ChatGPT

Project Plan: Machine Learning for Cross-Current Prediction at IJmuiden Harbor

Introduction and Problem Context

IJmuiden is the entrance to the North Sea Canal connecting the Port of Amsterdam to the North Sea, and it experiences strong **cross-currents** that can hinder ship navigation [1]. Currently, a daily forecast of cross-currents is produced using a neural network model developed ~25 years ago [2]. This legacy model takes **tide** and **wind** as inputs and outputs the "experienced" cross-current (originally inferred from ship steerage angles) [2]. Over the decades, however, the harbor environment and ships have changed, and the old model increasingly **misses significant cross-current events** [3]. In 2017, a measurement campaign deployed Acoustic Doppler Current Profilers (**ADCPs**) in and near the channel to collect actual current velocity profiles [4]. These measurements, along with a 2D hydrodynamic model study, revealed that **freshwater discharge** (outflow from the canal to sea) is likely an important factor that was not included in previous models [5]. As a result, there is a clear opportunity to improve cross-current predictions by leveraging newly available data (tide, wind, **freshwater outflow**, etc.) and modern machine learning techniques.

The goal of this project is to develop a robust **machine learning (ML) solution in Python** to accurately predict cross-currents at IJmuiden Harbor. The following plan outlines a roadmap, data sources, modeling strategy, and implementation approach for an intern at VORtech to tackle this challenge.

Roadmap for Solution Development

To address the cross-current prediction problem, we propose a phased **roadmap** with clear steps and milestones:

- **Phase 1: Stakeholder Engagement & Problem Understanding** – Begin by reviewing existing documentation and meeting with domain experts. Speak with harbor pilots, the Port of Amsterdam, and the Hydro Meteo Centre (HMC) that produces forecasts [6] to gather insight into when/why the current model fails. If possible, take a field trip on a ship in the IJmuiden area to get a firsthand sense of the conditions [6]. This will ground the project in real-world context and stakeholder needs.

- **Phase 2: Data Collection and Exploration** – Identify and obtain all relevant data sources (detailed in the next section). This includes historical **tide levels**, **wind measurements**, **freshwater discharge rates**, and any available **current velocity** measurements for the IJmuiden channel. Consolidate these datasets and perform Exploratory Data Analysis (EDA): visualize time series, check correlations (e.g. how wind and tide correlate with cross-current), and note data quality issues or gaps.

- **Phase 3: Data Preprocessing & Feature Engineering** – Clean and preprocess the data for modeling. Align timestamps from different sources (likely using a common time basis, e.g. hourly or

10-minute intervals). Handle missing values or outliers (e.g. interpolate short gaps or remove faulty sensor readings). Engineer informative features, for example:

- Lagged values of tide, wind, and discharge (to capture delayed effects on currents).
- Derived features like wind stress or direction components (e.g. splitting wind into along-channel vs cross-channel components).
- Tidal phase indicators (e.g. whether tide is flooding or ebbing).
- Possibly categorical flags for events (storms, heavy rain periods) if relevant.

- **Phase 4: Baseline Modeling** – Reproduce or retrain the existing neural network model as a baseline. Using the collected data, input the same features (tide and wind) to a simple neural network or regression model to see how it performs on recent data. This helps establish a performance baseline and ensures the data processing pipeline is correct. Compare baseline predictions to actual observed currents (from ADCP data or pilot reports) to quantify current accuracy.

- **Phase 5: Enhanced ML Model Development** – Incrementally build more advanced models to improve prediction accuracy. Incorporate the new **freshwater discharge** input and other engineered features into the model. Start with relatively simple models for quick iteration (e.g. a multivariate linear regression or a tree-based model) and then progress to more complex architectures (discussed later). Evaluate multiple model types (neural network, gradient-boosted trees, etc.) to identify what works best for this problem.

- **Phase 6: Model Training and Validation** – Train models on historical data and validate their performance on held-out periods (for example, using the most recent year as a test set to simulate forecasting future conditions). Use appropriate **evaluation metrics** (detailed below) to assess accuracy, and ensure the model captures the important cross-current events that the old model missed. Iterate on feature engineering and model hyperparameters based on validation results.

- **Phase 7: Incorporate Domain Feedback** – Present the improved model results to stakeholders (HMC forecasters, harbor masters, pilots) in review sessions. Gather feedback on whether the predictions align with their operational experience. For example, if the model still struggles in certain wind scenarios or extreme discharge events, use that insight to refine the model (add new features or adjust training data). This iterative loop ensures the solution is not just statistically sound but also meets practical needs.

- **Phase 8: Deployment Strategy** – Plan how to integrate the new model into the existing forecasting workflow. Determine whether to **replace or run in parallel** with the current neural network initially. If the new model is an entirely different implementation, ensure it can be deployed in the operational environment (which might have constraints on runtime or require a specific format for outputs). Develop a retraining schedule or adaptation mechanism so the model can be periodically updated as new data comes in (ensuring it remains accurate over time).

- **Phase 9: Monitoring and Maintenance** – After deployment, set up monitoring to track the model's performance in production. Compare its forecasts to subsequent actual measurements of cross-current and log any significant errors. Implement alerts or fallbacks for cases of model uncertainty.

Over time, gather more data (especially for any new extreme events) and feed that back into model improvements. This phase ensures the solution remains reliable and continues to improve with use.

