ChatGPT

CNC Tool Wear Anomaly Detection System – Implementation Specification

Dataset Overview (Exploratory Analysis)

- **Data Composition:** The provided `tool_wear_dataset.csv` contains 2,000 samples of multi-sensor measurements from CNC machining operations [1]. Each row represents a machining cycle with **extracted features** from multiple sensors (Cutting Force, Vibration, Acoustic Emission) along with tool wear readings and a wear label. The dataset columns include:
- **Cutting Force features (CF_Feature_1 … CF_Feature_5):** Numeric features derived from a cutting force sensor.
- **Vibration features (Vib_Feature_1 … Vib_Feature_5):** Numeric features derived from vibration sensor signals.
- **Acoustic Emission features (AE_Feature_1 … AE_Feature_5):** Numeric features from acoustic emission sensor signals.
- **Tool Wear Measure (VB_mm):** Continuous value of tool flank wear in millimeters for that cycle.

- **Wear_Class:** Categorical label of the tool's health/state for that cycle (possible values: **Healthy**, **Moderate**, **Worn** [2]). "Healthy" indicates little to no wear, "Worn" indicates high wear (close to failure), and "Moderate" is intermediate wear.

- **Data Types:** All sensor feature columns and `VB_mm` are floating-point numeric values. `Wear_Class` is a string/category indicating the wear condition [2]. There are **no missing values** in the dataset (clean dataset).

- **Label/Anomaly Indicators:** The presence of `Wear_Class` provides labels for supervision ("Healthy" vs "Worn", etc.), but for anomaly detection we treat these labels as **optional** ground truth. In an unsupervised scenario, we assume **"Healthy" cycles as normal data** and **"Worn" cycles as anomalies** that should be detected. The "Moderate" class represents partially worn tools, which may be considered **early anomalies or borderline cases**. There is no explicit anomaly flag column, but `Wear_Class` can serve to validate our model's detections.

- **Key Features and Signal Insights:** The dataset captures multiple sensor modalities: Cutting force, vibration, and acoustic emission – all known to reflect tool condition changes [1]. We expect as the tool wears out, some combination of these sensor features will shift or exhibit out-of-normal patterns (e.g. increased vibration or acoustic emission bursts when worn). The actual flank wear measurement `VB_mm` increases from ~0 mm (new tool) up to ~0.37 mm (worn tool). The wear labels correlate with `VB_mm` ranges (e.g. Healthy $\leq$0.1 mm, Worn $\geq$0.25 mm). Individual sensor features may not have a simple linear correlation with wear, but collectively they should differentiate healthy vs worn states. This multi-sensor feature set is suitable for anomaly detection since **normal operating cycles (healthy tool)** form one cluster of patterns, while **worn/faulty tool cycles** should deviate and appear as outliers in feature space.

- **Supports Anomaly Detection:** Yes. The data supports an unsupervised or semi-supervised anomaly detection approach. We have many examples of normal operation (Healthy cycles) and some examples of tool degradation/fault (Worn cycles). An unsupervised model can be trained on the entire dataset (or only on healthy data for a semi-supervised approach) to learn the normal pattern and then flag deviations. The existence of labeled wear states can help evaluate the model (e.g. check if "Worn" cycles are flagged as anomalies), but the model itself can be developed without using labels (unsupervised), which aligns with a real-world scenario where faults may not be labeled in advance.

Data Ingestion and Preprocessing

Goal: Quickly load and prepare the data for modeling, ensuring the pipeline is simple and reproducible.

- **Data Loading:** Use Python (pandas) to read the CSV file. This will be integrated into the Streamlit app or a preprocessing script. Confirm the data schema (17 columns: 15 sensor features, 1 wear measurement, 1 class label) on load. For example, using `pandas.read_csv` to ingest the data into a DataFrame.

- **Feature Selection:** For unsupervised anomaly detection, use the 15 sensor features (CF_, *Vib_*, AE_*) as the primary input features. The* `VB_mm` *(wear) and* `Wear_Class` *will not** be used by the model during training (since we prefer unsupervised), but can be retained for reference/ visualization and for validating results. We ensure that `Wear_Class` is not leaked into the unsupervised model.

