Debias

EECS4314 - Advanced Software Engineering

Professor Marin Litoiu

Prepared by - The Agile Alliance

Team members
Siddharth Bhardwaj
Michael Borg
Ruizhe Liu
Arshdeep Saini
Hengchao Xiang

Table of Contents

Introduction

Debias is a news aggregation tool. It aims to solve the difficulty in finding articles with different viewpoints by aggregating daily news articles and displaying their tone to allow users to differentiate articles pertaining to a topic based on their tone. Users can search for news articles in the front end, and articles pertaining to these topics will be selected from the database of processed news articles and returned to the user displaying the name, author, date, of the article as well as scores indicating the tone of the articles. The users can further filter down the articles within the topic by the tone, which can allow them to get a wider range of viewpoints from a topic.

We developed a conceptual architecture for Debias then derived the concrete architecture in the last phase. In this phase, we make an enhancement to our concrete architecture-prototype 0.2 implementation.

Enhancement Proposal

Motivation

Self-healing is the ability to recover from a failing component by first detecting improper operations (either proactively through predictions or otherwise) and then initiating corrective action without disrupting applications[9]. Thus, making it an invaluable feature in modern applications. Thus, as proposed by Professor.Litou and his team for EECS4314, our existing application will incorporate self-healing capabilities in the form of a new feature.

Continually, through this enhancement proposal- the Debias system use-case model will now account for a DevOps team member.

Conceptual design

The goal of this enhancement is to enable the system to log basic information about daily operations, including request, exception, and some completion flags of the pipe process, so that devops and developers can debug based on the log information when the system running into error state, and use the log analysis result to improve the fault tolerance of the system, and establish a certain degree of self healing loop.

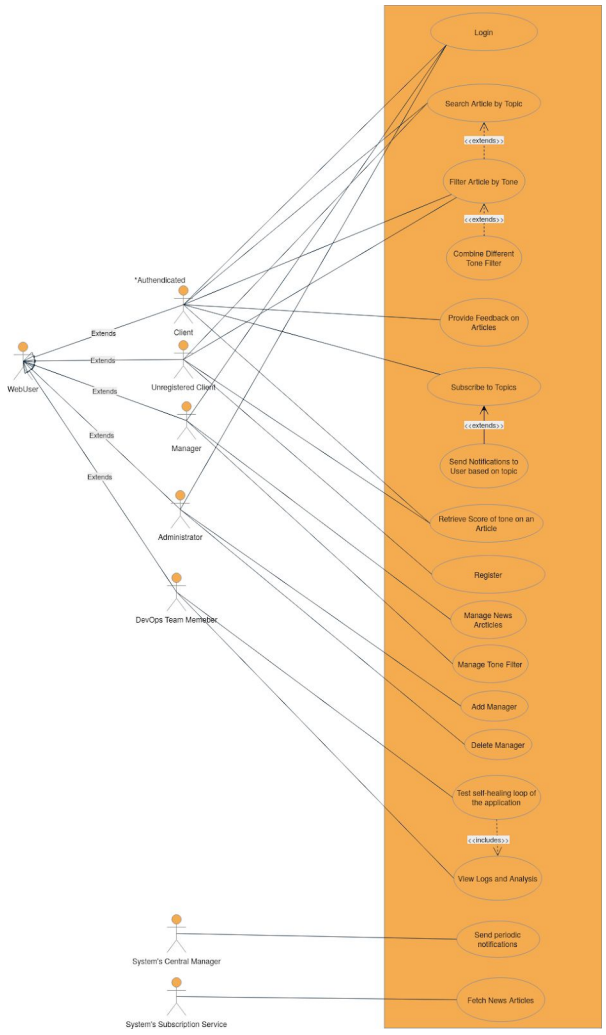

Figure1 shows the use-case model of Debias after adding the enhancement feature.

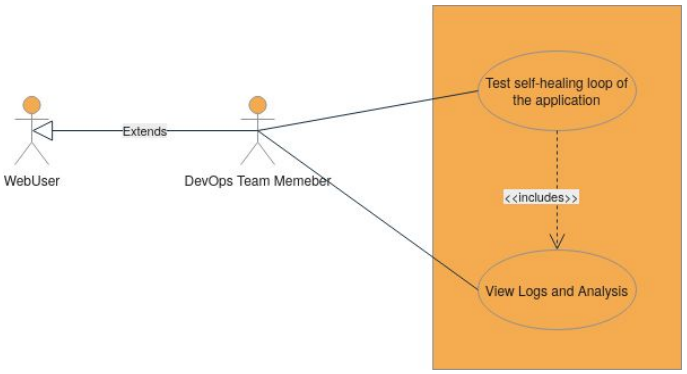

Figure2 shows the particular use case(s) that was achieved for the new actor- DevOps Team member.

