Graphelier

Team members

Name and Student id	GitHub id	Number of story points that member was an **author** on.
Lucas Turpin 40029907	Lercerss	14+19+20
Patrick Vacca 40028886	pvacca97	10+16+26
Duy-Khoi Le 40026393	alvyn279	11+18+21
Panorea Kontis 40032422	panoreak	10+14+21
Chirac Manoukian 40028500	chimano	18+16+26
Christoffer Baur 40028050	chris-baur	10+10+18
Riya Dutta 40028085	riyaGit	12+11+27

Project summary

Graphelier is a web application that displays historical financial market data through an intuitive graphical representation. The application allows the user to explore the state of the market's order book across time, viewing events that modify its contained orders. On each price level, not only is the total order volume displayed but also how it is divided into individual orders. Users can explore the order book at varying time granularities from hours or minutes to individual events separated by only a few nanoseconds. Historical data can be played back starting from any point in time to view the changes to the order book as they occurred in real time. News articles relating to different financial instruments are presented in a timeline, allowing users to explore important events and how they affect trading patterns around the release of new public information.

Risk

The project's biggest risks are related to the data required to achieve its features. The first one comes from the fact that data is not easily accessible as it is distributed on a subscription basis or sold from various vendors. The second is the size of the data set which can reach multiple gigabytes per day. To allow for an enjoyable user experience, the application must be able to efficiently handle this large data set.

The first risk was tackled by using a data sample from LOBSTER then using that sample as a basis to build a larger extended data set.

The second was tackled by building a large data set early. A key frame algorithm is used for building an orderbook efficiently for any given time. Due to the inherently partitioned nature of the data, the application is scalable to achieve higher throughput if necessary.

Legal and Ethical issues

The market data required to accomplish this project cannot be distributed openly and therefore poses a challenge in both acquiring it and displaying it publicly. As such, our project uses fake data based on a sample from June 2012 to circumvent those issues.

Velocity

Project Total: 26 stories, 119 points over 19 weeks

Iteration 1 (2 stories, 6 points)
Containerized the project using docker and docker-compose. Wrote a parser that parses the market data and saves it in MongoDB. Backend successfully generated an order book for a fixed point in time and the react application displayed it to the user.

Iteration 2 (1 story, 3 points)
Set up the continuous integration environment for the project using Circle-CI. Set up the linters and tests for all the components in the project.

Iteration 3, (2 stories, 5 points)
Implemented the keyframes algorithm to build the order books for any given time. The react application now allows users to pick the time they would like to generate an order book for. Modified the parser so that it can extend our sample to be longer than an hour. Styled the frontend.

Iteration 4, (4 stories, 17 points)
The application now displays the list of messages. Users can now see the order book as it is being modified one message at a time.

Iteration 5, (2 stories, 4 points)
The instrument whose orderbook is analyzed can now be selected. Furthermore, the application is now adaptive to custom screen sizes (i.e.: the computer screens at Squarepoint).

Iteration 6, (1 story, 2 points)
Timezone-related bugs were fixed. The action of clicking on a row of the message table will lead to the update of the orderbook being shown.

Iteration 7, (2 stories, 7 points)

Migration of front-end source code to a typed version of JavaScript, namely TypeScript. The application can now be accessed through a public URL. The continuous integration and continuous delivery pipeline is now set up.

Iteration 8, (5 stories, 24 points)
Graph displaying the best bid and best ask over time for an entire business day. Added an icon while loading a new orderbook state. Prompted the most recent message in the message list. Convert native DatePicker to be compatible with material-ui DatePicker. Updated milestone document.

Iteration 9, (0 stories, 0 points)
Documentation was updated to include class diagrams of scripts. Fixed bug where graph did not display all points for a day. Fixed x-axis window resize bug. Fixed bug with the graph displaying the incorrect time scale. Bug related to different integer types (32 and 64 bit) was fixed to make storing of integers consistent.

Iteration 10, (1 story, 8 points)
Migration of front-end and back-end source code for implementing the zooming feature. Migration of the front-end source code, completing the order-information feature.

