AS-IS System Analysis Report

*Migration Assessment and Planning Document*

|  |  |
| --- | --- |
| **Repository** | openshift-voting-app |
| **Analysis Date** | July 16, 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, based on the provided semantic analysis, here's a technical narrative of the data flow through the voting application, broken down by user interaction, service communication, data transformation, storage, and response:  
  
\*\*1. User Entry Point & Voting:\*\*  
  
\* The application likely has a user interface that allows users to vote. This entry point is through the `/` endpoint of the `voting-app/vote/src/app.py` service, which handles both `POST` requests (for submitting votes) and `GET` requests (likely for initial display or status).  
\* When a user submits a vote (via a `POST` request to `/` on the `vote` service), this triggers a `REDIS\_OP` within the `vote` service (python). This suggests the vote is initially stored, likely as a key-value pair, in a Redis instance. This acts as a queue or buffer for vote processing.  
  
\*\*2. Asynchronous Vote Processing via Worker Service:\*\*  
  
\* The `worker` service is responsible for asynchronously processing the votes stored in Redis. The code snippets show both `Worker.java` and `Program.cs` within the `worker` directory, indicating potentially multiple worker service implementations or remnants of different versions.  
\* \*\*Worker (Java):\*\* This worker performs the following operations. It reads data from Redis (`REDIS\_OP`), likely pulling vote data. It then interacts with a relational database (likely PostgreSQL or MySQL), performing an `INSERT` operation to record the vote (if it's a new vote) or an `UPDATE` operation (if the vote already exists for a user/session). It also executes `CREATE TABLE IF NOT EXISTS votes` for initialization.  
\* \*\*Worker (C#):\*\* This worker has `ADO\_EXECUTE` operations, suggesting it directly executes SQL commands against a database. This likely serves a similar function as the Java worker - fetching votes from Redis, and then writing/updating vote data in a database.  
\* The use of Redis between the `vote` service and the `worker` service decouples the voting process from the database write operations, improving responsiveness and resilience.  
  
\*\*3. Data Storage and Aggregation:\*\*  
  
\* The final persistent storage is a relational database (SQL database). The `worker` service is responsible for writing the vote data into a `votes` table. The table likely has columns for `id` (user or session identifier) and `vote` (the user's choice).  
  
\*\*4. Result Retrieval and Display:\*\*  
  
\* The `voting-app/result/src/server.js` service exposes a `/` endpoint. This service is responsible for displaying the voting results.  
\* When a user accesses the `/` endpoint on the `result` service, it performs a `SELECT` query: `SELECT vote, COUNT(id) AS count FROM votes GROUP BY vote`. This query aggregates the vote counts for each choice from the database.  
\* The `voting-app/result/src/views/angular.min.js` suggests the frontend likely uses AngularJS. The MongoDB operations within this file (`MONGO\_FIND`) seem inconsistent with the rest of the application, as the other components use Redis and relational databases. This could be a remnant of a previous design, an unused code path, or a misinterpretation of the code analysis.  
  
\*\*5. Response Flow:\*\*  
  
\* The `result` service receives the aggregated vote counts from the database.  
\* The `result` service then renders this data (likely as JSON or HTML) and sends it back to the user's browser.  
\* The AngularJS code in `angular.min.js` then processes and displays the voting results within the user interface.  
\* `socket.io.js` within the `result` service implies the voting results are broadcasted to all connected clients using Websockets.  
  
\*\*In Summary:\*\*  
  
1. A user votes through a web interface, sending a `POST` request to the `/` endpoint of the `vote` service.  
2. The `vote` service stores the vote in Redis as a temporary holding place.  
3. The `worker` service(s) (Java or C#) asynchronously pulls votes from Redis.  
4. The `worker` service writes the vote to a SQL database, either inserting a new vote or updating an existing one.  
5. The `result` service queries the SQL database to retrieve the aggregated vote counts.  
6. The `result` service renders the results in a format suitable for the user interface (likely JSON or HTML), which is then displayed via a Javascript framework.  
7. The voting results are broadcasted to all clients through websockets by the `result` service.  
  