Explanation of Figure 1 & 2: Through viewing logs and performing analysis, a DevOps team-member will be able to test the self-healing functionality of Debias. (Note: Fault was injected ahead of time, for the purposes of this assignment, so DevOps team-member can simply view that log data collected and perform analysis). Viewing logs use-case will be achieved through implementation- adding an additional interface to the website, which is accessible by the DevOps team member. Also, perform analysis use-case will be achieved through this same interface accessible by the DevOps team-member, where an explanation of each error/anomaly logged is briefly explained.

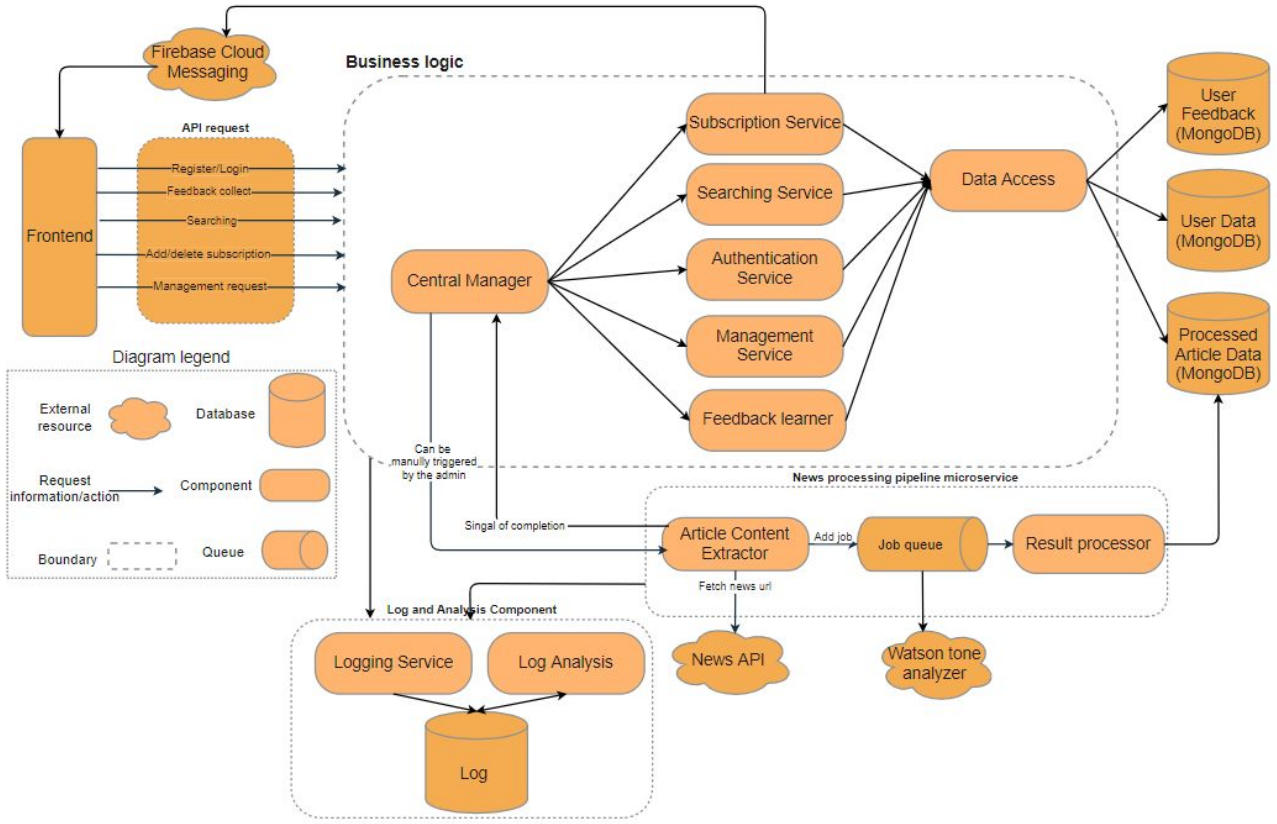

Figure 3: Conceptual architecture of the enhancement

Conceptually, the entire system will add a Log and Analysis component, which is responsible for collecting logs from all parts of the system, doing basic processing and analysis, and storing the system logs and analysis results in the corresponding database and taking some necessary action to deal with the detected failure . For Debias, both the backend and the independent microservice news processing pipeline should have log and analysis functions respectively.

