# DAT250 FeedApp Report

Erlend André Høntorp, Jonas Heen Opsahl and Thomas Tolo Jensen

November 19, 2025

**Abstract**

Live polling applications need to keep data consistent across many users while staying fast and responsive. FeedApp is a real-time polling platform where users can create polls, vote, and discuss results in comment threads, with updates appearing instantly for everyone. This project evaluates Apache Kafka against RabbitMQ as event streaming platforms for handling real-time updates in web applications. We built the system with a Spring Boot backend and React frontend, using Kafka's topic-per-poll approach along with Redis for caching and WebSockets for push notifications. Through benchmarking with OpenMessaging Benchmark framework, we tested hypotheses around throughput, scalability, and latency under different workloads simulating high-volume vote ingestion and real-time fan-out to clients. Our experiments showed that Kafka achieved significantly higher throughput and better horizontal scaling for write-heavy workloads, while RabbitMQ demonstrated lower latency at low message rates due to its push model. However, Kafka's performance advantage increased under high load as RabbitMQ's per-message overhead became more costly. Creating separate Kafka topics for each poll worked well for keeping things isolated, though it required extra management overhead. The main takeaways from this project are understanding the throughput-latency trade-offs between log-based and queue-based messaging systems, when event-driven architectures like Kafka justify their complexity, and how to handle cache invalidation properly in distributed systems.

## Contents

# 1 Introduction

## 1.1 Project Overview

FeedApp is a small web app for running live polls. Users can create a poll with options, vote, and discuss the results in a comment thread. The goal is to make collecting feedback simple and fast. Instead of refreshing the page or counting results by hand, the app pushes updates in real time so everyone sees new votes and comments right away.

The system has two parts: a backend and a frontend. The backend exposes REST endpoints for polls, votes and comments, and it emits events when data changes. The frontend shows the latest state, listens for WebSocket messages, and keeps the UI in sync with the server. The project is also used to try out event-driven ideas and see how they scale when more users join.

## 1.2 Technology Stack Summary

Backend: Java with Spring Boot and Spring Security for REST APIs, validation and auth. During development we store data in an in-memory H2 database. Redis is used as a cache for frequently read data like poll results. Kafka is our messaging backbone for domain events (for example, a vote that was cast).
Frontend: React with TypeScript for a simple, component-based UI that reacts to server events over a single WebSocket connection.
Infra: Docker and Docker Compose run all services with the same settings on any machine.

We chose this stack because it is familiar, easy to set up, and supports horizontal growth. Kafka helps decouple writes from notifications, Redis improves read performance, and WebSockets keep the UI responsive.

## 1.3 Project Outcome

The prototype supports creating polls, voting with instant updates, and threaded comments with edits and replies. We also compared Kafka with RabbitMQ for the event stream. In short: Kafka handled higher throughput and scaled better under heavy write load, while RabbitMQ showed slightly lower latency at low message rates. Our topic-per-poll approach in Kafka kept events neatly separated, and Redis caching reduced repeated work on popular polls. Everything is containerized so the full stack starts with one command.

### 1.4 Report Organization

This report is organized as follows. Section 2 explains the design: main use cases, the domain model, and the overall architecture. Section 3 introduces Kafka (and the supporting tech), and outlines our experiment plan and assumptions. Section 4 describes how we built the prototype, how services talk to each other, and how to run it locally. Finally, Section 5 sums up what we learned and lists possible next steps.

## 2 Design

Around 5 pages about functional aspects of the FeedApp application.

### 2.1 Use Cases

Describe:

- Main functionality from the user's perspective,
- Actors: Guest User, Registered User, System,
- Core use cases: Register/Login, Create Poll, Vote, Comment and Managing Polls,
- Brief explanation of real-time updates and visibility (public/private polls)

Optional:

- Use Case Diagram (Figure: Use case overview)

### 2.2 Domain Model

Describe:

- Core domain entities (User, Poll, Vote, VoteOption, Comment(?)),
- Relationships between them (one-to-many, many-to-one),
- Persistence through JPA/Hibernate

Optional:

- UML Class Diagram (Figure: Domain Model Overview)
  Show main entities, attributes, and associations

### 2.3 Architecture (including applied technologies)

Describe:

