**2. Verification of Implemented Steps (What is Incorrect or Partially Implemented )**

**A. Data Preprocessing**

* **Date/Time Parsing:**
  + **Issue:** The notebook crashes with an OutOfBoundsDatetime error on pd.to\_datetime('nov.18'). The developer correctly identified this and wrote a parse\_date\_safely function to handle this format. HoIver, the fix is in a later cell (In[40]) that doesn't seem to have been run, while the failing cell (In[20]) that applies the conversion remains unfixed. The KeyboardInterrupt in cell In[40] suggests the time-series loading is also broken.
  + **Documentation Requirement:** Load all data successfully.
  + **Verdict:** **Incorrectly Implemented.** The fix exists but is not applied correctly, leaving the notebook non-functional.
* **Time-Series Data Handling:**
  + **Issue:** The load\_time\_series\_data function concatenates *all 25 time-series files into one giant DataFrame*. This is highly inefficient and conceptually wrong.
  + **Documentation Requirement:** "For each batch, extract the segment from the corresponding time-series file (matching on product code and batch ID)". This implies a lookup, not a single massive table.
  + **Verdict:** **Incorrectly Implemented.** The current approach makes it very difficult and slow to isolate the time-series window for a specific batch, which is likely why the cell was interrupted.
* **Handling Missing Data & Downtime:**
  + **Issue:** The notebook uses a blanket df\_merged.dropna(), which removes any row with at least one missing value. The documentation suggests more nuanced handling (imputation for lab data). Furthermore, there is **no code** to handle machine downtime in the time-series data (e.g., masking or removing periods where tbl\_speed=0).
  + **Documentation Requirement:** "Impute... rare lab gaps; remove or mask downtime in sensor streams."
  + **Verdict:** **Partially and Incorrectly Implemented.** The missing value handling is too aggressive, and the critical step of cleaning downtime from sensor data is completely missing.

**B. Model Implementation**

* **LSTM Forecasting Model:**
  + **Issue:** The notebook defines an "Aggregated Features LSTM" that tries to predict a single value (e.g., total\_waste) for a future *batch* based on a sequence of *past batches*.
  + **Documentation Requirement:** The goal is to build a multivariate time-series model that forecasts future *sensor values* (e.g., main\_comp, tbl\_fill) within a single batch run, 30-60 minutes ahead.
  + **Verdict:** **Incorrectly Implemented.** The "aggregated" LSTM is a misinterpretation of the task. The model should not be predicting batch-level outcomes from sequences of other batches but rather high-frequency sensor data within one batch.
* **Classification Model (Major Flaw):**
  + **Issue:** The classification model is trained directly on the final, aggregated features from df\_merged. These features (e.g., main\_CompForce mean, total\_waste) are summary statistics calculated *after the batch is complete*.
  + **Documentation Requirement:** The classifier's inputs must be derived from the **forecasted sensor values**. "The classifier sees a feature vector representing the *expected near-future process state* combined with known inputs." This is to simulate a real-time prediction pipeline.
  + **Verdict:** **Incorrectly Implemented (Data Leakage).** Training on actual final outcomes instead of predicted ones means the model is learning from perfect future information. Its performance will be artificially inflated and useless in a real-world scenario where it only has access to forecasts.

**C. Validation Strategy**

* **Issue:** The notebook uses a standard, random train\_test\_split.
* **Documentation Requirement:** The documentation is very specific, recommending a **time-based split** (e.g., train on 2018-2019, test on 2021) and/or a **sub-family (product code) split** to ensure the model generalizes to new data and new product types.
* **Verdict:** **Incorrectly Implemented.** A random split leaks information across time and product types, leading to an over-optimistic evaluation of model performance.