- **Preprocessing Steps:** Minimal preprocessing to keep things quick:

- **Scaling:** Apply feature scaling (e.g., standardization or min-max normalization) to sensor features. This ensures that features on different scales (force vs vibration vs acoustic) are comparable for the anomaly model. *Implementation:* use `scikit-learn`'s `StandardScaler` or `MinMaxScaler` to fit on the training data (likely just healthy samples if semi-supervised) and transform features for modeling.
- **Handling Outliers/Missing:** The dataset appears clean (no missing values). No complex outlier removal is planned initially (the anomaly detection model will handle outliers inherently). Simply verify no obvious data errors.

- **Training/Validation Split:** Since this is unsupervised, we may not strictly split the data; however, for evaluation we could set aside a small portion of data containing a mix of conditions to simulate real-time input. For the initial POC, the model can be trained on all **Healthy** data (semi-supervised approach) or on the full dataset unsupervised with an assumed contamination fraction. **Plan:** Likely train on healthy cycles only (using `Wear_Class` to filter during training phase), so the model learns the normal pattern. The remaining data (with moderate/worn cases included) can then be used to test and identify anomalies. This simulates deploying the model on new data where anomalies may occur.

- **Data Pipeline:** Organize the ingestion and preprocessing into a small module or functions that can be reused:

- **load_data()** – reads the CSV and returns DataFrame.
- **preprocess_data(df)** – applies scaling (fit on train portion), returns scaled feature matrix and possibly the scaler object for inverse-transform if needed.
- These will be called at app startup or prior to model training. Keep it simple – no complex ETL since data is local CSV.

Model Selection and Training (Initial Baseline)

Goal: Develop a baseline anomaly detection model that can quickly flag unusual tool behavior, favoring unsupervised methods for speed and simplicity.

- **Model Choice:** Use an **unsupervised anomaly detection algorithm** from Python's ecosystem. Given the need for quick implementation and no heavy dependencies, a good choice is `IsolationForest` from scikit-learn, or alternatively `LocalOutlierFactor` or a one-class SVM. **Isolation Forest** is preferred for the baseline because:
- It handles high-dimensional data reasonably well and requires minimal parameter tuning.
- It gives an anomaly score for each sample and can work with a specified expected anomaly fraction.
- It's fast to train on 2000 samples (near real-time training).
- **Training Procedure:**
- If following a **semi-supervised** approach: filter the training set to only include "Healthy" data points (normal condition). Train the Isolation Forest on these normal samples to learn their distribution. Set the contamination parameter to a small value (e.g., `contamination=0.05` or similar) to make the model initially assume a small fraction of anomalies in training (or use `contamination='auto'` if available to let it infer). The model will then produce an **anomaly score** for new samples, labeling those that fall outside normal range.
- If fully **unsupervised on all data:** train on the full dataset but specify a contamination fraction roughly equal to the proportion of expected anomalies (for example, if we treat all "Worn" as anomalies, that's about ~28% of data; we might set contamination ~0.3). However, this is less accurate since moderate cases are also included. The semi-supervised (train on healthy only) approach is likely more precise for this context, but we mention both for completeness.
- **Model Output:** Once trained, the model can output:
- **Anomaly score or label:** For each sample/cycle, the model can give a binary label (anomaly or normal) and/or a score (e.g., Isolation Forest's `decision_function` or `score_samples`). We will consider any sample with an anomaly score beyond a chosen threshold as "flagged anomaly." In practice, IsolationForest can directly predict `.predict()` where -1 = anomaly.
- We will initially rely on the model's default threshold (which is based on the contamination fraction). This keeps it simple. The threshold can be later tuned (perhaps using the known labels for evaluation).
- **Integration in App:** The training can be done at app startup or offline:
- Given the small data size, we can **train the model at runtime** when the app starts (this avoids shipping a pre-trained model and ensures the latest data is used). This is quick (milliseconds to train on 1000-2000 points).
- Alternatively, we could train offline and load a saved model (using pickle or joblib), but since training is fast, doing it on-the-fly adds flexibility (e.g., re-train easily after receiving new feedback).

- For clarity and speed of delivery, we proceed with on-the-fly training inside the app (with a clear function encapsulating it, e.g., `train_model(data)`).

- **Initial Baseline Expectations:** The model should flag most of the **Worn** cycles (high wear) as anomalies, possibly some of the **Moderate** ones as well if they differ sufficiently from healthy patterns. The exact performance will be reviewed, but since this is unsupervised, we focus on getting a working detection pipeline rather than perfect accuracy on first pass. We will use the known `Wear_Class` labels after detection *only for validation or demonstrating the effectiveness* (e.g., highlight if flagged anomalies indeed correspond to "Worn" in the dataset, without the model having seen those labels).