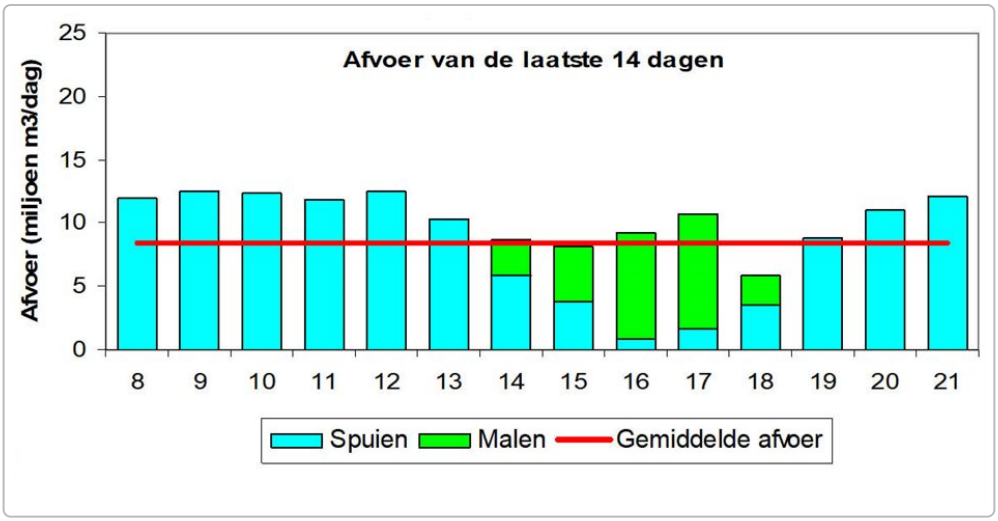
Each phase should be documented and include clear deliverables (e.g. data report, baseline model code, evaluation report, etc.). Regular meetings with the supervisor and relevant stakeholders will keep the project on track. Next, we detail the critical data sources needed and how to obtain them.

Data Sources and Acquisition

Accurate **data** is the backbone of this project. We need three main categories of data: (1) current measurements in the channel, (2) freshwater discharge through the canal, and (3) meteorological and tidal conditions. Below is a guide to relevant data sources and how to access them:

- **Current Velocity Measurements (ADCP data)** – The primary target variable is the cross-current velocity in the IJmuiden harbor channel. A permanent measuring pole with an **ADCP** is located near the channel, recording water current profiles (speeds at various depths) [7]. These **historical current measurements** are collected by Rijkswaterstaat (RWS). RWS operates an open data portal called **Waterinfo** that provides historical observations for Dutch waterways [8]. Through Waterinfo, one can request time series of current speeds (stroomsnelheid) at specific locations. For IJmuiden, the relevant station might be an offshore buoy or pier near the harbor entrance (sometimes referred to as the IJmuiden buoy or "IJ-geul" measurement). Using the Waterinfo download tool or API, the intern can query current data by specifying the parameter (current speed/direction) and location, then export it (e.g. as CSV via email or via a REST call) [9]. The Waterinfo portal provides an English interface and even a Python example on their GitHub for how to retrieve data programmatically [10]. If the 2017 measurement campaign data is not directly on Waterinfo, VORtech or RWS contacts may provide those datasets separately (possibly as Excel/CSV or via the **Hydro Meteo Centre** archives). In summary, start by checking Waterinfo for any station at IJmuiden with current measurements; if needed, follow up with RWS's data desk for access to the 2017 ADCP campaign data.

- **Freshwater Discharge Data (Canal Outflow)** – The **freshwater outflow** from the North Sea Canal into the North Sea is a crucial input that was missing in the old model [11]. This flow is controlled by a system of sluices and pumps at IJmuiden that regulate canal water levels by releasing water to the sea [12]. Rijkswaterstaat monitors and records these discharges (both **"spui"** i.e. gravity-driven sluice flow during low tide, and **"maal"** i.e. pumped flow) on a continuous basis. The data is typically in cubic meters per second (or sometimes aggregated in millions of cubic meters per day). These records can likely be obtained via RWS Waterinfo as well, by selecting the **"afvoer"** (discharge) parameter for the IJmuiden sluice complex. Historical discharge data might also be found in RWS reports or data portals – for instance, RWS Waterinfo-Extra provides reports that include **IJmuiden discharge graphs** [13]. According to a Deltares report, the average long-term discharge through IJmuiden is on the order of ~95 m^3/s [14], but it varies significantly with weather and season. We will want the **time series of discharge (flow rate)** on at least an hourly or daily resolution. Likely, there are specific gauges or calculations for total flow through the locks (combining all sluices and pumps). If an API query is difficult, the intern might retrieve this from Waterinfo's "Download historical data" by choosing parameter "Afvoer – debiet Oppervlaktewater (m3/s)" and selecting the IJmuiden location. If needed, contact the Waterinfo Servicedesk for assistance in locating the correct dataset.



Freshwater discharge through the IJmuiden sluices and pumping station (blue = sluicing, green = pumping) over a 14-day period, with the red line indicating the long-term average. This example (May 2001) illustrates the variable outflow that can range roughly 50–200+ m^3/s on daily timescales [13] [14] . Such variability in freshwater discharge is believed to significantly influence cross-currents in the canal.

- **Tide and Water Level Data** – Tidal water level oscillations at IJmuiden are a major driver of currents. The **IJmuiden tidal gauge** (managed by RWS or the Port) provides continuous records of water levels relative to NAP (Amsterdam Ordnance Datum). This data can be accessed via RWS Waterinfo as well, by selecting the parameter "waterhoogte" (water level) at IJmuiden (there may be stations like "IJmuiden Buitenhaven" or "IJmuiden Zuidpier"). Additionally, predicted tide curves can be obtained from the Dutch Hydrographic Service or through KNMI's public data if needed for forecasting. For modeling, we should use observed water levels as an input feature. If the water level data is at high frequency (e.g. 10-min intervals), consider downsampling or extracting features like tidal stage (phase in the tidal cycle). RWS Waterinfo allows exporting historical water level data over long periods (with some limits on request size) [15] . Ensure that the time frame of water level data aligns with the current measurements and other inputs. If only shorter-term records are available on the public portal (it sometimes limits to last 28 days for public view [16]), use the "Download historical data" function or the API to get longer time series.

