Orchestration

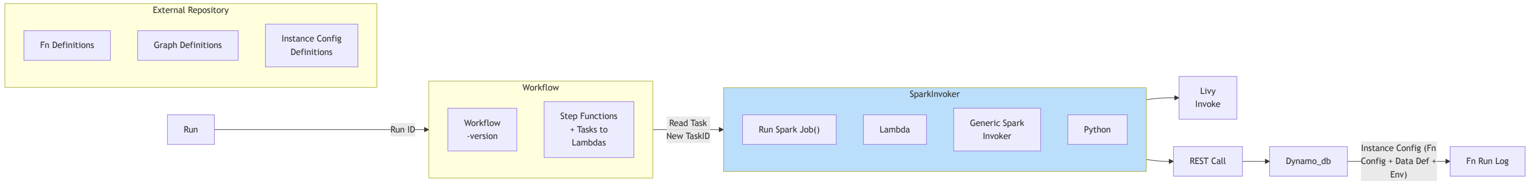
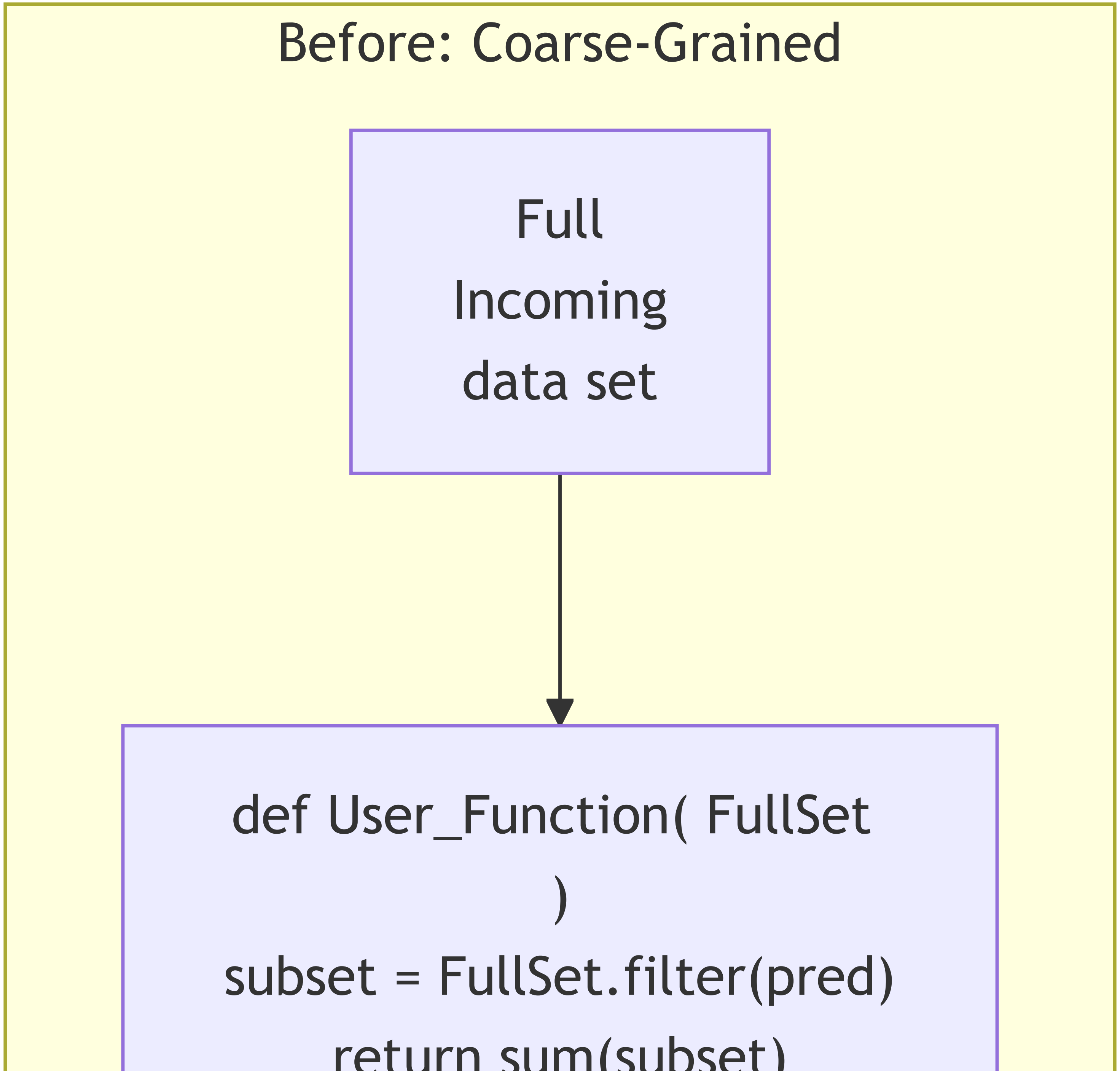


Diagram 11

Slide 13: Fine-Graining Your Functions

Improving Lineage Through Decomposition

Problem: User-defined function does filtering internally, reducing trackable granularity.



Before: Coarse-Grained

Full Incoming data set

```
def User_Function( FullSet
)
subset = FullSet.filter(pred)
return sum(subset)
```

