Ball Mill Process Optimization Plan

Multi-Model Machine Learning Approach for Minimizing +200 μm Scrap

---

1. Problem Overview

Goal

Minimize the +200 μm pulp fraction (scrap) in a ball milling process using machine learning optimization.

Available Controls (Manipulated Variables - MVs)

- **Ore feed rate** (t/h) - How much ore enters the mill
- **Mill water flow** (m^3/h) - Water added to the mill
- **Sump water flow** (m^3/h) - Water added to the sump
- **Ball dosage** (t/h) - Fresh grinding balls added

Process Measurements (Controlled Variables - CVs)

- **Mill motor power** (kW) - Energy consumption indicator
- **Pulp density** (kg/L) - Solid-to-liquid ratio
- **Pulp flow rate** (m^3/h) - Volumetric throughput
- **Hydrocyclone inlet pressure** (bar) - Hydraulic condition

External Factors (Disturbance Variables - DVs)

- **Ore quality parameters** (hardness, grindability, etc.) - From lab analysis

---

2. Strategy Overview: Multi-Model Approach

Why Multi-Model?

The ball milling process has a clear cause-and-effect chain:

```
MVs (what we control) → CVs (what we measure) → Quality (what we want to optimize)
```

Instead of one complex model, we build:

1. **Process models**: Predict how our controls affect measurements (MV → CV)
2. **Quality model**: Predict how measurements affect final quality (CV → Quality)

Benefits

- **Interpretable**: Each model represents a physical relationship
- **Actionable**: Optimization directly provides control setpoints
- **Robust**: If one relationship changes, we only retrain that specific model
- **Realistic**: Incorporates process constraints naturally

---

3. Data Preparation

Step 3.1: Data Collection and Cleaning

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

# Load your historical process data
df = pd.read_csv('ball_mill_data.csv')

# Define variable categories
MVs = ['ore_feed_rate', 'mill_water_flow', 'sump_water_flow', 'ball_dosage']
CVs = ['motor_power', 'pulp_density', 'pulp_flow', 'hydrocyclone_pressure']
DVs = ['ore_hardness', 'grindability_index', 'ore_size_p80']  # Add your ore
quality params
target = 'plus_200_micron_percentage'

# Data cleaning
print(f"Original data shape: {df.shape}")

# Remove outliers (example: beyond 3 standard deviations)
def remove_outliers(df, columns):
    for col in columns:
```

```python
        mean = df[col].mean()
        std = df[col].std()
        df = df[(df[col] > mean - 3*std) & (df[col] < mean + 3*std)]
    return df

df_clean = remove_outliers(df, MVs + CVs + [target])
print(f"After outlier removal: {df_clean.shape}")

# Handle missing values
df_clean = df_clean.dropna()
print(f"After removing NaN: {df_clean.shape}")
```

Step 3.2: Feature Engineering

```python
# Create derived features that capture process physics
df_clean['water_to_ore_ratio'] = (df_clean['mill_water_flow'] +
df_clean['sump_water_flow']) / df_clean['ore_feed_rate']
df_clean['specific_energy'] = df_clean['motor_power'] / df_clean['ore_feed_rate']
# kWh/t
df_clean['solids_concentration'] = df_clean['pulp_density'] * df_clean['pulp_flow']

# Add these to your feature lists if they improve models
engineered_features = ['water_to_ore_ratio', 'specific_energy',
'solids_concentration']
```

Step 3.3: Data Scaling

```python
# Scale features for better model performance
scaler_mvs = StandardScaler()
scaler_cvs = StandardScaler()

# Fit scalers on training data (we'll split later)
X_mvs_scaled = scaler_mvs.fit_transform(df_clean[MVs])
X_cvs_scaled = scaler_cvs.fit_transform(df_clean[CVs])

# Store scalers for later use during optimization
import joblib
joblib.dump(scaler_mvs, 'scaler_mvs.pkl')
joblib.dump(scaler_cvs, 'scaler_cvs.pkl')
```

---

4. Model Building

Step 4.1: Train Process Models (MV → CV)

These models predict how your controls affect process measurements.

```
import xgboost as xgb
from sklearn.metrics import mean_squared_error, r2_score

# Model 1: All MVs → Motor Power
print("Training Model 1: MVs → Motor Power")
X_model1 = df_clean[MVs]  # All 4 manipulated variables
y_model1 = df_clean['motor_power']

X_train1, X_test1, y_train1, y_test1 = train_test_split(X_model1, y_model1,
test_size=0.2, random_state=42)

model1 = xgb.XGBRegressor(
    n_estimators=200,
    max_depth=6,
    learning_rate=0.1,
    subsample=0.8,
    colsample_bytree=0.8,
    random_state=42
)
model1.fit(X_train1, y_train1)