Implementation Alternatives

ELK

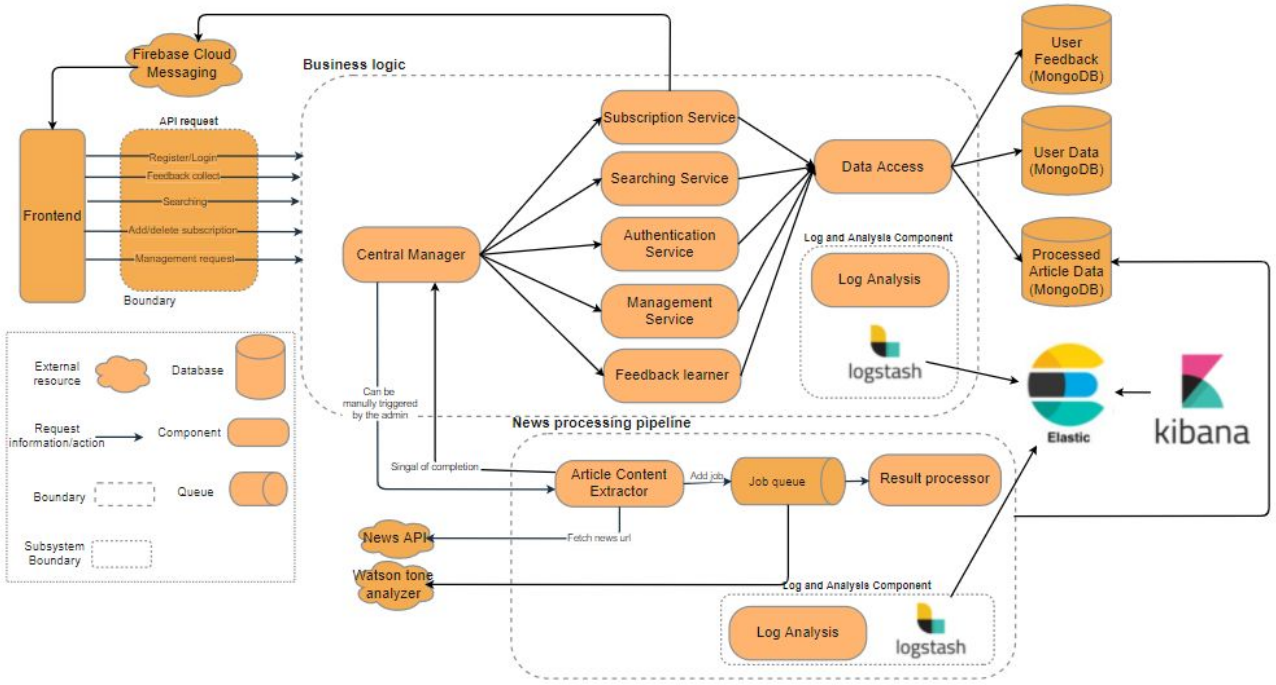

Figure 4: Concrete architecture of the enhancement with ELK stack

ELK Stack is designed to allow users to take data from any source, in any format, and to search, analyze, and visualize that data in real-time.

For us, the most basic way to use ELK stack is to deploy logstash in each app server separately, deploy elastic search and kibana separately, and then employ logstash to send logs to elastic, and realize the visualization through kibana.

ELK provides a mature process for aggregating logs. Logstash supports retrieving logs from many sources, including syslog, messaging (such as RabbitMQ) and JMX, and supports outputting data in a variety of ways; Elasticsearch is a real-time full-text search and analysis engine, provides the functions of collecting, analyzing, and storing data, while Kibana provides a rich visualization interface.

Because ELK is a mature log management stack, it has many advantages, such as good scalability, no-schema log storage, and mature concercy control. And using ELK we can avoid the cost of additional development of front-end log UI. But in our case, first of all, ELK needs three parts to be deployed and maintained separately, for small web applications, it will undoubtedly increase the burden of maintenance and more possible failure points, which is a challenge to maintainability. Secondly, deploying logstash separately on each app server will consume more computation resources, memory, CPU, etc., and there is a risk of reducing performance. Finally, ELK is a general log

management platform. We are aiming to achieve application-specific log analysis. Even if ELK is adopted, we still need to build our own log analysis component. But we will still use ELK as a system evolutionary solution, because as the application architecture gradually becomes complex, a mature general log management platform will provide a lot of convenience.