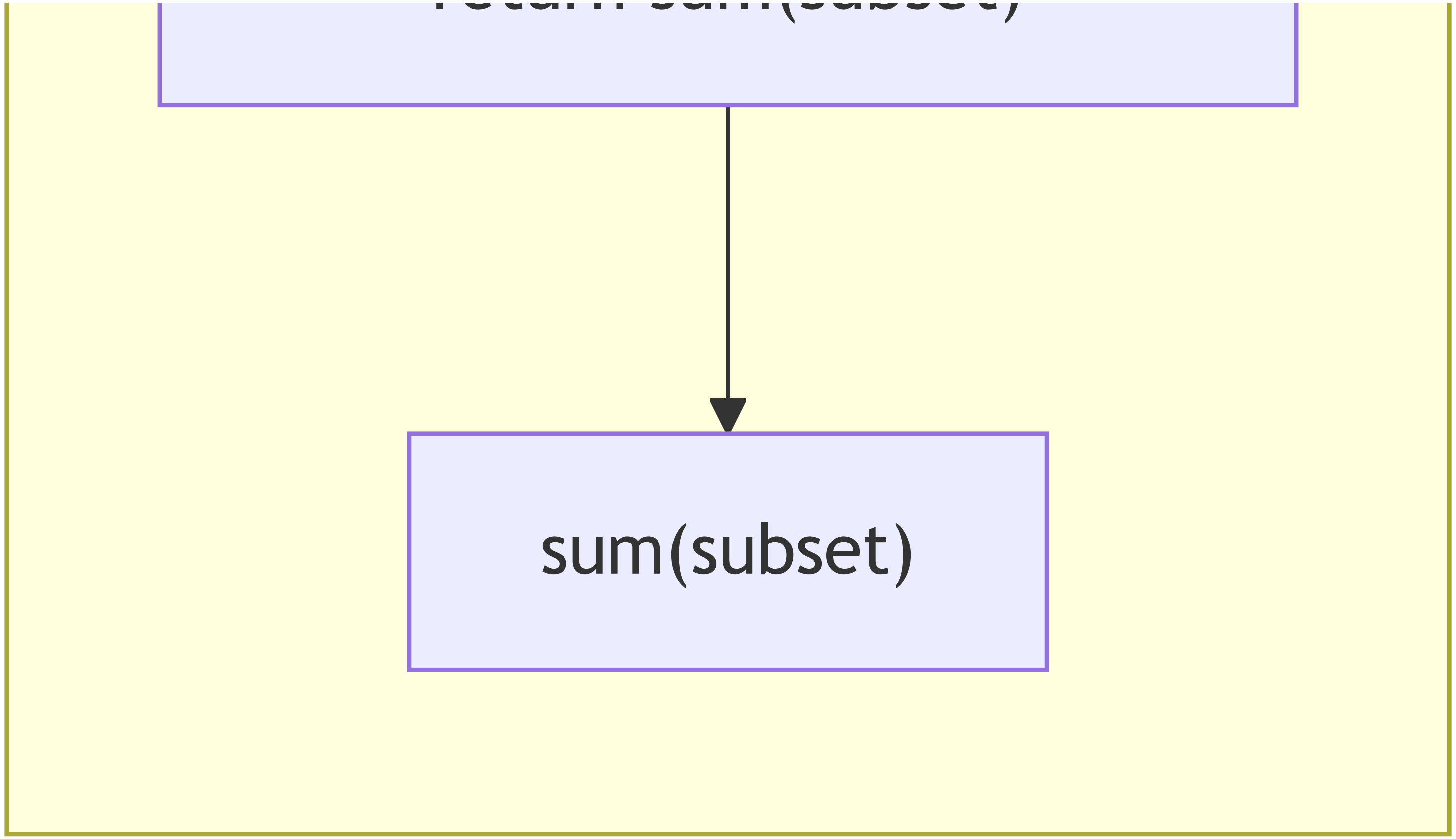
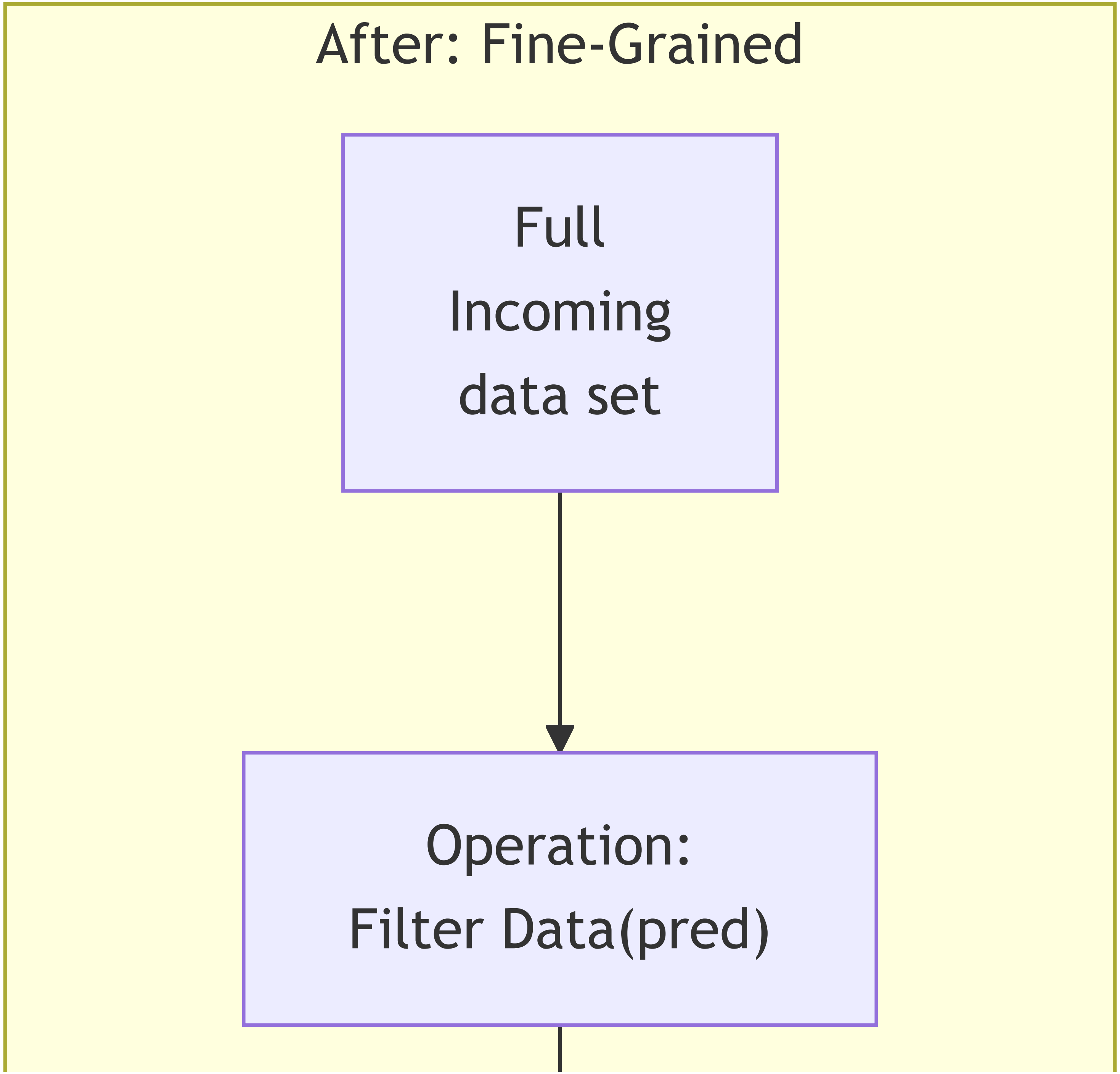
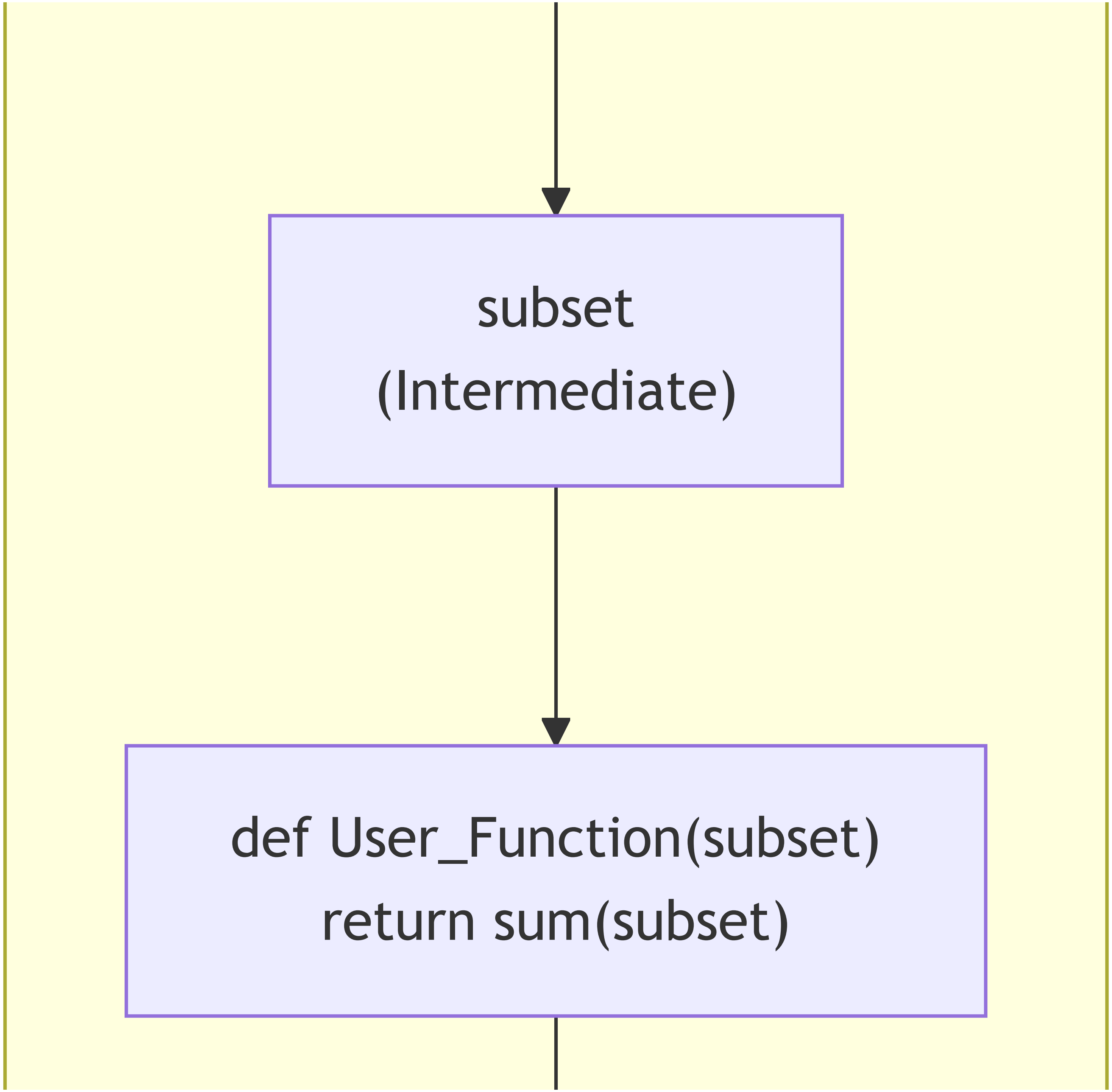


