DAT250 FeedApp Report

Erlend André Høntorp, Jonas Heen Opsahl and Thomas Tolo Jensen

November 22, 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 report compares RabbitMQ (message queuing) and Kafka (event sourcing) for handling real-time updates in a web application. Our aim is to show how a queue-oriented broker differs from a log-centric event stream in throughput, latency, scaling, and fan-out. 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. We used the OpenMessaging Benchmark to test both systems under different loads that simulate many votes coming in and updates being pushed to users. In our tests, Kafka handled more messages per second and scaled better when there were lots of writes, and RabbitMQ did not have lower latency at low message rates in our setup. As the load increased, Kafka pulled ahead, and RabbitMQ's per-message overhead became a bottleneck. Using one Kafka topic per poll kept events clean and isolated, but it added some extra management work. Overall, we learned about the trade-off between throughput and latency in log-based (Kafka) vs queue-based (RabbitMQ) systems, when the extra complexity of an event-driven design makes sense, and why careful cache invalidation matters in a distributed app.

Contents

1 Introduction

1.1 Project Overview

FeedApp is a small web app for running live polls. Users can create polls, vote, and discuss results, and the app updates everyone in real time so new votes and comments appear instantly. The goal is to make collecting and viewing feedback simple and fast.

The system has two main parts: a backend and a frontend. The backend provides REST endpoints for polls, votes, and comments, and emits events when data changes. The frontend displays the current state, listens for WebSocket updates, and stays synchronized with the server. The project also serves as a way to explore event-driven patterns and how they scale as more users interact with the system.

1.2 Technology Stack Summary

Backend: Java with Spring Boot and Spring Security for REST APIs, validation, and authentication. During development we use an in-memory H2 database, with Redis as a cache for frequently accessed poll data. Kafka handles domain events such as vote changes.

Frontend: React with TypeScript provides a simple, component-based UI that updates in real time through a single WebSocket connection.

Docker and CI: Docker Compose ensures consistent local deployment of all services, and GitHub Actions builds and tests the full stack on every push to maintain reliability.

We chose this stack because it is familiar, easy to set up, and scales horizontally. Kafka decouples writes from notifications, Redis improves read performance, and WebSockets keep the UI responsive

1.3 Project Outcome

The prototype supports creating polls, casting votes with instant updates, and adding threaded comments. We also compared Kafka and RabbitMQ as messaging systems for the event stream. In our experiments, Kafka outperformed RabbitMQ in both throughput and latency, remaining stable even under heavy write and fan-out load. RabbitMQ was simpler to work with but showed higher and more variable latency across all workloads. Using one topic per poll kept Kafka events organized, and Redis reduced repeated work on popular polls. The entire system is containerized so the full stack can be started with a single 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

This section presents the design of the FeedApp system. It covers the main functional use cases, the domain model that structures the application's core entities, and the overall software architecture. Together, these elements show how the system supports real-time behaviour, modular business logic, and scalability. Each part is accompanied by a diagram to provide a high-level structural overview.

2.1 Use Cases

FeedApp is an interactive polling platform that allows users to vote on and discuss polls in real time. The system distinguishes between two user roles:

- **Anonymous user**: can browse public polls, vote on them, register a new account, and log in.
- **Registered user**: has all anonymous permissions and can additionally create polls, comment on polls, vote on private polls if invited, and manage polls they have created.

For these roles, the following use cases emerge:

- **Register and log in:** An anonymous user may register by providing a username, email, and password. Upon successful registration, the system issues a JWT token which is used for stateless authentication. A registered user can log in at any time to obtain a new token.
- **Create Poll:** Registered users can create a poll by specifying a question, a set of options and visibility (public or private). When a poll is created, it is persisted to the database, a Kafka topic dedicated to that poll's future vote events is created, and a WebSocket update is broadcast to notify connected clients.
- **Vote on Poll:** Both anonymous and registered users may vote on public polls. Private polls are restricted to invited registered users. Voting triggers several actions: updating the database,

publishing a `voteChange` event to Kafka, invalidating the relevant Redis cache entry and broadcasting a WebSocket signal so that all clients refresh their displayed results.

- **Comment:** Registered users may write comments on polls and reply to existing comments.
- **Manage Polls:**Poll creators may add extra options or delete their own polls. Deletions cascade appropriately, removing options, votes, and comments associated with the poll.

A simplified use case diagram is shown in Figure 1.