Iteration 11, (0 stories, 0 points)
Docker initialization for implementing the news-article service. Fixed incorrect data bug for the graph relating to the zooming feature. Added animations for the playback feature.

Iteration 12, (0 stories, 0 points)
Fixed bug where unable to zoom back to a full day. Implemented animations for modifications to the orderbook (front-end and back-end source code). Implementation of playback route using websocket. Implementation of basic flask server for news api. Back-end source code for summarizing an article for the news feature. Fixed timezone handler bug in the front-end.

Iteration 13, (6 stories, 43 points)
Documentation to update architecture. Completed news-article feature. Completed playback feature. Fixed exposed port bug for the news service. Fixed timestamp bug for playback. Implementation of cluster route for news. Fixed division by zero bug in news summarizer. Fixed bug for missing direction in playback modification.

Release 1 Total: 9 stories, 31 points over 8 weeks
Release 2 Total: 10 stories, 37 points over 11 weeks
Release 3 Total: 7 stories, 51 points over 11 weeks

Overall Arch and Class diagram

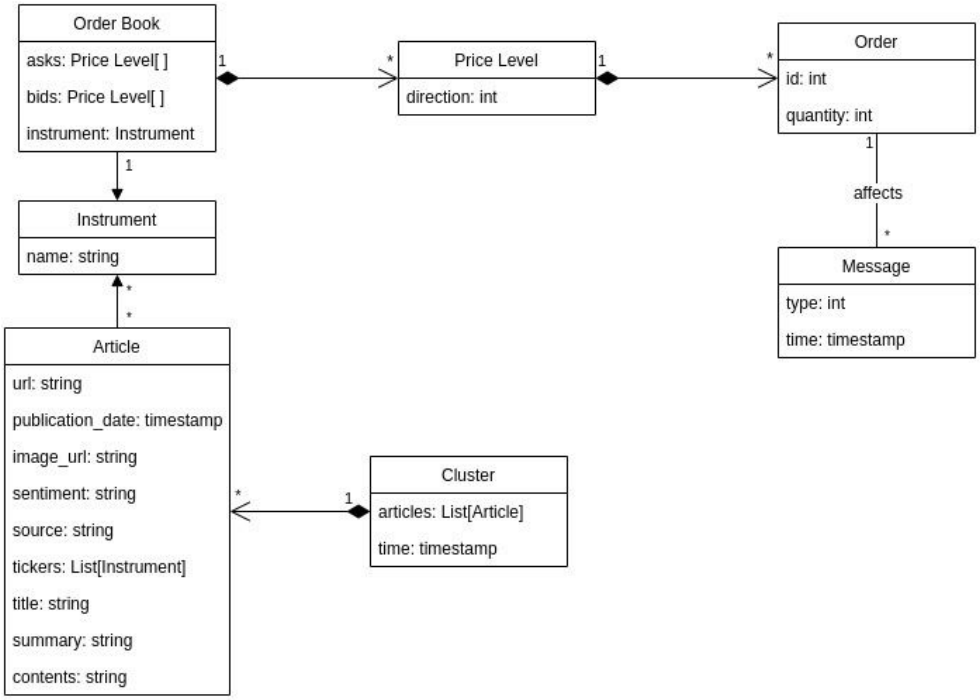

Figure 1: Domain Model

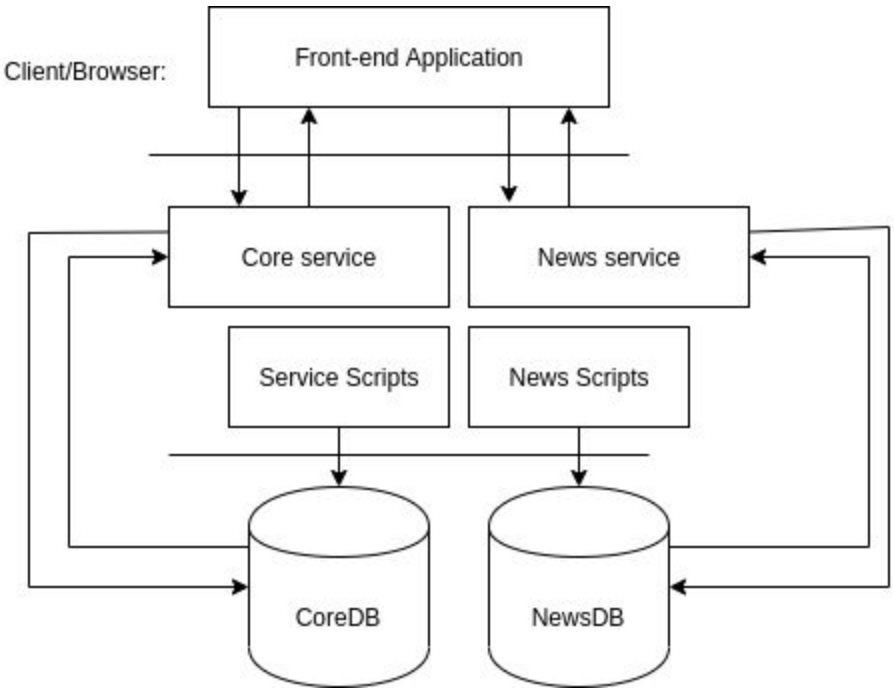

Figure 2: High-Level Architecture

A new service was added (News service). We have decided to go with a micro-service oriented architecture to lower the coupling between our different services. This would allow one service to go down without affecting the other.

From a high-level architecture perspective, the project is separated into three separate layers composed of the front-end application, the back-end service with its associated scripts, and the mongoDB database.