- High-level architecture (frontend–backend–infrastructure),
- Backend layers: Controller, Service, Repository,
- Key technologies: Spring Boot, H2, Redis, Kafka, WebSocket, JWT,
- Frontend technologies: React, TypeScript, REST API communication,
- Interaction between components (data flow from frontend to backend),

Optional:

- Architecture Diagram (Figure: System Architecture)
  Show containers/components (Frontend, Backend, Redis, Kafka, DB, WebSocket)

### 2.4 Summary

Summarize:

- How the design supports scalability, modularity, and real-time updates,
- The rationale behind chosen technologies,
- How the design connects to later implementation and deployment steps

# 3 Technology Assessment

(Remove later: Introduce in sufficient depth the key concepts and architecture of the chosen software technology. As part of this, you may consider using a running example to introduce the technology.)

At its core, Apache Kafka is a distributed event streaming platform. Think of it as a highly scalable, fault-tolerant, and durable ”log file” that different parts of an application can write to and read from simultaneously. The `FeedApp` project uses these core concepts to decouple tasks. When a user votes, the system does not immediately try to update every other user's screen. Instead, it publishes an "event" to Kafka, and other services (like the WebSocket broadcaster) react to that event.

**Event:**

An event is the most basic unit of data, representing the fact that "something happened." In the `FeedApp`, an event is a `Map<String, Object>` that gets serialized into JSON. For example, when a poll is created, the `PollEventListener` creates an event with the `pollId`, `question`, and `eventType`. The consumer listens for vote change events, which it receives as a `Map` containing the `pollId` and `optionOrder`.

**Topic:**

A topic is a named "category" or "feed" where events are stored and published. A cluster can host many topics, and with KRaft mode, the number of topics can now scale into the millions [2]. The `FeedApp` uses a dynamic topic strategy. Instead of one giant `votes` topic, the `PollTopicManager` creates a unique topic for every single poll. For example, for a poll with ID `10`, the topic name would be `poll.voteChange.10`.

**Producer:**

A producer is a client application that writes events to a Kafka topic. The `ProducerService` in the FeedApp project is a classic producer, using Spring's `KafkaTemplate` to send events. This service is used by the `PollEventListener` to send a message after a new poll is successfully saved to the database. With the 3-node cluster, the producer (our backend) is given a list of all three brokers (`kafka-1:29092,kafka-2:29092,kafka-3:29092`) via its environment variables. The Kafka client uses this list to discover the entire cluster.

**Consumer:**

A consumer is a client application that reads (subscribes to) events from one or more Kafka topics. The `ConsumerService` acts as the consumer . It uses a powerful feature to match the dynamic topic strategy: it listens to a `topicPattern` (`"poll.voteChange.*"`) instead of a fixed topic name. This allows it to automatically discover and consume from new poll topics as soon as they are created.

**Consumer Group:**

A set of consumers that work together to process events from topics. Kafka guarantees that each event in a topic's partition is delivered to only one consumer instance within that group. The `FeedApp` defines the consumer group ID as `"poll-app"` in its `application.properties` file. If one were to run multiple instances of the backend service for scalability, they would all share this group ID. Kafka would then automatically balance the load of all the poll topics among them.

**Partition:**

Partitions are the core of Kafka's scalability. A topic is split into one or more partitions. Each partition is an ordered, immutable log of events, which can be hosted on different brokers. The `FeedApp` project makes a specific design choice. When the `PollTopicManager` creates a new topic, it is configured with one partition to guarantee strict event ordering for that poll. To match the 3-node cluster's fault-tolerant design, the replication factor is set to 3. This is implemented by calling `new NewTopic` with the parameters `(topicName, 1, (short) 3)`. This configuration ensures the single partition is copied to all three brokers, providing fault tolerance and high availability.

**Broker:**

A broker is a single Kafka server. Brokers receive messages from producers, assign offsets to them, and commit them to the partition log on disk, which provides Kafka's durability. The provided `docker-compose.yml` file defines three distinct broker services: `kafka-1`, `kafka-2`, and `kafka-3`. Each of these nodes is configured with the `broker` role.

