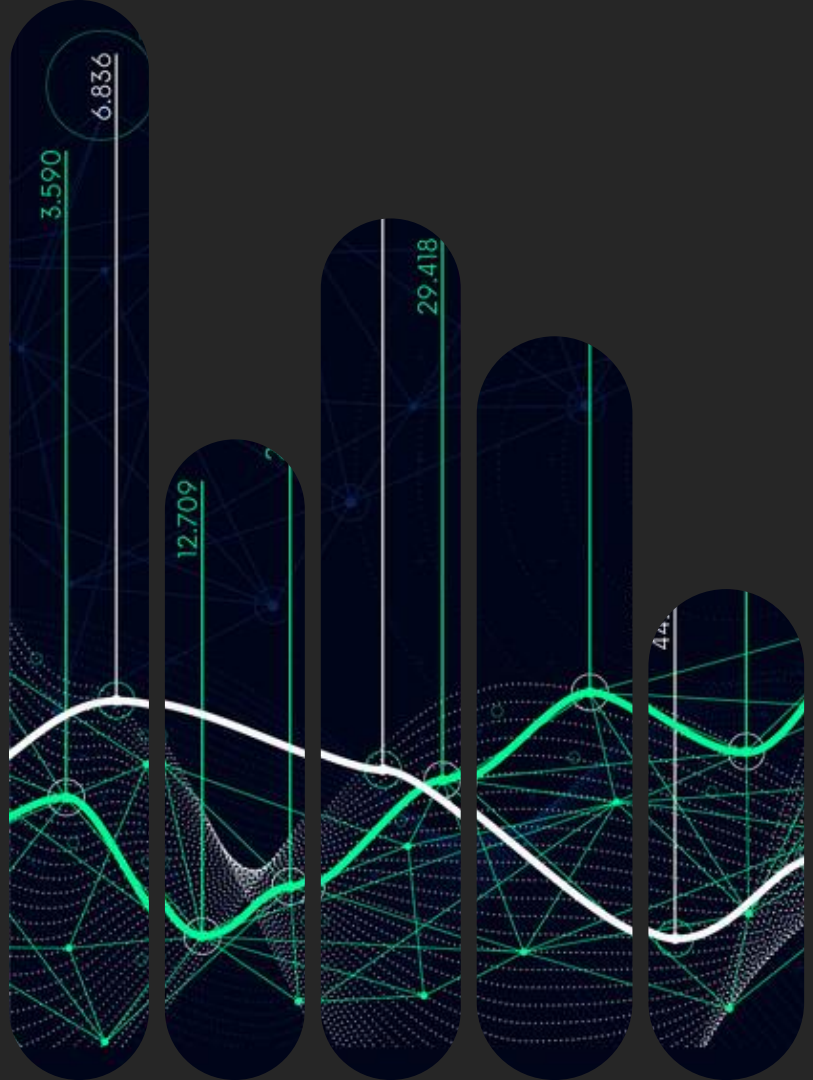


ChronoWave

Time Series Repository Web Application

Isaac Perkins, Geoffrey Brendel, Aleks Stevens, Cynthia Meneses

Objective/Concept of Operations

A web application that acts as a central repository for the storage and sharing of time series data sets for people/teams interested in creating and testing Time series Forecasting models.

Core Features:

- Allow time series contributors to upload their time series data sets along with forecasting tasks to the repository for sharing with the community.
- Offer a centralized repository for the browsing and downloading of time series datasets for use in training new Time series Forecasting models
- Allow users to upload forecasts made for specific time series sets and generate error metrics that let the user know how their model performed on the testing dataset(which is kept hidden).
- For forecasts, generate visualizations that show the difference between your prediction and the ground truth data.
- Allow the browsing of forecast performance data for all forecasts uploaded for a time series set so you can see how your model performed in comparison to other peoples models
- Allow the website administrator to remove certain time series sets from the repository at their own discretion

Overall, a web application for time series sharing and forecast evaluation will help foster a community of data-sharing and open-source research that will allow the field to progress.