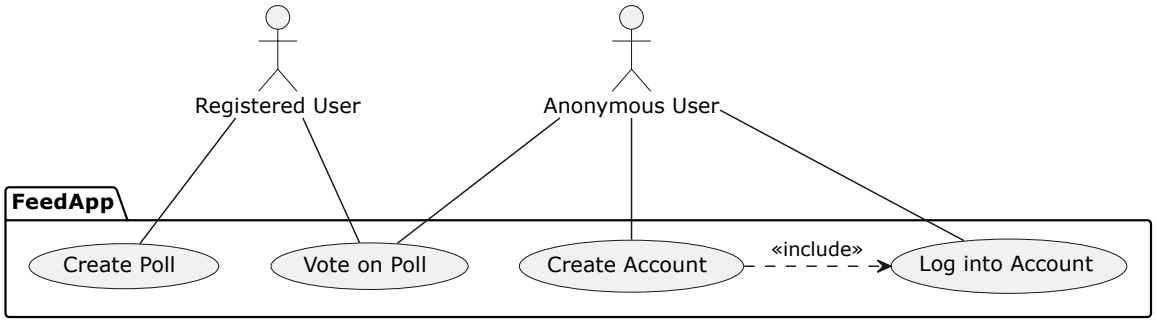


Figure 1: Use Case Diagram

2.2 Domain Model

The domain model captures the core business concepts of FeedApp and describes how polls, users, votes, and comments relate to each other. These entities are implemented as JPA `@Entity` classes in the Spring Boot backend and persisted using Hibernate. The model is designed around aggregate boundaries, allowing the application to remain modular and scalable.

User

The `User` entity represents a registered account with a unique identifier, username, email address, and password hash. A user may create multiple polls, vote and comment. In the case of anonymous voting, the `Vote` entity stores no user reference, making the user-vote relationship optional. A `User` may create many polls.

Poll

The `Poll` entity represents a question created by a user. It contains metadata such as visibility (public or private), validity period and a collection of vote options and comments. A `Poll` can have two to many `VoteOptions` and one to many `Comments`.

Vote

The `Vote` entity represents an individual vote cast on a specific `VoteOption`. Votes reference both the associated `VoteOption` and the (optional) user who cast the vote.

VoteOption

The `VoteOption` entity represents a selectable option for a poll. A poll must have at least two options, and each option is uniquely identified by its `pollID`. A `VoteOption` may receive many `Votes`.

Comment

The `Comment` entity represents text written by a user connected to a `Poll`. Comments support nested replies through a recursive `parent` relationship. A `Comment` is of course written by exactly one `User`, and may have nested child (comments via its `parent` field). Comments are deleted automatically if their parent or poll is removed.

