AS-IS System Analysis Report

*Migration Assessment and Planning Document*

|  |  |
| --- | --- |
| **Repository** | openshift-voting-app |
| **Analysis Date** | July 17, 2025 |
| **Components Analyzed** | 1 |
| **Dependencies Identified** | 0 |

Executive Summary

|  |  |
| --- | --- |
| **Total Components** | 1 |
| **Total Files** | 7 |
| **Total Lines of Code** | 350 |
| **Total API Endpoints** | 2 |
| **External Dependencies** | 0 |

**Criticality Assessment:** This analysis identified 0 critical components and 0 high-priority components that require special attention during migration planning.

**Key Findings:** Okay, let's analyze the data flow within this microservices application based on the provided semantic analysis.  
  
\*\*Overall System Purpose (Inferred):\*\*  
  
Given the component names ("voting-app", "vote", "worker", "result") and database operations related to "votes", it's highly likely this application allows users to vote, processes those votes, and displays the results.  
  
\*\*Data Flow Narrative:\*\*  
  
1. \*\*User Interaction (Vote Submission):\*\*  
 \* The user interacts with the `/` endpoint of the `vote` service (written in Python). This service handles both `POST` (likely for vote submissions) and `GET` requests (likely for retrieving current voting state or options).  
 \* When a user submits a vote (via a `POST` request), the `vote` service performs a `REDIS\_OP`. This indicates the service is likely using Redis as a fast, in-memory data store. The vote is likely stored in Redis temporarily.  
  
2. \*\*Vote Processing (Background Worker):\*\*  
 \* The `worker` service (written in Java or C#) is responsible for persisting the votes from Redis into a more persistent database. We have two potential `worker` implementations.  
 \* \*\*Java Implementation:\*\* The Java `worker` performs several database operations. It executes a query to `CREATE TABLE IF NOT EXISTS votes`, which likely creates the `votes` table in a relational database if it doesn't already exist. It then `INSERT` new votes into the `votes` table (after retrieving them from Redis), and `UPDATE` existing votes. Additionally, it has a `REDIS\_OP` which is likely a read operation to retrieve votes from Redis.  
 \* \*\*C# Implementation:\*\* The C# `worker` performs several `ADO\_EXECUTE` operations, which indicates it's using ADO.NET to interact with a database (likely SQL Server or similar). These operations likely mirror the Java implementation: retrieving votes (or vote updates) from Redis and writing them to a relational database using `INSERT` and `UPDATE` statements.  
 \* \*\*Communication:\*\* The `vote` service likely publishes a message (perhaps via Redis pub/sub or a queue) when a vote is received. The `worker` service subscribes to this message or polls Redis, retrieves the vote data, and persists it into the relational database.  
  
3. \*\*Result Retrieval (Result Service):\*\*  
 \* The `result` service (written in JavaScript/Node.js) is responsible for displaying the voting results to the user.  
 \* It exposes a `/` endpoint, which when accessed, retrieves the vote counts from the relational database. The `server.js` file performs a `SELECT vote, COUNT(id) AS count FROM votes ...` query.  
 \* The `angular.min.js` file shows `MONGO\_FIND` operations, which implies that the result service is also retrieving data from a MongoDB database. This likely means that results from the relational database are stored in a MongoDB database. This aggregation likely happens to optimize the retrieval of results for display to the user.  
 \* The service also uses `socket.io.js`, indicating that it uses WebSockets. This allows real-time updates to be pushed to the user interface as new votes are cast.  
  
4. \*\*User Presentation:\*\*  
 \* The `app.js` file (presumably for the frontend) fetches data from the `/` endpoint of the `result` service. This data is then used to update the user interface, showing the current vote counts. The WebSocket connection allows the frontend to receive real-time updates without requiring constant polling.  
  
5. \*\*Response Flow:\*\*  
 \* A user requests the voting results via a GET request to the `/` endpoint of the `result` service.  
 \* The `result` service retrieves the aggregated results from MongoDB (or possibly directly from the relational database).  
 \* The `result` service formats the data and sends it back to the user's browser.  
 \* Subsequent votes are reflected in real time on the user's browser via the WebSocket connection.  
  