Diagram 12

Lineage Outcome: ResultSet was generated by User_Function using InputSet (no detail on filter)


After: Fine-Grained
Full Incoming data set
Operation: Filter Data(pred)


subset
(Intermediate)
def User_Function(subset)
return sum(subset)

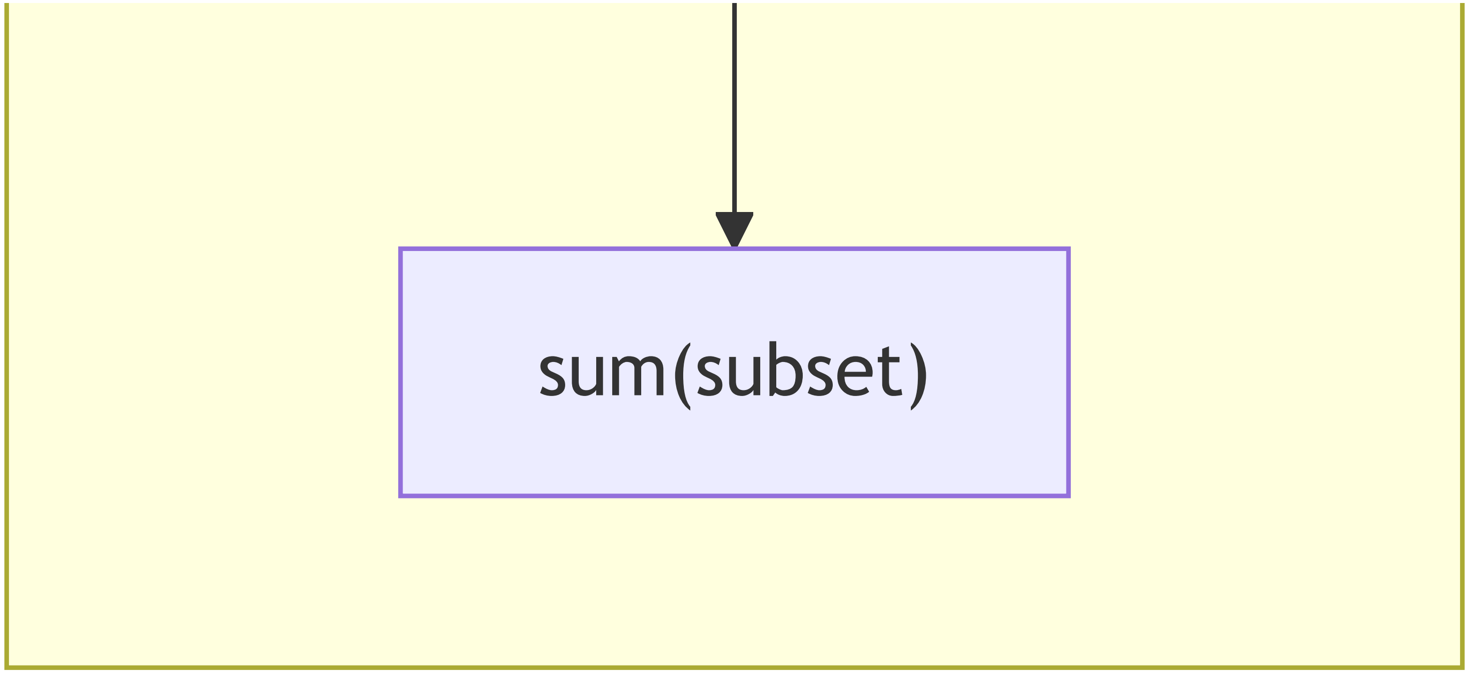


Diagram 13

To improve granularity: The filtering functionality should be a separate function.