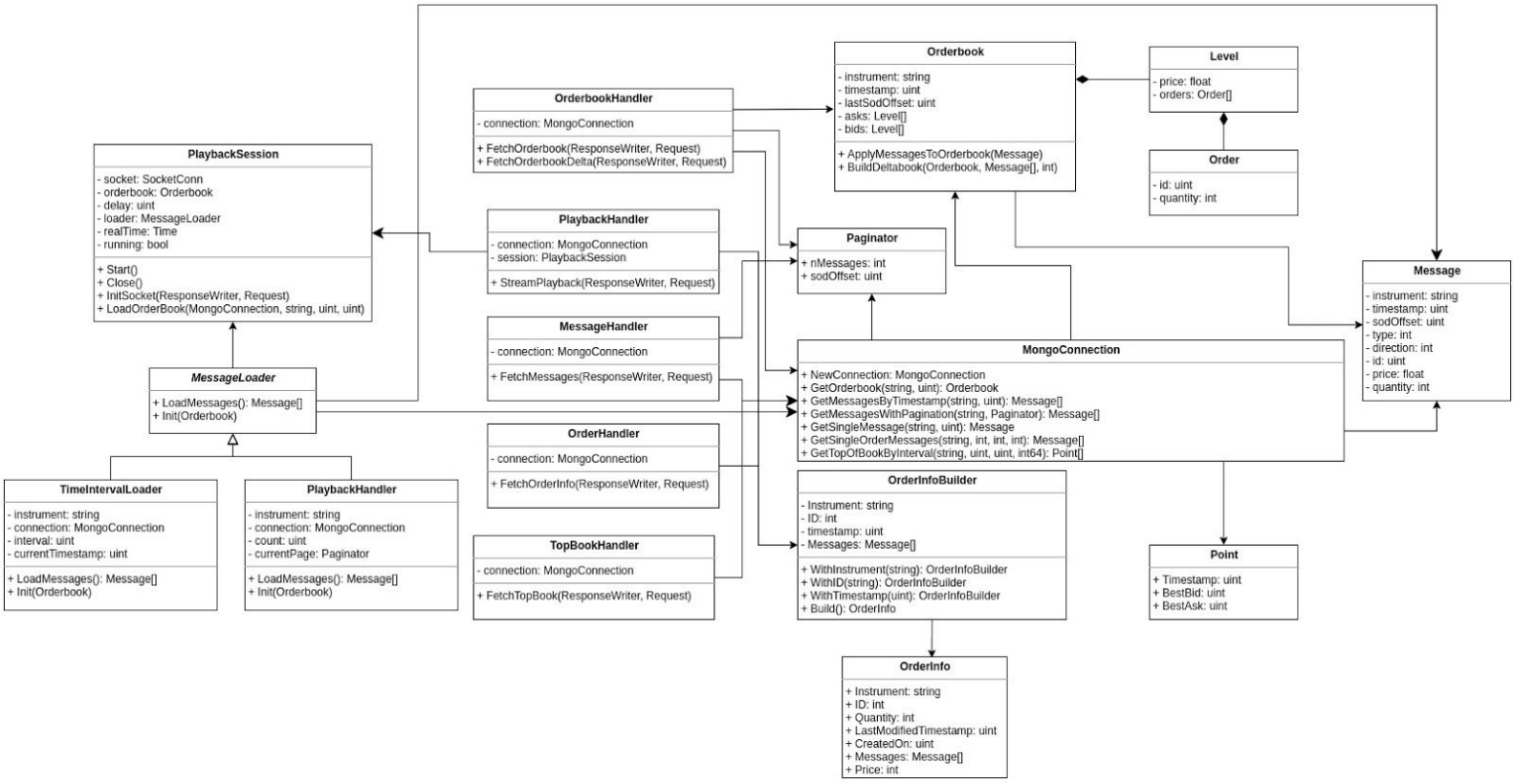

Figure 3: Class Diagram - Service Implementation

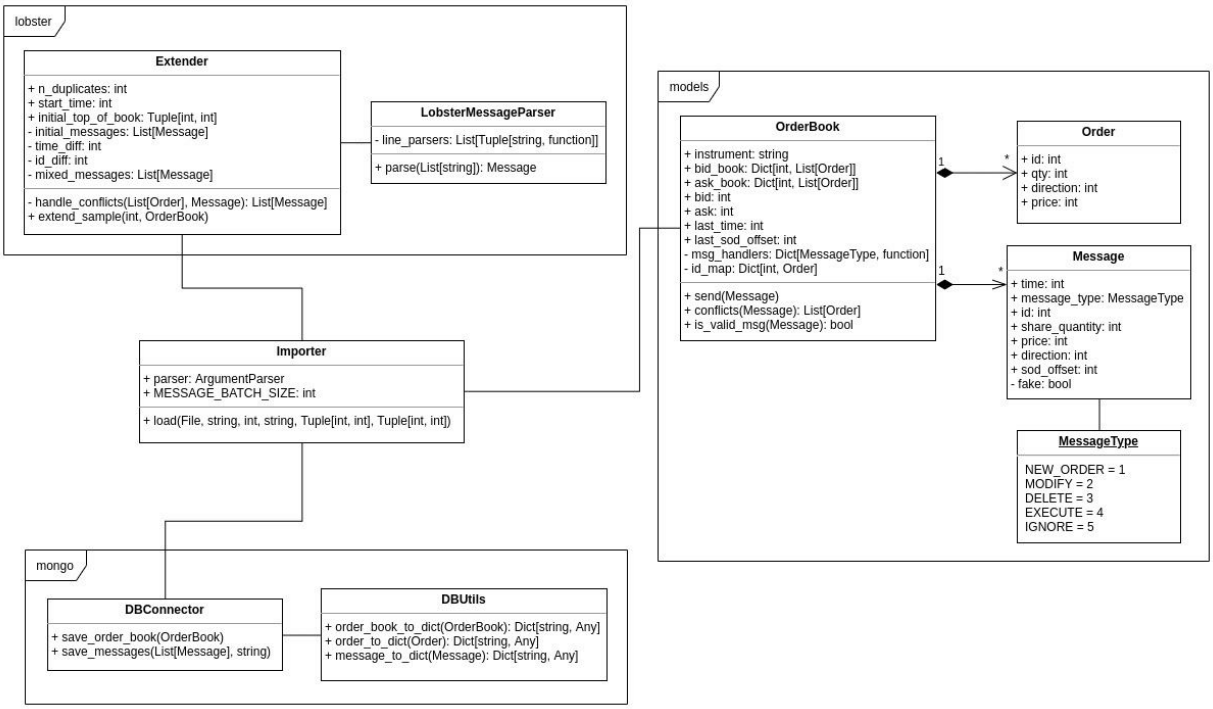

Figure 4: Class Diagram - Scripts Implementation

The OrderBook contains an array of PriceLevels which contains an array of Orders. The OrderBook collection represents snapshots of the orderbook at every interval specified in the Meta collection. The Message collection is used to store messages broadcasted by the market. The data view can be seen in figure 7.