\*\*Summary of Data Transformations:\*\*  
  
\* \*\*Vote Service:\*\* Receives raw vote data. Stores temporarily in Redis.  
\* \*\*Worker Service:\*\* Retrieves vote data from Redis. Persists (transforms) the data into relational database format (structured data).  
\* \*\*Result Service:\*\* Aggregates vote counts (potentially performs further transformations). Stores results in a MongoDB database. Formats data for display to the user.  
  
\*\*Storage Technologies:\*\*  
  
\* \*\*Redis:\*\* Used for temporary storage of votes, likely for fast ingestion and asynchronous processing.  
\* \*\*Relational Database (e.g., PostgreSQL, MySQL, SQL Server):\*\* Used for persistent storage of votes, likely for accurate vote counting and auditing.  
\* \*\*MongoDB:\*\* Used for storing aggregated voting results, likely for optimized retrieval and display in the user interface.  
  
\*\*Improvements and Considerations (Beyond the Analysis):\*\*  
  
\* \*\*Message Queue:\*\* A dedicated message queue (like RabbitMQ or Kafka) might be a better choice than relying solely on Redis pub/sub for communication between the `vote` and `worker` services. This provides better reliability and scalability.  
\* \*\*Data Consistency:\*\* Ensure proper transaction management between Redis, the relational database, and MongoDB, to maintain data consistency in case of failures.  
\* \*\*Scalability:\*\* Consider load balancing across instances of each service to handle increased traffic.  
\* \*\*Error Handling:\*\* Implement robust error handling and logging throughout the system.  
\* \*\*Security:\*\* Address security concerns such as input validation, authentication, and authorization.

Table of Contents

1. System Overview

2. Component Analysis

3. Dependency Analysis

4. Criticality Assessment

5. Security Analysis

6. Architecture Insights

7. Migration Recommendations

8. Appendices

System Overview

Technology Stack

|  |  |
| --- | --- |
| **Language/Framework** | **Components** |
| Javascript | 1 |

Architecture Patterns

* Database-Centric - High number of database operations

Component Analysis

voting-app

|  |  |
| --- | --- |
| **Language** | javascript |
| **Files** | 7 |
| **Lines of Code** | 350 |
| **API Endpoints** | 2 |
| **Database Operations** | 13 |
| **External HTTP Calls** | 0 |

Criticality Assessment

|  |  |
| --- | --- |
| **Business Criticality** | low |
| **Technical Complexity** | high |
| **User Impact** | high |
| **Data Sensitivity** | low |
| **Risk Score** | 0.36 |

**Assessment Reasoning:** Moderate technical complexity; High user impact with direct user interaction

Dependency Analysis

Dependency Types

|  |  |
| --- | --- |
| **Dependency Type** | **Count** |

Key Dependencies

Criticality Assessment

Criticality Distribution

|  |  |
| --- | --- |
| **Critical** | 0 |
| **High** | 0 |
| **Medium** | 0 |
| **Low** | 1 |
| **Total** | 1 |

Security Analysis

No security findings available.

Architecture Insights

**Insight 1:** Okay, let's analyze the data flow within this microservices application based on the provided semantic analysis.  
  
\*\*Overall System Purpose (Inferred):\*\*  
  
Given the component names ("voting-app", "vote", "worker", "result") and database operations related to "votes", it's highly likely this application allows users to vote, processes those votes, and displays the results.  
  
\*\*Data Flow Narrative:\*\*  
  
1. \*\*User Interaction (Vote Submission):\*\*  
 \* The user interacts with the `/` endpoint of the `vote` service (written in Python). This service handles both `POST` (likely for vote submissions) and `GET` requests (likely for retrieving current voting state or options).  
 \* When a user submits a vote (via a `POST` request), the `vote` service performs a `REDIS\_OP`. This indicates the service is likely using Redis as a fast, in-memory data store. The vote is likely stored in Redis temporarily.  
  