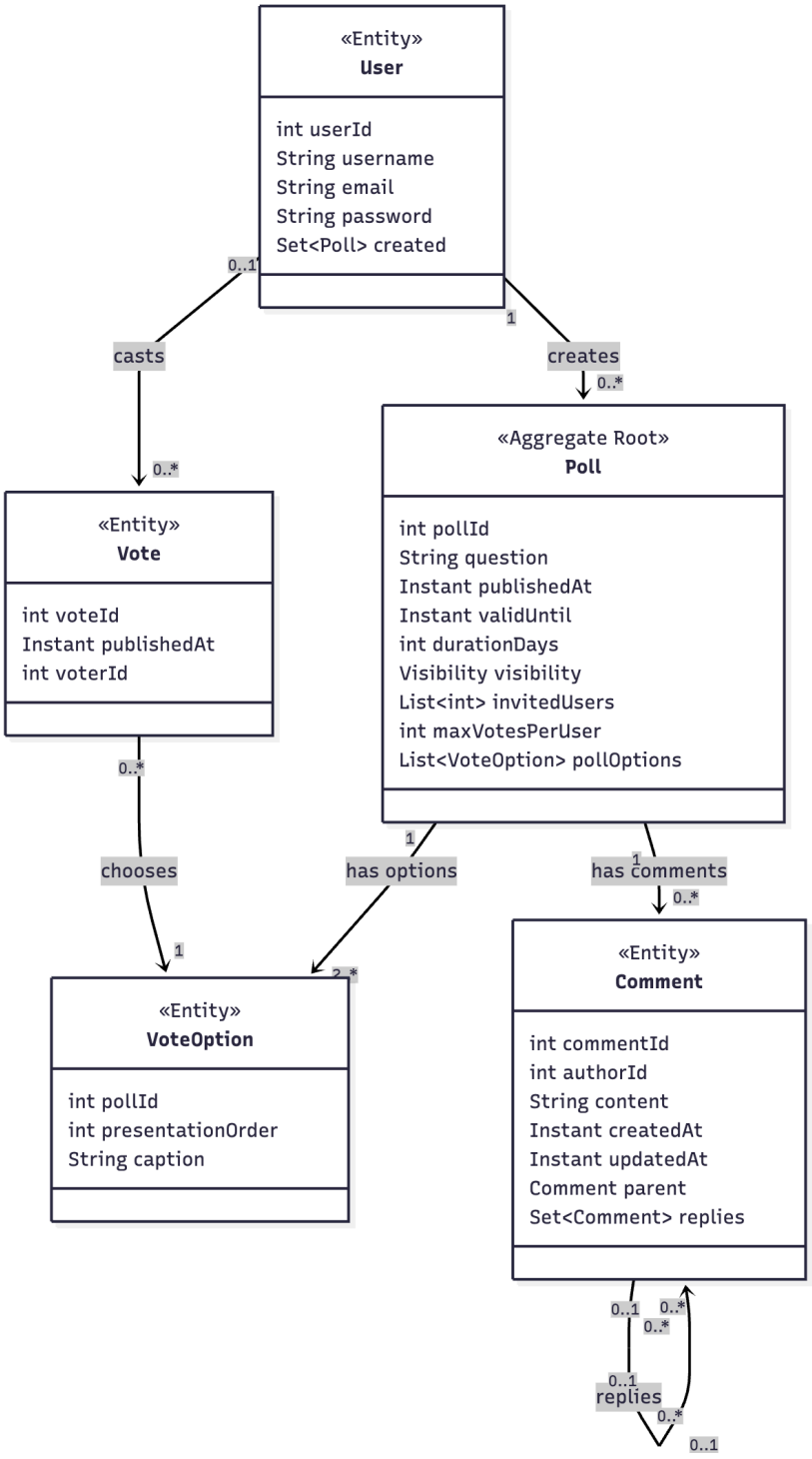


Figure 2: Domain Model overview.

2.3 Architecture

The architecture of FeedApp follows a layered and event-driven design. The system is divided into a React/TypeScript frontend, a Spring Boot backend and supporting infrastructure services (H2, Redis, Kafka) orchestrated using Docker Compose. The design ensures clear separation, maintains responsiveness, and enables horizontal scalability. Figure 3 shows an overview of the main components.

Frontend (React + TypeScript)

The user interface of the application. It communicates with the backend using REST API calls for CRUD operations, and a persistent WebSocket connection for real-time updates. The frontend never accesses data directly. The UI listens for events such as `pollsUpdated` or `commentsUpdated` and refreshes data automatically.

Backend (Spring Boot)

The backend implements the core application logic. The controllers maps HTTP endpoints to methods in the service layer. The service layer contains domain logic for polls, votes, comments and events. This includes caching, authentication and Kafka integration. The persistence layer uses JPA/Hibernate to store and retrieve entities from the database. The backend also hosts the WebSocket endpoint (`/rawws`), used to notify clients of changes, and applies stateless JWT-based authentication for all protected routes.

Infrastructure Layer

Several external components support performance, scalability and reliability. The H2 database is a lightweight SQL database used for storing users, polls, options, votes and comments during development. The Redis Cache stores precomputed poll results to reduce the database load. Cache entries are invalidated when related Kafka events are consumed. Kafka clusters are used as a distributed event log for publishing vote events. It decouples producers and consumers and enables potential future scaling of services.

Docker

FeedApp is packaged and executed as a multi-container system using Docker and Docker Compose. Each major component of the architecture runs in its own container, ensuring a reproducible and uniform environment. The frontend and backend are published as public Docker Hub images (erlendandre/feedapp-frontend and erlendandre/feedapp-backend), allowing anyone to start the full

system with a single command. Redis and Kafka rely on standard official images, while Kafka is deployed as a three-node replicated cluster, matching the design principles explored in the technology assessment. All containers are orchestrated through a docker-compose.yml file that wires together the network, environment variables (e.g., Kafka bootstrap servers, Redis host), and persistent volumes for Kafka. This containerized structure simplifies onboarding, testing, and scaling, and reflects modern deployment practices for distributed, event-driven applications.