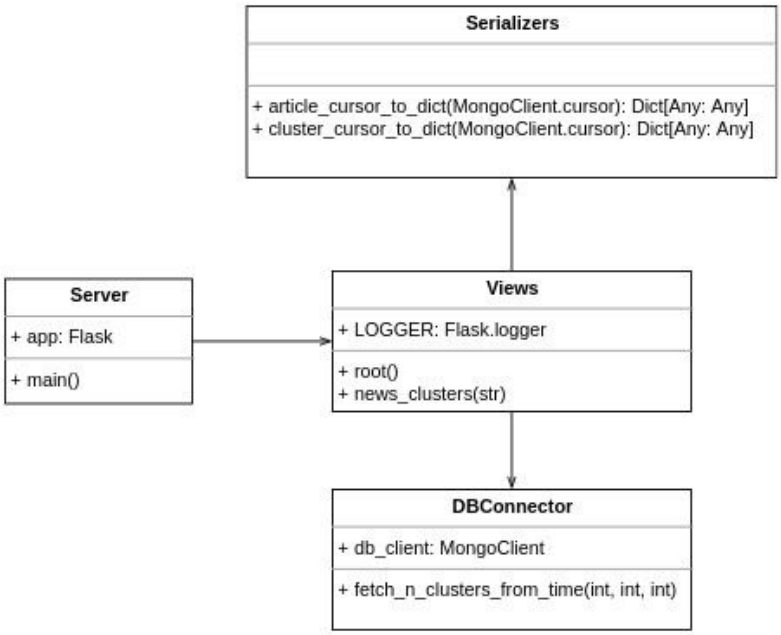

Figure 5: Class Diagram - News Service Implementation

The Graphelier news service is simplistic. The views serve news clusters based on an epoch timestamp through a single route: */news_clusters/<timestamp>*. This requires a query to the news database and the use of serializers to make the information front-end processable.

