BEARS-1101: Handle EVENT_BETTING_CONTROLS_UPDATED Kafka Message

SARATH RAMACHANDRAN

Apprenticeship Activity Report

Introduction

In this activity, I was assigned the BEARS-1101 ticket during a sprint planning session by my team lead. The business need was driven by a requirement from Base Pricing stakeholders to improve customer fairness during in-play betting by implementing **bet delays**. These delays help prevent users from exploiting timing gaps between event outcomes and system updates.

The application is a **Kafka-based Spring Boot microservice** that consumes **event-driven kafka messages**, processes them, and updates a **DynamoDB** table with the latest event data. My responsibility was to handle the `*EVENT_BETTING_CONTROLS_UPDATED*` Kafka message and correctly integrate the bet delay information into the existing `Event` schema stored in DynamoDB.

I worked in a cross-functional agile team of developers, QA engineers, DevOps, and business stakeholders, attending stand-ups, retrospectives, and sprint ceremonies. The work was logged in Jira and GitLab, with clear acceptance criteria, source control, and CI pipelines.

Requirement Gathering

The requirement was delivered via a Jira ticket (BEARS-1101) with acceptance criteria, a sample Kafka message payload, and a Confluence reference page. The goal was to:

- Parse the incoming message
- Extract and write `**betDelayDefault**` to the event level
- Optionally, if `**marketBettingControlsUpdated**` exists, apply the delay to that market

Tools & Sources:
- Jira (Kanban board)
- Confluence (GTP Outbound Message Spec)
- Kafka Schema

Stakeholders Involved:
- Base Pricing Product Owner
- Lead Developer
- QA
- DevOps Engineer

We worked within an Agile sprint. Requirements were tracked in Jira using Kanban-style boards, with daily stand-ups and planning sessions providing further clarification. The team follows Agile methodology with two-week sprints. Tickets progress through *To Do → In Progress → Code Review → Testing → Done stages*.

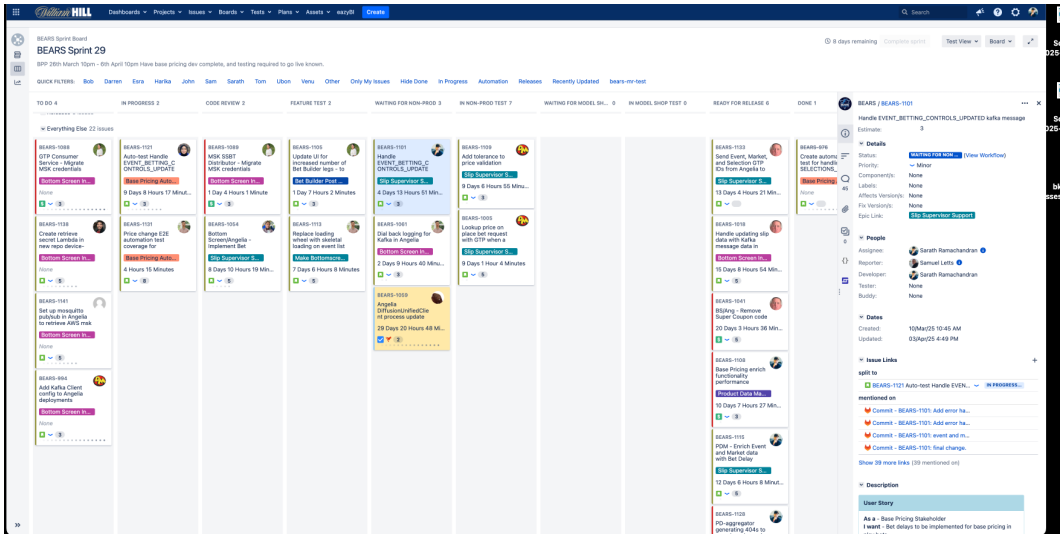

Figure 1 Sprint Board

Analysis

To understand where the new message fits:
- I reviewed the **DynamoDbService.java**, which handles existing event types.
- I explored related test files like `**MessageListenerTest**` and `**EventPricedMergerTest**`.
- I analyzed the `Event` model and verified how the `betDelay` field is structured at both event and market levels.

Key decisions:
- Reuse existing builder-based POJOs (`Event`, `Market`, etc.)
- Follow the structure of similar Kafka handlers (`EventStatusUpdatedMessage`, etc.)

Design

The plan was to:
- Extend the service class to support the new message type
- Use the builder pattern to modify nested structures safely (e.g., `Market.builder().betDelay(...)`)
- Reuse existing `mergeMarkets()` logic to ensure selections remain intact

A UML sketch was created showing the message flow from Kafka → `MessageListener` → `DynamoDbService` → DynamoDB write operation.

The design was simple: leverage existing structure and add logic to process EVENT_BETTING_CONTROLS_UPDATED messages. If betDelayDefault is > 0, add it to the

event object. If market-level controls exist, attach those delays to the appropriate markets. We reused the Event builder and merger utility where applicable.
The method `handleEventBettingControlsUpdated` was introduced in DynamoDbService, using the Enhanced DynamoDB Client API to insert or update records.

Development

All work was done in a feature branch `feature/BEARS-1101-betDelay`, rebased frequently against `develop`.

Tools and Stack:
- Java 21 (Amazon Corretto)
- Spring Boot 3.4
- IntelliJ IDEA
- DynamoDbEnhancedClient
- Mockito (for test mocking)
- JUnit 5
- Maven / GitLab CI

Key implementations:
- Created method:
`handleEventBettingControlsUpdated(EventBettingControlsUpdatedMessage)`
- Handled nested updates to `Market` objects
- Reused `EventPricedMerger.mergeMarkets()`
- Ensured `getItem`, `putItem` methods are mockable/testable

I created a new method `handleEventBettingControlsUpdated` within `DynamoDbService`. This method reads values from the Kafka message and inserts them into the DynamoDB table via an enhanced client. I used builder-style POJOs to compose the new Event and Market records. Unit tests were written to validate conditional handling of `betDelayDefault` and nested `marketBettingControlsUpdated` values.

Challenges included understanding builder chaining for deeply nested objects, mocking table reads and writes and resolving merge conflicts when working with teammates. I learned to use `argThat()` for verifying deep object graphs, and used reflection to inject mocks during testing.

Code changes were made to DynamoDbService.java, Event.java, Market.java, and supporting utility classes. The key method implemented was `handleEventBettingControlsUpdated`, using the Enhanced DynamoDbClient to retrieve existing records and update them with delays. Builder patterns and safe null handling were adopted throughout.
The code was committed and pushed for review via GitLab merge requests.