Slide 14: Managed Data Sets

Technology-Agnostic Data Storage

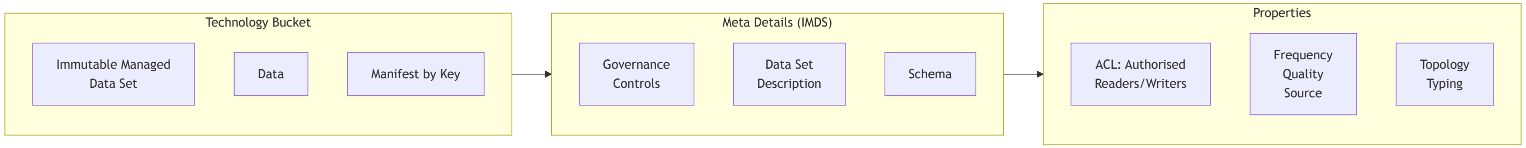


Diagram 14

S3 Implementation: | Property | Value | |— — —-|— —-| | Security | IMS | | Governance | Configurable | | Physical Location | Regional Replication | | Access Methods | Sockets, Object Model | | Encoding | Configurable |

Slide 15: CDH Domain Model Modules

Data Storage Architecture

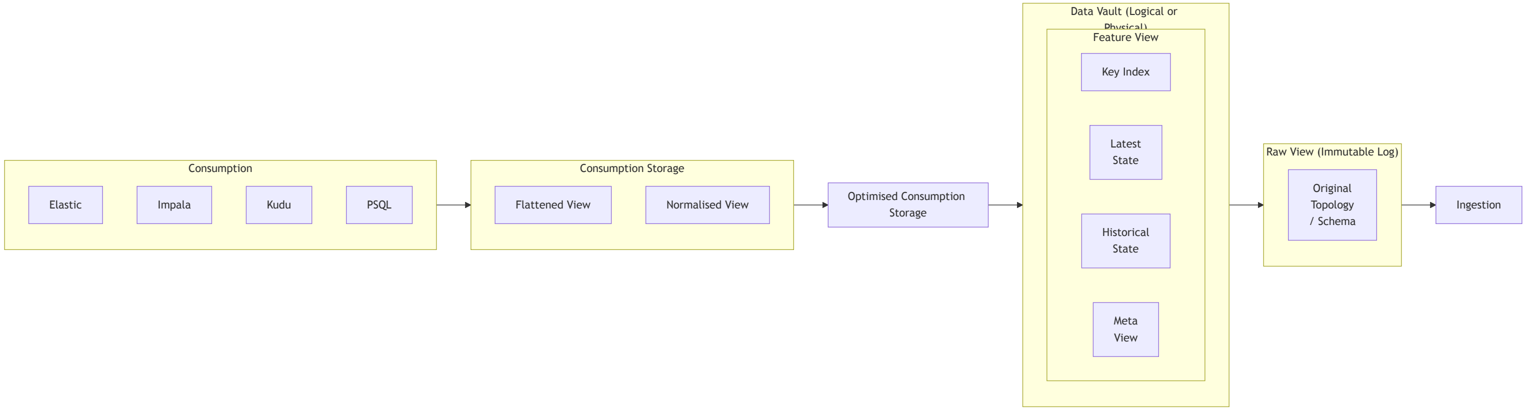


Diagram 15

Slide 16: Refactoring - You Got to Have a Plan

The Two-Phase Approach

There is no point refactoring towards a technology unless it is in service to your model.

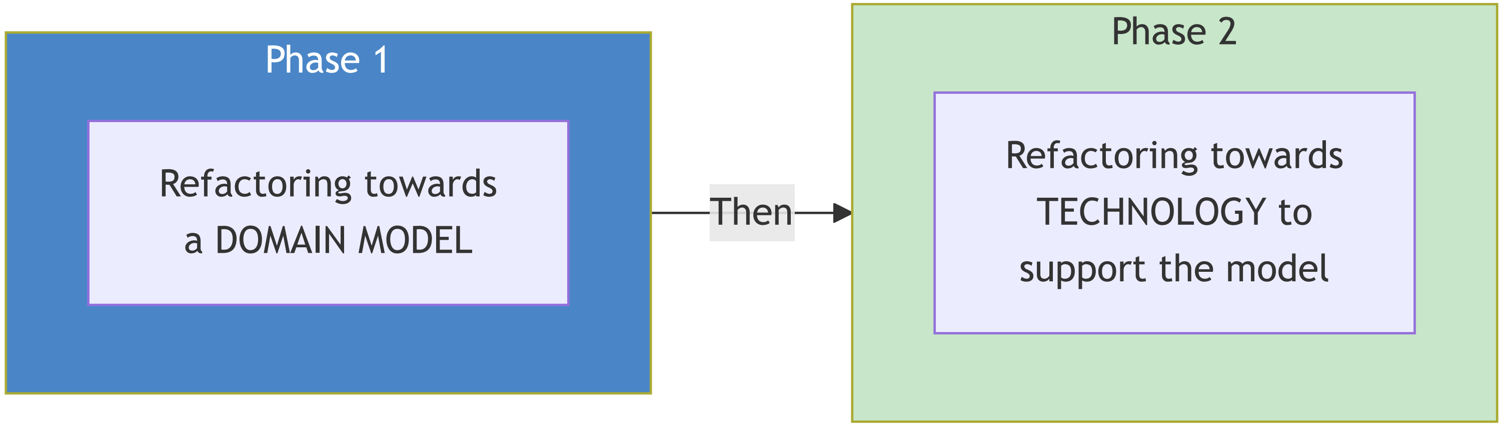

Phase 1
Refactoring towards a DOMAIN MODEL
Then
Phase 2
Refactoring towards TECHNOLOGY to support the model

Diagram 16

Key insight: Technology serves the model, not the other way around.