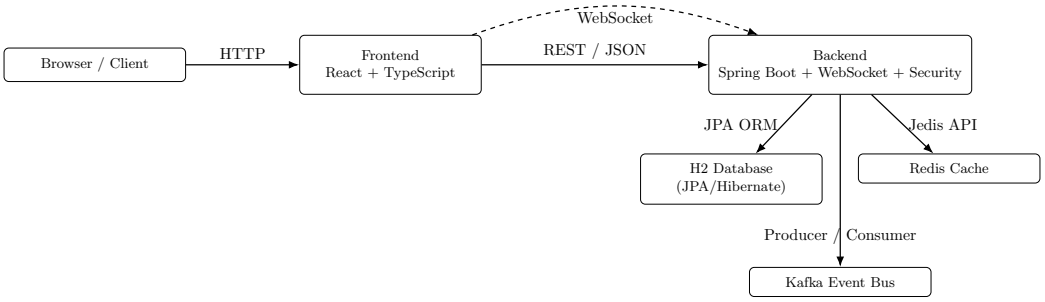


Figure 3: System Architecture

The FeedApp design uses a combination of layered backend logic and event-driven infrastructure. Redis improves read performance, Kafka ensures decoupled communication and future extensibility, and WebSockets provide responsive updates. All components run in isolated Docker containers, and the use of public Docker Hub images ensures that the entire system can be reproduced consistently on any machine. The three-node Kafka cluster deployed through Docker Compose demonstrates real distributed behaviour, including replication and fault tolerance, in a controlled development environment. Together, these design choices result in a modular, scalable, and extensible architecture that illustrates key concepts of contemporary distributed systems and serves as a solid educational example for event-driven application design.

3 Technology Assessment

At its core, Apache Kafka is a distributed event streaming platform acting as a fault-tolerant log. The **FeedApp** uses Kafka to decouple tasks: user votes publish events that trigger asynchronous reactions from services (like the WebSocket broadcaster) rather than blocking immediate updates.

Event: An event is the most basic unit of data. 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 category where events are stored. **FeedApp** employs a dynamic strategy: the `PollTopicManager` creates a unique topic for every poll (e.g., `poll.voteChange.10`) rather than using a single monolithic topic. KRaft mode allows this to scale to millions of topics[2].

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 of the report aims to follow this software technology evaluation framework. 4

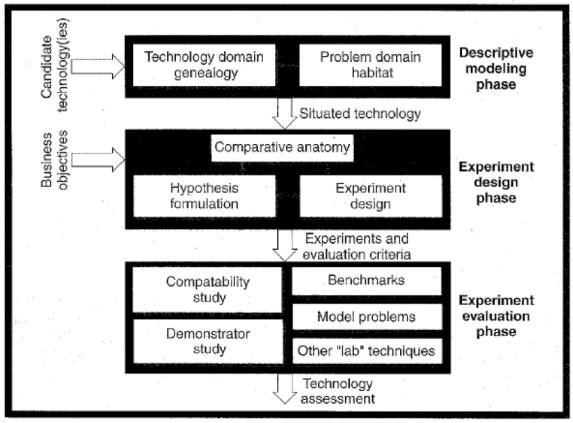


Figure 4: Software technology evaluation framework.

3.1 Descriptive Modeling

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 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.[1] 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

Following the descriptive modeling phase, we now design the experiments to empirically evaulate our candidate technologies. The goal of this phase is to formulate refutable hypotheses based on our problem domain habitat and design experiments to generate quantitative data to test them.

Our design focuses on getting objective data for our key requirements: throughput, latency and scalability.

3.2.1 Hypothesis Formulation

Our hypotheses are derived directly from the two habitats identified in the FeedApp backend and they key feature deltas between Kafka, which is a log based pull model, and RabbitMQ which is a queue based, push model.

Habitat 1 prioritizes high throughput and scalability.

Hypothesis 1.A (Throughput): In a many producers, high volume workload, Kafka will achieve significantly higher maximum throughput than RabbitMQ before latency thresholds are breadched. Here, throughput will be measured as messages per second.

Hypothesis 1.B (Scalability): As the number of producers increases, Kafka's throughput will scale better than RabbitMQ's. Here, better scaling will be how much more load can be handled when increasing cluster size.

Habitat 2 prioritizes low end-to-end latency.

Hypothesis 2.A (Low-load latency): In a low message rate workload, RabbitMQ's push model will demonstrate lower p99 end-to-end latency than Kafka's pull model.

Hypothesis 2.B (High-load latency): As the message rate in this habitat increases, Kafka's pull model will become more efficient, and its average latency will become lower than RabbitMQ's, which will suffer from per-message overhead.