Ticket details and code change snippets -

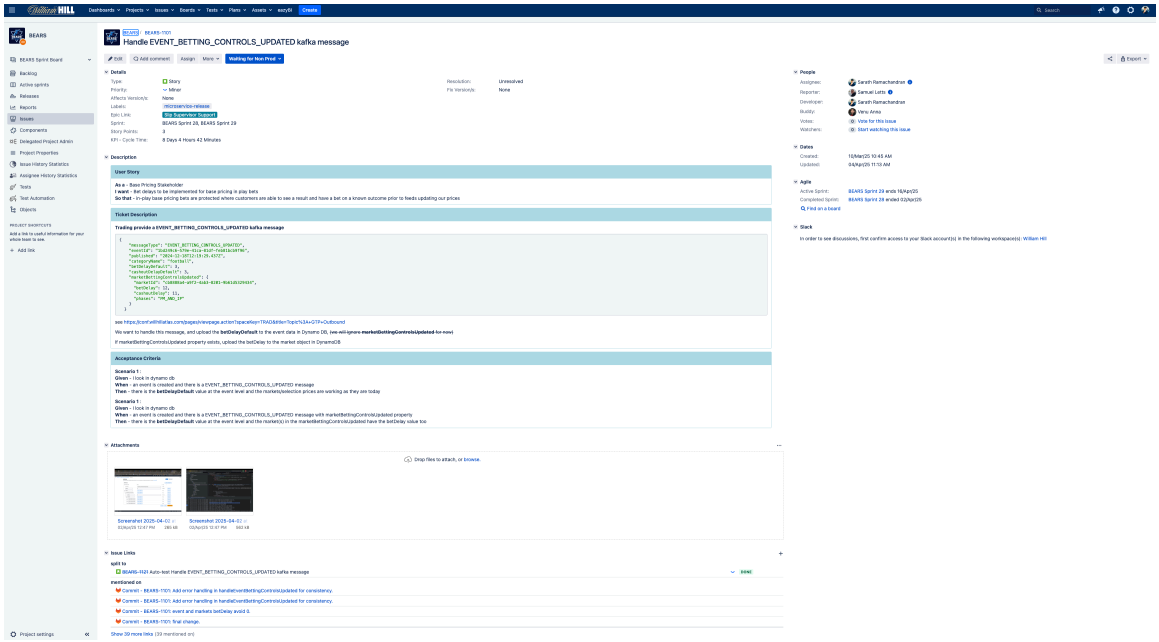

Figure 2 Ticket Description

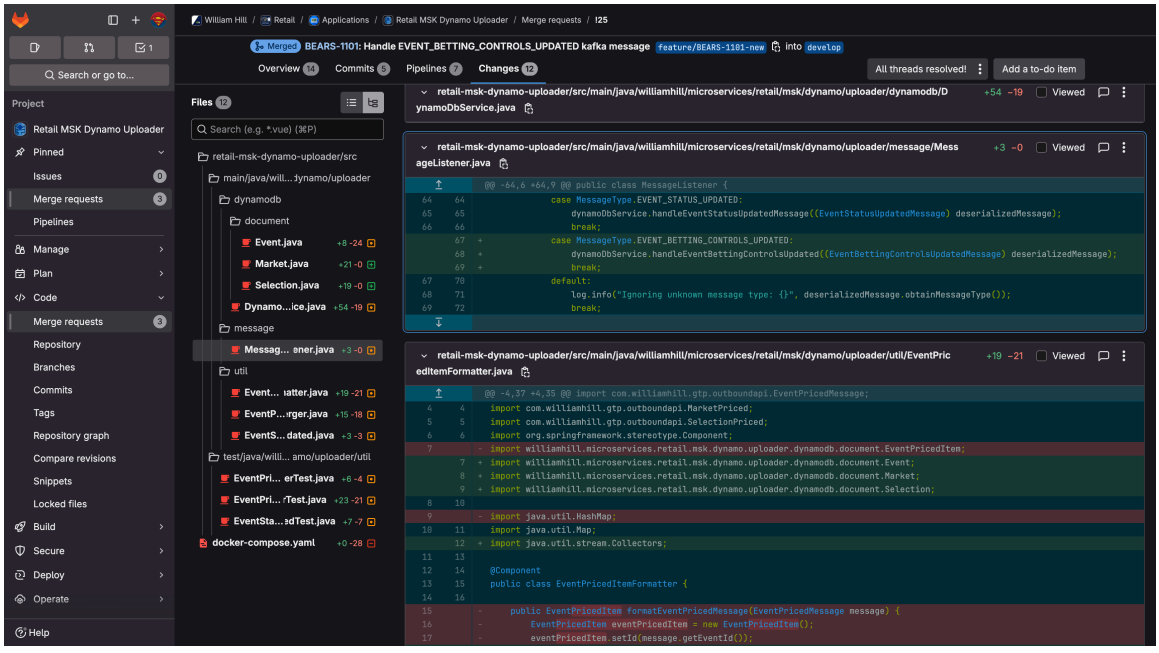

Figure 3 Code changes - MessageListener

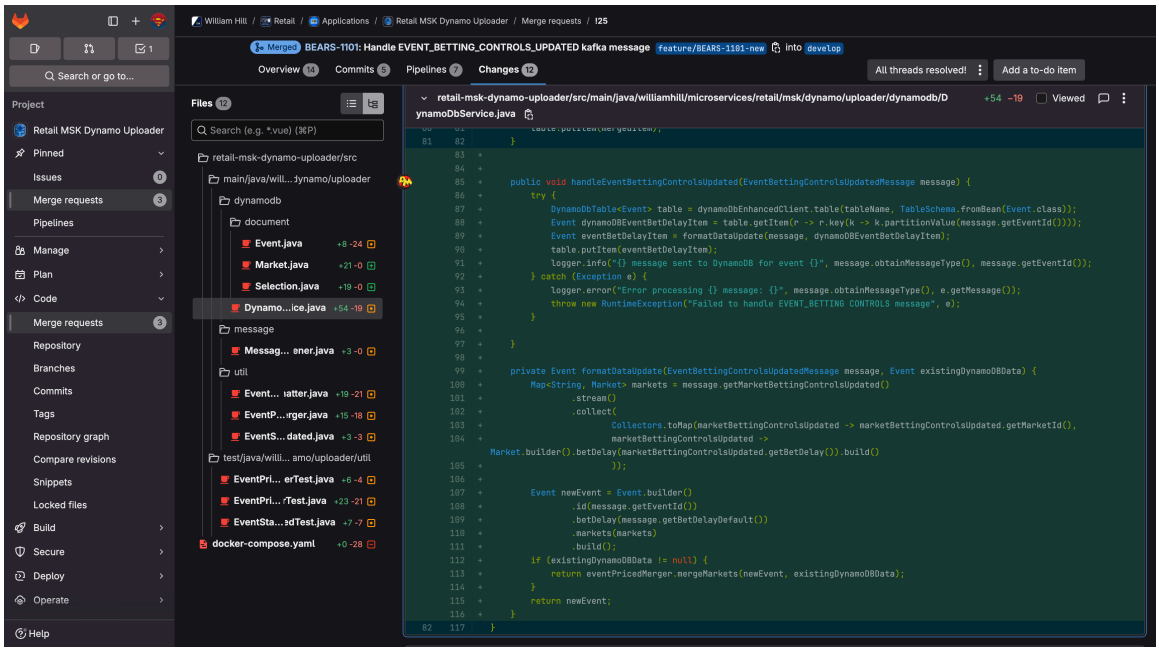

Figure 4 Code changes - DynamoDbService

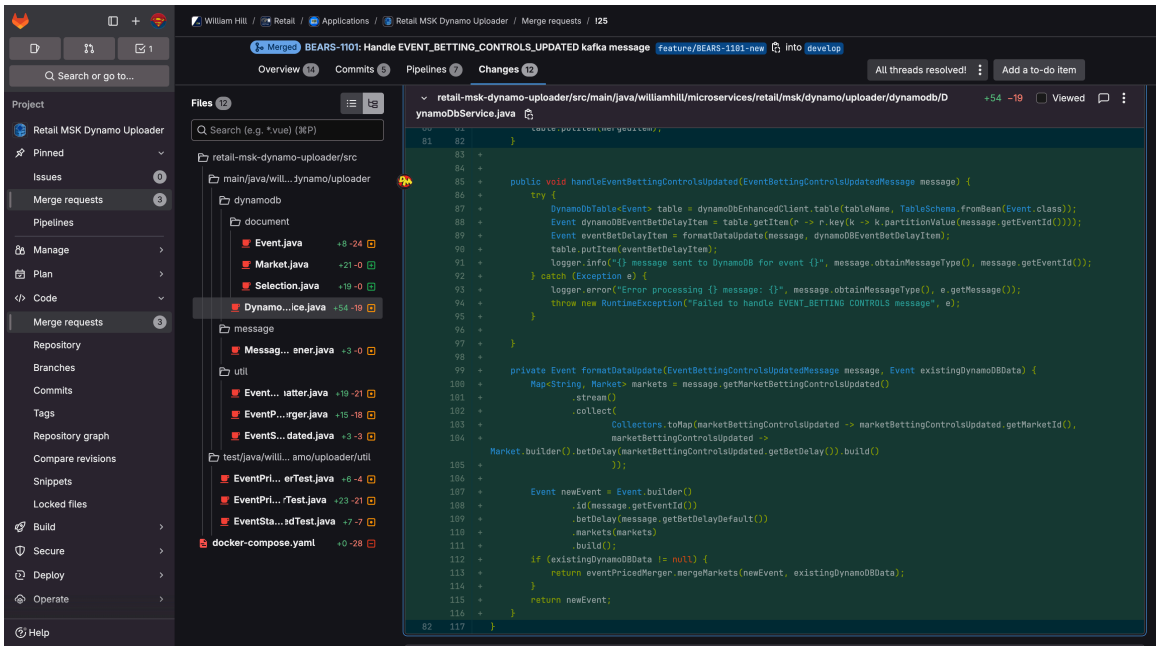

Figure 5 Code changes - DynamoDbService

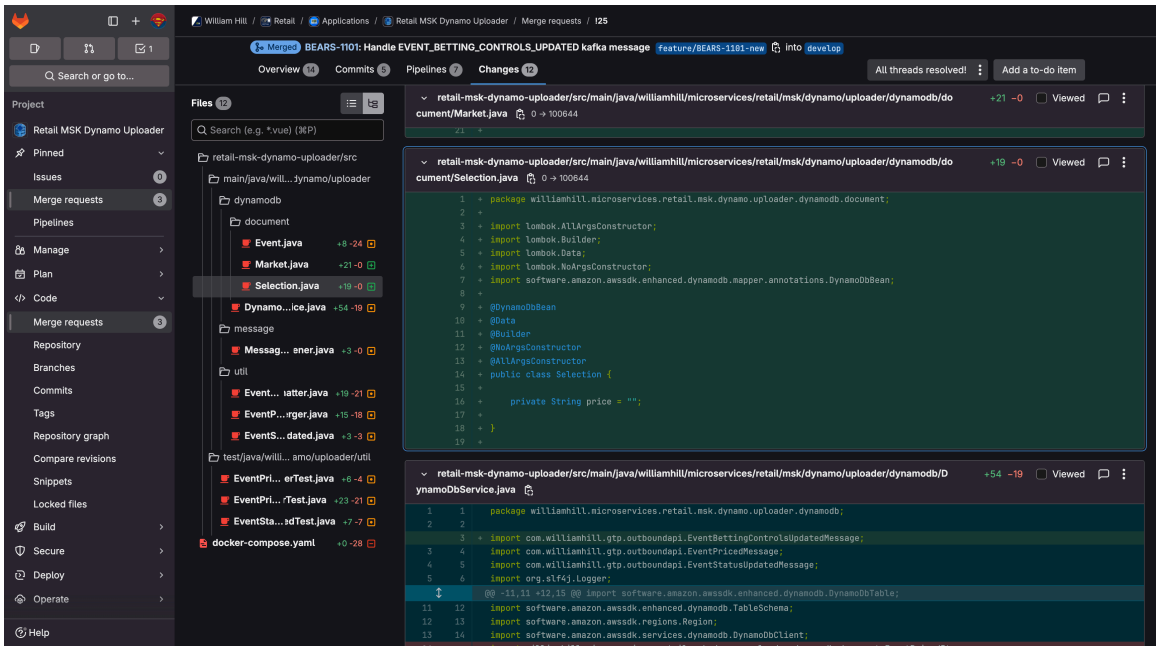

Figure 6 Code changes – Selection

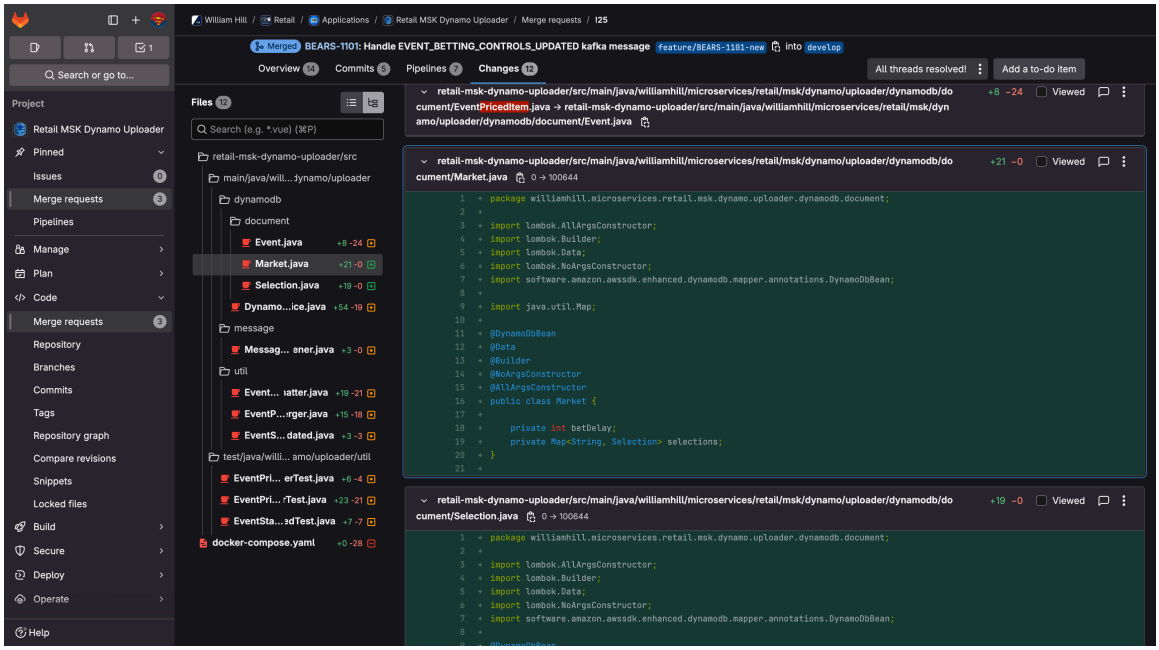