**3. Missing Implementations (What is Not Implemented )**

1. **Correct Time-Series Alignment:** The logic to extract the specific time-series window for each batch from its corresponding file is missing.
2. **Sensor Selection:** The notebook uses a hardcoded list of main\_sensors. The documentation suggests a data-driven approach (correlation analysis, model-based importance) to select the most predictive sensors.
3. **Downtime Handling:** The logic to detect and mask/remove idle periods from the time-series data (tbl\_speed=0 or produced=0) is missing.
4. **Correct Classification Pipeline:** The entire pipeline of Forecast -> Extract Features from Forecast -> Classify is not implemented.
5. **Robust Validation:** Time-based and product-based cross-validation schemes are not implemented.
6. **Normalization of Count-Based Metrics:** The documentation states, "apply the batch-size normalization factors to any count-based or time-dependent metrics (such as waste, produced counts)". This step is missing from the preprocessing.

**4. Actionable & How to Fix**

Here is how to correct the notebook to align with the documentation.

**Fix 1: Data Loading and Preprocessing**

**A. Correct the Date Parsing Integration:**  
Move the parse\_date\_safely function definition to an earlier cell (e.g., right after imports) and ensure the cell that processes the start column uses it.

**B. Implement Correct Time-Series Loading and Alignment:**  
Do not concat all files. Instead, load them into a dictionary and create a function to extract data for a specific batch.

**Fix 2: Implement the Correct Two-Stage Model Pipeline**

The entire modeling approach needs to be restructured.

**A. Stage 1: Train the Forecasting Model**  
Use the correctly loaded time-series data to train the LSTM. The sequences should be generated from within each batch run.

**B. Stage 2: Train the Classification Model on Forecasts**  
This is the most critical correction.

1. Split Ir df\_merged data into a train and test set using a **time-based split**.
2. For **each batch in Ir training set**:  
   a. Get its time-series data.  
   b. Pick a point in time (e.g., 60 minutes before the end of the run).  
   c. Feed the preceding sensor data into Ir **trained LSTM model** to get a 30-60 minute forecast.  
   d. From this **forecast**, calculate summary features (e.g., predicted\_mean\_main\_comp, predicted\_max\_srel).
3. Create a new training DataFrame where the features are these *predicted* statistics.
4. Train Ir XGBoost classifier on this new DataFrame against the true defect label.

**Fix 3: Implement Correct Validation**

**Summary Table: Notebook vs. Documentation**

|  |  |  |  |
| --- | --- | --- | --- |
| Feature/Step | Documentation Requirement | Notebook Implementation | Status & Verdict |
| **Time-Series Loading** | Extract segment for each batch from its file. | Concatenates all 25 files into one. | **Incorrect** |
| **Date Parsing** | Handle "nov.18" format. | Fails execution due to this error; fix exists but is not properly integrated. | **Incorrect** |
| **Downtime Handling** | Mask or remove idle periods (tbl\_speed=0). | Not implemented. | **Missing** |
| **Forecasting Model** | Forecast future sensor values within a batch. | Attempts to forecast batch-level outcomes from sequences of other batches. | **Incorrect** |
| **Classification Model** | Train on features derived from **forecasts**. | Trains on actual, final batch statistics (data leakage). | **Incorrect** |
| **Validation Strategy** | Use time-based or product-based splits. | Uses a standard random train\_test\_split. | **Incorrect** |

RL

**Reinforcement Learning System for Pharmaceutical Process Optimization: A Detailed Implementation Guide**

**1. Conceptual Framework: From Prediction to Control**

The goal of Phase 2 is to move from passive prediction (forecasting defects) to active control (optimizing parameters to prevent defects). The RL system will act as an intelligent agent that observes the state of the manufacturing process and recommends optimal actions (e.g., adjusting press speed or compression force) to maximize batch quality and efficiency.

I will formalize this problem as a **Markov Decision Process (MDP)**, which is the foundation of RL. The key is to define the four core components: State, Action, Reward, and Transition.

**The Integrated Pipeline:**  
The RL agent will leverage the models built in Phase 1:

1. **Live Sensor Data** is collected.
2. The **LSTM Forecasting Model (Phase 1)** predicts the future trajectory of key sensors.
3. The current sensor data and the LSTM's forecast are combined to form the **State** for the RL agent.
4. The **RL Agent** observes this state and suggests an **Action**.
5. This action is checked against a **Safety Layer** to ensure it's within operational limits.
6. The **XGBoost/GB Classification Model (Phase 1)** is used within the reward function to estimate the quality impact of the agent's actions.