# Evaluate
y_pred1 = model1.predict(X_test1)
print(f"Model 1 R² Score: {r2_score(y_test1, y_pred1):.4f}")
print(f"Model 1 RMSE: {np.sqrt(mean_squared_error(y_test1, y_pred1)):.2f} kW")

# Model 2: MVs (excluding balls) → Pulp Density
print("\nTraining Model 2: MVs → Pulp Density")
X_model2 = df_clean[['ore_feed_rate', 'mill_water_flow', 'sump_water_flow']]  #
Balls don't affect density much
y_model2 = df_clean['pulp_density']

X_train2, X_test2, y_train2, y_test2 = train_test_split(X_model2, y_model2,
test_size=0.2, random_state=42)

model2 = xgb.XGBRegressor(
    n_estimators=200,
    max_depth=6,
    learning_rate=0.1,
    subsample=0.8,
    colsample_bytree=0.8,
    random_state=42
)
model2.fit(X_train2, y_train2)

y_pred2 = model2.predict(X_test2)
print(f"Model 2 R² Score: {r2_score(y_test2, y_pred2):.4f}")
print(f"Model 2 RMSE: {np.sqrt(mean_squared_error(y_test2, y_pred2)):.4f} kg/L")

# Model 3: MVs (excluding balls) → Pulp Flow
```

```python
print("\nTraining Model 3: MVs → Pulp Flow")
X_model3 = df_clean[['ore_feed_rate', 'mill_water_flow', 'sump_water_flow']]
y_model3 = df_clean['pulp_flow']

X_train3, X_test3, y_train3, y_test3 = train_test_split(X_model3, y_model3,
test_size=0.2, random_state=42)

model3 = xgb.XGBRegressor(
    n_estimators=200,
    max_depth=6,
    learning_rate=0.1,
    subsample=0.8,
    colsample_bytree=0.8,
    random_state=42
)
model3.fit(X_train3, y_train3)

y_pred3 = model3.predict(X_test3)
print(f"Model 3 R² Score: {r2_score(y_test3, y_pred3):.4f}")
print(f"Model 3 RMSE: {np.sqrt(mean_squared_error(y_test3, y_pred3)):.2f} m³/h")

# Model 4: MVs (excluding balls) → Hydrocyclone Pressure
print("\nTraining Model 4: MVs → Pressure")
X_model4 = df_clean[['ore_feed_rate', 'mill_water_flow', 'sump_water_flow']]
y_model4 = df_clean['hydrocyclone_pressure']

X_train4, X_test4, y_train4, y_test4 = train_test_split(X_model4, y_model4,
test_size=0.2, random_state=42)

model4 = xgb.XGBRegressor(
    n_estimators=200,
    max_depth=6,
    learning_rate=0.1,
    subsample=0.8,
    colsample_bytree=0.8,
    random_state=42
)
model4.fit(X_train4, y_train4)

y_pred4 = model4.predict(X_test4)
print(f"Model 4 R² Score: {r2_score(y_test4, y_pred4):.4f}")
print(f"Model 4 RMSE: {np.sqrt(mean_squared_error(y_test4, y_pred4)):.3f} bar")
```

Step 4.2: Train Quality Model (CV → Quality)

CRITICAL: This model is trained on **real measured CVs**, not predictions!

```python
print("\nTraining Quality Model: CVs → +200 μm Fraction")

# Use REAL measured CVs from historical data
X_quality = df_clean[CVs]  # Real sensor measurements
y_quality = df_clean[target]  # Real +200 μm measurements
```

```python
# Include disturbance variables if available
if DVs and all(dv in df_clean.columns for dv in DVs):
    X_quality = pd.concat([X_quality, df_clean[DVs]], axis=1)
    print("Including disturbance variables in quality model")

X_train_q, X_test_q, y_train_q, y_test_q = train_test_split(X_quality, y_quality,
test_size=0.2, random_state=42)

# Quality model - use more complex parameters since this is your main model
quality_model = xgb.XGBRegressor(
    n_estimators=500,  # More trees for complex relationships
    max_depth=8,       # Deeper trees
    learning_rate=0.05, # Lower learning rate
    subsample=0.8,
    colsample_bytree=0.8,
    reg_alpha=0.1,     # L1 regularization
    reg_lambda=0.1,    # L2 regularization
    random_state=42
)

quality_model.fit(X_train_q, y_train_q)

# Evaluate quality model
y_pred_q = quality_model.predict(X_test_q)
print(f"Quality Model R² Score: {r2_score(y_test_q, y_pred_q):.4f}")
print(f"Quality Model RMSE: {np.sqrt(mean_squared_error(y_test_q,
y_pred_q)):.2f}%")

# Feature importance for the quality model
feature_importance = pd.DataFrame({
    'feature': X_quality.columns,
    'importance': quality_model.feature_importances_
}).sort_values('importance', ascending=False)
print("\nFeature Importance in Quality Model:")
print(feature_importance)
```

Step 4.3: Save All Models