Utilize existing architecture

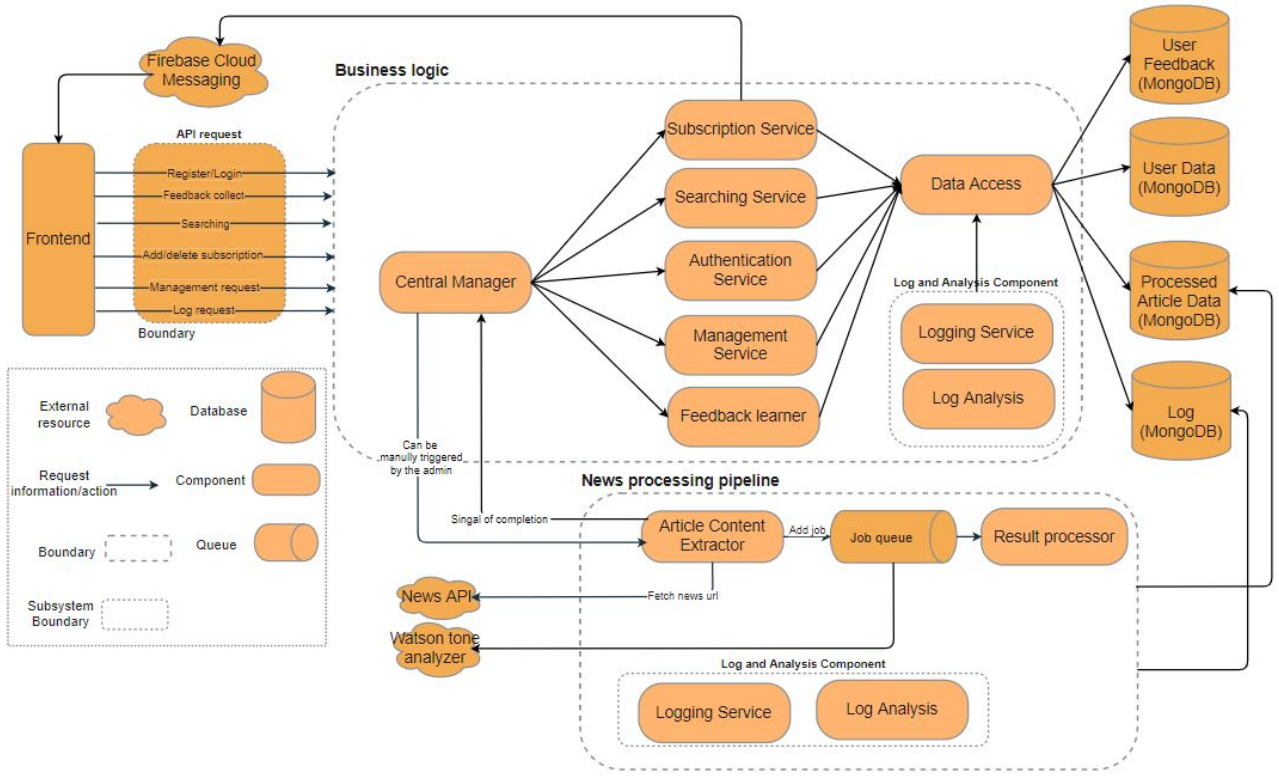

Figure 5: Concrete architecture of the enhancement with current architecture(Interactions with Log and Analysis component is omitted as it basically need to do interactions with every subsystem, to keep the diagram clean and clear, we omit it in this diagram)

Another solution is to directly use the current architecture and add more components to achieve a self-healing loop. As you can see in the architecture diagram, we need to add log analysis components to each independently running service, and develop the front-end UI to realize the visualization of logs.

But compared to deploying the three parts of ELK separately, for small applications, the time cost and complexity of this solution are relatively lower. In order to aggregate and view the logs, both the backend and the independent microservices will store the collected logs in the MongoDB database after simple analysis. Since the two parts of the service have communicated with the database before, we can reuse the existing parts of them.

The Log and Analysis component basically adopts the structure of pipe and filter form. The log is processed by unifying format, simple analysis, persistence, and cumulative analysis.

For the logging part, in our application, Flask runs with standard Python logging, so the default logger is used to log the errors. After the error is logged, a simple analysis will be

done on the logged error message to get the actual failure. In order to store the log and the analysed result in MongoDB, a new logger handler is created. Once a new error is logged, the handler will put the log and failure in the database.