**2. Defining the RL Environment (The "PharmaGym")**

The first and most critical step is to create a custom environment, compliant with the OpenAI Gym (now Gymnasium) API. This PharmaGym will simulate the manufacturing process using Ir historical data. It will need reset() and step(action) methods.

The state is what the agent "sees" at each time step. It must be comprehensive enough to make informed decisions. Based on Ir data and models, the state vector at each step t should include:

* **Current Process State (from the time-series data):**
  + The most recent values of the key process sensors selected in Phase 1: main\_comp, tbl\_speed, SREL, ejection, stiffness, produced, waste.
  + Short-term trends (e.g., rolling average or slope over the last 5-10 minutes) for these sensors to capture momentum.
* **Forecasted Process State (from the Phase 1 LSTM model):**
  + This is the "look-ahead" capability. For each key forecasted sensor, generate summary statistics from the 30-minute forecast horizon.
    - forecast\_main\_comp\_mean, forecast\_main\_comp\_std, forecast\_main\_comp\_max
    - forecast\_tbl\_speed\_mean, forecast\_tbl\_speed\_std
    - forecast\_SREL\_mean, forecast\_SREL\_max
    - ...and so on for other key forecasted sensors.
* **Static/Contextual Features (from Process.csv and Laboratory.csv):**
  + These features do not change during a batch but are crucial for context.
  + product\_code (one-hot encoded).
  + Key raw material properties for the current batch: api\_content, api\_water, lactose\_water, smcc\_water.
  + normalization\_factor for the batch.
* **Time-based Features:**
  + time\_since\_batch\_start: A counter indicating how far into the production run I are.

This composite state vector provides the agent with a snapshot of "what is happening now," "what is likely to happen next," and "what is the context of this batch."

The action space defines the decisions the agent can make. Based on the project blueprint, this is a MultiDiscrete action space, allowing simultaneous adjustments to multiple parameters.

* **Action Vector:** (speed\_adjustment, compression\_force\_adjustment, sampling\_rate\_adjustment)
* **Discrete Levels for each Action:**
  1. **Speed Adjustment:** 3 discrete actions.
     + 0: Decrease speed by a fixed step (e.g., 5%).
     + 1: Maintain current speed.
     + 2: Increase speed by a fixed step (e.g., 5%).
  2. **Compression Force Adjustment:** 7 discrete actions.
     + 0: Decrease force by a large step (e.g., -1.5 kN).
     + 1, 2: Decrease force by smaller steps.
     + 3: Maintain current force.
     + 4, 5: Increase force by smaller steps.
     + 6: Increase force by a large step (e.g., +1.5 kN).
  3. **Sampling Rate Adjustment:** 2 discrete actions.
     + 0: Nominal sampling rate.
     + 1: Increased sampling rate (triggering a "yellow flag" for operator attention).

This MultiDiscrete([3, 7, 2]) space gives the agent a flexible but constrained set of controls.

The reward function guides the agent's learning. It must align with the business objectives of maximizing quality and efficiency while minimizing costs and risk. Use the multi-objective function from the project blueprint:

reward = (100 \* (1 - predicted\_defect\_prob)) - (5 \* test\_cost) + (50 \* reg\_compliance) - (10 \* downtime)

To implement this at each step:

1. **predicted\_defect\_prob**: After an action is taken and the next\_state is determined, feed this next\_state into the **trained XGBoost/GB defect classifier (from Phase 1)**. The output probability of defect is this value. This directly links the RL agent's actions to the quality prediction model. A loIr predicted defect probability yields a higher reward.
2. **test\_cost**: This is directly tied to the sampling rate action. If sampling\_rate\_action == 1, test\_cost is 1; otherwise, it is 0. This penalizes the agent for frequently requesting more tests.
3. **reg\_compliance**: This is a penalty for suggesting actions that would violate safety limits. If the proposed action is clipped by the safety layer (see Section 3.2), reg\_compliance is -1; otherwise, it is 0.
4. **downtime**: This is a penalty for reducing throughput. If speed\_adjustment\_action == 0 (decrease speed), downtime is 1; otherwise, it is 0.