```python
# Save all trained models
joblib.dump(model1, 'model1_mvs_to_power.pkl')
joblib.dump(model2, 'model2_mvs_to_density.pkl')
joblib.dump(model3, 'model3_mvs_to_flow.pkl')
joblib.dump(model4, 'model4_mvs_to_pressure.pkl')
joblib.dump(quality_model, 'quality_model_cvs_to_plus200.pkl')

print("All models saved successfully!")
```

5. Model Validation

Step 5.1: Validate Individual Process Models

Test how well each MV→CV model predicts on unseen data:

```
def validate_process_models():
    """Validate each process model individually"""

    # Test on a subset of validation data
    n_test = 100
    test_indices = np.random.choice(len(X_test1), n_test, replace=False)

    results = {}

    # Model 1 validation
    X_val1 = X_test1.iloc[test_indices]
    y_val1_true = y_test1.iloc[test_indices]
    y_val1_pred = model1.predict(X_val1)
    results['power'] = {
        'r2': r2_score(y_val1_true, y_val1_pred),
        'rmse': np.sqrt(mean_squared_error(y_val1_true, y_val1_pred))
    }

    # Similar for models 2, 3, 4
    # ... (repeat for each model)

    return results

validation_results = validate_process_models()
print("Process Models Validation Results:")
for model_name, metrics in validation_results.items():
    print(f"{model_name}: R² = {metrics['r2']:.3f}, RMSE = {metrics['rmse']:.3f}")
```

Step 5.2: Validate Complete Chain (MV → CV → Quality)

This is crucial - test the entire prediction chain:

```
def validate_complete_chain(n_samples=200):
    """Test the complete MV → CV → Quality prediction chain"""

    # Select random historical points
    test_indices = np.random.choice(len(df_clean), n_samples, replace=False)
    test_data = df_clean.iloc[test_indices]
```

```
    predictions = []
    actuals = []

    for idx, row in test_data.iterrows():
        # Get actual MVs that were used
        actual_mvs = row[MVs].values

        # Predict CVs using process models
        pred_power = model1.predict([actual_mvs])[0]
        pred_density = model2.predict([actual_mvs[:3]])[0]  # Exclude balls
        pred_flow = model3.predict([actual_mvs[:3]])[0]
        pred_pressure = model4.predict([actual_mvs[:3]])[0]

        # Predict quality using predicted CVs
        predicted_cvs = np.array([pred_power, pred_density, pred_flow,
pred_pressure])
        if DVs:  # Add disturbance variables if available
            predicted_cvs = np.concatenate([predicted_cvs, row[DVs].values])

        pred_quality = quality_model.predict([predicted_cvs])[0]

        predictions.append(pred_quality)
        actuals.append(row[target])

    # Calculate chain performance
    chain_r2 = r2_score(actuals, predictions)
    chain_rmse = np.sqrt(mean_squared_error(actuals, predictions))

    print(f"Complete Chain Validation:")
    print(f"R² Score: {chain_r2:.4f}")
    print(f"RMSE: {chain_rmse:.2f}%")
    print(f"Mean Absolute Error: {np.mean(np.abs(np.array(actuals) -
np.array(predictions))):.2f}%")

    return predictions, actuals

chain_predictions, chain_actuals = validate_complete_chain()
```

---

6. Optimization Setup

Step 6.1: Define Operating Constraints

```
# Define realistic operating ranges for your process
MV_BOUNDS = {
    'ore_feed_rate': (50, 150),      # t/h - typical range for your mill
    'mill_water_flow': (10, 50),    # m³/h
    'sump_water_flow': (5, 30),     # m³/h
    'ball_dosage': (0.5, 2.0)       # t/h
```

```python
}

# Define acceptable CV ranges (process constraints)
CV_CONSTRAINTS = {
    'motor_power': (500, 1200),       # kW - don't overload motor
    'pulp_density': (1.2, 1.6),      # kg/L - flotation requirements
    'pulp_flow': (80, 200),          # m³/h - hydraulic limits
    'hydrocyclone_pressure': (1.0, 3.0)  # bar - equipment limits
}

print("Operating constraints defined:")
print("MV Bounds:", MV_BOUNDS)
print("CV Constraints:", CV_CONSTRAINTS)
```

Step 6.2: Create Prediction Function

```python
def predict_quality_from_mvs(ore_feed, mill_water, sump_water, ball_dosage,
ore_quality_params=None):
    """
    Complete prediction chain: MVs → CVs → Quality

    Args:
        ore_feed, mill_water, sump_water, ball_dosage: Manipulated variables
        ore_quality_params: List of disturbance variables (if available)

    Returns:
        predicted_quality: Predicted +200 μm percentage
        predicted_cvs: Dictionary of predicted controlled variables
        is_feasible: Boolean indicating if constraints are met
    """