Interactions with other features

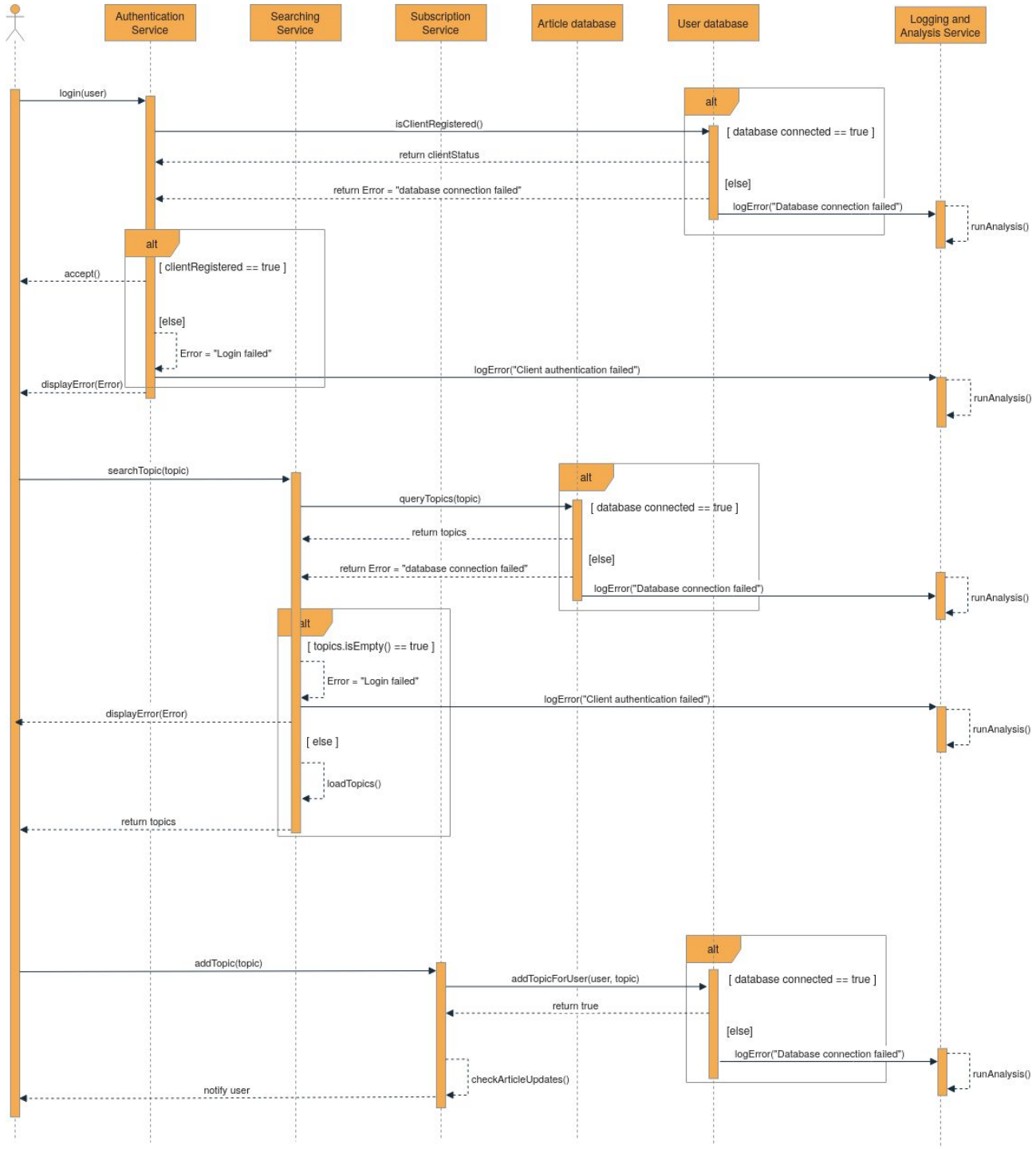

Figure 6 shows the sequence diagram capturing flow of login process, searching articles by topic process, adding topic to database process with the enhancement/fault mitigation tactic- logging and analysis service, implemented in the system.

Effects of the enhancement

Improvement

- The self-healing loop will strengthen the fault tolerance of the system, thereby improving the reliability of the system. Log and log analysis will also make debugging easier. From this perspective, the maintainability of the system is also improved.
- The enhancement improves the testability of the system. Since the log captures the information of the daily operation of the application, and we developed a front-end UI to view the log information, this will make the functional test more reliable with the confirmation from log, and it can also help developers quickly locate problems.

