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, based on the provided semantic analysis, here's a narrative of the data flow in the voting application:  
  
\*\*1. User Interaction & Vote Submission:\*\*  
  
\* The user interacts with the application, likely through a web interface (details not provided but assumed).  
\* When a user casts a vote, a `POST` request is sent to the `/` endpoint of the `vote-app` service (`app.py`, Python).  
\* The `vote-app` service then performs a `REDIS\_OP`. Based on common usage patterns, we can infer that this operation likely pushes the vote data onto a Redis queue.  
  
\*\*2. Asynchronous Vote Processing (Worker Service):\*\*  
  
\* The `worker` service (Java and C# versions are present, implying redundancy or different deployment strategies) consumes votes from the Redis queue.  
\* The `worker` service (Java, `Worker.java`) performs these actions:  
 \* It first does a `REDIS\_OP`, most likely to fetch the vote from the Redis queue.  
 \* It then interacts with a relational database. It potentially creates a table `votes` (if it doesn't exist), inserts a new vote using an `INSERT` statement, or updates an existing vote using an `UPDATE` statement.  
\* The `worker` service (C#, `Program.cs`) performs `ADO\_EXECUTE` operations, likely mirroring the behavior of the Java `worker` service by interacting with a relational database to persist the vote. The specific actions (insert/update/create table) cannot be determined precisely based on "ADO\_EXECUTE" alone.  
  
\*\*3. Result Aggregation & Presentation:\*\*  
  
\* The `result-app` service (`server.js`, JavaScript) handles the presentation of voting results.  
\* It exposes a `GET` endpoint at `/`.  
\* When a user requests the results (by accessing `/`), the `result-app` service executes a `SELECT` query against the database to aggregate the vote counts. The specific SQL query `SELECT vote, COUNT(id) AS count FROM votes GROUP BY vote` aggregates the votes by vote type.  
\* The `result-app` service may be using `angular.min.js`, `app.js` and `socket.io.js` to dynamically update the user interface with the voting results, potentially using MongoDB for some functionalities. The data flow related to these files is unclear from the analysis provided. The presence of `MONGO\_FIND` operations suggests the potential usage of MongoDB to cache or store the voting results.  
  
\*\*4. Response to User:\*\*  
  
\* The `result-app` service formats the aggregated vote counts (obtained from the database) into a suitable format (likely JSON or HTML) and sends it back to the user's browser as a response to the `GET` request to `/`.  
  
\*\*Summary of Data Flow:\*\*  
  
1. User votes are submitted to the `vote-app` service.  
2. `vote-app` pushes vote data to a Redis queue.  
3. The `worker` service consumes votes from the Redis queue and persists them in a relational database (e.g., Postgres, MySQL).  
4. The `result-app` service queries the database to aggregate vote counts.  
5. The `result-app` service presents the aggregated results to the user, potentially using MongoDB for caching.  
  
\*\*In summary, the architecture follows a common pattern for voting applications: vote submission -> queuing (Redis) -> asynchronous processing (worker) -> data persistence -> result aggregation and display.\*\*

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 narrative of the data flow in the voting application:  
  
\*\*1. User Interaction & Vote Submission:\*\*  
  
\* The user interacts with the application, likely through a web interface (details not provided but assumed).  
\* When a user casts a vote, a `POST` request is sent to the `/` endpoint of the `vote-app` service (`app.py`, Python).  
\* The `vote-app` service then performs a `REDIS\_OP`. Based on common usage patterns, we can infer that this operation likely pushes the vote data onto a Redis queue.  
  
\*\*2. Asynchronous Vote Processing (Worker Service):\*\*  
  
\* The `worker` service (Java and C# versions are present, implying redundancy or different deployment strategies) consumes votes from the Redis queue.  
\* The `worker` service (Java, `Worker.java`) performs these actions:  
 \* It first does a `REDIS\_OP`, most likely to fetch the vote from the Redis queue.  
 \* It then interacts with a relational database. It potentially creates a table `votes` (if it doesn't exist), inserts a new vote using an `INSERT` statement, or updates an existing vote using an `UPDATE` statement.  
\* The `worker` service (C#, `Program.cs`) performs `ADO\_EXECUTE` operations, likely mirroring the behavior of the Java `worker` service by interacting with a relational database to persist the vote. The specific actions (insert/update/create table) cannot be determined precisely based on "ADO\_EXECUTE" alone.  
  
\*\*3. Result Aggregation & Presentation:\*\*  
  
\* The `result-app` service (`server.js`, JavaScript) handles the presentation of voting results.  
\* It exposes a `GET` endpoint at `/`.  
\* When a user requests the results (by accessing `/`), the `result-app` service executes a `SELECT` query against the database to aggregate the vote counts. The specific SQL query `SELECT vote, COUNT(id) AS count FROM votes GROUP BY vote` aggregates the votes by vote type.  
\* The `result-app` service may be using `angular.min.js`, `app.js` and `socket.io.js` to dynamically update the user interface with the voting results, potentially using MongoDB for some functionalities. The data flow related to these files is unclear from the analysis provided. The presence of `MONGO\_FIND` operations suggests the potential usage of MongoDB to cache or store the voting results.  
  
\*\*4. Response to User:\*\*  
  
\* The `result-app` service formats the aggregated vote counts (obtained from the database) into a suitable format (likely JSON or HTML) and sends it back to the user's browser as a response to the `GET` request to `/`.  
  
\*\*Summary of Data Flow:\*\*  
  
1. User votes are submitted to the `vote-app` service.  
2. `vote-app` pushes vote data to a Redis queue.  
3. The `worker` service consumes votes from the Redis queue and persists them in a relational database (e.g., Postgres, MySQL).  
4. The `result-app` service queries the database to aggregate vote counts.  
5. The `result-app` service presents the aggregated results to the user, potentially using MongoDB for caching.  
  
\*\*In summary, the architecture follows a common pattern for voting applications: vote submission -> queuing (Redis) -> asynchronous processing (worker) -> data persistence -> result aggregation and display.\*\*

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 |