3.2.2 Experiment Design and Workloads

To test these hypotheses, we will use the OpenMessaging Benchmark (OMB) framework. This provides a standardized, vendor-neutral toolset to conduct repeatable performance tests easily. We will configure

OMB with two distinct workloads, each designed as a model problem that simulates one of our defined habitats.

Workload 1 for Habitat 1: Ingestion test

The purpose of this workload is to test hypotheses 1.A and 1.B. This will be the OMB workload configuration for this test.

```
name: "workload1"
topics: 1
partitionsPerTopic: 3
messageSize: 100
useRandomizedPayloads: true
randomBytesRatio: 0.5
randomizedPayloadPoolSize: 1000
subscriptionsPerTopic: 1
consumerPerSubscription: 1
producersPerTopic: 100
producerRate: 500000
consumerBacklogSizeGB: 0
testDurationMinutes: 3
```

We will have many producers, a single topic/exchange, and a low number of consumers. This simulates thousands of concurrent users casting votes and creating polls in the FeedApp. There are three metrics we wish to capture in this test.

1. Maximum throughput: The highest sustainable message rate each platform can ingest

2. Producer latency (p99): The 99th percentile latency for a producer to successfully publish a message.

3. Scalability curve: Run the test with 1,5 and 10 producer instances to observe how throughput scales.

Workload 2 for Habitat 2: Fan-out test

The purpose of this workload is to test hypotheses 2.A and 2.B. This will be the OMB workload configuration for the test.

```
name: "workload2"
topics: 1
partitionsPerTopic: 3
messageSize: 1024
useRandomizedPayloads: true
randomBytesRatio: 0.5
randomizedPayloadPoolSize: 1000
subscriptionsPerTopic: 70
consumerPerSubscription: 1
```

```
producersPerTopic: 1
producerRate: 10000 # This will vary, 100,1000,10000
consumerBacklogSizeGB: 0
testDurationMinutes: 3
```

We will have a single producer and many competing consumers on a shared subscription. This simulates real time fanout of poll updates to all connected clients. There are two metrics we wish to capture in this test:

1. End-to-end latency (p99): The time from message publish to message receipt by a consumer.

2. Latency vs. Throughput Curve: We will specifically run this workload at varying throughput rates (100 msg/sec, 1000 msg/sec and 10,000 msg/sec) to find the crossover point predicted in 2.B.

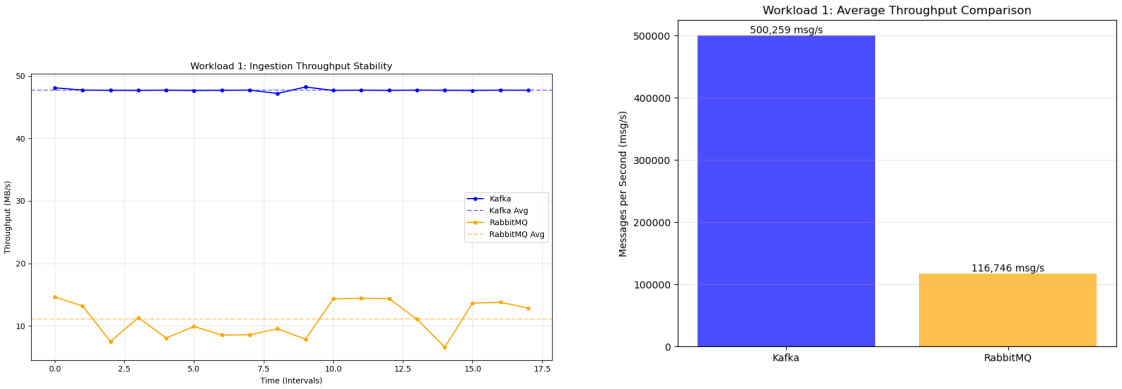
3.2.3 Comparative Feature Analysis

We acknowledge that our locally run benchmarks will be limited by our specific hardware and test duration. Therefore, as part of our comparative feature analysis, we will supplement out primary findings with a review of existing, larger scale benchmarks published by others. This aims to validate our results and provide a broader context for the performance characteristics that we cannot easily reproduce.

3.3 Experiment Evaluation

The experimental phase was aimed to gather empirical data to validate the suitability of the technologies for the **FeedApp** high volume ingestion (Habitat 1) and real time fan-out (Habitat 2) requirements.