Slide 17: Refactoring a Slice Pattern

Consumer-Backward Approach

Step 1: Go to the end point and identify the model the consumer needs - E.g., FP&A CTOs report has a specific ‘bounded context’ - It may have a unique language and requirement

Step 2: Work backwards from the Consumer - Identify entities needed to fulfill the consumer’s model - Build the model for those entities

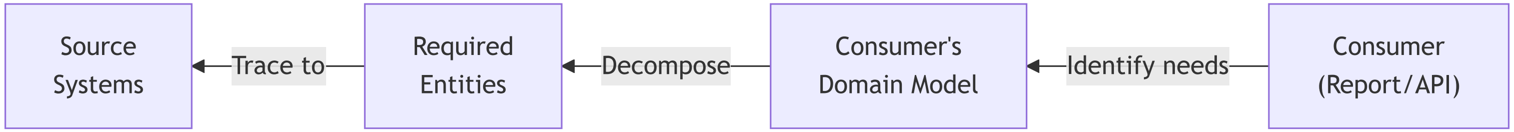


Diagram 17

Slide 18: Section Outlines - The Refactoring Roadmap

From Current State to Future State

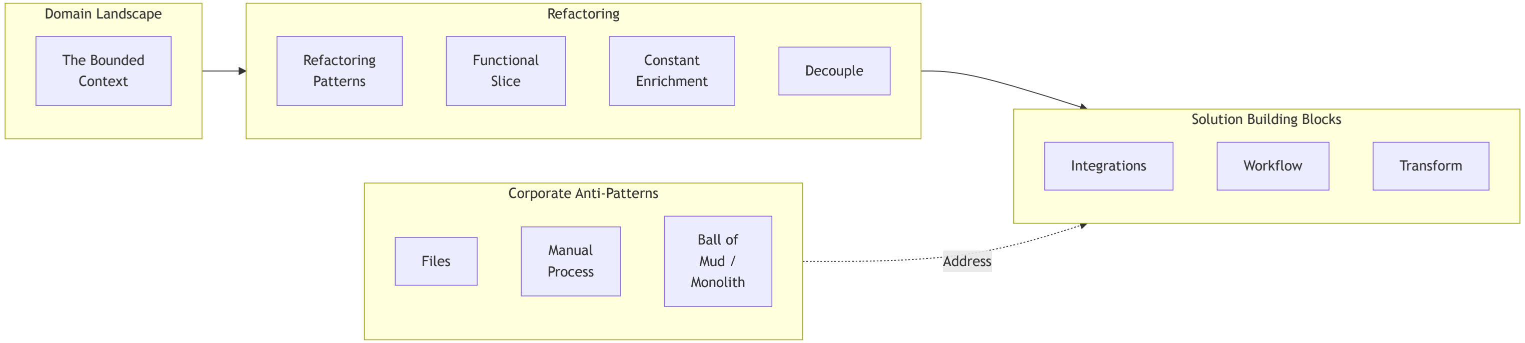


Diagram 18

The Journey: 0. Example Future State 1. Domain Model Consumer 2. Domain Model Sources 3. Create APIs over each Model 4. Integrate 5. Solution Building Blocks 6. Parallel Verification

Slide 19: Orchestra - Philosophy of Product Development

Domain-Driven, Event-Sourced Architecture

Philosophy of Designing for Product: 1. Background in Start-ups & Product-driven development 2. Elements of a start-up culture: - Use a real-world problem to bootstrap a product - Separate the Business Domain from the Capabilities needed - Discover requirements through iteration - don’t be paralysed by lack of requirements

Philosophy of Orchestra: 1. **Domain-Driven Design** to define your services 2. **Event-Driven Architecture / Event Sourcing** for modeling 3. **Orchestrating Domain Services**: - Avoiding creating dependency chains - Introducing the **Saga Pattern**