    # Step 1: Predict CVs from MVs
    mvs_full = np.array([ore_feed, mill_water, sump_water, ball_dosage])
    mvs_partial = np.array([ore_feed, mill_water, sump_water])  # For models 2,3,4

    pred_power = model1.predict([mvs_full])[0]
    pred_density = model2.predict([mvs_partial])[0]
    pred_flow = model3.predict([mvs_partial])[0]
    pred_pressure = model4.predict([mvs_partial])[0]

    predicted_cvs = {
        'motor_power': pred_power,
        'pulp_density': pred_density,
        'pulp_flow': pred_flow,
        'hydrocyclone_pressure': pred_pressure
    }

    # Step 2: Check if predicted CVs meet constraints
    is_feasible = True
    for cv_name, cv_value in predicted_cvs.items():
        min_val, max_val = CV_CONSTRAINTS[cv_name]
        if not (min_val <= cv_value <= max_val):
```

```
            is_feasible = False
            break

    # Step 3: Predict quality if feasible
    if is_feasible:
        cv_array = np.array([pred_power, pred_density, pred_flow, pred_pressure])
        if ore_quality_params is not None:
            cv_array = np.concatenate([cv_array, ore_quality_params])

        predicted_quality = quality_model.predict([cv_array])[0]
    else:
        predicted_quality = 999.0  # High penalty for infeasible solutions

    return predicted_quality, predicted_cvs, is_feasible

# Test the prediction function
test_quality, test_cvs, feasible = predict_quality_from_mvs(100, 25, 15, 1.2)
print(f"\nTest prediction: {test_quality:.2f}% +200μm, Feasible: {feasible}")
print("Predicted CVs:", test_cvs)
```

---

7. Bayesian Optimization with Optuna

Step 7.1: Define Objective Function

```
import optuna
from optuna.samplers import TPESampler

def create_objective_function(current_ore_quality=None):
    """
    Create objective function for Optuna optimization

    Args:
        current_ore_quality: Current ore quality parameters (if available)

    Returns:
        objective: Function that Optuna will minimize
    """

    def objective(trial):
        # Step 1: Sample manipulated variables within bounds
        ore_feed = trial.suggest_float('ore_feed_rate',
                                       MV_BOUNDS['ore_feed_rate'][0],
                                       MV_BOUNDS['ore_feed_rate'][1])

        mill_water = trial.suggest_float('mill_water_flow',
                                         MV_BOUNDS['mill_water_flow'][0],
                                         MV_BOUNDS['mill_water_flow'][1])

        sump_water = trial.suggest_float('sump_water_flow',
```

```python
                                        MV_BOUNDS['sump_water_flow'][0],
                                        MV_BOUNDS['sump_water_flow'][1])

        ball_dosage = trial.suggest_float('ball_dosage',
                                         MV_BOUNDS['ball_dosage'][0],
                                         MV_BOUNDS['ball_dosage'][1])

        # Step 2: Predict quality using the complete chain
        predicted_quality, predicted_cvs, is_feasible = predict_quality_from_mvs(
            ore_feed, mill_water, sump_water, ball_dosage, current_ore_quality
        )

        # Step 3: Add penalties for infeasible solutions
        if not is_feasible:
            return 100.0  # High penalty

        # Step 4: Optional - add soft constraints for better operation
        penalty = 0.0

        # Penalty for very high power consumption (operational cost)
        if predicted_cvs['motor_power'] > 1000:  # kW
            penalty += 0.5

        # Penalty for very low density (poor flotation downstream)
        if predicted_cvs['pulp_density'] < 1.3:  # kg/L
            penalty += 0.5

        return predicted_quality + penalty

    return objective
```

Step 7.2: Run Optimization

```python
def optimize_process(current_ore_quality=None, n_trials=1000):
    """
    Run Bayesian optimization to find best MV settings

    Args:
        current_ore_quality: Current ore parameters (if available)
        n_trials: Number of optimization trials

    Returns:
        best_params: Dictionary of optimal MV values
        study: Optuna study object for analysis
    """

    # Create Optuna study
    study = optuna.create_study(
        direction='minimize',  # Minimize +200 μm fraction
        sampler=TPESampler(seed=42),
        study_name='ball_mill_optimization'
    )
```

```
    # Create objective function
    objective_func = create_objective_function(current_ore_quality)

    # Run optimization
    print(f"Starting optimization with {n_trials} trials...")
    study.optimize(objective_func, n_trials=n_trials)

    # Get best results
    best_params = study.best_params
    best_value = study.best_value

    print(f"\nOptimization completed!")
    print(f"Best +200 µm fraction: {best_value:.2f}%")
    print(f"Best parameters:")
    for param, value in best_params.items():
        print(f"  {param}: {value:.2f}")