- **Wind Data (Meteorological)** – Wind is another key input, as strong winds can induce surface currents and interact with tides to produce cross-currents. The Royal Netherlands Meteorological Institute (KNMI) operates weather stations across the country, including near the coast. There is a KNMI station **IJmuiden (station 225)** which has historical records of wind speed and direction (and other weather variables) going back decades [17] . KNMI's **open data platform** provides hourly and 10-minute observational data. For convenience, one can download daily data files that include wind speed (in m/s) and wind direction for IJmuiden [17] , or use the KNMI API for finer resolution. The KNMI Data Platform documentation confirms that automatic weather stations report every 10 minutes on parameters like wind speed and direction [18] . The intern should retrieve at least the wind time series (at 10m height) covering the same period as the current data. If multiple nearby stations exist (e.g. IJmuiden Lighthouse, Schiphol airport, Wijk aan Zee), the closest and most relevant should be used (likely the IJmuiden station or a dedicated wind mast at the harbor). **Feature note:** It may be useful to break wind into **u/v components** (eastward, northward components) or

relative to the channel orientation (along vs cross-channel wind), since a wind blowing directly into the canal vs across it can have different effects.

- **Additional Data (if available)** – Other environmental factors might include **water density/salinity stratification** (fresh vs salt water interface), which can be relevant for cross-currents (freshwater tends to flow out at the surface, saltwater intrudes at depth) [19] [20]. Direct measurements of salinity in the canal or any density stratification data could help if available (e.g. from the 2017 campaign or earlier surveys). Another possible input is **rainfall** in the catchment feeding the canal (which would influence discharge), though this is indirectly captured by the discharge data itself. If the intern identifies that heavy precipitation or scheduled water management actions cause spikes in outflow, including a precipitation forecast or an upstream water level variable might be beneficial. For a first model iteration, tide, wind, and discharge are the core inputs as specified.

All the above data should be gathered and stored in a structured format (CSV files, NetCDF, or a time series database). It's wise to build a **data fetching script** in Python (using `requests` for APIs or downloading files) to automate the retrieval. This script can also perform initial merging of datasets by timestamp. After acquisition, proceed to clean and integrate the data as outlined in the next section.

Reusing vs. Replacing the Existing Neural Network

A critical decision is whether to **reuse (retrain)** the current neural network model in production or **replace** it with a new model. Given that the existing model is 25 years old and was developed with technology of that era, it likely has a very simple architecture and is implemented in an older software environment. We assume we have the freedom to either retrain it (if its code is accessible and can accept new weights) or build a new model from scratch.

Assessment of the Current Model: The current model has known limitations – it was trained on an old dataset (ship steerage angles as a proxy for currents) and **omits important inputs** like freshwater outflow [5]. It also appears to be missing modern improvements in ML. If the original model's architecture is extremely outdated (e.g. a single hidden layer neural net) and the code interface is not easily extensible, trying to simply "update" it may be more effort than benefit. Moreover, incorporating new inputs (discharge) would anyway require retraining from scratch on new data.

Retrain as Baseline: For initial comparison, it might be useful to **retrain the old network** using the current data and the same inputs (tide & wind) to see how well it performs relative to its original calibration. This could be done by building an equivalent model (for example, if it was a 3-layer perceptron with X neurons, replicate that in a modern framework like TensorFlow/PyTorch) and training it on the historical record of tide, wind -> cross-current (from ADCP measurements). If this "legacy architecture" retraining yields improved performance over the 25-year-old version (likely, because it will adjust to current data), that model could serve as a baseline going forward. It may be straightforward to then extend that model by adding the third input (discharge) and training again.

Building a New Model: Ultimately, we anticipate that a **new model** will be needed to achieve the best performance. Modern ML frameworks allow us to experiment with advanced architectures (e.g. recurrent networks for sequence modeling) that were not available decades ago. Replacing the model gives us full flexibility in design and training. The new model can be deployed either by **integrating into the existing**

forecast system (replacing the old model's component) or running alongside it and outputting predictions in the required format. Since VORtech specializes in operational forecasting software, integrating a Python ML model (perhaps by exporting it or rewriting it in a compatible form) should be feasible. In making the switch, we should ensure: - The new model is **at least as fast and reliable** as the old one (inference time likely negligible for a small neural net, but ensure the pipeline is optimized). - It produces **probabilistic or at least well-calibrated outputs** if needed (the old model likely gave a single deterministic prediction; the new one could potentially give confidence intervals or flags for uncertainty). - There is a **fallback** plan: for a transitional period, the old and new models could run in parallel, and any large divergences can be examined. This is a safeguard to build trust in the new system.

Conclusion on Strategy: We propose retraining the existing model only as a short-term baseline exercise, while focusing on developing a superior replacement model. Given that the old network "misses substantial cross-current events with regularity" [3], a fresh approach is warranted. We will design the new model to directly address those missed events (e.g. by including discharge and using techniques to better predict extreme values). The old model's 25-year legacy is valuable domain knowledge, but modern data and algorithms offer a clear path to improvement.

Model Development Strategy

Developing a high-quality predictive model will involve careful data handling, selecting an appropriate ML architecture, and rigorous evaluation. This section outlines the strategy for model development, including data preprocessing, feature engineering, model selection, and training/validation methodology.

Data Preprocessing & Feature Engineering

Before modeling, ensure the data is well-prepared:

- **Synchronization and Resampling:** All input time series (tide, wind, discharge) and the output series (cross-current) need to share a common timeframe and interval. Decide on a time step for prediction – for example, forecasting hourly average cross-current or 10-minute instantaneous values. Resample or aggregate the data as needed (e.g. compute hourly means from higher-frequency measurements, if high frequency noise is not essential for the model). Make sure to align the timestamps (taking care of time zones – use UTC or local time consistently, as KNMI data is usually in local time + timezone info, and water data might be in CET).

- **Cleaning:** Scan for missing data or obvious errors. For instance, anemometer data might have gaps or spikes during maintenance; water level sensors might drop out during storms. Use interpolation for short gaps or flag and fill with model predictions if needed. Outliers should be investigated – e.g. an extremely high discharge value could correspond to an emergency pumping event and might be real (so include it), whereas a spike due to sensor error should be removed or capped.

- **Scaling/Normalization:** It often helps to normalize features (especially for neural networks). Compute mean and standard deviation for continuous features like wind speed, discharge, tide range, etc., and scale them (or use min-max scaling) so that inputs are of similar magnitude. This prevents one input (e.g. discharge in m^3/s which might be tens of units) from dominating another (wind in m/s). Save these scaling parameters for use in model inference later.