Slide 20: The Saga Pattern

Orchestrating Without Dependency Chains

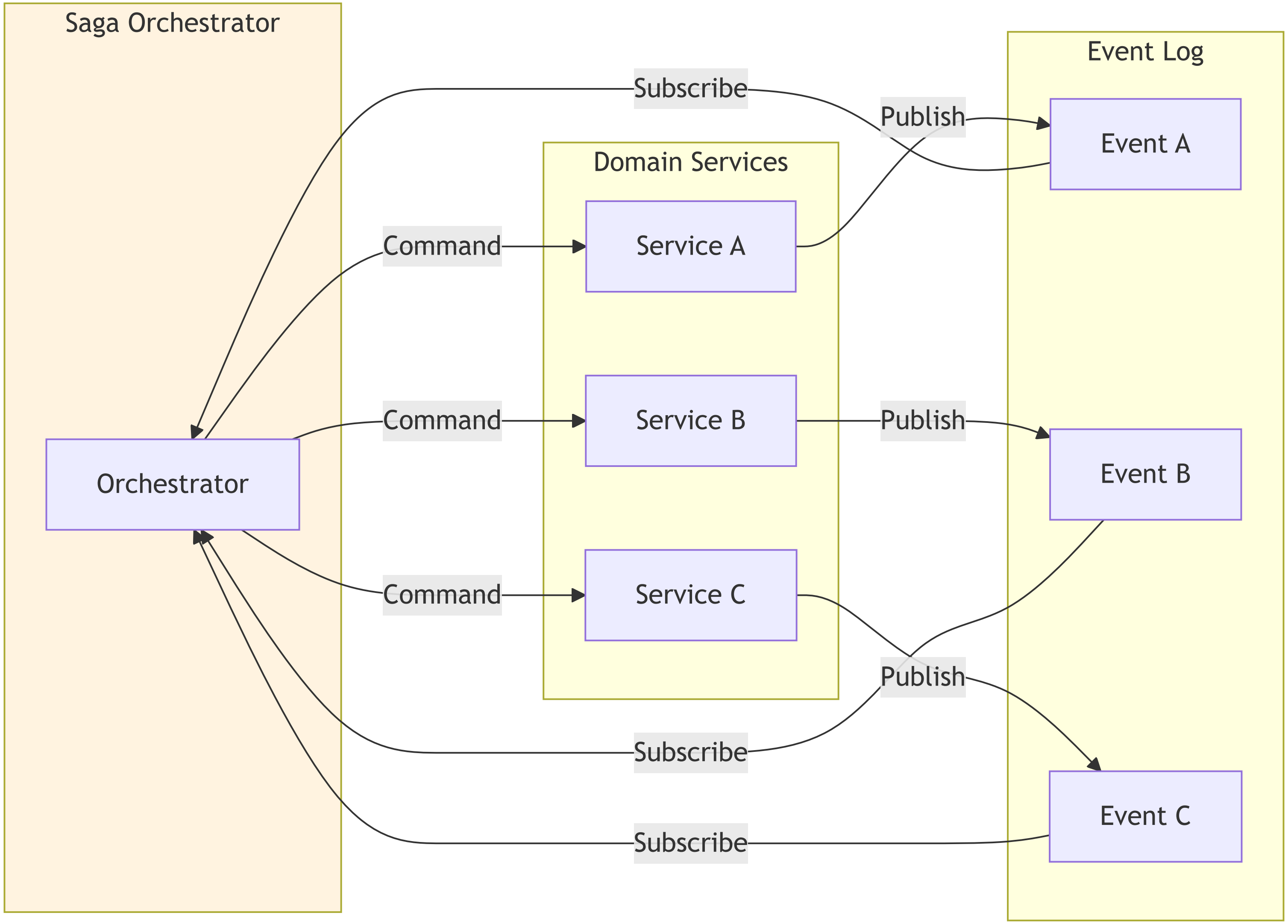
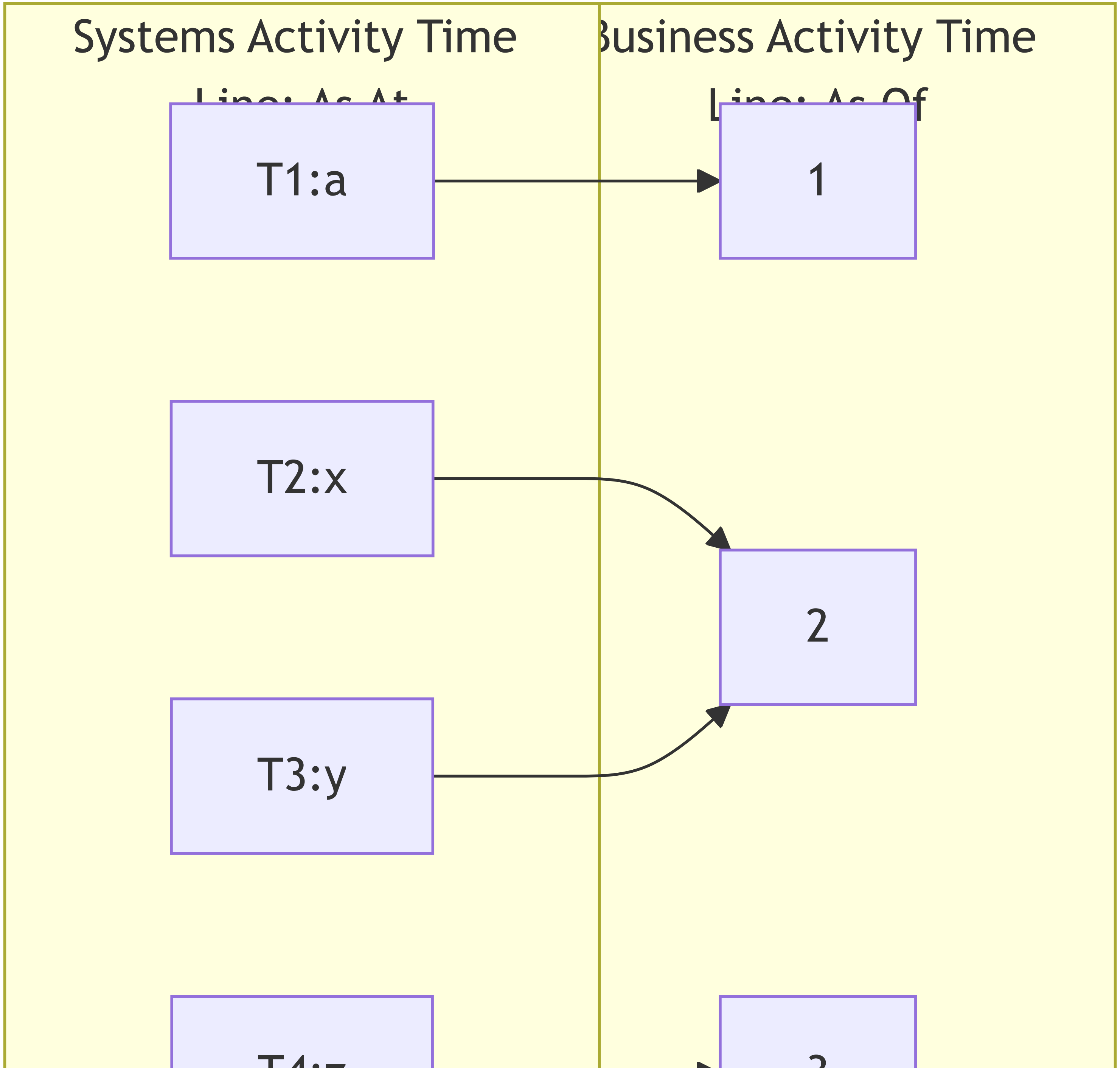