Potential risks

- The logging adds more database read and write operations, making the communication between the back-end and the databases become more frequent, and the database also needs to deal with more concurrency issues, which will increase the burden on the server to a certain extent and affect performance.
- With the gradual increase of the self-healing loop, the complexity of the system and the code base will gradually increase, which will have some negative effects on maintainability, but at the current stage, we believe that the impact of this enhancement on maintainability is still positive.

Self-healing loop

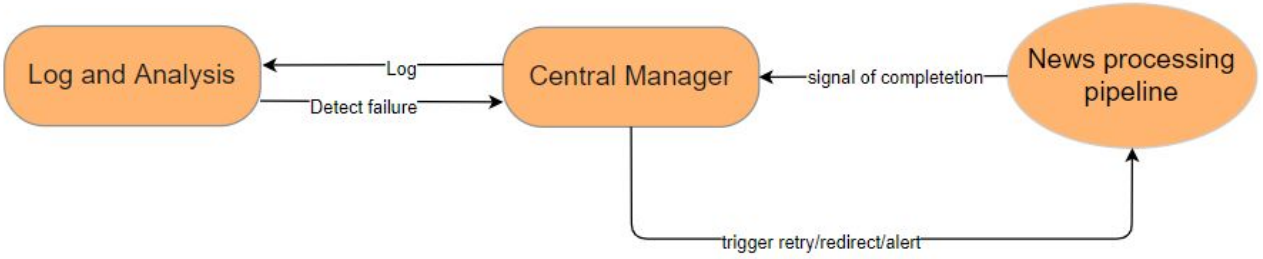

Figure 7: Self-healing loop between central manager and news processing pipeline

An example of a self-healing loop could be enabled by the Log and Analysis component is a self-healing case between the central manager and the news processing pipeline.

By our design, the news processing pipeline is an independent microservice with a pre-set scheduler, it should update the processed article database periodically, so every time it finishes the updating task it should send a signal to the central manager as a heartbeat to inform the job completion and the health of the microservice. After we added the Log and Analysis component, this completion signal will be logged the log analyzer will expect to see this log as a health check of the microservice in a certain around time, if this log is not detected or reported as an error, Log and analysis component will try to manually trigger the news processing pipeline through the central manager by the retry policy, if the retry

still failed after multiple times, we can escalate this issue to try to redirect the updating request to other instance of news processing pipeline or even send alert.

Plans for testing

There are two approaches our team explored for testing the self-healing capabilities of our system:

1) Before deploying to production, test with fault injection: we inject faults to test the resiliency of the system, either by triggering actual fault or by simulating them. A successful test is the system being handled fault or failure gracefully without impacting client experience.
2) While in production perform chaos engineering: randomly inject failures or abnormal conditions into deployed production (ec2) instances. This approach extends the notion of fault injection. A successful test is such that the application should be able to recover from “chaos”.

In essence, we chose to use the first approach for the implementation, while keeping the second approach in the backlog for the evolution of our system. We injected fault into Debias- disconnected system component, and throttled request handler. Then refactored system code to capture logs, store them into a database, and then perform analysis. The logs, and results of analysis are accessible through an interface implemented (in this project phase) for the DevOps team members.

Working prototype

Our team included the proposed enhancement (first approach) to our existing functional prototype. The screenshot below represents some logs captured and presented for the DevOps team members.