- **Feature Engineering:** Create additional features that could enhance model learning:

- **Lagged features:** Because the effect of wind or discharge on cross-current might not be instantaneous (e.g. a surge of outflow might create a current a bit later), include lagged versions of inputs. For example, include discharge at T-1 hour, T-2 hours, etc., up to a reasonable lag (maybe up to 6-12 hours if looking for effects of prior tides). Similarly, wind an hour ago or a moving average of wind over past 3 hours could be relevant if currents take time to respond to sustained wind.
- **Tide phase indicators:** Instead of just raw water level, consider breaking it into **tide phase** – e.g. time since last high tide or low tide, or a binary feature "rising tide vs falling tide" (since cross-currents might be strongest during certain phases of the tide). Another approach is to include the **tidal current prediction** from a simple physical model (like if we know at flood tide water comes in, at ebb goes out – but since our model is data-driven, it might learn this itself).
- **Wind direction decomposition:** Convert wind direction and speed into **X and Y components** (eastward and northward wind). Then, determine the orientation of the harbor/channel (roughly East-West for the North Sea Canal). We can then derive **along-channel wind** (which would either push water into or out of the canal) and **cross-channel wind** (which might cause lateral currents at the entrance). These directional features can be important because a 10 m/s north wind vs west wind have different effects.
- **Interaction terms:** If suspected, create features that are interactions of inputs, e.g. wind * tide (perhaps when a low tide coincides with strong outbound wind, the outflow current is especially strong). Tree-based models can capture interactions implicitly, but linear models and even some neural nets might benefit from explicit interaction features.

- **Seasonal or time-of-day signals:** It might help to include the **month or season** (to account for seasonal differences, like generally wetter winter meaning more outflow) or a **binary storm flag** for known storm event periods. Also, daily or tidal periodicity might be captured by using cyclic encodings (sine/cosine of time variables) if needed.

- **Target variable processing:** Define the exact target for prediction. It could be the **maximum cross-current** in the channel at a given time, or the current at a specific location/depth. Since the ADCP gives a vertical profile, we might need to reduce that to one representative value. One approach is to use the surface current or depth-averaged current at the location most relevant to ships (maybe mid-depth current in the navigation channel). Alternatively, if multiple locations are measured, we might focus on the one that historically caused issues. For simplicity, assume we predict the **surface cross-current velocity** at the harbor entrance channel. We may also smooth the target slightly (e.g. a 10-min running average) to avoid very noisy target that's hard to predict. If the target is directional (current direction matters), we might break that into two components as well; but likely the main concern is the magnitude of cross-current perpendicular to the channel, which can be taken as a single value (with sign indicating direction maybe).

By the end of this stage, we should have a comprehensive feature matrix and target vector, ready for input into modeling. Document the preprocessing steps in code (so it can be reused for new data) and save the processed dataset for training/validation.

Candidate Model Architectures

We will experiment with a few **model architectures** to identify the best approach for this time-series prediction problem. It's wise to start simple and increase complexity as justified:

- **Baseline Linear/MLR Model:** Start with a multiple linear regression or another simple regression (like a linear model with regularization). This will set a performance floor and sometimes reveals if basic linear combinations of features can explain a lot of variance. Given the relationships (tide, wind, discharge -> current) might have nonlinearities and thresholds, the linear model likely won't suffice, but it's a quick check.

- **Traditional Neural Network (MLP):** Implement a basic multi-layer perceptron (feed-forward neural network) with perhaps one or two hidden layers. This could be similar to the old model's structure but trained on the new data. It will capture some nonlinear relationships. We can treat each time step's features independently (i.e. not explicitly sequence modeling, just feed the feature vector at time *t* to predict current at time *t*). This ignores temporal autocorrelation except as captured by lag features that we include. An MLP can serve as a baseline non-linear model.

- **Recurrent Neural Network (LSTM/GRU):** Because currents are a time series and have sequential dependencies (the current now is related to currents a short time ago), a **Long Short-Term Memory (LSTM)** network or a **GRU (Gated Recurrent Unit)** network is a promising choice. LSTMs are designed to handle sequences and can maintain an internal state to capture long-term dependencies. We can set up an LSTM model that takes as input a sequence of recent timesteps (e.g. the last 6 hours of data at 10-min resolution, which would be 36 timesteps) and outputs a prediction for the next time step (or next few timesteps). LSTMs have been successfully used in oceanographic time-series predictions; for example, researchers have applied LSTM networks to ocean current forecasting and found they can learn the temporal patterns effectively [21]. We will configure an LSTM with a reasonable number of units (e.g. 50-100) and train it on sliding windows of our time series data. The advantage is the model may learn the dynamic behavior of the system (like how a combination of rising tide and strong wind over several hours builds up the current). However, LSTMs require more data and tuning to avoid overfitting, so we will need to monitor that.

- **Tree-Based Ensemble (e.g. XGBoost)**: As an alternative to neural networks, **eXtreme Gradient Boosting (XGBoost)** or similar ensemble methods (Random Forests, LightGBM) can be effective for tabular data with many features. XGBoost is known for its strong performance in many structured data problems, including hydrological forecasting [22]. We can train XGBoost to predict the current at time *t* given features at time *t* (including some lagged features to provide context). It will automatically handle nonlinear interactions to some extent. One study on river flow forecasting found XGBoost outperformed other ML models in accuracy for streamflow predictions [22], highlighting its potential here. The downside is that XGBoost, as a purely regression approach, won't inherently capture sequential dependencies beyond the lagged inputs we feed it, and it might not extrapolate as well for situations outside the training range (whereas a physics-based or recurrent model might handle sequences more flexibly). Nonetheless, it's worth including in the model suite for comparison.

- **Hybrid or Advanced Models:** Depending on results, we could try combinations, such as an LSTM that feeds into a small fully-connected network (to combine sequence output with static features), or

even an **attention-based sequence model (Transformer)** if we have a lot of data (though with limited data a full Transformer may be overkill). Another possibility is a **multi-output model** that predicts not just the next time step but a short-term forecast (say currents 1 hour, 2 hours ahead, etc.); this could be useful if pilots need forecasts slightly in advance. However, initially we focus on nowcasting or very short lead predictions.