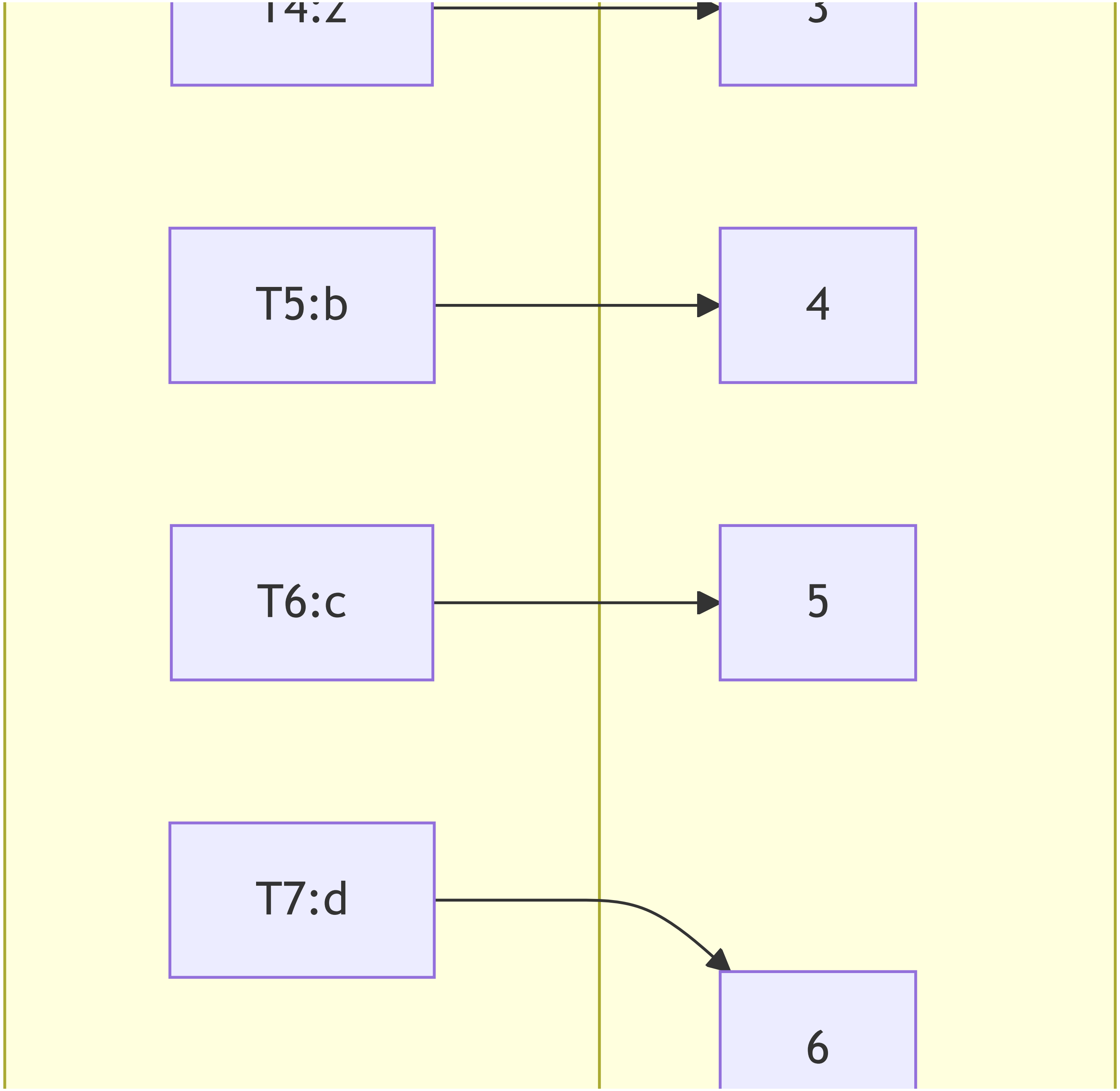
Diagram 19

Key benefits: - No direct service-to-service dependencies - Compensating transactions for rollback - Event log provides complete audit trail - Services remain independently deployable

Slide 21: Bi-Temporal Views

Business View vs Systems View


Systems Activity Time Line: As At
Business Activity Time Line: As Of
T1:a
1
T2:x
2
T3:y
T4:z
3


T4:Z
3
T5:b
4
T6:c
5
T7:d
6

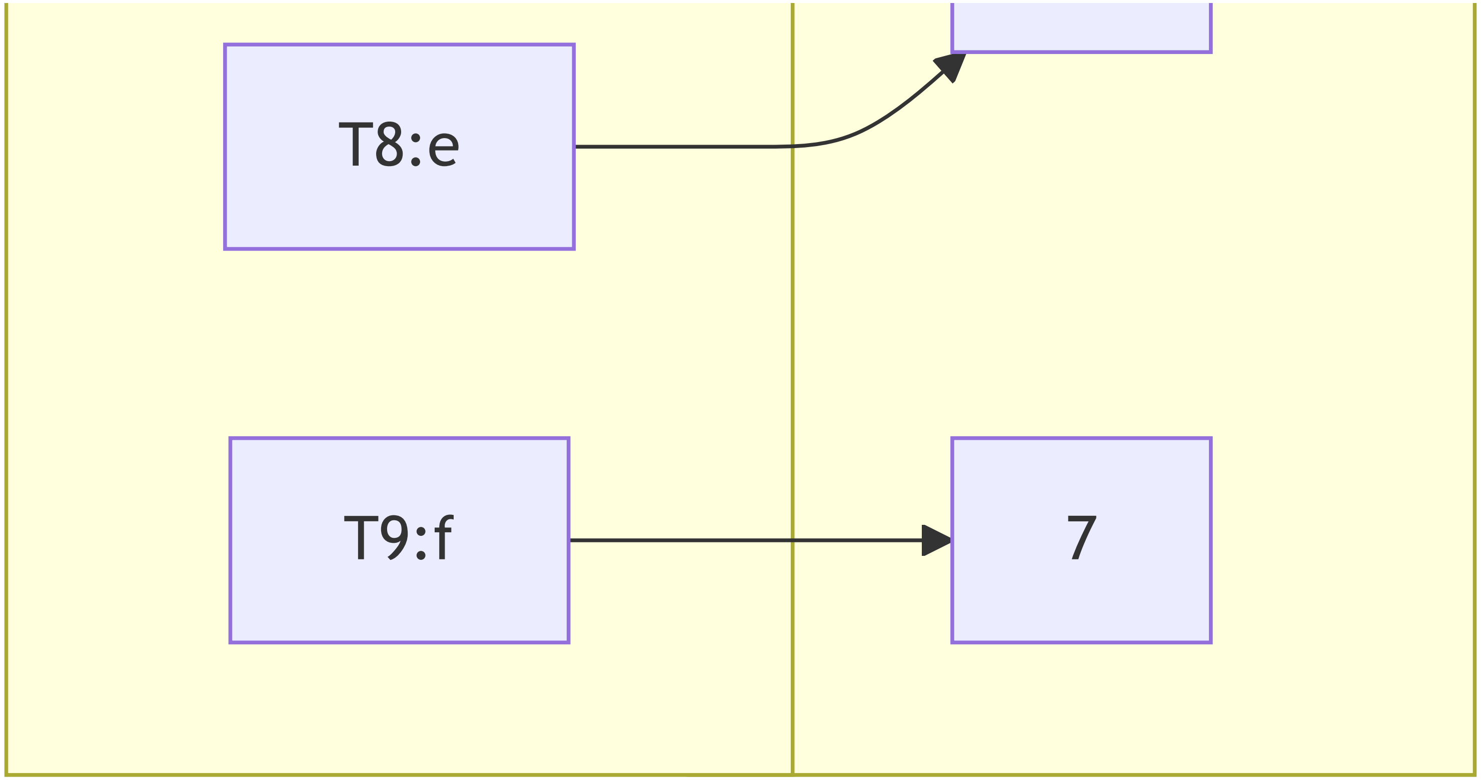


Diagram 20

Two timelines: - **As At**: When the system recorded the event - **As Of**: When the business event actually occurred

View: As of Till - Business activity timeline is a view of the underlying system events.

Slide 22: Information Systems & Intentional Design

From Reality to Composable Architecture

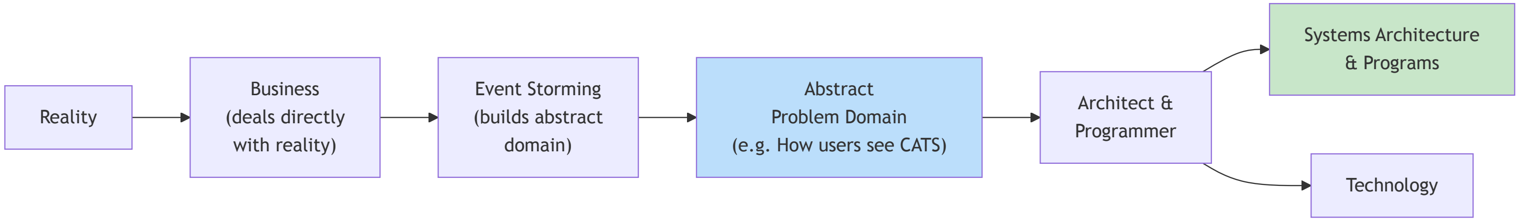

Reality
Business (deals directly with reality)
Event Storming (builds abstract domain)
Abstract Problem Domain (e.g. How users see CATS)
Architect & Programmer
Systems Architecture & Programs
Technology

Diagram 21

Event Storming helps build the abstract domain by focusing on what actually happens in the business.