Figure 7 Code changes - Market

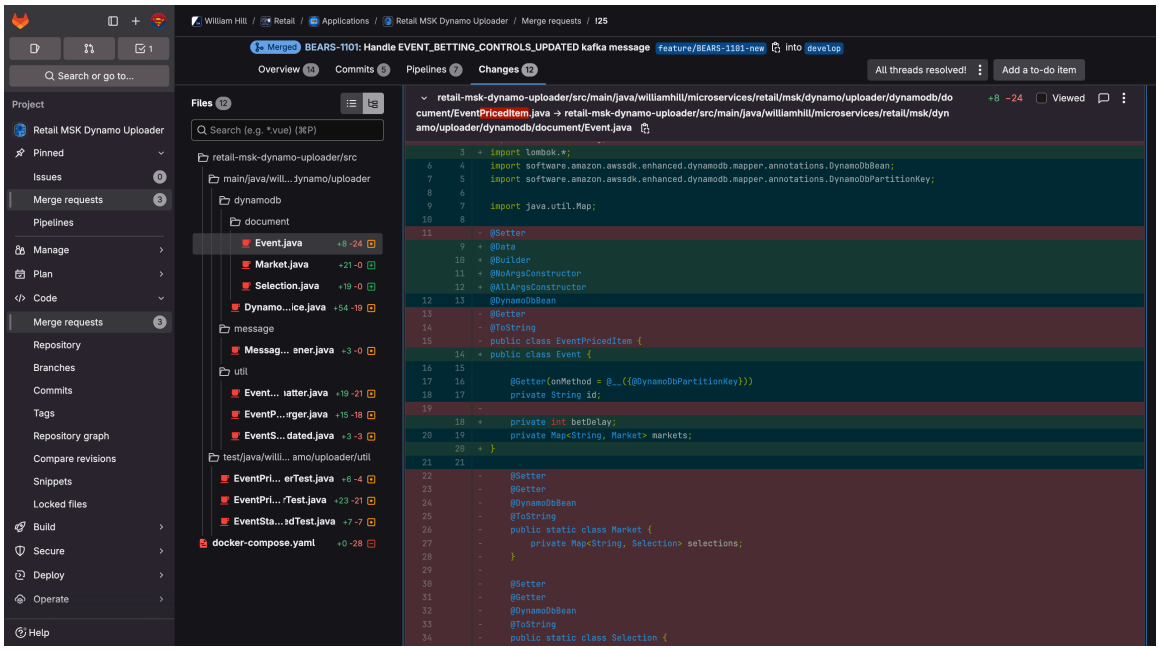

Figure 8 Code changes - Event

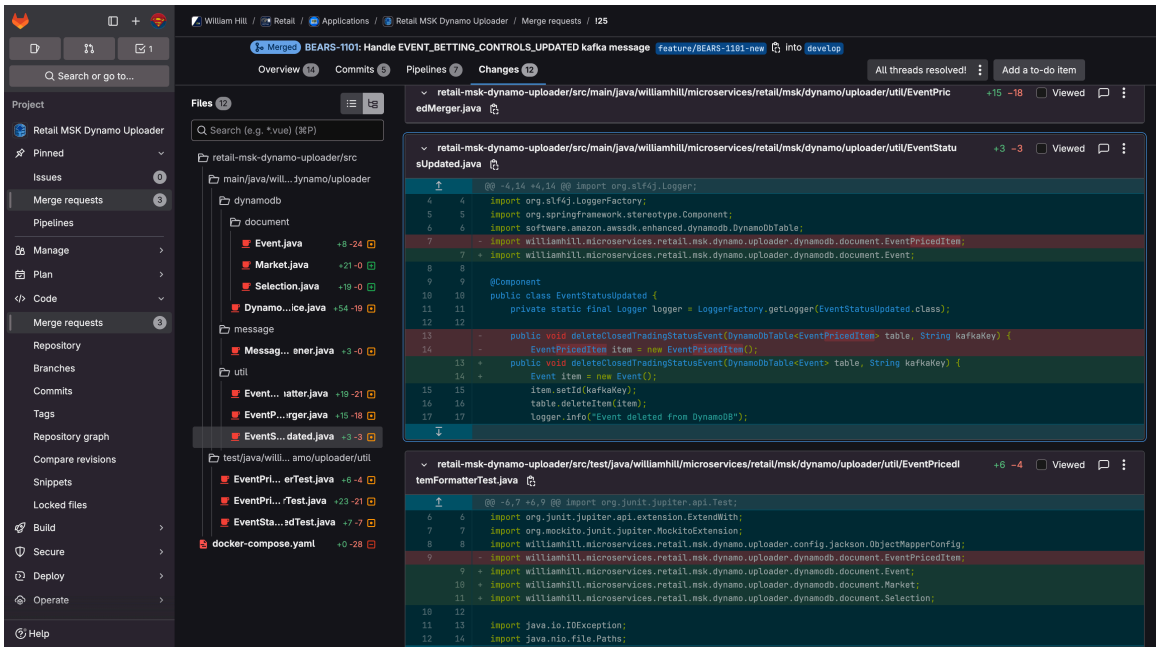

Figure 9 Code changes - EventStatusUpdated

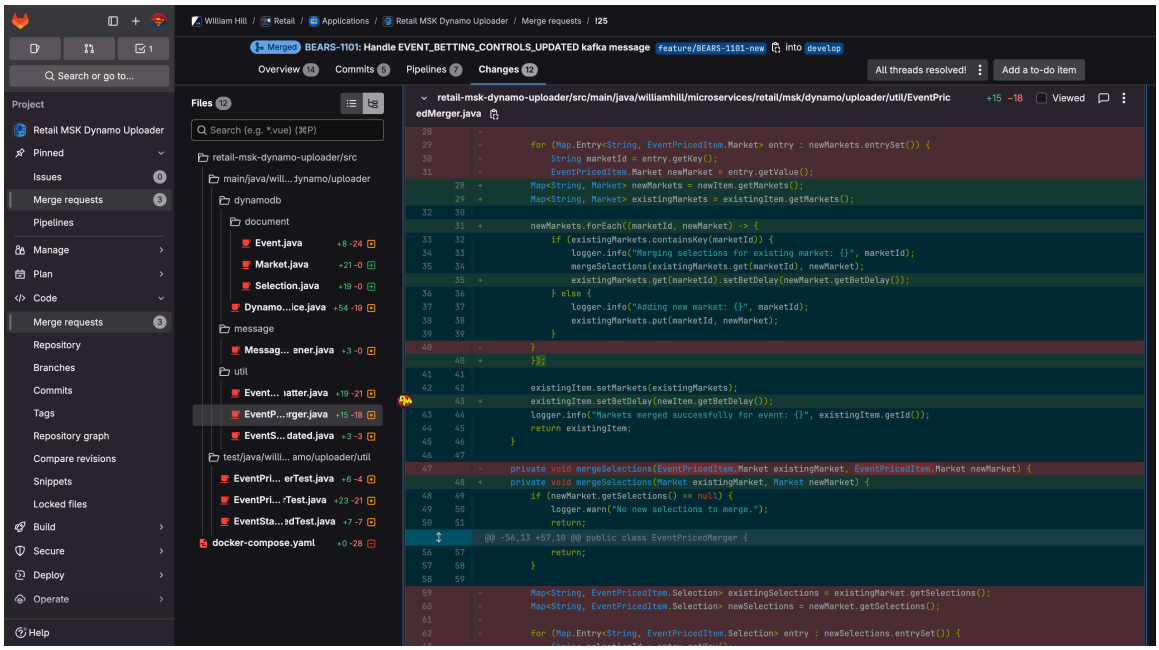

Figure 10 Code changes - EventPricedMerger

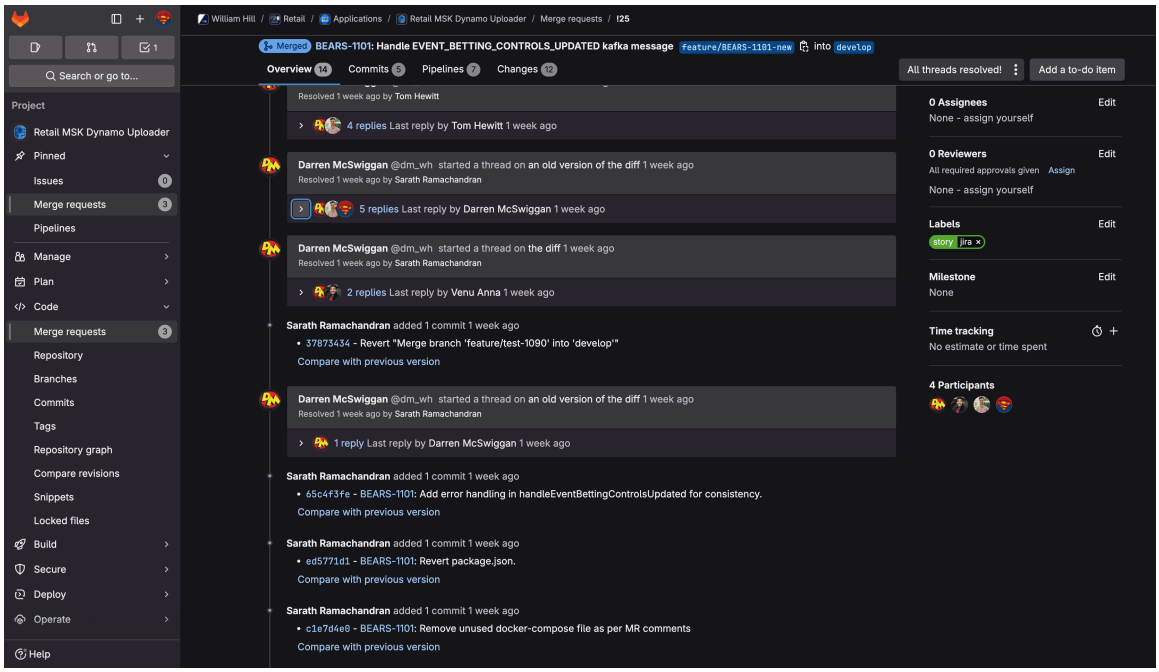

Figure 11 Git logs

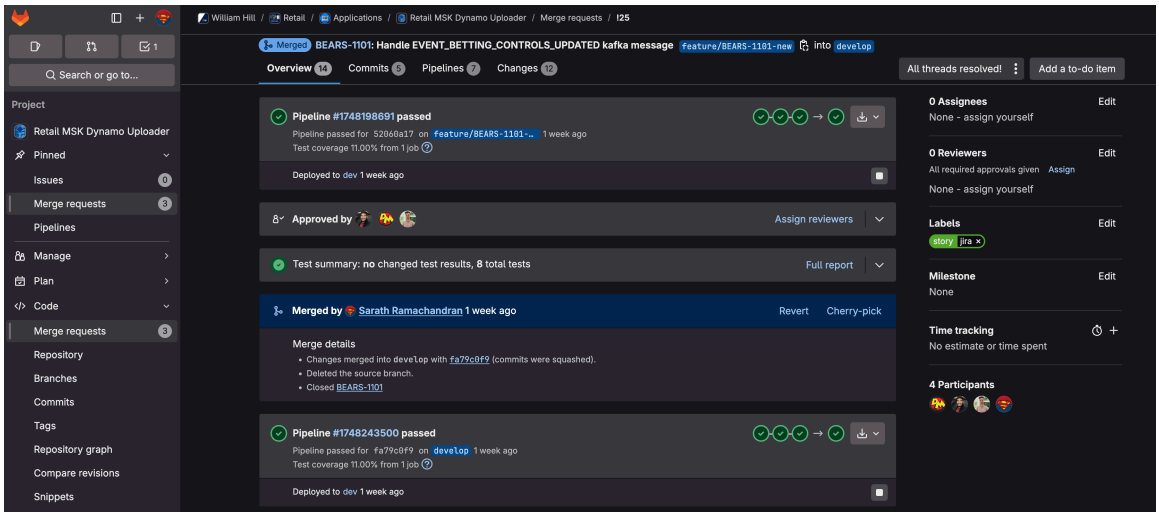

Figure 12 Git Overview

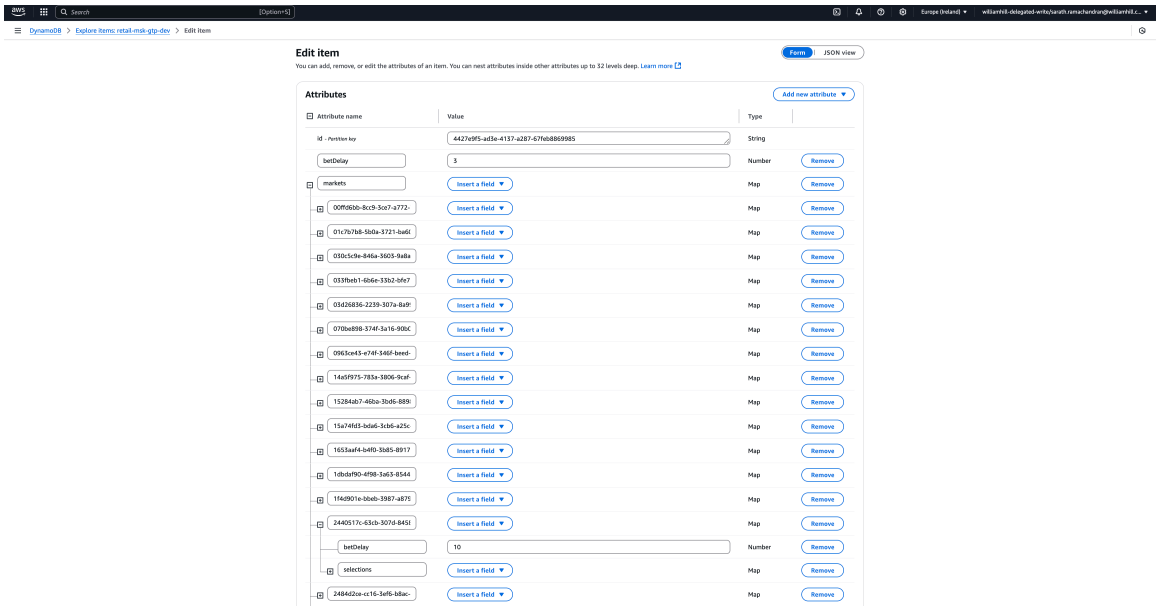

Figure 13 Updated Dynamo DB table

Debugging

Debugging & Issue Resolution During development, I encountered several issues:

- Lombok builder methods not working due to missing annotation processing
- IntelliJ misconfiguration leading to unresolved symbols
- Maven dependency resolution failures (1.0.0 artifact not found)

To resolve them:

- I enabled annotation processing in IntelliJ
- Reloaded Maven projects from disk and cleared caches
- Used mvn -U clean install and updated dependency versions

Debugging was done by adding log statements, checking stack traces, comparing working test files, and using IntelliJ's debugger to step through object creation logic.

Testing

Unit tests were written in DynamoDbServiceTest.java using JUnit 5 and Mockito. Three scenarios were validated:

- Skip update if betDelay is zero
- Insert new event with default and market-level delay
- Update existing event with delay values

These tests ensured no writes occurred when `betDelayDefault` was 0 and verified correct behavior with full payloads. Unit tests validate message transformation, merging logic, and DynamoDB writes. We used Maven to build and test locally, and tests ran successfully in the CI pipeline.

Unit testing allowed quick validation and confidence in the integration logic. I mocked Kafka message classes and nested builder objects, and asserted the final objects passed to DynamoDB using `argThat()` with deep matching logic.