- **Iteration and Tuning:** The spec emphasizes quick deployment, so the initial training will use default parameters. However, we will structure the code such that improvements can be made:

- e.g., Parameterize the contamination rate or the features used so we can adjust if needed.
- Possibly experiment with a very simple alternative model (like a PCA-based anomaly score or one-class SVM) if IsolationForest doesn't perform well, but only if time permits. The baseline will likely suffice.

Front-End Features (User Interface)

Goal: Develop a Streamlit web application that visualizes sensor data over time and allows user interaction to confirm anomalies.

Main UI Components:

- **Time-Series Chart of Sensor Readings:**
  A core part of the UI is a line chart showing sensor readings across cycles (x-axis as cycle index or time progression, y-axis as sensor feature value). We will likely plot the **tool wear progression or a representative sensor signal over cycles**:
- One simple approach: plot the actual wear measure `VB_mm` over cycle index, since it trends upward as the tool wears. This is intuitive for users to see normal vs worn regions. We can overlay anomaly markers on this plot (since anomalies correlate with high wear regions).
- Additionally or alternatively, plot one of the key sensor features (or an aggregated metric) over time. Plotting all 15 features at once would be cluttered; instead, we might include a dropdown for the user to select which sensor feature to visualize. By default, we can show `VB_mm` or perhaps an average vibration/force metric to give a sense of how signals change.
- **Anomaly Highlighting:** Anomalies detected by the model will be highlighted on the chart. For example, flagged cycles can be marked with a red dot or a different color on the time-series. In Streamlit, we might use Altair or Plotly to enable coloring points by anomaly vs normal. The chart will clearly indicate where the model believes an anomaly occurred in the sequence of cycles. This helps the user quickly identify the suspect regions.

- The chart should have a legend or annotation explaining the highlight (e.g., red = flagged anomaly). We'll ensure the chart is large enough and possibly zoomable (if using Plotly) for usability.

- **Anomaly Details and Feedback Form:**
  Below the chart, we will provide a list or table of the specific cycles that were flagged as anomalies by the model, with a way for the user to give feedback on each:

- List the **Cycle index or ID** (implicitly, the row number or an index if provided) for each anomaly, along with key info like the sensor readings or wear value at that point. This gives context to the user about what was happening in that cycle (we might show `Wear_Class` label here if we want to reveal whether it was actually worn or not, but that might bias the user feedback. In a real scenario, the user might only know if it was a true fault after inspection).
- For each flagged anomaly, present a **confirmation control** – for example, a checkbox or radio button labeled “Mark as true fault” vs “false alarm.” The user can review the anomaly and indicate whether the model was correct.
    - Simpler UI: a checkbox list where each anomaly has a checkbox if the user confirms it was a real issue. If unchecked, it’s considered a false alarm. Or use two radio buttons (True Anomaly / Not Anomaly) for clarity.
    - We might also include a comments text box for each anomaly in case the user wants to note something, but this may be overkill for POC.

- A **submit button** to save feedback: At the bottom of the list, a button “Submit Feedback” which, when clicked, will collect all the user inputs (which anomalies were marked true/false) and then store this feedback (and possibly update the UI to acknowledge it). In Streamlit, this could trigger a callback or simply run on button press event to gather states.

- **User Experience Considerations:**

- Keep the UI clean and focused: one page app with the chart on top and feedback list below.
- Add a brief explanation at the top of the app: e.g., title “CNC Tool Wear Anomaly Detection” and a subtitle/description instructing the user to inspect the chart and provide feedback on anomalies. This guides a first-time user.
- Ensure the app is interactive but not cluttered: for POC, we won’t implement multi-page navigation; everything is on one page for simplicity.
- The chart and anomaly detection results should update whenever the model is (re)run. If we allow dynamic model updates (not initially required), the UI should refresh accordingly.
- (Optional nice-to-have) Possibly allow the user to select which sensor’s time-series to view. For example, a dropdown with options: “View wear over time”, “View vibration signal feature over time”, etc. This can help in analysis, but by default we’ll show the most relevant one (wear or an aggregate).

Backend Architecture (Data and Feedback Management)

Goal: Outline how the system will manage data, model, and user feedback behind the scenes in the Streamlit app.