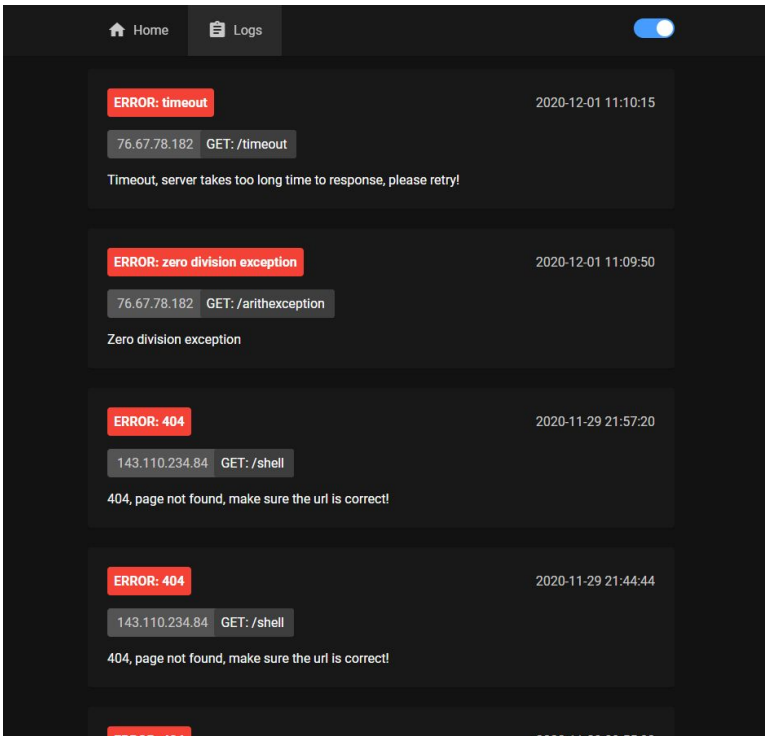

Figure 8: shows the functional prototype updated with our enhancement.

URL to resource

The updated prototype can be accessed here: http://18.206.238.196:5000/logs

Screenshot of Code Repository

Our concrete system v0.2's code can be found on our team's git repository here, or alternatively through our team's webpage for EECS4314 here.

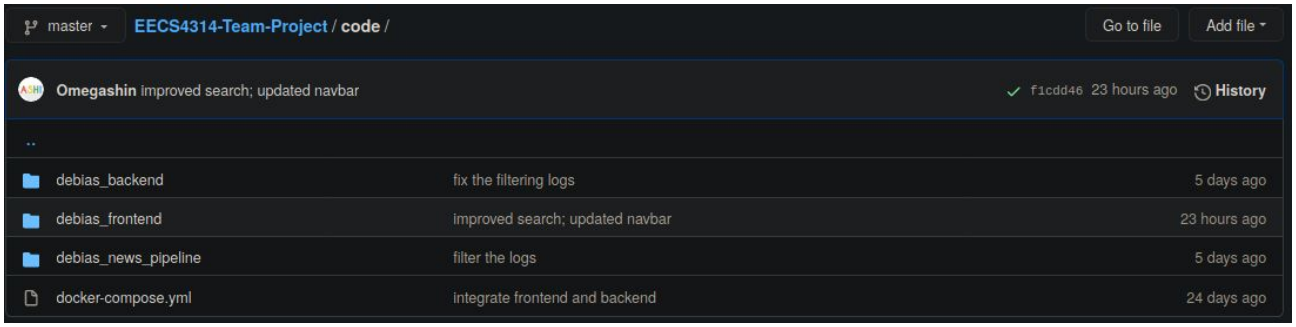

Figure 9: shows our live website's code. It is proof that we finished implementing the enhancement feature we proposed in the beginning of the report.

Conclusion

In conclusion, we proposed and implemented an enhancement to our concrete architecture-prototype 0.2 implementation. This enhancement enabled our system to log basic information about daily operations, including request, exception, and some completion flags of the pipe process. In essence, this feature is accessible on live website, under the

“Logs and Analysis” view- primarily intended for a DevOps team members, as these actors will be perform job function of debugging system log information when the system is running into error state, using analysis of log messages provided by our system. Thereafter, use the log analysis result to improve the fault tolerance of the system, and establish a certain degree of self healing loop.

Lessons Learned

Initially when we were considering what kind of architecture would be most suitable for our project we considered ways in which we could reduce the complexity of development and maintenance and decided to plan our architecture using the monolithic model, this would make it easier to plan the subsystems, the corresponding deployment strategy, and the scaling without having to consider the complex communication issues between various services at the beginning, we would then only need to consider the horizontal scaling of a single layer.

But as we began to consider more quality attributes and we observed the interaction of the subsystems, we gradually found that some of the subsystems are inherently less coupled with other subsystems and are more suitable for independent running. At the same time, monolithic limits the flexibility of scaling, because we must scale the entire backend, not just target on the certain subsystems that may become performance bottlenecks. But the microservice architecture does reduce the reusability of the code, for example, we must connect to the database multiple times, however using the microservice architecture has big advantage for developers as the codebase is more split up allowing for fewer conflicts when submitting code, and it allows for more focused testing, which can be combined with end to end testing to ensure the functionality works as intended. On the devops side using the microservice architecture means that one microservice going down does not bring down the entire product, this can be used to bring down and update various microservices, as one instance of the container can be running, while others are being updated, which allows us to perform maintenance without bringing down our uptime.

Data Dictionary

Central Manager: subsystem component responsible for routing the api and function as a scheduler-periodically triggering the task of updating article data and coordinating the function of applying user feedback to the article metadata.