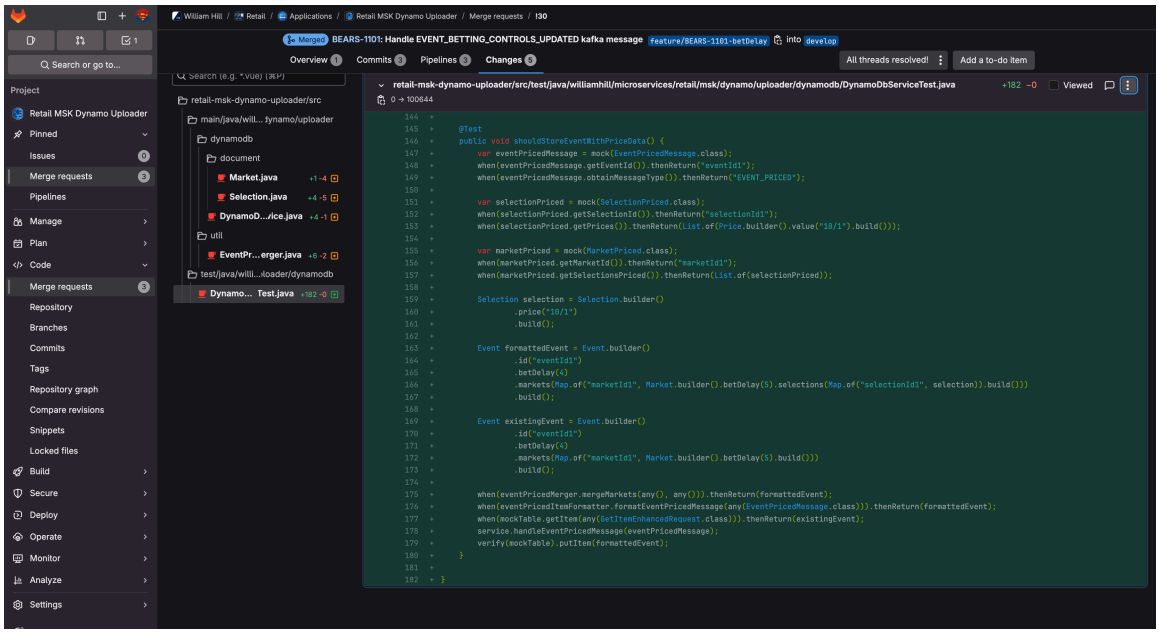

Figure 14 Code changes - DynamoDbServiceTest

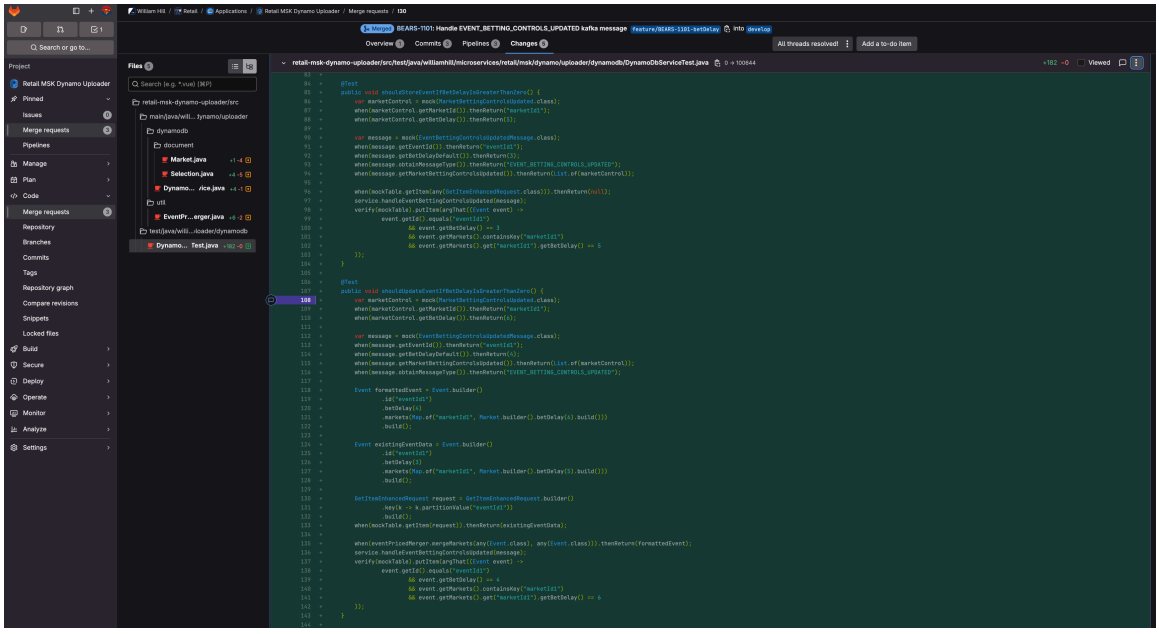

Figure 15 Code changes - DynamoDbServiceTest

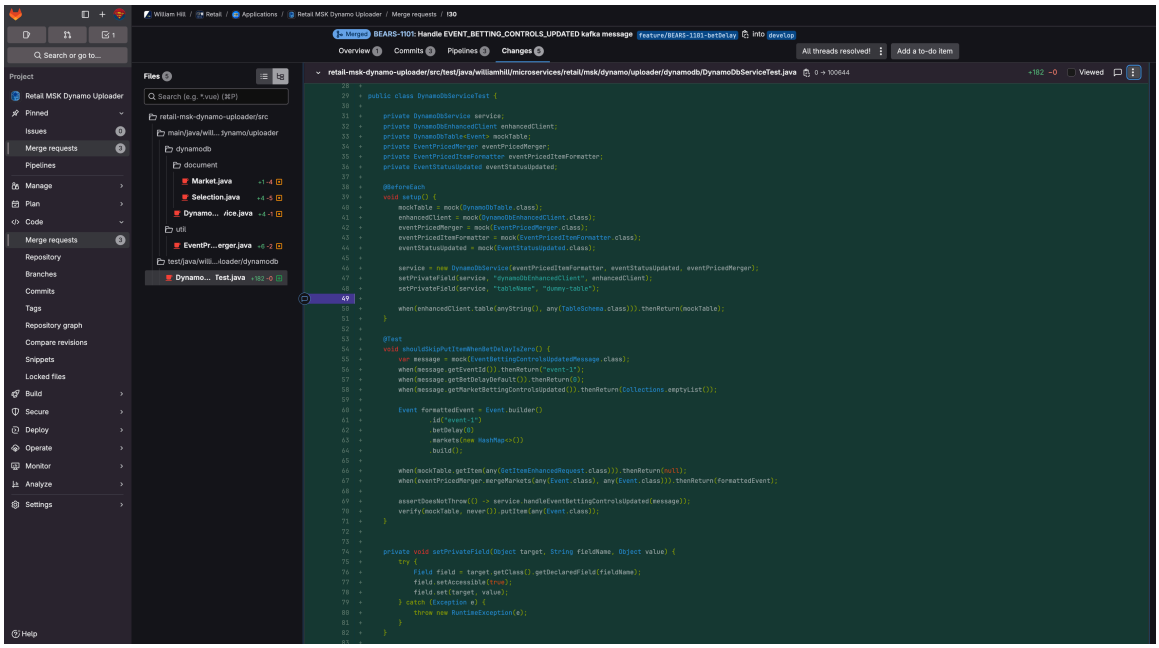

Figure 16 Code changes - DynamoDbServiceTest

Communication

Throughout this task, I maintained regular communication with my team to ensure clarity and alignment. Key interactions included:

• **Daily Stand-ups**: I provided updates on the ticket's progress during morning team calls, discussing blockers and next steps.