2. \*\*Vote Processing (Background Worker):\*\*  
 \* The `worker` service (written in Java or C#) is responsible for persisting the votes from Redis into a more persistent database. We have two potential `worker` implementations.  
 \* \*\*Java Implementation:\*\* The Java `worker` performs several database operations. It executes a query to `CREATE TABLE IF NOT EXISTS votes`, which likely creates the `votes` table in a relational database if it doesn't already exist. It then `INSERT` new votes into the `votes` table (after retrieving them from Redis), and `UPDATE` existing votes. Additionally, it has a `REDIS\_OP` which is likely a read operation to retrieve votes from Redis.  
 \* \*\*C# Implementation:\*\* The C# `worker` performs several `ADO\_EXECUTE` operations, which indicates it's using ADO.NET to interact with a database (likely SQL Server or similar). These operations likely mirror the Java implementation: retrieving votes (or vote updates) from Redis and writing them to a relational database using `INSERT` and `UPDATE` statements.  
 \* \*\*Communication:\*\* The `vote` service likely publishes a message (perhaps via Redis pub/sub or a queue) when a vote is received. The `worker` service subscribes to this message or polls Redis, retrieves the vote data, and persists it into the relational database.  
  
3. \*\*Result Retrieval (Result Service):\*\*  
 \* The `result` service (written in JavaScript/Node.js) is responsible for displaying the voting results to the user.  
 \* It exposes a `/` endpoint, which when accessed, retrieves the vote counts from the relational database. The `server.js` file performs a `SELECT vote, COUNT(id) AS count FROM votes ...` query.  
 \* The `angular.min.js` file shows `MONGO\_FIND` operations, which implies that the result service is also retrieving data from a MongoDB database. This likely means that results from the relational database are stored in a MongoDB database. This aggregation likely happens to optimize the retrieval of results for display to the user.  
 \* The service also uses `socket.io.js`, indicating that it uses WebSockets. This allows real-time updates to be pushed to the user interface as new votes are cast.  
  
4. \*\*User Presentation:\*\*  
 \* The `app.js` file (presumably for the frontend) fetches data from the `/` endpoint of the `result` service. This data is then used to update the user interface, showing the current vote counts. The WebSocket connection allows the frontend to receive real-time updates without requiring constant polling.  
  
5. \*\*Response Flow:\*\*  
 \* A user requests the voting results via a GET request to the `/` endpoint of the `result` service.  
 \* The `result` service retrieves the aggregated results from MongoDB (or possibly directly from the relational database).  
 \* The `result` service formats the data and sends it back to the user's browser.  
 \* Subsequent votes are reflected in real time on the user's browser via the WebSocket connection.  
  
\*\*Summary of Data Transformations:\*\*  
  
\* \*\*Vote Service:\*\* Receives raw vote data. Stores temporarily in Redis.  
\* \*\*Worker Service:\*\* Retrieves vote data from Redis. Persists (transforms) the data into relational database format (structured data).  
\* \*\*Result Service:\*\* Aggregates vote counts (potentially performs further transformations). Stores results in a MongoDB database. Formats data for display to the user.  
  
\*\*Storage Technologies:\*\*  
  
\* \*\*Redis:\*\* Used for temporary storage of votes, likely for fast ingestion and asynchronous processing.  
\* \*\*Relational Database (e.g., PostgreSQL, MySQL, SQL Server):\*\* Used for persistent storage of votes, likely for accurate vote counting and auditing.  
\* \*\*MongoDB:\*\* Used for storing aggregated voting results, likely for optimized retrieval and display in the user interface.  
  
\*\*Improvements and Considerations (Beyond the Analysis):\*\*  
  
\* \*\*Message Queue:\*\* A dedicated message queue (like RabbitMQ or Kafka) might be a better choice than relying solely on Redis pub/sub for communication between the `vote` and `worker` services. This provides better reliability and scalability.  
\* \*\*Data Consistency:\*\* Ensure proper transaction management between Redis, the relational database, and MongoDB, to maintain data consistency in case of failures.  
\* \*\*Scalability:\*\* Consider load balancing across instances of each service to handle increased traffic.  
\* \*\*Error Handling:\*\* Implement robust error handling and logging throughout the system.  
\* \*\*Security:\*\* Address security concerns such as input validation, authentication, and authorization.

Migration Recommendations

**Recommendation 1:** Plan extra time for 1 high-complexity components

Appendices

Appendix A: Glossary

|  |  |
| --- | --- |
| **Term** | **Definition** |
| API | Application Programming Interface |
| LOC | Lines of Code |
| HTTP | Hypertext Transfer Protocol |
| REST | Representational State Transfer |
| JSON | JavaScript Object Notation |
| SQL | Structured Query Language |
| NoSQL | Not Only SQL |
| CI/CD | Continuous Integration/Continuous Deployment |