**Cluster:**

A Kafka cluster is a group of brokers working together to provide scalability, availability, and fault tolerance. Partitions are replicated on multiple brokers based on the topic's replication factor. The three brokers (`kafka-1`, `kafka-2`, `kafka-3`) form the cluster. This is where the concept of a replication factor of 3 becomes reality. When `PollTopicManager` creates `poll.voteChange.10` with 3 replicas, that topic's single partition is copied to all three brokers. One broker is elected "leader" for that partition (handling all writes), while the other two are "followers." If the leader fails, a follower is promoted, ensuring no data loss. This entire process is managed by the KRaft controller quorum, which in this 3-node setup consists of all three nodes (`KAFKA_CONTROLLER_QUORUM_VOTERS: '1@kafka-1:9093,...'`).

This part and other parts of the report probably needs to refer to figures. Figure 1 from [1] just illustrates how figure can be included in the report.

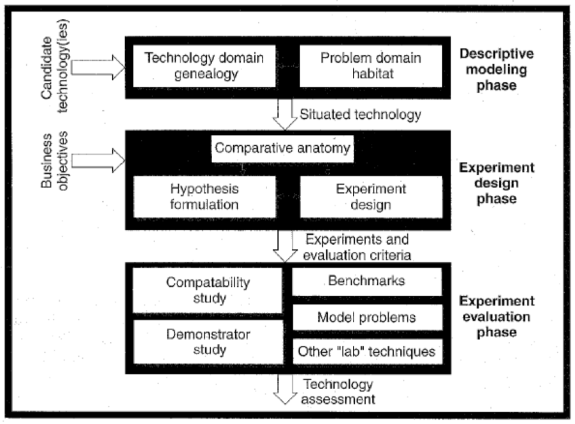

Figure 1: Software technology evaluation framework.

## 3.1 Descriptive Modeling

Remove later: write where the technology comes from, its history, its context and what problem it solves. Consider drawing a graph like in [1].

The purpose of this phase is to establish a formal model for evaluating two candidate technologies, Apache Kafka (in KRaft mode) and RabbitMQ for adoption within our project. First, we will define the Technology Domain Genealogy for both candidates. This model will establish Kafka (KRaft) as an evolution to its ”precursor” (Kafka with ZooKeeper), explaining the problems it was designed to solve. It will also define RabbitMQ’s distinct architectural heritage. Second, we will define our project’s Problem Domain Habitat, the specific usage contexts and project needs, to determine which candidate is a better fit.

### 3.1.1 Technology Domain Genealogy

This model describes the ancestry and core design philosophy of the two competing technologies. Candidate technology 1 is Apache Kafka (KRaft mode). Kafka is, as stated earlier, a distributed event streaming platform. Its core architecture is a persistent, immutable log. It uses a ”dumb broker/smart consumer” model, where the consumers ”pull” data by tracking their position using an offset within the log. The KRaft mode we are looking at is a significant evolution. Its ”precursor” architecture required a separate, external Apache ZooKeeper cluster. This external dependency was known to cause:

1. High operational complexity: Requiring the management and security of two separate distributed systems.

2. Scalabiliy bottlenecks: Limiting clusters to tens of thousands of partitions.

3. Slow recovery times: Controller failover and metadata loading were slow.

KRaft replaces ZooKeeper by integrating a consensus protocol inside Kafka. This was designed to provide simpler operations, massive scalability (to millions of partitions), and near instant, sub-second failover.

Candidate technology 2 is Kafka's peer, RabbitMQ. RabbitMQ is a traditional message broker with a long heritage rooted in reliable delivey protocols like AMQP. It functions as a ”smart broker/dumb consumer” platform. The smart broker manages complex, flexible routing via exchanges and actively pushes messages to consumers. Its core data structure is a queue. Messages are stateful and are typically deleted from the queue once successfully consumed and acknowledged.

### 3.1.2 Problem Domain Habitat

As defined by the evaluation framework, the ”Problem Domain Habitat” describes the specific usage contexts, problem characteristics, and key requirements where the candidate technologies will be deployed. Our habitat is the FeedApp backend, a real-time, interactive polling application built on Spring Boot and Java. The primary objectives for this habitat are to find a messaging solution that is highly scalable, supports high throughput, and simultaneously provides good (low) latency. These objetives can be split into two separate problem domain habitats.

Habitat 1: High-throughput event ingestion

The application must reliably ingest a high volume of events, such as poll creation, comments, and most critically votes from a potentially large and concurrent user base. A user action is sent via the web API, processed, and then published as an immutable event by a producer service. There are a few key requirements to this problem habitat. We prioritize high throughput (handling thousands of writes per second) and scalability (the ability to add more nodes to handle increased load). The durability of these events is critical, as they effecitvely form the application's historical log.