Slide 23: Parallel Registration Strategy

Phased Migration Approach

Stage 1: Full Segregation - Separation on Product Boundaries - Users register independently - **Data unified at Registered Person Records** coming out of both systems - Advantages: Easiest, cheapest, quickest to stand up - Disadvantages: Double registration for cross-product users

Stage 2: Registration Integration - WSA implements registration synchronization from existing system - Consistent user experience with minimal re-keying

Stage 3: Product Migration - Selective controlled product migration from existing systems to WSA

Slide 24: Patterns of Integration

The Two Fundamental Patterns

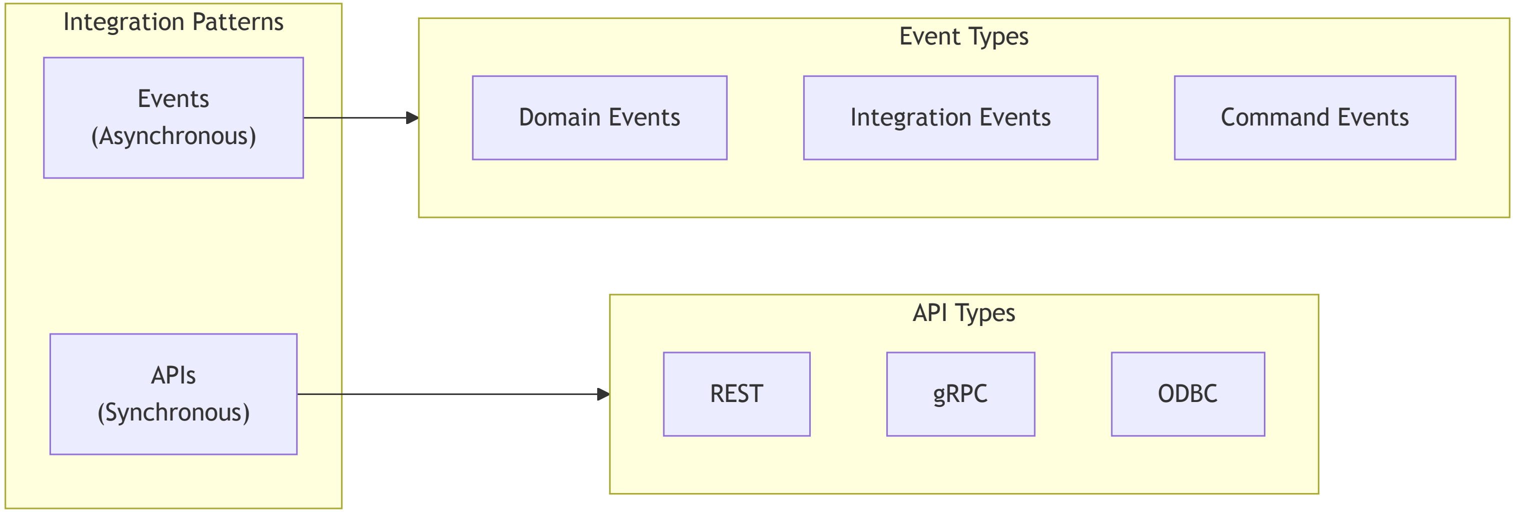


Diagram 22

APIs include all programmatic interfaces such as REST & ODBC.

Slide 25: Highest Value Automation Testing

Outside-In Testing Strategy

For highest value, test from the outside in:

1. **Test from outside to inwards** of your releasable Product

2. Examples:

   - Releasing an **Application** → test its interfaces, imports and exports
   - Releasing a **Library** → test its interface calls

3. **If modules need refactoring** → automate testing their interfaces where possible

4. **BUT**: If your automated API tests exercise the execution paths, then the need for internal testing is greatly mitigated

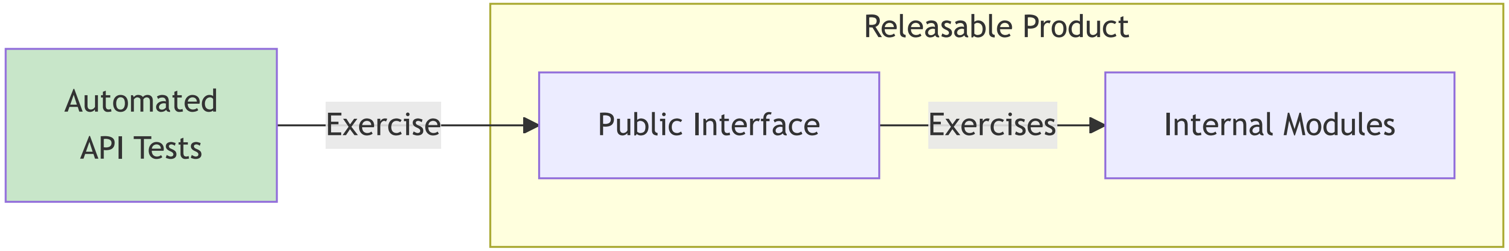


Diagram 23

Slide 26: Summary - The Complete Picture

On-Demand Data Architecture

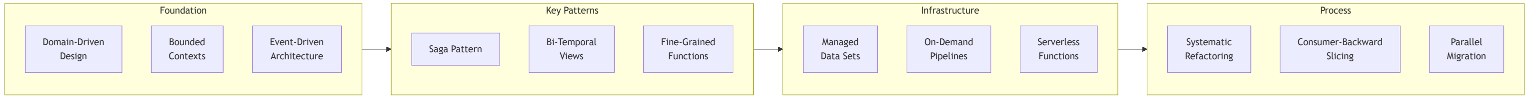


Diagram 24

Key Takeaways:

1. **Domain First** - Refactor towards domain model, then technology
2. **Bounded Contexts** - Manage complexity through separation
3. **Events as First-Class** - Every business event captured and available
4. **Saga for Orchestration** - Avoid dependency chains
5. **Fine-Grained Functions** - Improve lineage through decomposition
6. **Consumer-Backward** - Start from the end and work backwards
7. **Parallel Migration** - Staged approach minimizes risk

This presentation covers the architecture patterns for building composable, event-driven, domain-modeled enterprise data systems.

Version: 1.0 **Date**: February 2026