- **Physics-guided Model:** A different approach is to incorporate some physics knowledge into the ML model. For instance, we could constrain the model or features such that at slack tide (no water level gradient) and no wind, the predicted current should be near zero. Or we ensure the model learns the periodic tidal component by including sinusoidal basis functions for tide. These considerations might improve physical realism. There's also an option to use the output of a physical model (like the 2D model that exists) as an input feature to the ML model – effectively doing model output statistics or bias correction with ML.

We will likely try several models in parallel. To manage this, consider using a framework like **scikit-learn** for simpler models (linear regression, random forest), and **PyTorch or TensorFlow** for neural network models (PyTorch is very user-friendly for prototyping, and TensorFlow/Keras is also fine especially if we want quick scaffolding of an MLP or LSTM). We can use scikit-learn's XGBoost wrapper or the `xgboost` Python package for gradient boosting models.

Crucially, whichever model performs best on validation data (in terms of error metrics and capturing big events) will be chosen for further refinement and eventual deployment. It's possible that a committee or **ensemble of models** could outperform any single model – for example, we might average the predictions of an LSTM and an XGBoost model to balance each's strengths. This could be explored if time permits.

Training, Validation and Evaluation Metrics

Model training will be supervised learning on historical data. We need to set up a training/validation scheme that properly evaluates how the model would perform on unseen data (simulating future predictions):

- **Train-Test Split:** Use the chronologically earlier portion of the dataset for training and the later portion for testing. For example, if we have data from 2017–2024, train on 2017–2023 and test on 2024. This ensures the test simulates forecasting new conditions. We might also carve out a validation set (e.g. use 2017–2022 train, 2023 validation, 2024 test) to tune hyperparameters without touching the final test. It's important **not to do random shuffle split** because of time correlations – always split by date to avoid data leakage from the future.

- **Cross-Validation (Time-series CV):** If sufficient data is available, we can perform a rolling-origin cross-validation. For instance, train on 2017–2019, validate on 2020; then train 2017–2020, validate on 2021, etc. This would give multiple validation evaluations to average. Tools like `sklearn.model_selection.TimeSeriesSplit` can help implement this. This approach helps in assessing stability of model performance over different periods (which might include different weather regimes).

- **Evaluation Metrics:** We will evaluate model predictions vs actual observed cross-current with several metrics:

- **Mean Absolute Error (MAE)** and **Root Mean Squared Error (RMSE)**: standard regression error metrics to gauge average prediction error. RMSE penalizes large errors more, which is useful if we care about big misses.
- **R^2 (Coefficient of Determination)**: to understand what fraction of variance in the currents is explained by the model.
- **Correlation coefficient** between predicted and observed time series, to see if the model is getting the timing and general pattern correct.
- **Event detection metrics:** Since one concern is the model missing "substantial cross-current events," we should define what constitutes an event (e.g. cross-current exceeding a certain threshold that is considered hazardous). Then we can compute metrics like **Hit Rate, False Alarm Rate, and Miss Rate** for those events. For example, how often does the model correctly predict a cross-current > 1 m/s event? We might use a **precision/recall** or **F1-score** for event detection. This will directly address the problem of missed events in the old model.

- **Visual Inspection:** Plot time series of predictions vs actual for critical periods (such as storms or spring tides) to qualitatively check performance. This often reveals if the model consistently lags or overshoots in certain conditions.

- **Training Procedure:** For neural networks, use a portion of training data as a mini validation to do **early stopping** (to avoid overfitting – stop training when validation error starts increasing). Use optimization algorithms like Adam (for neural nets) with an appropriate learning rate. For XGBoost, use built-in cross-validation to find the optimal number of trees (rounds) to prevent overfit. We will likely need to tune hyperparameters:

- Neural net: number of layers/neurons, learning rate, batch size, etc.
- LSTM: sequence length (window size), number of layers, units, dropout regularization.

- XGBoost: tree depth, learning rate, number of estimators, regularization terms. This tuning can be done via manual search guided by experience, or using a library like `Optuna` or scikit-learn's GridSearchCV (though grid search on time series must be done carefully to not leak data).

- **Handling Overfitting:** Keep model complexity in check relative to data volume. If we find that a complex model (like a deep LSTM) is overfitting (training error much lower than test error), consider regularization (dropout, L2 weight decay) or gather more training data (perhaps augment with synthetic data – see next section). Simpler models like a shallow neural net or XGBoost with regularization may generalize better with limited data.

- **Model Selection:** Based on validation results and stakeholder input, select the model that best balances overall accuracy and the capture of critical events. It may not necessarily be the one with lowest RMSE; for example, if Model A has slightly higher RMSE but predicts extreme surges correctly while Model B smooths them out, Model A might be preferable for this application.

All evaluation results should be documented. We will prepare a **technical report on model performance** including tables of metrics and plots of predictions vs actual, to justify the choice of final model.

Utilizing Numerical 3D Model Data (Synthetic Data Augmentation)

One of the research questions is whether a **3D hydrodynamic model** might be needed to solve this problem, or if its outputs can be used to train an AI model offline [23]. Incorporating a numerical model can be beneficial in two ways:

1. **Direct 3D Modeling for Simulation:** A full 3D flow model (like **Delft3D** or a Finite-Element coastal model) could in principle simulate the currents in and around IJmuiden given inputs like tide, wind, and freshwater inflow. Such a model could capture complex physics (e.g. stratification of fresh and salt water, 3D circulation patterns) that a data-driven model might struggle with. The drawback is that running a 3D model operationally is computationally heavy and requires careful calibration; hence it might not be feasible to run it in real-time for daily forecasts, which is why the AI approach is appealing as a faster surrogate.

2. **Synthetic Training Data from 3D Model:** A promising approach is to run the 3D model **offline** (not in real-time, but as a one-time or occasional effort) to generate **synthetic data** under a wide range of conditions, and then use those simulations to train or enhance the ML model [24]. For example, we could create a series of Delft3D simulations varying inputs: different constant wind directions and speeds, different tidal ranges (spring vs neap), and different freshwater discharges. Each simulation would output cross-current values. This would effectively create a large dataset of scenarios, including extreme ones that might not be well represented in historical data (e.g. an extremely high discharge combined with a gale from an unusual direction). The AI model can be trained on this expanded dataset to recognize patterns and outcomes it hasn't seen in the real record. Essentially, the 3D model acts as a data generator to fill gaps in experience.