    # Predict resulting CVs for the best parameters
    best_quality, best_cvs, feasible = predict_quality_from_mvs(
        best_params['ore_feed_rate'],
        best_params['mill_water_flow'],
        best_params['sump_water_flow'],
        best_params['ball_dosage'],
        current_ore_quality
    )

    print(f"\nPredicted process conditions with optimal settings:")
    for cv_name, cv_value in best_cvs.items():
        print(f"  {cv_name}: {cv_value:.2f}")

    return best_params, study

# Run the optimization
# Example: Use average ore quality from your dataset
current_ore_quality = df_clean[DVs].mean().values if DVs else None
optimal_params, optimization_study = optimize_process(current_ore_quality,
n_trials=1000)
```

---

8. Results Analysis and Visualization

Step 8.1: Analyze Optimization Results

```
import matplotlib.pyplot as plt
import seaborn as sns

def analyze_optimization_results(study):
    """Analyze and visualize optimization results"""
```

```python
    # Convert trials to DataFrame for analysis
    trials_df = study.trials_dataframe()

    # Plot optimization history
    fig, axes = plt.subplots(2, 2, figsize=(15, 10))

    # Convergence plot
    axes[0,0].plot(trials_df['number'], trials_df['value'])
    axes[0,0].set_xlabel('Trial Number')
    axes[0,0].set_ylabel('+200 μm Fraction (%)')
    axes[0,0].set_title('Optimization Convergence')
    axes[0,0].grid(True)

    # Parameter distribution for best trials (top 10%)
    n_best = len(trials_df) // 10
    best_trials = trials_df.nsmallest(n_best, 'value')

    # Ore feed rate distribution
    axes[0,1].hist(best_trials['params_ore_feed_rate'], bins=20, alpha=0.7)
    axes[0,1].set_xlabel('Ore Feed Rate (t/h)')
    axes[0,1].set_ylabel('Frequency')
    axes[0,1].set_title('Optimal Ore Feed Rate Distribution')

    # Mill water distribution
    axes[1,0].hist(best_trials['params_mill_water_flow'], bins=20, alpha=0.7)
    axes[1,0].set_xlabel('Mill Water Flow (m³/h)')
    axes[1,0].set_ylabel('Frequency')
    axes[1,0].set_title('Optimal Mill Water Distribution')

    # Ball dosage distribution
    axes[1,1].hist(best_trials['params_ball_dosage'], bins=20, alpha=0.7)
    axes[1,1].set_xlabel('Ball Dosage (t/h)')
    axes[1,1].set_ylabel('Frequency')
    axes[1,1].set_title('Optimal Ball Dosage Distribution')

    plt.tight_layout()
    plt.savefig('optimization_analysis.png', dpi=300, bbox_inches='tight')
    plt.show()

    return best_trials

best_trials = analyze_optimization_results(optimization_study)
```

Step 8.2: Compare Current vs. Optimal Operation

```python
def compare_current_vs_optimal():
    """Compare current operation with optimized settings"""

    # Current typical operation (calculate from recent data)
    recent_data = df_clean.tail(100)  # Last 100 data points
```

```
    current_mvs = recent_data[MVs].mean()
    current_quality = recent_data[target].mean()
    current_cvs = recent_data[CVs].mean()

    # Optimal operation
    optimal_mvs = optimal_params
    optimal_quality, optimal_cvs, _ = predict_quality_from_mvs(
        optimal_mvs['ore_feed_rate'],
        optimal_mvs['mill_water_flow'],
        optimal_mvs['sump_water_flow'],
        optimal_mvs['ball_dosage']
    )

    # Create comparison table
    comparison = pd.DataFrame({
        'Parameter': ['Ore Feed (t/h)', 'Mill Water (m³/h)', 'Sump Water (m³/h)',
                      'Ball Dosage (t/h)', '', 'Motor Power (kW)', 'Pulp Density
(kg/L)',
                      'Pulp Flow (m³/h)', 'Pressure (bar)', '', '+200 µm Fraction
(%)'],
        'Current': [current_mvs['ore_feed_rate'], current_mvs['mill_water_flow'],
                    current_mvs['sump_water_flow'], current_mvs['ball_dosage'], '',
                    current_cvs['motor_power'], current_cvs['pulp_density'],
                    current_cvs['pulp_flow'], current_cvs['hydrocyclone_pressure'],
'',
                    current_quality],
        'Optimal': [optimal_mvs['ore_feed_rate'], optimal_mvs['mill_water_flow'],
                    optimal_mvs['sump_water_flow'], optimal_mvs['ball_dosage'], '',
                    optimal_cvs['motor_power'], optimal_cvs['pulp_density'],
                    optimal_cvs['pulp_flow'], optimal_cvs['hydrocyclone_pressure'],
'',
                    optimal_quality],
        'Improvement': ['', '', '', '', '', '', '', '', '', '',
                        f"{current_quality - optimal_quality:.2f}%"]
    })

    print("Current vs. Optimal Operation Comparison:")
    print(comparison.to_string(index=False))