Habitat 2: Real-time asynchronous fan-out

The application must feel responsive and as if event processing happens in real-time, as this is important for a good user experience. When a poll-related event is successfully processed, the system must immediately notift all subscribed clients. A consumer service listens for new events from a topic, it triggers a notification service, which pushes updates to clients via WebSockets. In this habitat we prioritize low end-to-end latency. The time from an events publication to its consuption must be minimal for the application to feel responsive.

## 3.2 Experiment Design

Write you hypotheses about what benefits the technology bring and how you can support or reject them via experiments.

## 3.3 Experiment Evaluation

Write about the results of your experiments, either via personal experience reports, quantitative benchmarks, a demostrator case study or a combination of multiple approaches.

For some reports you may have to include a table with experimental results are other kinds of tables that for instance compares technologies. Table 1 gives an example of how to create a table.

| Config | Property | States | Edges | Peak | E-Time | C-Time | T-Time |
|---|---|---|---|---|---|---|---|
| 22-2 | A | 7,944 | 22,419 | 6.6 % | 7 ms | 42.9% | 485.7% |
| 22-2 | A | 7,944 | 22,419 | 6.6 % | 7 ms | 42.9% | 471.4% |
| 30-2 | B | 14,672 | 41,611 | 4.9 % | 14 ms | 42.9% | 464.3% |
| 30-2 | C | 14,672 | 41,611 | 4.9 % | 15 ms | 40.0% | 420.0% |
| 10-3 | D | 24,052 | 98,671 | 19.8 % | 35 ms | 31.4% | 285.7% |
| 10-3 | E | 24,052 | 98,671 | 19.8 % | 35 ms | 34.3% | 308.6% |

Table 1: Selected experimental results on the communication protocol example.

# 4 Prototype Implementation

This section summarizes how the FeedApp prototype has been implemented, focusing on the core technical components and how developers can run or extend the system.

## 4.1 Setup and Deployment

The FeedApp prototype runs as a multi-container application orchestrated by Docker Compose with four services: backend (Spring Boot, port 8080), frontend (React/Vite, port 5173), Redis (cache, port 6379), and Kafka (event bus, ports 9092/29092). The backend reads infrastructure dependencies from environment variables: `SPRING_KAFKA_BOOTSTRAP_SERVERS=kafka:29092` and `SPRING_REDIS_HOST=redis`.

To run locally, developers execute `docker compose up -d` and access the frontend at `http://localhost:5173`. The system can be stopped with `docker compose down`.

```
backend:
  ports: ["8080:8080"]
  environment:
    - SPRING_KAFKA_BOOTSTRAP_SERVERS=kafka:29092
    - SPRING_REDIS_HOST=redis
frontend: { ports: ["5173:5173"], depends_on: [backend] }
```

The frontend communicates with the backend via REST APIs and maintains a WebSocket connection for real-time push notifications.

## 4.2 Backend Implementation

The backend uses Spring Boot with a layered architecture: Controllers handle HTTP mapping, Services contain business logic, and Repositories manage persistence. The system supports two persistence strategies via Spring profiles: `HibernatePollService` (JPA/Hibernate for production) and `InMemoryPollService` (for testing). Activate the desired mode with `spring.profiles.active=in-memory` or `spring.profiles.active=database` (for example in `application.properties` or as an environment variable). A `CachingPollService` decorator adds Redis-based caching with Jedis, improving performance for frequently accessed data like poll results.

### 4.2.1 Authentication and Security

The application implements stateless JWT authentication with BCrypt password hashing. The `/api/auth/login` endpoint authenticates users and returns a JWT token, while `/api/auth/register` creates new accounts. The `JwtAuthFilter` validates tokens on each request, making the authenticated user available via `@AuthenticationPrincipal`.

```java
// SecurityConfig (rules excerpt)
http.csrf(AbstractHttpConfigurer::disable)
    .sessionManagement(s -> s.sessionCreationPolicy(STATELESS))
    .authorizeHttpRequests(a -> a
        .requestMatchers("/api/auth/**", "/rawws").permitAll()
        .requestMatchers(HttpMethod.GET, "/api/polls/**").permitAll()
        .requestMatchers(HttpMethod.POST, "/api/polls/*/vote").permitAll()
        .anyRequest().authenticated());
```

This configuration allows public access to viewing polls and voting while protecting write operations. CORS is configured for localhost development.