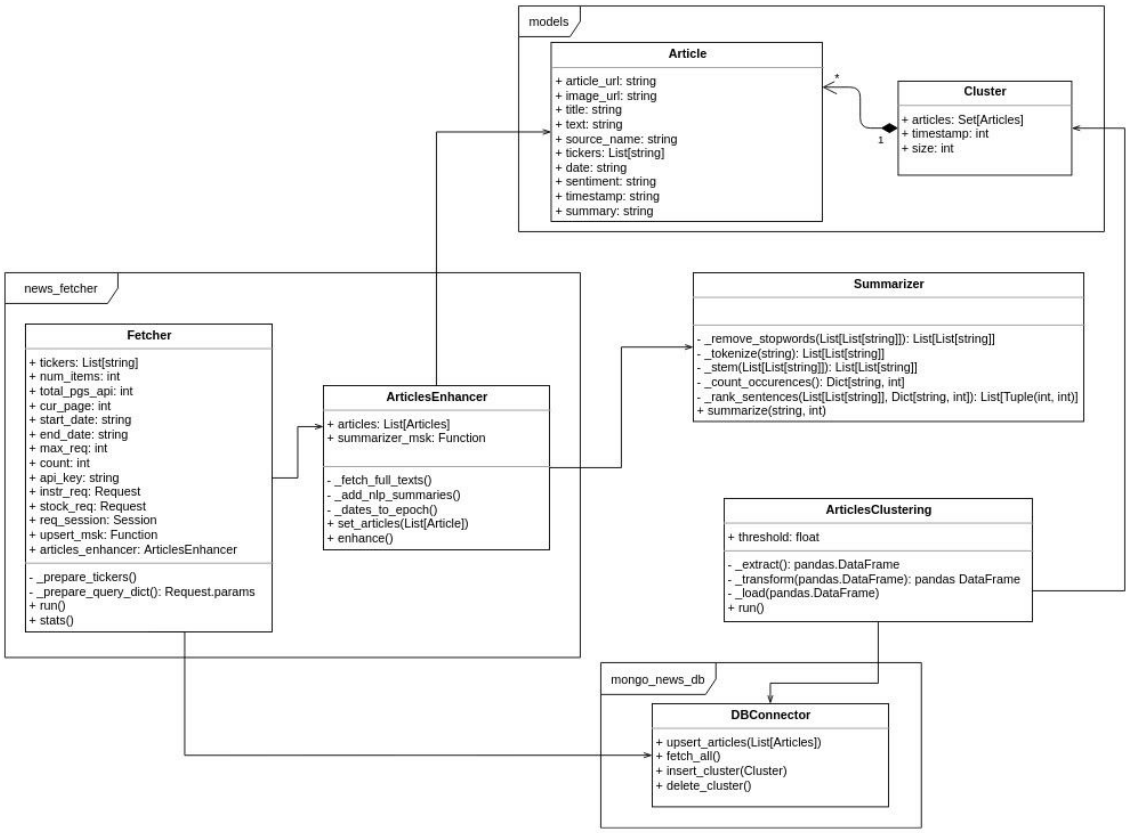

Figure 6: Class Diagram - News Scripts Implementation

Graphelier's news module rests around two main objects: Article and Cluster. A Cluster is an aggregate of Article objects, as it holds an array of articles. The scripts use these models to perform operations and store to the NewsDB, as illustrated in Figure 7.