- **Application Structure:** The Streamlit app will essentially serve as both front-end and back-end in one Python script, but we will structure it logically:
- On startup, the app will **load and preprocess data** (using the functions described earlier). This gives us `data_df` (the original data) and `features` (scaled features for modeling).
- Then, **model training/inference** is performed: train the Isolation Forest model on normal data and get anomaly predictions for each cycle in the dataset. The results (e.g., an array of anomaly flags or scores) will be stored.

- These results are then used to populate the UI components (e.g., the chart and the anomalies list).

- **Data Storage:** Since this is a lightweight POC:

- The sensor data will reside in memory as a Pandas DataFrame (sufficient for 2000 samples, very light).
- No external database is required for the sensor data; we rely on the CSV file as the source of truth, loaded at start.

- **Feedback Management:** For user feedback on anomalies, we need to capture and persist the inputs:

- In-memory approach: We can use Streamlit's session state or global variables to store a dictionary or DataFrame of feedback. For example, `feedback = {cycle_index: user_label, ...}` where `user_label` might be True/False for true anomaly. This would reset if the app restarts, but it's simplest for initial testing.
- Persisting feedback to file: To allow analysis of feedback or retraining later, we will also **append the feedback to a local file** (e.g., a CSV or JSON in the app directory). This can be as simple as writing a CSV row with cycle id and user verdict each time feedback is submitted, or writing the whole feedback dictionary at once. This ensures feedback isn't lost between sessions. Given the POC nature, writing to a small CSV (e.g., `anomaly_feedback.csv`) is sufficient (no need for a full database server).
- **Data Model for Feedback:** Each feedback entry should include at least:
    - Cycle identifier (could be an index or if there is a timestamp or cycle number, use that for clarity).
    - The model's anomaly prediction (for reference).
    - User confirmation flag (true anomaly or false alarm).
    - (Optionally) timestamp of feedback or user name if multi-user (not needed now).

- The app will have logic such that on clicking "Submit Feedback," it updates an internal structure and also appends to the file. Streamlit can show a success message like "Feedback saved" to confirm to the user.

- **Retraining with Feedback:** Although not implemented initially, the architecture will prepare for it:

- We will modularize model training so that we could easily call it again incorporating new knowledge. For example, if the user marks some anomalies as true faults, we might in the future use that information to label those samples and include them (or exclude them) in retraining. Initially, we will **not retrain automatically** (to keep things simple and avoid altering the model live during a session). The feedback is just stored for later use.
- We will note in code/comments where retraining could be triggered (e.g., a button "Retrain model with feedback" could be added in future that uses the feedback file to separate confirmed normals and anomalies and retrains a supervised or semi-supervised model).

- For now, the backend's main job is to capture feedback reliably and keep the system state (data, model, predictions) consistent.

- **State Management:** Ensure that expensive operations (like training model, generating full chart data) are not needlessly repeated on each Streamlit re-run:

- Use `st.cache_data` or `st.cache_resource` decorators for data loading and perhaps model training if appropriate, so that interactions (like checking a box) don't reload everything each time.

- The model and processed data can be cached after first computation. This way, the UI interactions (which cause the script to rerun) won't rebuild the model from scratch unless data changes. This will keep the app snappy.

- **Security/Privacy:** Not a major concern for a local POC. The data is static and no external connections. Just ensure file writes (for feedback) are handled properly. No sensitive info involved aside from normal operational data.

Deployment Method

Goal: Deploy the app quickly and simply, focusing on local or easily accessible deployment without complex infrastructure.

- **Environment & Dependencies:** The solution will be contained in a Python script (or a small set of scripts) using:
- Streamlit for the web interface.
- Pandas for data handling.
- Scikit-learn for the anomaly detection model.
- (Possibly Altair or Plotly for better charting, which can be installed via pip as needed.)

- These are all pip-installable and lightweight. We will provide a `requirements.txt` including these libraries for easy setup.

- **Local Deployment:** For a fast POC, the app can run on a local machine. The engineer should be able to start the app with:

  ```
  streamlit run app.py
  ```

  This will launch the web UI on a local port. The system is lightweight enough to run on a standard laptop.

- **Sharing Options:** If Jasmin or the team needs to share the app with others quickly:

- The app could be deployed to **Streamlit Cloud** or similar free hosting by just pushing the code (no special config needed since no cloud data sources). This would give a public URL for demonstration. This is quick and avoids infrastructure overhead.
- Alternatively, package the app in a **Docker container** for portability. A simple Dockerfile can base on a Python image, install requirements, copy the app code, and specify the entrypoint to run Streamlit. This container can be run on any machine or server if needed. (This is optional if local run suffices, but noted for completeness.)