System Demo

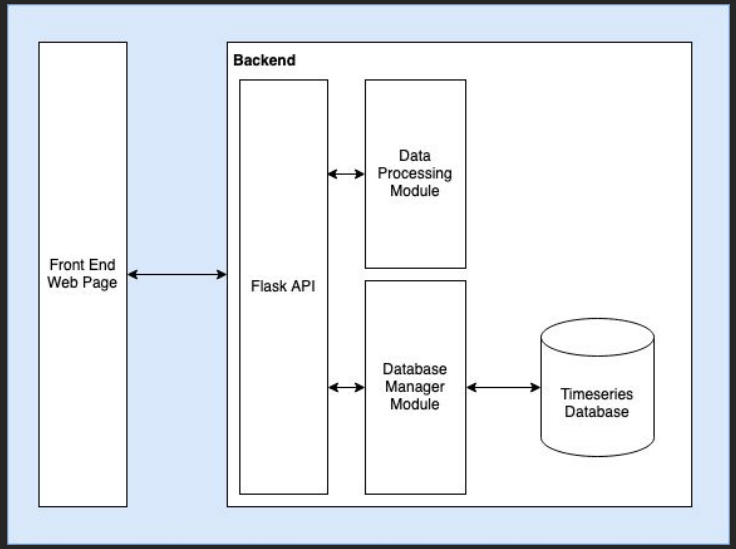
High Level System/Software Design

The web application will have a **Flask API** as the intermediary between the front-end and back-end modules. The Flask App will use the back-end software modules to generate HTML web pages with dynamic content.

The **Database Manager Module** that handles all data ingestion, data storage, and data retrieval/export jobs.

The **Data Processing Module** is responsible for comparing test data with forecast results and creating visualizations.

The **Time Series Database**, is the DBMS that the **Database Manager Module** interacts with. This is a MongoDB Atlas Cluster.



This software architecture clearly defines the communication boundaries between different software modules which allows for quick development by pre-defining what the interfaces between any two software modules are.

This model also allows for high software maintainability, as the use of predefined abstract interfaces allows for the swapping out of software components in a way that minimizes the amount of code that needs to be changed(ie. We can switch DBMS systems and only have to rewrite the **Database Manager Module** since the communication interface stays the same)

Flask API (main.py)

Main.py:

- Served as the main file for the flask app instance and the flask app routes.
- The routes have a url for each page and a function that processes any operations we want to send to the html or receive from user input.
- In our example we see the use of the Database manager instance db_manager.
- In this example if we get a “Post” request from the html page we then create a variable to the input file.
- Then we make sure its a valid file and store it using the database manager method “store_timeseries_set()”
- And the database manager does the work communicating with the mongodb server.

(Example) Flask Upload data flask route:

```
@app.route("/upload_data", methods=["POST", "GET"])
def upload_data():

    if (request.method == "POST"):
        if (request.files):
            file = request.files["dataset"]
            file_name = secure_filename(file.filename)

            if (file and allowed_file(file.filename)):
                file_path = os.path.join(app.config["WORKING_DIR"], file_name)
                file.save(file_path)
                db_manager.store_timeseries_set(file_path)

            return redirect(request.url)

    return render_template("upload_data.html")
```

Flask API (templates)

Templates:

- Templates is a folder where we stored all of our html files.
- We use Flasks template system to structure each page
- Each template defines blocks such as “description” or “content” which will be displayed in the same way
- Using templates allows for quick development of new tools and pages because we don’t have to worry about the structure
- Alongside templates we use Bootstrap 5 for styling
- Having a solid page structure and style setup from the beginning allowed for efficient development of the tools that make the site functional

(Example) base.html:

```
<body style = "background-color: #FAFAFA;">
    <div class="container-fluid">
        <div class="row flex-nowrap">
            {% include "navigation.html" %}

            <main class="col ps-md-2 pt-2">
                <div class="page-header pt-3">
                    <h2>{{ self.title() }}</h2>
                </div>
                <p class="lead">{% block description %}{% endblock %}</p>
                <hr>
                <div class="row">
                    <div class="col px-5">
                        {% block content %}{% endblock %}
                    </div>
                </div>
            </main>
        </div>
    </div>
```

Flask API (templates)

Templates:

- These html files may be static or dynamic.
- For our dynamic files we used Ajax and Jquery to dynamically display any meta data or metrics required.
- In our example to the right we show our most simple jquery/ajax script that dynamically displays a list of Time-series sets in the database.
- The user can then select a set and upload their forecast to that set.
- #select-dataset represents our html selection.
- Ajax recognizes changes in the selection and sends dataset_id to the flask API for proper uploading of the forecast.