3.3.1 Evaluation of Workload 1: Ingestion Performance



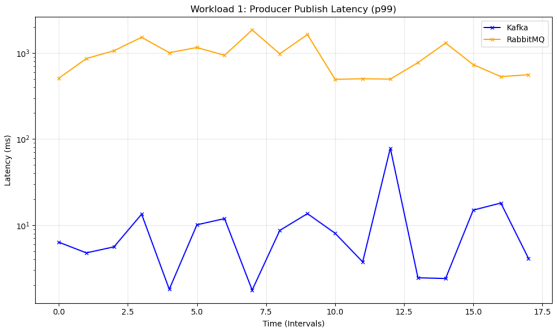
(a) Throughput Stability over Time

(b) Average Throughput Comparison

Figure 5: **Workload 1 Ingestion Performance.** Comparing the stability and raw average throughput of Kafka vs. RabbitMQ.

Workload 1 simulated the **FeedApp** backend receiving a storm of user votes and poll creation events. The primary metric for this habitat is throughput (messages/second and MB/s) and producer stability.

Throuput Analysis As illustrated in Figure5, the results strongly validate **Hypothesis 1.A**. Apache Kafka demonstrated a significantly higher write capacity than RabbitMQ. Looking at5b, Kafka maintained a sustained average throughput of approximately 500,259 msg/s. In contrast, RabbitMQ saturated at 116,746 msg/s. It is also important to mention that the producer rate was set at 500,000, which suggests that Kafka could reach even higher if this was increased. Figure5a reveals the stability of this throughput. Kafka (blue line) maintains a near perfectly flat ingestion rate, indicating that the broker was not resource constrained by the producer load. RabbitMQ (orange line), while relatively stable, operates at a much lower ceiling, suggesting it hit an internal processing bottleneck, likely the CPU cost of managing exchange routing and per-message persistence. Figure5a also clearly illustrates the increased throughput capcaity (MB/s) between the two brokers.



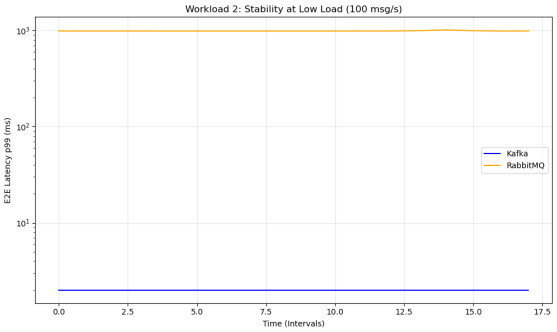
(a) Producer Publish Latency (p99)

(b) Consumer Backlog Size (Log Scale)

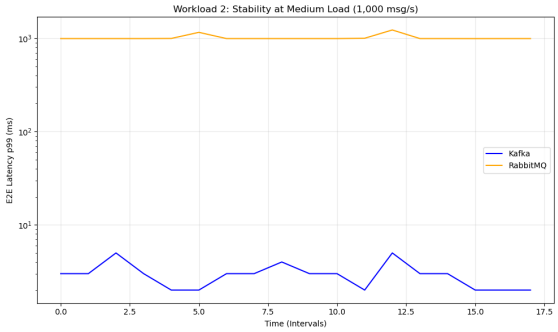
Figure 6: **Workload 1 System Stability.** Correlating producer latency with consumer backlog accumulation.

System Stability and Backlog The cost of RabbitMQ's smart broker architecture becomes evident when analyzing latency and backlogs in Figure6. Figure6a shows the p99 public latency, which is the time it takes for the producer to get an acknowledgement from the broker. Kafka consistently acknowledges writes in single-digit milliseconds. RabbitMQ, conversely, exihibits high volatility with p99 latencies oscillating between 500ms and 1,500ms. This latency disparity is directly explained by6b. RabbitMQ accumulates a significant consumer backlog, peaking over 10^5 messages. Because RabbitMQ stores messages in a queue structure that requires memory management and index updates for every message, the high ingestion rate cause the broker to slow down producers to allow consumers to catch up. Kafka, utilizing a sequential append-only log, handled the backlog without impacting producer write latency, effectively decoupling the producer from the consumer.