If we take this route, we should consider: - Does a calibrated 3D or **2D/3D Delft3D model for the North Sea Canal** already exist (maybe from Rijkswaterstaat or Deltares)? The internship description mentions a 2D model was tried but it lacked freshwater flow [5], suggesting that a 3D model would be the next step. If a model is available, leveraging it is easier than building one from scratch within the internship timeframe. - Using 3D model outputs requires interpreting them into the same terms as our ML model output. For instance, Delft3D might give current velocity at various points in the harbor – we'd need to extract the "cross-current at the critical location" from those results. - The consistency between synthetic and real data: ensure the ML model trained on synthetic data does not learn artifacts that don't generalize to real world. One way is to use synthetic data to pre-train the model, then fine-tune with real data (transfer learning). Or use synthetic data to augment training: mix it with real samples. - Evaluate if synthetic-trained model actually improves performance on real test cases. If the physics model is accurate, it can help the AI to learn fundamental relationships (e.g. how a sudden large freshwater release will always create a seaward current regardless of wind, except maybe under extreme incoming tide).

We will keep the option of synthetic data in our plan. Initially, focus on real data to get a working model. In parallel (or if time allows), set up a simple Delft3D or similar model for the area: - Possibly use Delft3D-FM (Flexible Mesh) if a grid for the canal exists, or utilize an existing model from previous studies. Boundary conditions would be the sea level at IJmuiden (tidal boundary), inflow at the canal end (from river/ Markermeer), and wind forcing. - Validate that the model reproduces known current measurements in some test conditions (e.g. a known storm event in 2017). - Run a few scenario simulations to generate training data.

Finally, even if we develop a pure ML model, the **3D model can serve as a benchmark**. If the ML approach fails to capture some phenomena, the project might conclude that a 3D model (or hybrid solution) is needed for ultimate accuracy [23]. In that case, our efforts would still be valuable in highlighting what the AI can and cannot do, and perhaps the AI could still be used as a post-processor to a 3D model (speeding it up or adjusting biases).

Tools, Libraries, and Platform

We will utilize open-source tools in Python for all aspects of this project. Below is a selection of recommended libraries and their purposes:

- **Python Data Analysis**:
- **pandas** – For loading CSV data, merging tables, cleaning time series, and general data manipulation. Pandas provides convenient data structures (DataFrames) to handle time-indexed data and will be heavily used in preprocessing and EDA.
- **numpy** – The fundamental package for numerical computations. Used under the hood by pandas and others, and directly for array manipulations, mathematical functions, etc.

- **xarray** – Useful for multi-dimensional labeled data, especially if dealing with **NetCDF files** or gridded model output. If we use any 3D model data or KNMI data in NetCDF format, xarray can simplify working with these by treating data with time, depth, location coordinates in a high-level way.

- **Data Access and Formats**:

- **requests** – For making HTTP requests to APIs (like the Waterinfo or KNMI APIs to download data). We might use this to automate data retrieval.
- **netCDF4** or **h5py** – If data comes in NetCDF or HDF5 format (common for model outputs or some KNMI datasets), these libraries allow reading those files. xarray actually can wrap around netCDF4 to directly open NetCDF files easily (`xarray.open_dataset()`).

- **pyOWM** or **knmi-api** – While not necessarily needed (as we can use direct HTTP requests), there are sometimes specific client libraries for certain data (e.g. PyOWM for weather, but KNMI has its own data portal which might not have a dedicated Python client yet). Likely not needed if we handle manually.

- **Machine Learning / Modeling**:

- **scikit-learn** – A rich library of machine learning algorithms for regression, classification, and preprocessing. We will use scikit-learn for things like linear regression, random forest, support vector regression (if tried), and utility functions (train_test_split, scaling, metrics calculations like MAE/RMSE). It's great for baseline models and quick experiments.
- **XGBoost** – The XGBoost library (installable via pip) for gradient boosted decision trees. Alternatively, scikit-learn's API can call XGBoost if installed. We'll use this for the tree ensemble modeling.
- **PyTorch** – A deep learning framework that is very useful for building custom neural networks (including LSTMs). PyTorch is highly Pythonic and good for research environments. We can define our neural net architecture and use PyTorch's autograd and optimization to train models. It also has good support for time-series via torch's RNN modules.

- **TensorFlow / Keras** – Another popular deep learning framework. Using TensorFlow 2.x with Keras API might be simpler for quick prototyping of standard architectures (like an LSTM network, which can be made with a few lines using Keras `Sequential`). Either PyTorch or TensorFlow can be used; it might come down to the intern's familiarity. Both have ample documentation and community support.
- **pandas.DataFrame.rolling / statsmodels** – For any time-series-specific transforms or even trying classical models (ARIMA, etc.). The statsmodels library has ARIMA and other time-series models which could be used as a baseline (e.g. an ARIMAX which includes exogenous variables like wind, tide could be tried). This isn't machine learning per se, but could be an interesting comparison.

- **Visualization**:

- **matplotlib** and **seaborn** – For plotting data and results. Line charts of time series, scatter plots of predicted vs actual, histograms of error, etc., will be needed for analysis and for presentations to stakeholders.
- **plotly** or **bokeh** – If interactive plots are useful (e.g. to explore data), these could be used, but matplotlib likely suffices.

- **Environment & Tools**:

- **Jupyter Notebooks** – Ideal for exploratory data analysis and for documenting the process. The intern can use JupyterLab or classic notebooks to interactively explore data and build initial models, combining code with notes.
- **Git** – Version control to track code changes. It's good practice to use a Git repository (possibly on a platform like GitHub or GitLab) to manage the project code, especially if multiple iterations and branching experiments are done.
- **Conda or venv** – To manage Python packages and environments. Ensures that the required libraries (especially heavy ones like TensorFlow) are installed in a stable environment.
- **MLflow or DVC (optional)** – Tools for experiment tracking and data versioning. If the project becomes complex with many experiments, MLflow can log parameters and results for each run, making it easier to compare models. DVC (Data Version Control) can version large data files if needed. These are advanced tools and might be optional depending on scope.