(Example) upload_forecast.html:

```
<script>
    $(document).ready(function() {

        // View selected metadata
        $("#select-dataset").change(function() {
            $.ajax({
                url: "",
                type: "get",
                contentType: "application/json",
                data: {
                    dataset_id: $("#select-dataset").val()
                },
                success: function(response) {}
            });
        });
    });
</script>
```

Data Representation(Upload)

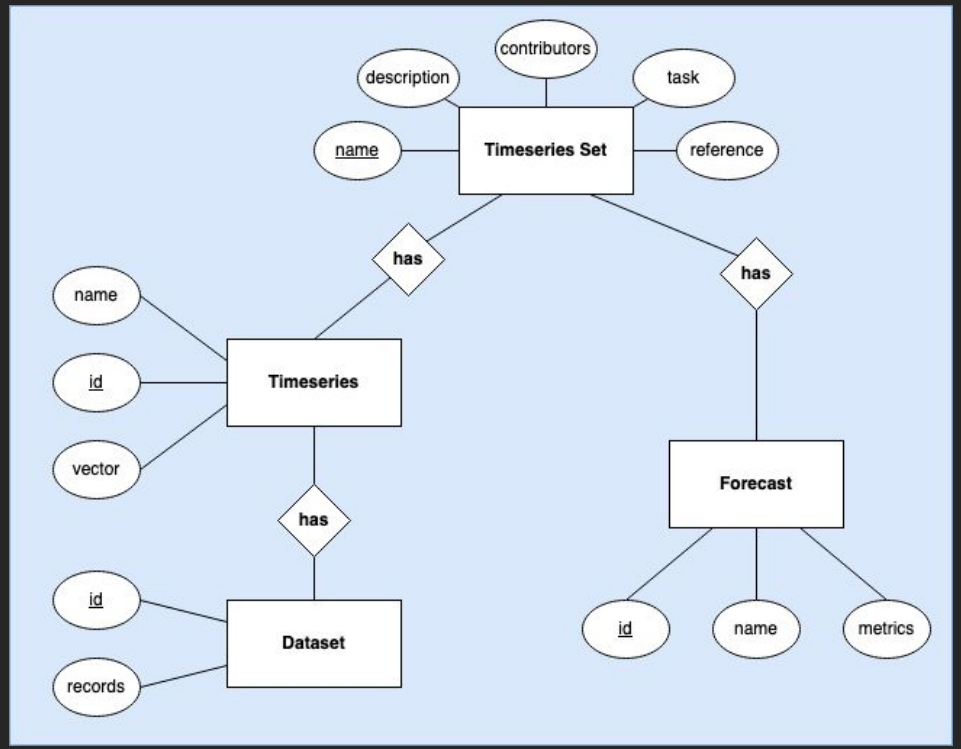
```
{
    "name": "timeseries_set_test",
    "timeseries": [
        {
            "name": "timeseries_test",
            "description": "description",
            "domains": ["domains"],
            "keywords": ["keywords"],
            "vector": {
                "timestep_label": "datetime",
                "measure_labels": {
                    "measure": "unit"
                }
            },
            "length": 5,
            "sampling_period": "1m",
            "training_filename": "training.csv",
            "testing_filename": "testing.csv"
        }
    ],
    "task": {
        "forecast_period": "1m",
        "count": 1,
        "timeseries_name": "timeseries_test"
    },
    "description": "description",
    "domains": ["domain"],
    "keywords": ["keyword"],
    "contributors": ["contributor"],
    "reference": "reference",
    "link": "link"
}
```

- Data is uploaded to the ChronoWave as a zip file containing a series of time series files(in either CSV, JSON, or Excel formats) along with a required *metadata.json* file that describes the time series set

- The *metadata.json* file indicates important attributes of the time series set such as it's name, description, contributors, etc

- The *metadata.json* file also includes a description of each time series in the time series set. This includes information such as the name of the time series, the description of the data format(such as time column name, units in the time series vector, etc), as well as the names of the time series files that contain the testing and training data.