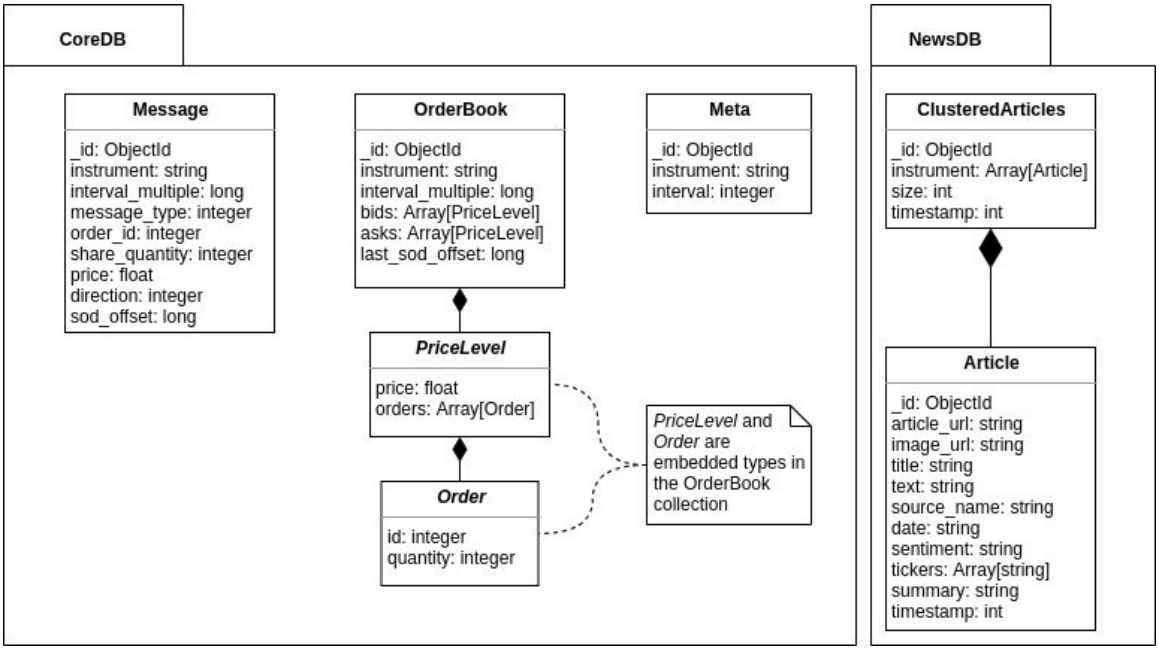

Figure 7: Data View

The NewsDB portion was added in this release. The application now handles the storage of news articles.

Infrastructure

On the front-end side, React bootstrapped with Create React App is used to implement a dynamic application on the browser. As per the language used, TypeScript is included on top of the React JSX code to enforce typing in the source code. This allows for design and implementation by contract. With models defined, the interaction between the front-end and back-end is made seamless through verifiable interfaces with respect to each other. Not only does this improve the development process for the developer, good code quality and standards are encouraged.

On the back-end side, two HTTP services are used to serve and process the data presented by the front-end. The first service is built using the golang built-in http library along with Gorilla Mux. MongoDB is used as the database for both services.

The scraping of news articles for the news service is performed on StockNewsAPI, which specializes on finance-specific articles. It includes features such as article filtering based on instruments and sentiment analysis. To enhance these fetched articles, the open-source library Newspaper is used. The latter internally uses natural language processing techniques to scrape an article's full relevant text.

For news enhancement, NLTK was used to summarize the news articles, because of all the tools it provides for natural language processing such as stemming and

tokenizing. Flask was used to serve data from the database because of how it is to set up and how light it is. A separate instance of MongoDB was used for this service to lower the coupling between the core service and the news service. This would make it so that if the core service somehow experienced downtime, the news service would not be affected. Finally, scikit-learn was used to cluster similar news articles together. Specifically, we used this module to transform text into TF-IDF vectors and to cluster them using the Agglomerative clustering algorithm.

Each component is run as its own Docker container to allow for environment encapsulation and orchestration of processes.

The production environment is very similar to the development environment in terms of backend and mongo. However, the frontend application gets served using an Nginx container. The production environment is hosted on AWS, making use of their EC2, ECS and ECR services. The docker containers are run on EC2 using ECS for deployments, and the docker images are pushed to ECR. The setup can be viewed in figure 8.