* **Episode:** A single episode corresponds to one full manufacturing batch from the historical dataset.
* **reset() function:** This function will randomly select a new batch\_id from the training set, load its corresponding time-series and static data, and return the initial state vector for t=0.
* **step(action) function:** This function advances the simulation by one time step (e.g., 1 minute). It finds the corresponding next state in the historical data, calculates the reward for the transition, and determines if the episode is done.
* **done flag:** The episode terminates (returns done=True) when the end of the historical time-series for that batch is reached.

**3. Offline RL Training Strategy**

Since I cannot train the agent on a live manufacturing plant, I must use **Offline Reinforcement Learning**. This involves training the agent on a fixed dataset of historical trajectories without further interaction with the environment.

I need to convert Ir 1005 historical batches into a dataset of (state, action, reward, next\_state, done) transitions. This is the most crucial data engineering step.

1. **Iterate through each batch** in Ir training set.
2. For each time step t within a batch:
   * Construct the state vector as defined in Section 2.1.
   * **Infer the historical action**: This is the most challenging part. I must infer what action was taken by the human operators at that time. This can be approximated by calculating the change in setpoints (e.g., action\_speed = tbl\_speed[t] - tbl\_speed[t-1]). I will then need to discretize these continuous changes to match Ir MultiDiscrete action space. This inferred action is known as the "behavioral policy."
   * Construct the next\_state vector for time t+1.
   * Calculate the reward for transitioning from state to next\_state using the inferred historical action.
   * Set the done flag.
3. Store these transition tuples in a format compatible with an offline RL library like d3rlpy.

* **Algorithm**: Start with **Conservative Q-Learning (CQL)**. CQL is specifically designed for offline RL. It learns a conservative Q-function that avoids overestimating the value of actions that Ire not present in the historical dataset, making it safer and more stable. The d3rlpy library has a robust implementation of CQL.
* **Safety Layer**: The project blueprint mandates a safety layer. This is a critical wrapper around the agent's actions. Before any action is executed in the step function, it must be passed through this layer.
  + **Implementation**: Create a function that takes the agent's proposed action and the current state as input.
  + **Check Limits**: It checks if the resulting parameter values (e.g., current\_force + force\_adjustment) would exceed predefined absolute limits (e.g., compression\_force must be in [10, 20] kN, speed must be <= 180k tablets/hr).
  + **Clip Actions**: If a limit is violated, the action is "clipped" or modified to the nearest safe value. The reg\_compliance component of the reward should then be set to -1 to penalize the agent for suggesting unsafe actions.

**4. Evaluation and Back-testing**

Evaluating an offline RL agent is non-trivial. The primary method is **Offline Policy Evaluation (OPE)**, or more intuitively, back-testing on a held-out set of batches.

1. **Hold-out Set**: Reserve a set of batches (e.g., the most recent 15%) that the agent has never seen during training.
2. **Back-testing Simulation**: For each batch in the hold-out set:
   * Initialize the environment with that batch.
   * At each time step, let the trained CQL agent choose an action.
   * Record the agent's action and the resulting reward.
   * Track the cumulative reward, the average predicted defect probability, and the number of times the safety layer was triggered.
3. **Benchmark**: Compare the agent's performance against the historical performance (the trajectory that actually occurred). The key success metrics are:
   * **Reduction in Predicted Defect Rate**: Does the agent's policy lead to states with a consistently loIr predicted defect probability than the historical policy?
   * **Improvement in Cumulative Reward**: Does the agent achieve a higher total reward over the course of a batch compared to the historical run?
   * **Safety Violations**: How often does the agent attempt to take an unsafe action? This should be minimal.

This back-testing framework will provide a strong indication of how the agent would perform in a real-world scenario.