News processing pipeline: microservice composed of three separate subcomponents-article content extractor, a job queue, and result processor. It is responsible for processing the article and presenting it to the user.

Feedback learner: subsystem component responsible for collecting user feedback data. Communicates with the Feedback database and Central manager to manage user feedback.

Authentication Service: subsystem component that communicates with the central manager to handle registration and login requests- following NIST guidelines to ensure compliance with security and privacy requirements.

Searching Service: subsystem component that relies on the article metadata database to provide search functionality. Uses keywords, tone filters, and topic tags (for subscription) to search.

Subscription Service: subsystem component that interacts with the central manager to handle requests to adding or deleting a topic tag from the front end, and manage publications and push subscriptions requests.

Management Service: subsystem component responsible for fulfilling the management requests of administrators and managers. Interacts with user data database and article metadata database to handle functions invovinging managing user permissions, article visibility to Clients, and visibility of tone filters available to Clients.

Log and Analysis Component：Responsible for formatting, analysing the system log.

Log Data Database: Stores the system log information.

User Data Database: Stores the user information, including user credentials, subscription list information etc.

Processed Article Database: Main database of the system, will store the article information with tone analysis scores, visibility, abstract, url and other necessary information.

User Feedback Database: Stores the user feedback data

Agile: Software development in which requirements are discovered and solutions are discovered through the collaborative effort of self-organizing and cross-functional teams, and the customer/end user.

User: a Client of the System. Extents WebUser.

Naming Conventions

TCP/IP address: **T**ransmission **C**ontrol **P**rotocol over **I**nternet **P**rotocol address. is a suite of communication protocols used to interconnect network devices on the internet.

SaaS: **S**oftware as a **S**ervice- The capability provided to the consumer is to use the provider's applications running on a cloud infrastructure. The applications are accessible from various client devices through either a thin client interface, such as a web browser (e.g., web-based email), or a program interface.

COTS: Commercially available **O**ff-**T**he-**S**helf**.** Software and hardware that already exists and is available from commercial sources. It is also referred to as off-the-shelf.

URL: Uniform **R**esource **L**ocator. Is a short string containing an address which refers to an object in the "web." URLs are a subset of URIs

API: **A**pplication **P**rogramming **I**nterface. Is an intermediary software that allows two applications to communicate with each other.

REST: **RE**presentational **S**tate **T**ransfer. A software architecture style that defines a set of constraints when creating web services.

SOAP: **S**imple **O**bject **A**ccess **P**rotocol. It defines an RPC mechanism using XML for client-server interaction across a network

MVC: **M**odel-**V**iew-**C**ontroller. Architectural pattern commonly used in developing client-facing applications.

NIST: The **N**ational **I**nstitute of **S**tandards and **T**echnology. Is a government entity that helps organizations to better understand and improve their management of cybersecurity risk.

Scrum: is an agile framework that helps teams work together. Encourages teams to learn through experiences, self-organize while working on a problem, and reflect on their wins and losses to continuously improve.

ELK: the acronym for three open source projects: **E**lasticsearch, **L**ogstash, and **K**ibana.

References

1. “Guide to Secure Web Services”. The National Institute of Standards and Technology. Retrieved from: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-95.pdf
2. “Tone Analyzer”. IBM Cloud Docs. Retrieved from https://cloud.ibm.com/docs/tone-analyzer?topic=tone-analyzer-about
3. Account Plans. IBM Cloud. Retrieved from: https://www.ibm.com/cloud/free
4. “Agile 101”. The Agile Alliance. Retrieved from: https://www.agilealliance.org/agile101/
5. “Software Processes Requirements”. Marin Litiou’s 4314 class slides. Retrieved from:https://eclass.yorku.ca/eclass/pluginfile.php/775191/mod_resource/content/2/Lecture%203-%20Software%20Processes.pdf
6. “Scrum”. Atlassian Agile Coach. Retrieved from https://www.atlassian.com/agile/scrum
7. The Complete Guide to the ELK Stack. Retrieved from https://logz.io/learn/complete-guide-elk-stack
8. What is Elastic stack which everyone is talking about. Retrieved from https://medium.com/@lakinisenanayaka/what-is-elastic-stack-which-everyone-is-talking-about-6c5d94d83983
9. “Self Adaptive Software Systems”. Professor.Litou. Class Slides. Retrieved from: https://eclass.yorku.ca/eclass/mod/resource/view.php?id=378496