These tools will support the entire workflow from data handling to model training and evaluation. All are open-source and widely used, which means plenty of documentation and examples are available online. For instance, PyTorch and TensorFlow have tutorials on time series forecasting; scikit-learn has examples of regression and hyperparameter tuning; and xarray has documentation on working with multi-dimensional data (which could be useful if analyzing e.g. a 3D model's output).

Experimentation and Prototyping Structure

Starting the project in an organized manner will save time in the long run. We suggest the intern sets up a clear structure for coding experiments and maintains discipline in recording results. Here are some guidelines for the experimentation process:

- **Project Repository Layout:** Organize the project directory with clear folders, for example:

```
crosscurrent-project/
 ├── data/              # raw data files or a link to data storage
 ├── notebooks/         # Jupyter notebooks for exploration and reporting
 ├── src/               # Python source scripts or modules for data
processing and modeling
 ├── models/            # saved model files (if any) and results
 └── docs/              # documentation, reports, figures
```

  Raw data can be kept in `data/` (possibly with subfolders per source). Notebooks are great for EDA and trying out ideas, but core functionality (like a function to fetch data or a class to define a model) can be moved into re-usable scripts under `src/` as they mature. This structure will help in both development and eventual handover/deployment.

- **Jupyter Notebook Workflow:** Begin with an **exploratory notebook**. For example, `notebooks/01_data_exploration.ipynb` can be used to load the datasets, plot them, and do initial analysis (like correlation between wind and current, or visualize an event). Use a combination of text and visuals to document insights (these notes can later contribute to the report or stakeholder presentations). Subsequent notebooks might be `02_baseline_model.ipynb` where the intern implements and evaluates the baseline models, and `03_advanced_models.ipynb` for the LSTM/XGBoost and so on. Each notebook should have a clear goal and not become too unwieldy; better to have multiple focused notebooks than one giant one.

- **Modular Code for Reuse:** As soon as certain code is tested (for instance, a block that downloads data or a function that computes features from raw input), move it to a script in `src/` (e.g. `data_utils.py` or `features.py`). This way, notebooks remain clean and high-level, while the actual logic lives in well-organized, version-controlled Python modules. For example, you might create a `prepare_data()` function that reads raw files and returns a processed DataFrame, and a `train_model()` function that given training data and parameters, trains a model and returns it. This makes it easier to run experiments programmatically or even integrate into a pipeline later.

- **Configuration and Parameters:** It's useful to have a single place where key parameters are defined (like the train/test split date, or model hyperparameters for a run). This could be a YAML/JSON config file or just a Python dictionary in a notebook. The intern can then easily adjust, say, the number of LSTM layers or the lag features considered, and rerun experiments. Keeping track of which configuration produced which result is important (this could be done manually in a log or via an experiment tracking tool).

- **Experiment Logging:** Encourage the intern to **log results** of each experiment, either in a spreadsheet or a simple markdown file. Note the date, the model version, parameters, and performance metrics on validation. This will help in comparing models. If using MLflow, this can be automated, but even manually is fine for a single user project.

- **Version Control & Backups:** Commit changes to Git regularly with meaningful messages (e.g. "Added lag features up to 6h and retrained XGBoost – improved RMSE by 10%"). This not only provides a safety net (you can revert if something breaks) but also creates a history of progress. Also, since data files can be large, avoid putting big raw data in Git; instead, keep a copy on a drive and perhaps just version control a small sample for testing code. Use Git LFS or DVC if needed for larger files.

- **Testing:** It's not a software product, but some basic tests can be invaluable. For instance, after writing a data loading function, test it on a small known input to ensure it correctly handles time zones and missing data. If writing a custom loss function or evaluation metric, test that it computes expected values. This can prevent hours of debugging later due to a silly data alignment bug or similar.

- **Iterative Approach:** Start with a small prototype and gradually expand:

- First, get one day's data or one week's data flowing through the pipeline to make sure you understand the formats.
- Then try a tiny model on that (even one training example) just to verify the training loop works.
- Scale up to the full dataset and more complex models once the pipeline is verified.
- At each stage, produce some output (like a plot of model fit or errors) to verify things make sense (e.g. check that the model's predicted current roughly follows the tide cycles, etc., else something might be off in data alignment).

By following these practices, the intern will create a **reproducible and extensible codebase**. This is important because after the internship, colleagues may continue the work – well-structured code and clear notebooks will enable them to understand what was done and build on it. It also means the intern can more easily write the final report, since results and figures will be easy to regenerate from the notebooks.

Implementation Strategy and Iteration with Stakeholders

Implementing the improved cross-current prediction model in a real operational setting requires close collaboration with stakeholders and iterative refinement. The strategy to roll out the solution is as follows:

- **Stakeholder Feedback Loops:** Throughout development, involve the end-users of the forecast (e.g. harbor pilots, HMC forecasters). Early in the project (after Phase 2 or 3), share preliminary findings – for example, "We found that during **XYZ** storm, the old model under-predicted the cross-current by 50%. Our new model with freshwater flow data captures this better." Get qualitative feedback: does this align with their experience? Are there specific **cases of concern** they want the model to handle (like a notorious storm or an unusual high discharge event)? By engaging them, the intern can ensure the model's improvements are noticed and trusted. It also helps in defining success criteria (stakeholders might say, "We need to know when cross-current exceeds 0.7 m/s at location X with at least 1-hour notice").

- **Demonstration and Training:** Once a prototype model is ready that clearly outperforms the old one, set up a demonstration. For example, create a small report or presentation for the Port Authority or VORtech team showing side-by-side comparisons of old vs new predictions on historical events. Highlight how the new model uses additional data (perhaps visualize how including discharge data helped to predict an event that was missed before). This builds confidence and also serves as a checkpoint to decide if the model is ready for deployment. If stakeholders see only marginal improvement, they might suggest further iteration or combining approaches.