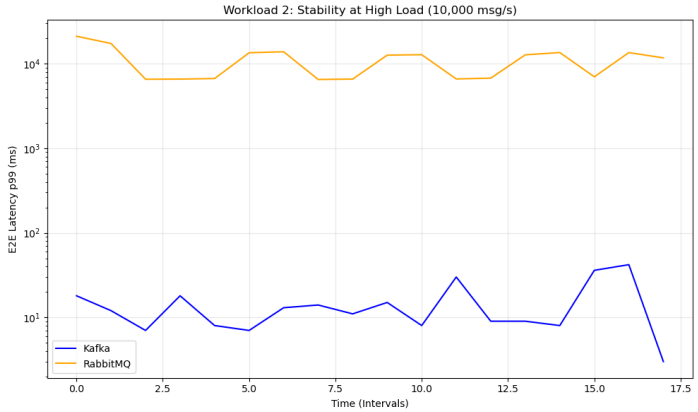
Conclusion on Habitat 1: Hypothesis 1.A is confirmed. In this experiment, Kafka provides approximately 4.2x the throughput of RabbitMQ, with significantly better stability.



(a) Low Load (100 msg/s)



(b) Medium Load (1,000 msg/s)



(c) High Load (10,000 msg/s)

Figure 7: **Workload 2 Fanout Stability.** Evolution of end-to-end latency as throughput increases.

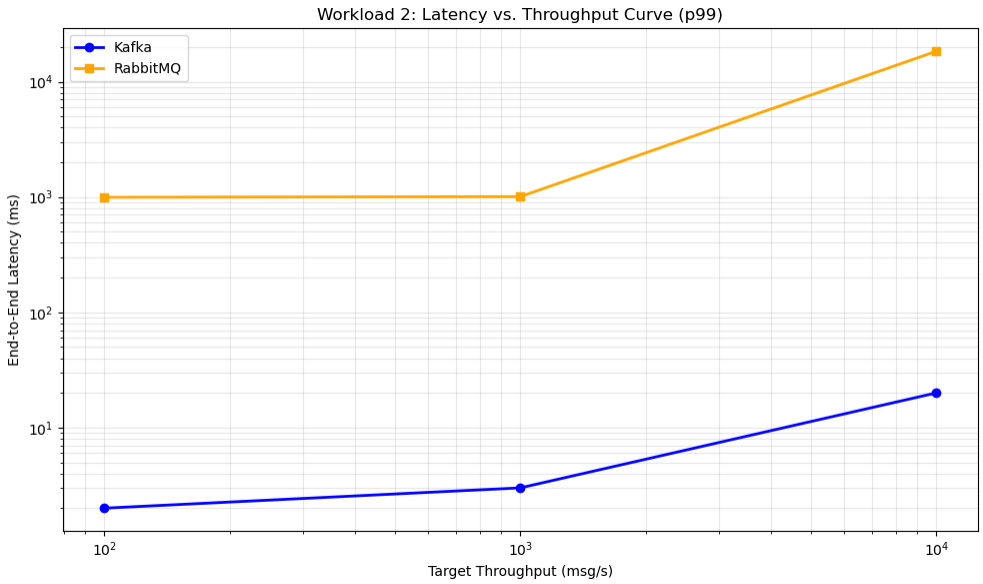


Figure 8: **Latency vs. Throughput (p99).** Comparison of fanout latency scaling. Note the logarithmic scales on both axes.

3.3.2 Evaluation of Workload 2: Fan-out Performance

Workload 2 simulated the real time component of the app, pushing updates to clients. The primary metric here is end-to-end (E2E) latency.

Low vs. High Load Latency The results for this workload yield unexpected findings regarding **Hypothesis 2.A**. We hypothesized that RabbitMQ would offer lower latency at low loads due to its push based model. However, the data refutes this. As seen in Figure7a, at a low load of 100 msg/s, Kafka maintains an E2E p99 latency of around 2ms. RabbitMQ exhibits a consistent latency floor of 1000ms. The high baseline for RabbitMQ suggests a configuration interaction with the test harness, but it highlights that push does not automatically guarantee lower latency than pull if the persistence guarantees are strict.

Scalabiliy Under Load Figure7 demonstrates the behavior as load increases to 10,000 msg/s (Figure7c). Kafka remains stable with latencies under 20ms. RabbitMQ's latency deteriorates further, spiking above 10,000ms. The relationship is best summarized further by the logarithmic scale curve in Figure8. Kafka's latency growth is linear and shallow, wheras RabbitMQ's latency growth is exponential relative to throughput. **Conclusion on Habitat 2: Hypothesis 2.A is refuted** and **Hypothesis 2.B is confirmed**. Even at low loads, Kafka's efficient zero-copy network transfer and sequential I/O provided lower latencies than RabbitMQ in this benchmark. As load increased, Kafka's performance gap widened, proving it to be the more scalable solution for the **FeedApp's** fan-out requirements.