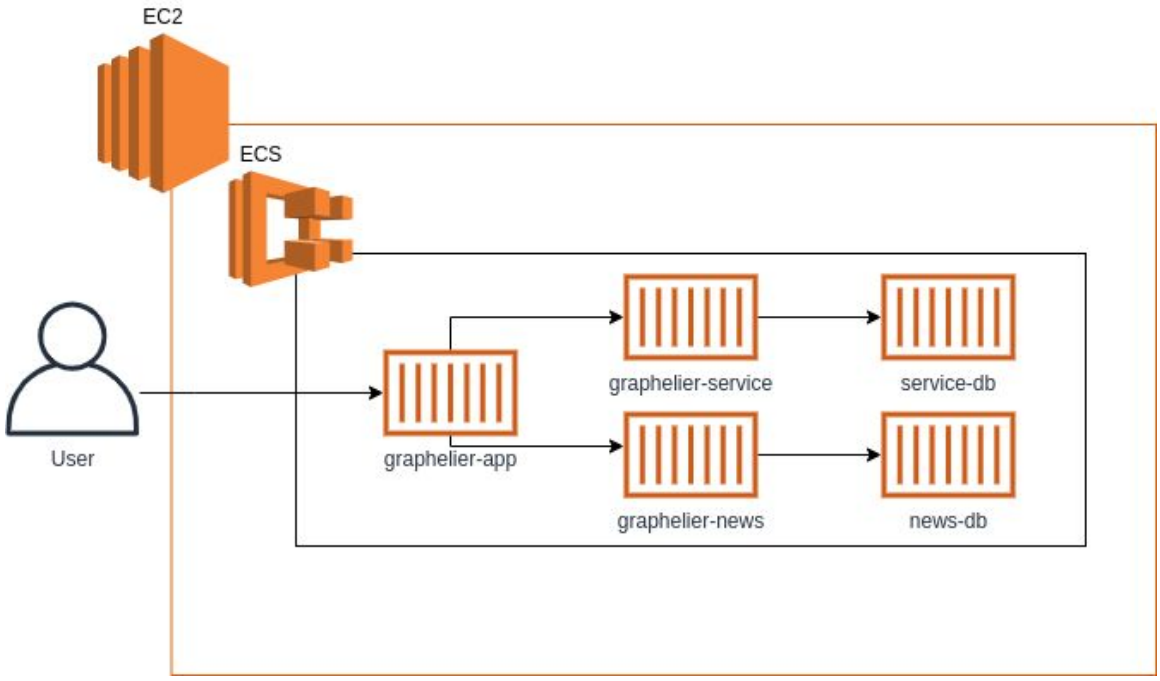

Figure 8: Production Infrastructure Diagram
Adding the new service resulted in the creation of 2 containers: graphelier-news, and news-db.

Name Conventions

JavaScript code should be formatted according to a custom set of rules extended from the Airbnb JavaScript Style Guide and ES6 standards. TypeScript standards are also applied at compile time in order to detect coding style errors.
Go code should be formatted according to Effective Go.

Python code should be formatted according to the PEP 8 Style Guide.

Code

Key files:

File path with clickable GitHub link	Purpose (1 line description)
graphelier/core/scripts/importer.py	This file is responsible for populating the database with fake orderbook and message data based on a given sample set.
graphelier/news/scripts/summarizer.py	This file is responsible for summarizing the news articles given based on ranking sentences.
graphelier/core/graphelier-service/models/orderbooks.go	This file is responsible for holding the orderbook structure as well as various manipulation functions on an orderbook itself.
graphelier/app/src/components/MessageList.tsx	This is the component holding a pageable list of messages that are applied on the orderbook.
graphelier/app/src/components/TopOfBookGraph.tsx	This file is responsible for displaying the best bid and best ask over the course of a full business day in the format of a graph in which the users can select various points.

Testing and Continuous Integration

Test File path with clickable GitHub link	What is it testing (1 line description)
graphelier/core/scripts/extender.spec.py	This test file is responsible for ensuring that the sample set given is properly extended over multiple days in order to have a larger set of data.
graphelier/core/graphelier-service/api/hndlrs/obhandler_test.go	This test file is responsible for ensuring the different orderbook and deltabook calls to the api return the correct information to the frontend application.
graphelier/core/graphelier-service/models/orderbooks_test.go	This test file is responsible for ensuring that the manipulation of the orderbook content is correct.
graphelier/app/src/tests/component-tests/OrderBookSnapshot.test.tsx	It tests the date and timestamp picking functionality and the handling of the initial orderbook raw data received from the back-end.

graphelier/app/src/tests/component-tests/TimestampOrderBookScroller.test.tsx	It tests the correctness in rendered orderbook rows and function calls related to message stepping and scrolling within the orderbook table.

Continuous integration is configured on CircleCI with builds for every pull request or new commits to *master*. Builds execute tests, static code analysis and packaging for each of the three components composing the application: the front-end application, the back-end service and the data-related scripts. An additional tool called GolangCI provides a static analysis report directly on GitHub pull requests specifically for the service code implemented in Go. These tools together enforce that commits are always functional and adhere to the code style rules for each programming language.

Continuous deployment is also configured on CircleCI. Upon merging a branch on master, docker images for each of the application's components are built then pushed to AWS' Elastic Container Registry. A final job then updates the Elastic Container Service's task definition of our application to use these new docker images. The service that is running our application then tears down the old containers and redeploys containers built off the new images.

Customer Feedback

The customer feedback can be found at the following link.