    # Calculate potential improvement
    improvement_percent = ((current_quality - optimal_quality) / current_quality) *
100
    print(f"\nPotential Quality Improvement: {improvement_percent:.1f}%")

    return comparison

comparison_results = compare_current_vs_optimal()
```

9. Implementation Strategy

Step 9.1: Gradual Implementation Plan

```python
def create_implementation_plan(current_mvs, optimal_mvs, n_steps=5):
    """
    Create gradual transition plan from current to optimal settings

    Args:
        current_mvs: Current MV values
        optimal_mvs: Optimal MV values
        n_steps: Number of implementation steps
    """

    implementation_steps = []

    for step in range(n_steps + 1):
        # Linear interpolation between current and optimal
        alpha = step / n_steps

        step_mvs = {}
        for mv in MVs:
            current_val = current_mvs[mv] if mv in current_mvs else optimal_mvs[mv]
            step_mvs[mv] = current_val + alpha * (optimal_mvs[mv] - current_val)

        # Predict expected results for this step
        quality, cvs, feasible = predict_quality_from_mvs(
            step_mvs['ore_feed_rate'],
            step_mvs['mill_water_flow'],
            step_mvs['sump_water_flow'],
            step_mvs['ball_dosage']
        )

        implementation_steps.append({
            'step': step,
            'mvs': step_mvs,
            'predicted_quality': quality,
            'predicted_cvs': cvs,
            'feasible': feasible
        })

    # Create implementation DataFrame
    impl_df = pd.DataFrame([
        {
            'Step': step['step'],
            'Ore Feed': step['mvs']['ore_feed_rate'],
            'Mill Water': step['mvs']['mill_water_flow'],
            'Sump Water': step['mvs']['sump_water_flow'],
            'Ball Dosage': step['mvs']['ball_dosage'],
            'Predicted +200μm': step['predicted_quality'],
            'Motor Power': step['predicted_cvs']['motor_power'],
            'Pulp Density': step['predicted_cvs']['pulp_density'],
            'Feasible': step['feasible']
        } for step in implementation_steps
    ])

    print("Gradual Implementation Plan:")
    print(impl_df.round(2))

    return impl_df
```

```python
# Example usage (replace with your current values)
current_operation = df_clean[MVs].tail(10).mean()  # Average of last 10 operations
implementation_plan = create_implementation_plan(current_operation, optimal_params)
```

Step 9.2: Real-Time Monitoring Setup

```python
def setup_monitoring_system():
    """
    Setup for monitoring actual vs. predicted performance during implementation
    """

    monitoring_template = {
        'timestamp': [],
        'actual_mvs': [],
        'actual_cvs': [],
        'actual_quality': [],
        'predicted_cvs': [],
        'predicted_quality': [],
        'cv_prediction_error': [],
        'quality_prediction_error': []
    }

    return monitoring_template

def update_monitoring(monitoring_data, actual_mvs, actual_cvs, actual_quality):
    """
    Update monitoring data with new measurements

    Args:
        monitoring_data: Dictionary to store monitoring info
        actual_mvs: List/array of actual MV values
        actual_cvs: List/array of actual CV values
        actual_quality: Actual +200 µm measurement
    """

    # Predict what we expected to see
    predicted_quality, predicted_cvs, _ = predict_quality_from_mvs(*actual_mvs)

    # Calculate prediction errors
    cv_errors = {}
    for i, cv_name in enumerate(CVs):
        cv_errors[cv_name] = abs(actual_cvs[i] - predicted_cvs[cv_name])

    quality_error = abs(actual_quality - predicted_quality)

    # Update monitoring data
    monitoring_data['timestamp'].append(pd.Timestamp.now())
    monitoring_data['actual_mvs'].append(actual_mvs)
    monitoring_data['actual_cvs'].append(actual_cvs)
    monitoring_data['actual_quality'].append(actual_quality)
    monitoring_data['predicted_cvs'].append(list(predicted_cvs.values()))
```

```
        monitoring_data['predicted_quality'].append(predicted_quality)
        monitoring_data['cv_prediction_error'].append(cv_errors)
        monitoring_data['quality_prediction_error'].append(quality_error)

        print(f"Quality prediction error: {quality_error:.2f}%")

        return monitoring_data

    # Initialize monitoring
    monitoring_system = setup_monitoring_system()
```

10. Advanced Optimization Strategies

Step 10.1: Multi-Objective Optimization

Sometimes you want to optimize multiple goals simultaneously:

```
def multi_objective_optimization():
    """
    Optimize for both quality and operational cost
    """

    def multi_objective(trial):
        # Sample MVs
        ore_feed = trial.suggest_float('ore_feed_rate',
                                       MV_BOUNDS['ore_feed_rate'][0],
                                       MV_BOUNDS['ore_feed_rate'][1])
        mill_water = trial.suggest_float('mill_water_flow',
                                         MV_BOUNDS['mill_water_flow'][0],
                                         MV_BOUNDS['mill_water_flow'][1])
        sump_water = trial.suggest_float('sump_water_flow',
                                         MV_BOUNDS['sump_water_flow'][0],
                                         MV_BOUNDS['sump_water_flow'][1])
        ball_dosage = trial.suggest_float('ball_dosage',
                                          MV_BOUNDS['ball_dosage'][0],
                                          MV_BOUNDS['ball_dosage'][1])

        # Predict outcomes
        predicted_quality, predicted_cvs, is_feasible = predict_quality_from_mvs(
            ore_feed, mill_water, sump_water, ball_dosage
        )

        if not is_feasible:
            return 100.0, 1000.0  # High penalties

        # Calculate operational cost (simplified)
        power_cost = predicted_cvs['motor_power'] * 0.10  # $/kWh
        water_cost = (mill_water + sump_water) * 0.50    # $/m³
```

```
        ball_cost = ball_dosage * 800.0                  # $/t

        total_cost = power_cost + water_cost + ball_cost

        return predicted_quality, total_cost

    # Multi-objective study
    study = optuna.create_study(
        directions=['minimize', 'minimize'],  # Minimize both quality and cost
        study_name='multi_objective_optimization'
    )

    study.optimize(multi_objective, n_trials=1000)

    return study

# Uncomment to run multi-objective optimization
# multi_obj_study = multi_objective_optimization()
```

Step 10.2: Robust Optimization

Account for uncertainty in ore quality and model predictions:

```
def robust_optimization(ore_quality_scenarios):
    """
    Optimize for robust performance across different ore qualities

    Args:
        ore_quality_scenarios: List of different ore quality conditions
    """

    def robust_objective(trial):
        # Sample MVs
        mvs = [
            trial.suggest_float('ore_feed_rate', *MV_BOUNDS['ore_feed_rate']),
            trial.suggest_float('mill_water_flow', *MV_BOUNDS['mill_water_flow']),
            trial.suggest_float('sump_water_flow', *MV_BOUNDS['sump_water_flow']),
            trial.suggest_float('ball_dosage', *MV_BOUNDS['ball_dosage'])
        ]

        # Test performance across all ore quality scenarios
        qualities = []
        feasible_count = 0

        for ore_scenario in ore_quality_scenarios:
            quality, cvs, feasible = predict_quality_from_mvs(*mvs, ore_scenario)

            if feasible:
                qualities.append(quality)
                feasible_count += 1
            else:
                qualities.append(100.0)  # Penalty
```

```
        # Robust metrics
        mean_quality = np.mean(qualities)
        worst_quality = np.max(qualities)  # Worst-case scenario
        feasibility_ratio = feasible_count / len(ore_quality_scenarios)

        # Penalize if not feasible across all scenarios
        if feasibility_ratio < 0.8:  # Must work for at least 80% of scenarios
            return 100.0

        # Optimize for worst-case performance (conservative approach)
        return 0.7 * mean_quality + 0.3 * worst_quality

    study = optuna.create_study(direction='minimize')
    study.optimize(robust_objective, n_trials=1500)

    return study

# Example: Create ore quality scenarios (modify based on your lab data)
ore_scenarios = [
    [7.5, 15.2],  # [hardness, grindability] - soft ore
    [8.5, 14.1],  # medium ore
    [9.8, 12.5],  # hard ore
]

# Uncomment to run robust optimization
# robust_study = robust_optimization(ore_scenarios)
```

---

11. Model Maintenance and Updates

Step 11.1: Model Drift Detection

Ball mills change over time due to liner wear, ball wear, and other factors:

```
def detect_model_drift(new_data, lookback_days=30):
    """
    Detect if models are becoming less accurate over time

    Args:
        new_data: Recent process data
        lookback_days: Period to analyze for drift
    """

    # Get recent data
    recent_data = new_data.tail(lookback_days * 24)  # Assuming hourly data

    # Test each process model
    drift_results = {}
```

```
    # Model 1 drift check
    X_recent = recent_data[MVs]
    y_recent_power = recent_data['motor_power']
    y_pred_power = model1.predict(X_recent)

    power_mae = np.mean(np.abs(y_recent_power - y_pred_power))
    drift_results['power_model'] = {
        'mae': power_mae,
        'drift_detected': power_mae > 50.0  # Threshold: 50 kW average error
    }