- **Iterative Refinement:** Expect a few cycles of iteration. After the initial model, you might discover, for instance, that **under certain wind directions the model still errs**. Investigate those cases specifically: maybe it needs a wind-direction-specific term or there's an external factor (like wave action or longshore currents) not in the model. Each iteration could involve adding a feature, adjusting the model, or even making minor process changes (e.g. calibrating the model output bias if it consistently over/underestimates). Maintain a **change log** of what is tried in each iteration and the effect on performance.

- **Integration into Production:** VORtech's forecasting systems may have a specific integration path. The model might need to run automatically every day at a set time to produce the forecast. For integration:

- Decide whether to embed the Python model in the existing system or to run it as a separate service. If the current system is not Python-based, one might use a REST API or a microservice to host the model. Alternatively, **export the trained model** (for example, if using scikit-learn or XGBoost, you can save the model object with pickle; if using TensorFlow, save it as a TensorFlow SavedModel).
- Ensure that data feeds (tide, wind, expected discharge for the day) are available in real-time to the model. If the model needs forecast input (like forecast wind or planned water release schedule), integrate those sources too. Possibly the daily forecast could use *predicted* wind for the next hours to predict cross-current (this would extend the model from nowcast to short-term forecast).

- Build any necessary **wrapper code** to fetch the latest input data, run the model prediction, and output results in the required format (maybe a CSV, or a database entry, or a visualization). This should be tested thoroughly in a staging environment.

- **Validation and Handover:** After deploying the model, closely monitor initial outcomes. Compare the model's predictions each day with what actually happened (using new ADCP data as it comes in or pilot reports). Keep the stakeholders in the loop; for example, after one month of operation, compile statistics: "The new model had an MAE of X, versus the old model's MAE of Y, and it successfully predicted N out of M events." This will formally validate the improvement. Document any issues that arose and how they were solved. Finally, prepare a **handover document** or session for the team that will maintain this system after the internship, including:

- How the model was developed and can be retrained (with code references).
- How to adjust parameters or retrain when new data is available.

- Any maintenance tasks (e.g. updating the discharge data source if the format changes, etc.).

- **Continuous Improvement:** Recommend a plan for continuous improvement. For instance, as more data (especially new extreme events) are observed, set a schedule to retrain the model every year or

after major events. Encourage stakeholders to keep providing feedback (maybe a channel where pilots can report if a forecast seemed off, which can then be investigated). Because machine learning models can degrade if conditions change (concept drift), having a mechanism to update the model or at least check its accuracy over time is important for longevity.

In summary, the implementation strategy emphasizes **transparency with stakeholders** and **iterative refinement**. By the end of the project, the aim is that stakeholders have confidence in the new ML-based forecast, and a clear path exists for integrating and maintaining this improved model in the operational forecasting system of IJmuiden harbor.

Conclusion and Next Steps

By following this project plan, the intern will systematically tackle the cross-current prediction challenge: from understanding the problem context and gathering high-quality data, through building and evaluating a series of machine learning models, to deploying an improved solution in collaboration with stakeholders. The roadmap ensures that along the way, knowledge from both data and domain experts is leveraged – incorporating new information like freshwater discharge and validating results with those who navigate the harbor daily.

This project not only aims to enhance forecast accuracy (especially for extreme events) but also serves as a case study in applying modern **AI techniques to maritime forecasting**, potentially informing similar efforts at other ports. As next steps, the intern can begin with data acquisition and exploratory analysis (Phase 1 and 2). Early identification of data issues or gaps should be communicated so that support (from RWS data services or others) can be obtained in time. With data in hand, the modeling phases will follow, and regular updates should be provided to VORtech mentors to ensure alignment with expectations.

Ultimately, success will be measured by a **safer and more reliable entry for ships into IJmuiden**, thanks to better predictions of cross-currents. By the conclusion of the internship, we expect to deliver a working ML model (or model suite), a robust codebase for ongoing use, and a detailed report (this document evolving into it) with findings and recommendations. With VORtech's guidance and the intern's efforts, IJmuiden's cross-current forecasting can be brought up-to-date, combining the best of domain experience and data-driven modeling.

References and Data Sources: The report has cited key sources of information and data, including Rijkswaterstaat data portals for water levels, flows, and currents [8] [12], KNMI for wind and weather data [18] [17], and the context of the IJmuiden challenge provided by VORtech [2] [4]. Additionally, relevant studies in ocean current prediction with ML were referenced to support the chosen methods [21] [22]. These resources will be useful throughout the project for both implementation and justification of approach. The intern should familiarize themselves with these and any further documentation provided.

---

[1] [2] [3] [4] [5] [6] [7] [11] [23] [24] Internship_Crosscurrent_IJmuiden.pdf
file://file-PKFwpa9reCMxx9uLn2mLpr

[8] [9] [10] [15] [16] Waterdata - Data Rijkswaterstaat
https://rijkswaterstaatdata.nl/waterdata/

12 Kaart waterverdeling bij droogte
https://www.rijkswaterstaat.nl/water/waterbeheer/droogte-en-watertekort/verdeling-water-bij-droogte/kaart-waterverdeling-bij-droogte

13 19 20 Zoutmetingen Noordzeekanaal - Waterinfo Extra
https://waterinfo-extra.rws.nl/projecten/@206881/zoutmetingen-noordzeekanaal/

14 publications.deltares.nl
https://publications.deltares.nl/1200339_007.pdf

17 KNMI - Daggegevens van het weer in Nederland
https://www.knmi.nl/nederland-nu/klimatologie/daggegevens

18 Documentation Actuele10mindataKNMIstations | Open Data | KNMI Data Platform
https://english.knmidata.nl/open-data/actuele10mindataknmistations

21 jaem.isikun.edu.tr
https://jaem.isikun.edu.tr/web/images/articles/vol.13.no.1/34.pdf

22 Weekly streamflow forecasting of Rhine river based on machine learning approaches | Natural Hazards
https://link.springer.com/article/10.1007/s11069-024-06962-x