The inconsistent MongoDB operations in the frontend remain an open question.

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, based on the provided semantic analysis, here's a technical narrative of the data flow through the voting application, broken down by user interaction, service communication, data transformation, storage, and response:  
  
\*\*1. User Entry Point & Voting:\*\*  
  
\* The application likely has a user interface that allows users to vote. This entry point is through the `/` endpoint of the `voting-app/vote/src/app.py` service, which handles both `POST` requests (for submitting votes) and `GET` requests (likely for initial display or status).  
\* When a user submits a vote (via a `POST` request to `/` on the `vote` service), this triggers a `REDIS\_OP` within the `vote` service (python). This suggests the vote is initially stored, likely as a key-value pair, in a Redis instance. This acts as a queue or buffer for vote processing.  
  
\*\*2. Asynchronous Vote Processing via Worker Service:\*\*  
  
\* The `worker` service is responsible for asynchronously processing the votes stored in Redis. The code snippets show both `Worker.java` and `Program.cs` within the `worker` directory, indicating potentially multiple worker service implementations or remnants of different versions.  
\* \*\*Worker (Java):\*\* This worker performs the following operations. It reads data from Redis (`REDIS\_OP`), likely pulling vote data. It then interacts with a relational database (likely PostgreSQL or MySQL), performing an `INSERT` operation to record the vote (if it's a new vote) or an `UPDATE` operation (if the vote already exists for a user/session). It also executes `CREATE TABLE IF NOT EXISTS votes` for initialization.  
\* \*\*Worker (C#):\*\* This worker has `ADO\_EXECUTE` operations, suggesting it directly executes SQL commands against a database. This likely serves a similar function as the Java worker - fetching votes from Redis, and then writing/updating vote data in a database.  
\* The use of Redis between the `vote` service and the `worker` service decouples the voting process from the database write operations, improving responsiveness and resilience.  
  
\*\*3. Data Storage and Aggregation:\*\*  
  
\* The final persistent storage is a relational database (SQL database). The `worker` service is responsible for writing the vote data into a `votes` table. The table likely has columns for `id` (user or session identifier) and `vote` (the user's choice).  
  
\*\*4. Result Retrieval and Display:\*\*  
  
\* The `voting-app/result/src/server.js` service exposes a `/` endpoint. This service is responsible for displaying the voting results.  
\* When a user accesses the `/` endpoint on the `result` service, it performs a `SELECT` query: `SELECT vote, COUNT(id) AS count FROM votes GROUP BY vote`. This query aggregates the vote counts for each choice from the database.  
\* The `voting-app/result/src/views/angular.min.js` suggests the frontend likely uses AngularJS. The MongoDB operations within this file (`MONGO\_FIND`) seem inconsistent with the rest of the application, as the other components use Redis and relational databases. This could be a remnant of a previous design, an unused code path, or a misinterpretation of the code analysis.  
  
\*\*5. Response Flow:\*\*  
  
\* The `result` service receives the aggregated vote counts from the database.  
\* The `result` service then renders this data (likely as JSON or HTML) and sends it back to the user's browser.  
\* The AngularJS code in `angular.min.js` then processes and displays the voting results within the user interface.  
\* `socket.io.js` within the `result` service implies the voting results are broadcasted to all connected clients using Websockets.  
  
\*\*In Summary:\*\*  
  
1. A user votes through a web interface, sending a `POST` request to the `/` endpoint of the `vote` service.  
2. The `vote` service stores the vote in Redis as a temporary holding place.  
3. The `worker` service(s) (Java or C#) asynchronously pulls votes from Redis.  
4. The `worker` service writes the vote to a SQL database, either inserting a new vote or updating an existing one.  
5. The `result` service queries the SQL database to retrieve the aggregated vote counts.  
6. The `result` service renders the results in a format suitable for the user interface (likely JSON or HTML), which is then displayed via a Javascript framework.  
7. The voting results are broadcasted to all clients through websockets by the `result` service.  
  
The inconsistent MongoDB operations in the frontend remain an open question.

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 |