### 4.2.2 Core APIs and Real-Time Updates

The REST API provides two main resource groups: `/api/polls` (CRUD operations, voting, comments) and `/api/users` (profile management with authorization checks).

```java
// PollController (excerpt)
@PostMapping("/{id}/vote")
public boolean castVote(@PathVariable Integer id,
                        @RequestParam Integer presentationOrder,
                        @AuthenticationPrincipal User user) {
    return pollService.castVote(id,
                                user == null ? Optional.empty()
                                             : Optional.of(user.getUserId()),
                                presentationOrder);
}
```

When data changes occur, the backend emits Kafka messages and broadcasts lightweight WebSocket notifications to connected clients:

```
// on create/update/delete
RawWebSocketServer.broadcast("pollsUpdated");
```

This ensures all clients stay synchronized without polling. Basic unit and integration tests in `backend/src/test` verify the main flows.

### 4.2.3 Kafka Integration

Kafka is used as the backend's event bus to decouple write operations from real-time updates and cache invalidation. When a vote or comment is stored, the service resolves a per-poll topic name (e.g., `poll.voteChange.<pollId>`) and publishes a compact JSON event using Spring Kafka's `KafkaTemplate`. A lightweight consumer subscribes to all poll topics using a pattern and reacts by invalidating Redis caches and nudging the WebSocket layer to refresh.

```
// ProducerService (excerpt)
public void sendEvent(String topicName, Map<String, Object> data) {
    kafkaTemplate.send(topicName, data);
}

// ConsumerService (excerpt)
@KafkaListener(topicPattern = "poll.voteChange.*", groupId = "poll-app")
public void onVoteChange(Map<String, Object> event) {
    // read pollId, invalidate caches, and (optionally) broadcast WS hints
}
```

The broker address is provided via `SPRING_KAFKA_BOOTSTRAP_SERVERS` (see Docker Compose). Topics are created or ensured at runtime through a small helper that uses `KafkaAdmin`. See Section 3 for the Kafka assessment and results.

## 4.3 Frontend Implementation

The frontend uses React with TypeScript, bundled via Vite. Key components include `CreatePoll`, `VoteOnPoll`, `ManagePolls`, `CommentSection`, and `Login`. The application uses a hybrid communication strategy: REST for data operations and a single WebSocket for real-time events.

```
// ws.ts
export const onWs = (f:(e:unknown)=>void) =>
    (listeners.add(f), () => listeners.delete(f));

export function connectWs() {
    const u = `${location.protocol==="https:"?"wss":"ws"}://
              ${location.hostname}:8080/rawws`;
    const ws = new WebSocket(u);
    ws.onmessage = ev => {
        try { emit(JSON.parse(ev.data)); }
        catch { emit(ev.data); }
    };
}
```

The `App` component listens to WebSocket events and selectively refreshes data:

```
// App.tsx
useEffect(() => onWs(m => {
    if (m === "pollsUpdated") refreshPolls();
    if (m === "commentsUpdated") bumpCommentsVersion(activePollId);
}), []);
```

When a poll is created, the UI automatically switches to the ”Vote” tab and refreshes, allowing immediate verification. A `WebSocketMessages` component displays a rolling log of events for development debugging.

## 4.4 Extensibility and Future Work

The layered architecture and Spring profiles make the system easily extensible. For example creating an app for iOS or Android would primarily involve building a new frontend that consumes the existing REST and WebSocket APIs. The modular design ensures incremental feature additions without disrupting existing functionality. Security improvements could be made by implementing refresh tokens and stricter CORS policies for production.

## 5 Conclusions

Concludes on the project, including the technology, its maturity, learning curve, and quality of the documentation.

The references used throughput the report should constitute a well chosen set of references, suitable for someone interesting in learning about the technology.

## References

[1] A.W. Brown and K. C. Wallnau. A framework for evaluating software technology. *IEEE Software*, 13(5):39–49, September 1996.

[2] Confluent. Kafka topics and producers faqs. Accessed: 2025-11-12.