• **Peer Reviews & Feedback**: I collaborated with a teammate who had also worked on the same feature branch previously. I reviewed their code and incorporated some of their logic for consistency and maintainability.

• **Slack Conversations**: I shared implementation doubts, test coverage ideas, and mocking strategies with the team via Slack. When I faced issues with ambiguous builder methods or mocking nested classes, I reached out for guidance.

• **Merge Request Discussions**: I submitted my code via a GitLab Merge Request, where I responded to reviewer comments and made necessary improvements. This helped refine both logic and test coverage.

• **Pair Debugging**: For test failures and runtime issues, I sat with my senior developer for pair debugging, which helped resolve builder-related edge cases effectively.

This open and collaborative approach helped ensure that the feature aligned with team expectations and coding standards and helped me upskill through real-time guidance.

Result

The final implementation was merged into the develop branch after successful review. DynamoDB now stores bet delays for in-play events as expected. Unit tests cover all functional scenarios.

I used Git to create a feature branch, commit changes, and resolve conflicts with teammate commits. Each commit message was clear and referenced the ticket. Screenshots of Git log and ticket board are attached. Version control was maintained via GitLab, with branches rebased and updated as needed. The Maven build used JaCoCo and other plugins for code coverage and compilation.

The feature was visible in the sprint demo, and acceptance criteria from the ticket were marked as completed in JIRA.

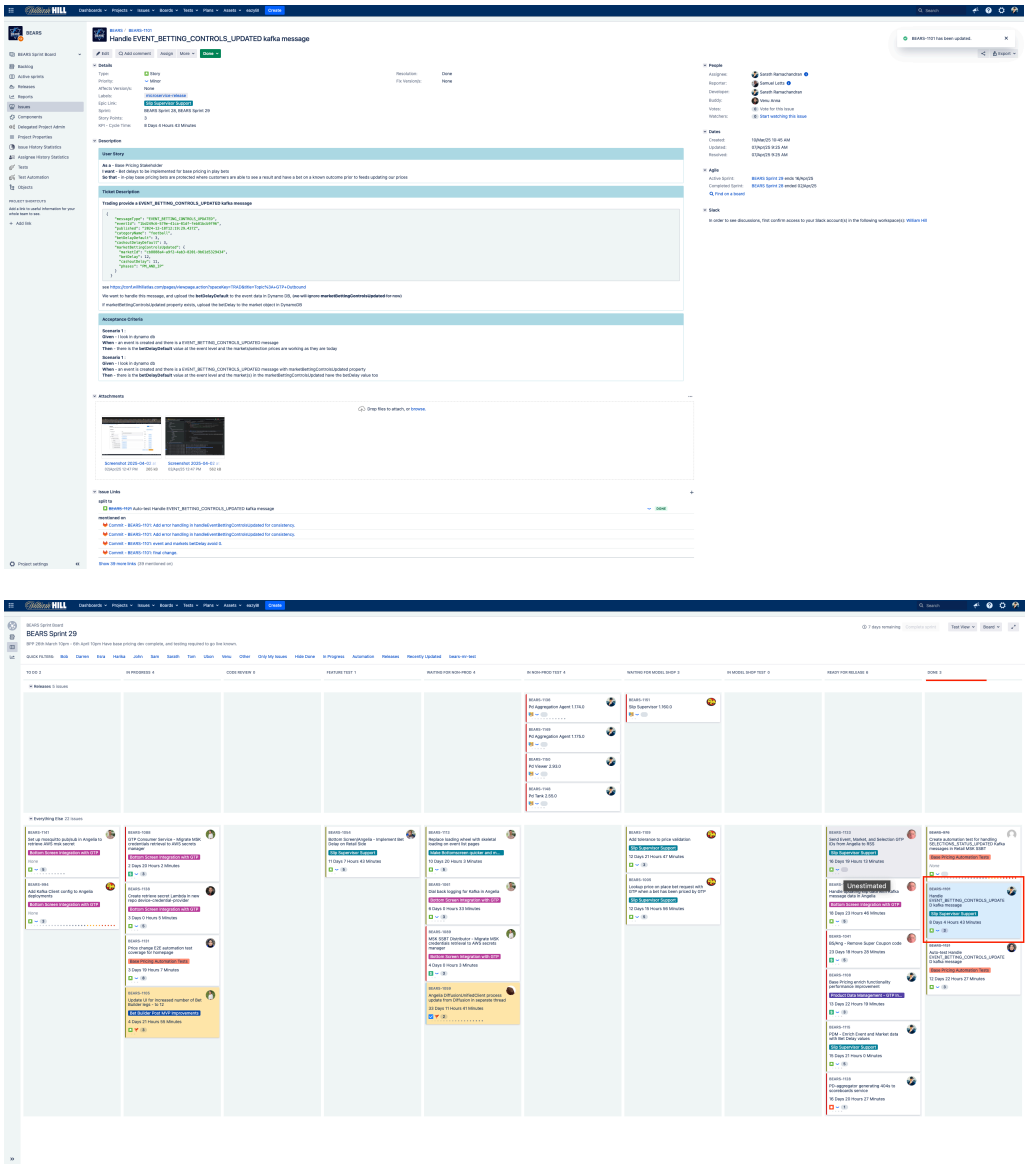

Figure 17 Ticket status after successful completion

Reflection

This activity significantly enhanced my understanding of real-world backend development and improved my confidence across full SDLC stages. I learned how to:

- Extend event-driven services with new Kafka message types
- Mock nested and builder-based POJOs using `Mockito`
- Write precise assertions using `argThat()` and lambda matchers
- Understand and modify code working with DynamoDbEnhancedClient
- Troubleshoot complex issues including IntelliJ settings, Lombok builder limitations, and Maven errors
- Perform Git rebase, conflict resolution, and align my work with `develop`

I also became more confident in:
- Contributing to shared unit test suites
- Writing production-safe, readable code with encapsulation and builder chaining
- Communicating progress effectively during stand-ups and merge requests

This hands-on work has been one of the most complete, satisfying features I've delivered independently. Overall, the feature delivery was smooth and aligned with Agile expectations. This task deepened my knowledge of our microservice stack, encouraged best practices in testing and mocking, and increased my ownership confidence from ticket analysis to test delivery.