4 Prototype Implementation

This section outlines how the FeedApp prototype is implemented and demonstrates the core mechanisms behind running and extending the system. Only key code snippets are included here to illustrate the most important architectural details in our FeedApp.

4.1 Setup and Deployment

FeedApp runs as a multi-container application using Docker Compose, consisting of a Spring Boot backend, a React/Vite frontend, Redis for caching, and Kafka as the event bus. Infrastructure configuration is passed to the backend through environment variables. The most important part of the Compose file is shown below:

```
backend:
  ports: ["8080:8080"]
  environment:
    - SPRING_KAFKA_BOOTSTRAP_SERVERS=kafka:29092
    - SPRING_REDIS_HOST=redis

frontend:
  ports: ["5173:5173"]
  depends_on: [backend]
```

The entire system can be started with:

```
docker compose up -d
```

The frontend is then reachable at `http://localhost:5173`. A persistent WebSocket connection is used alongside REST APIs to support real-time updates across clients.

4.2 Backend Implementation

The backend follows a layered Spring Boot structure. Controllers expose REST endpoints, Services implement poll, vote, and comment logic, and Repositories handle persistence via JPA or an in-memory store. Redis is used as a simple cache for frequently read poll results.

4.2.1 Authentication and Security

Authentication is implemented with stateless JWT tokens. Viewing polls and voting are public operations, while creating polls and comments requires authentication. The security rules are configured as:

```
// SecurityConfig (selected rules)
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());
```

The `JwtAuthFilter` validates tokens on each request and injects the authenticated user into controller methods using `@AuthenticationPrincipal`.

4.2.2 Core APIs and Real-Time Updates

Endpoints under `/api/polls` handle CRUD operations, comments, and voting. A shortened version of the voting endpoint is shown below:

```
// PollController (vote endpoint)
@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);
}
```

On each update, the backend broadcasts a minimal WebSocket message to prompt clients to refresh their view:

```
RawWebSocketServer.broadcast("pollsUpdated");
```

This keeps the UI synchronized without polling.

4.2.3 Kafka Integration

Kafka serves as the event bus for vote and poll updates. Each poll receives a dedicated topic to maintain ordering and isolate events.

Publishing events:

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

Consuming events:

```
// ConsumerService (pattern subscription)
@KafkaListener(topicPattern="poll.voteChange.*", groupId="poll-app")
public void onVoteChange(Map<String, Object> event) {
  // invalidate Redis cache and notify WebSocket layer
}
```

This approach separates writes from cache invalidation and real-time UI updates, improving system responsiveness under load.

4.3 Frontend Implementation

The frontend is built in React and TypeScript. REST APIs fetch poll data, while a single WebSocket connection delivers push updates. The WebSocket setup is shown below:

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

Components subscribe to events and handle updates selectively:

```
// App.tsx
useEffect(() => onWs(msg => {
```

```
  if (msg === "pollsUpdated") refreshPolls();
  if (msg === "commentsUpdated") bumpCommentsVersion(activePollId);
}), []);
```

This pattern keeps the UI responsive with minimal network traffic.

4.4 Extensibility and Future Work

The modular architecture allows further development, such as new poll types, sharing features, and improved authentication. A mobile client can reuse the existing REST and WebSocket APIs. Future improvements include stronger production security settings and persistent database storage.

5 Conclusion

This project resulted in a working real-time polling app and a comparison of Kafka and RabbitMQ as messaging solutions for event-driven systems. The technologies work differently and have different strengths. RabbitMQ uses a simpler model based on queues and exchanges, which makes it easier to understand and suitable for smaller or more straightforward setups. Kafka requires learning more concepts, such as partitions, replication, and consumer groups, but is designed to handle larger amounts of data and more demanding workloads.

Our experiments provided a clear contrast in how the two technologies behave under the kinds of workloads a live polling service must handle. Kafka consistently delivered higher throughput and more stable latency as load increased, while RabbitMQ showed larger spikes and less predictable performance when both write pressure and fan-out activity rose. This makes Kafka a better fit for high-volume, real-time feeds like the one FeedApp demonstrates.

The project also strengthened our understanding of event-driven design, caching, and containerized deployment. The evaluation showed that technology choice depends on the scale and responsiveness requirements of the system: RabbitMQ remains an accessible and capable broker, but for high-throughput, low-latency event streams, Kafka offers clear advantages. Overall, the prototype and benchmark results provide direction for how FeedApp could be taken further toward a real production setup.

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 **&** producers faqs. Accessed: 2025-11-12.