Data Representation(Storage)



- MongoDB(document-based NoSQL database) is used as the storage mechanism for time series data, metadata, and forecast statistics
- Four main data entities: Time series Set, Time series, Dataset, and Forecast
- Data model designed for good locality of relevant information for fast lookup while preventing unnecessary reads

System Design: What did we learn?

1. A good software design involves careful planning and consideration of the various modules that will make up the application. In this case, the web application was designed with several modules, including the Flask API, Database Manager Module, and Data Processing Module, each with a specific role to play. Clearly defining requirement before engineering minimizes rewrites and makes working in a team a smooth process.

2. A good software design considers the user's needs and provides a user-friendly interface. In our case, the Flask API allows us to create well stylized dynamic web pages that make it easy for users to download time series, upload forecasts, and view metrics.

3. A good software design leverages existing technologies and tools to facilitate development and improve performance. For example, ChronoWave's use of popular modern technologies like MongoDB, Flask, and Pandas allowed us to quickly iterate on the web app from sprint to sprint.

Overall, software design is a critical aspect of software development that requires careful planning, consideration of the user's needs, and using existing technologies to improve performance and simplify development.

Integration and Testing

1. Integration testing: Once we completed the individual backend modules and Flask App. We tested whether or not they could communicate properly without error. When errors were found(usually due to one of the modules making assumptions about the output), we strengthened the interface definitions for module communication and modified to code base.
2. System testing: Once integration testing between components was done, we tested the whole system together to see if there were any real-world use cases that would cause any of the software systems to break(such as badly formatted input data). When discovered, we added additional validation and data sanitization to prevent errors.
3. Acceptance testing: Once system testing was done and core features were working, we tested the system over a longer period of time to see if we could find anything that would unexpectedly break(possible deadlock in long running program, time series deletions not propagating to the datastore, etc.) When found, we would fix the bugs.

What did we learn? Integrating and testing the system is an important step in the software development process to ensure that all the individual components work together seamlessly and that the application meets the user's requirements.

Working as a Team

1. From the get go, we specified what general area each team member was going to focus on and designed to software architecture with that in mind.
2. We used a weekly SCRUM system for project management. Every week we did design/code sprints and then once a week would talk about what tasks we completed during the sprint, as well as plan what tasks each member was going to work on for the next sprint. We used Trello to track our tasks.
3. In addition, we stayed in constant communication via discord so everyone was aware of possible immediate issues/changes that may have come up so that no one was ever in the dark. This ensured that we were able to pivot effectively in the event that unexpected changes came up in between sprint meetings.

What did we learn? It is important to have a clear understanding of the goals and objectives of the project, as well as the roles and responsibilities of each team member so that everyone is working towards the same goal and that there is no duplication of efforts. In addition, it is important to break down the project into smaller, more manageable tasks and assign them to specific team members. This can help ensure that each task is completed on time and to the required standard.

The Team

Front-End / Flask Back-End

Isaac

- Created the base file structure of the front end web page using html documents and a python-flask file.
- Made the appearance presentable and navigable.
- Implemented Navigation between pages of frontend
- Set up Github
- Transitioned upload and download page to use database manager
- Setup dynamic metadata rendering on download page using ajax

Geoffrey

- Added to the Upload page a way to insert a file and have it stored in a folder/(local hosted database): should be easily changed to adapt to the actual database and data format.
- Added to Download page a way to retrieve data from local hosted database (MongoDB)
- Worked on full transition to using database modules for frontend.
- Implemented performance metrics page routing and rendering with flask and Jquery/ajax.

Back-End/Pandas/Numpy

Cynthia

- Set up trello organization and updating information based on tem advances.
- Converting input files into correct format for database storage.
- Making error calculations with numpy.
- Making plots of calculation results.
- Converting output files into proper output.
- Assisting in front end adjustments.

Back-End/Database

Aleks

- Designed UML diagram for software and system architecture design
- Designed MongoDB NoSQL database schema and Chen Entity-Relationship diagram
- Architected and Implemented Database Manager application code for use in interfacing with the database management system
- Designed expected file format for uploaded time series sets by contributors