- No Azure or external cloud integrations are needed; the dataset is local and the app is self-contained.

- **File Management:** Ensure that the CSV dataset file (`tool_wear_dataset.csv`) is included or accessible in the deployment environment (if using Docker, copy it into the image or mount it). The feedback file (if used) should be stored in a writeable location. For Streamlit Cloud, the app directory is fine (feedback will reset if the app restarts fresh, which is acceptable for POC).

- **Testing Deployment:** The engineer should test the end-to-end pipeline locally with a couple of sample feedback submissions to ensure everything works. Since speed is key, extensive unit tests are not required, but basic manual testing (does the app start, does it detect anomalies, does it save feedback) is expected.

Optional Future Enhancements

(These are not required for the initial POC, but design is flexible to allow them later.)

- **Automated Retraining/Continuous Learning:** Use the accumulated user feedback to improve the model. For instance, after a number of feedback entries, retrain the anomaly detector:
- Could implement a **semi-supervised approach** where the model initially trained on healthy data is periodically retrained by incorporating confirmed normal points and excluding or separately treating confirmed anomalies. Over time, this adapts the model to reduce false alarms.

- Alternatively, train a supervised classification model (using the feedback as labels) if enough data accumulates, to directly predict anomaly vs normal. This could complement the unsupervised model.

- **Advanced Modeling:** For better accuracy, consider more complex models:

- **Autoencoder neural network** to learn normal patterns of all sensor features and reconstruct, with high reconstruction error indicating anomalies. This would require a bit more setup (TensorFlow/PyTorch) but could improve detection of subtle anomalies. It can be integrated later if needed.
- **Ensemble of models:** combine multiple detectors (e.g., one per sensor type or different algorithms) to improve robustness.

- **Time-series analysis:** If data were streaming, incorporate temporal context (e.g., an LSTM-based detector). Currently each cycle is independent, but future work could consider sequence anomalies (like an abrupt change in sensor patterns).

- **UI/UX Improvements:**

- Add the ability to **filter or zoom** into certain cycle ranges on the chart for closer inspection.
- Include **multiple charts** or an interactive dashboard: e.g., small multiples for each sensor type (force, vibration, AE) so user can see how each signal behaves when an anomaly is flagged.
- **Threshold slider:** Allow user to adjust the anomaly sensitivity threshold from the UI and immediately see how the flagged anomalies change. This could help domain experts fine-tune the detector without code changes.

- **Display metrics:** If the dataset has labels, show basic metrics like how many anomalies were found vs how many actual worn instances, once the user provides feedback or if ground truth is known. A confusion matrix or summary could be displayed to evaluate performance (for development purposes).

- **Persistence and Scaling:**

- If the tool were to be used by multiple users or over a long term, integrate a small database to store feedback and possibly the data. For example, an SQLite DB or a simple REST backend could hold feedback so it doesn't rely on a single file.

- Allow the app to ingest **new data in real-time** (e.g., streaming sensor readings). For now we use a static CSV, but future enhancements might connect to a machine data source or at least periodically refresh with new CSV data.
- Deploy on an internal server or cloud when moving beyond POC, with proper authentication if needed (since feedback might be considered sensitive in a real setting).

- **Integration with Tool Maintenance Workflow:** In a production scenario, this system could integrate with maintenance systems (e.g., trigger an alert or log entry when a true anomaly is confirmed). For now, we note this as a future integration point.

Conclusion: This specification provides a clear blueprint for a minimal yet functional anomaly detection system for CNC tool wear. The focus is on fast development and deployment: using straightforward data processing, a baseline unsupervised model [2], and a lightweight Streamlit app for visualization and feedback. By following the above requirements, a competent engineer can implement the system within a short timeframe, while ensuring the design is extensible for future improvements in accuracy and usability.

---

[1] Multi-Sensor CNC Tool Wear Dataset | Kaggle
https://www.kaggle.com/datasets/ziya07/multi-sensor-cnc-tool-wear-dataset

[2] Multi-Sensor CNC Tool Wear Dataset - Kaggle
https://www.kaggle.com/datasets/ziya07/multi-sensor-cnc-tool-wear-dataset/data