    # Similar checks for models 2, 3, 4
    # ... (implement for each model)

    # Quality model drift check
    X_recent_cvs = recent_data[CVs]
    y_recent_quality = recent_data[target]
    y_pred_quality = quality_model.predict(X_recent_cvs)

    quality_mae = np.mean(np.abs(y_recent_quality - y_pred_quality))
    drift_results['quality_model'] = {
        'mae': quality_mae,
        'drift_detected': quality_mae > 2.0  # Threshold: 2% average error
    }

    # Summary
    models_drifted = [name for name, result in drift_results.items()
                      if result['drift_detected']]

    if models_drifted:
        print(f"⚠️ Model drift detected in: {', '.join(models_drifted)}")
        print("Consider retraining these models with recent data.")
    else:
        print("✅ No significant model drift detected.")

    return drift_results
```

Step 11.2: Incremental Model Updates

```
def update_models_incrementally(new_data, retrain_threshold=0.05):
    """
    Update models when performance degrades beyond threshold

    Args:
        new_data: New process data
        retrain_threshold: R² decrease threshold for retraining
    """

    # Test current model performance on new data
    X_new_mvs = new_data[MVs]
    X_new_cvs = new_data[CVs]
```

```
    y_new_quality = new_data[target]

    # Quality model performance on new data
    y_pred_new = quality_model.predict(X_new_cvs)
    current_r2 = r2_score(y_new_quality, y_pred_new)

    print(f"Current quality model R² on new data: {current_r2:.4f}")

    # If performance dropped significantly, retrain
    if current_r2 < (original_r2 - retrain_threshold):  # original_r2 from initial
training
        print("🔄 Retraining quality model with recent data...")

        # Combine old and new data (weighted toward recent)
        old_weight = 0.7
        new_weight = 0.3

        # You would implement weighted training here
        # This is a simplified example
        combined_X = pd.concat([X_train_q * old_weight, X_new_cvs * new_weight])
        combined_y = pd.concat([y_train_q * old_weight, y_new_quality *
new_weight])

        # Retrain quality model
        quality_model.fit(combined_X, combined_y)

        # Save updated model
        joblib.dump(quality_model, 'quality_model_updated.pkl')
        print("✅ Quality model updated and saved")

    return current_r2
```

---

12. Practical Implementation Checklist

Pre-Implementation Validation

- ☐ **Validate all models** on held-out test data ($R^2 > 0.8$ recommended)
- ☐ **Test complete chain** (MV → CV → Quality) on historical data
- ☐ **Verify constraints** are realistic and properly implemented
- ☐ **Check edge cases** - what happens at operating limits?
- ☐ **Review with process engineers** - do the relationships make sense?

Optimization Setup

- ☐ **Define realistic MV bounds** based on equipment limitations

- ☐ **Set CV constraints** based on downstream process requirements
- ☐ **Test objective function** manually with known good/bad operating points
- ☐ **Run small optimization** (100 trials) to check for issues
- ☐ **Validate optimal solution** - does it make physical sense?

Implementation Preparation

- ☐ **Create gradual transition plan** - don't jump to optimal settings immediately
- ☐ **Setup monitoring system** to track actual vs. predicted performance
- ☐ **Prepare rollback plan** - how to return to previous settings if needed
- ☐ **Train operators** on new setpoints and what to watch for
- ☐ **Establish update schedule** for model retraining

---

13. Troubleshooting Common Issues

Issue 1: Models Give Conflicting MV Recommendations

Problem: When you try to achieve optimal CVs, different models suggest different MVs.

Solution: This is why we optimize in MV space directly. The forward models will naturally find a compromise.

```
# Don't do this (leads to conflicts):
# target_power = 900  # kW
# mv1_from_power_model = inverse_power_model.predict([target_power])
# mv1_from_density_model = inverse_density_model.predict([target_density])
# # mv1_from_power ≠ mv1_from_density ❌

# Do this instead (automatic compromise):
def objective(trial):
    mvs = sample_mvs(trial)
    predicted_cvs = predict_all_cvs(mvs)  # Natural compromise
    return quality_model.predict(predicted_cvs)
```

Issue 2: Optimization Suggests Unrealistic Settings

Problem: Optuna finds "optimal" settings that are operationally impractical.

Solution: Add more constraints and penalties:

```
def constrained_objective(trial):
    mvs = sample_mvs(trial)
    quality, cvs, feasible = predict_quality_from_mvs(*mvs)

    if not feasible:
        return 100.0

    # Add operational penalties
    penalty = 0.0

    # Penalty for high power consumption (cost)
    if cvs['motor_power'] > 1000:
        penalty += (cvs['motor_power'] - 1000) * 0.01

    # Penalty for extreme water usage
    total_water = mvs[1] + mvs[2]  # mill_water + sump_water
    if total_water > 60:  # m³/h
        penalty += (total_water - 60) * 0.1

    # Penalty for density too far from flotation optimum
    optimal_density = 1.45  # kg/L for flotation
    density_penalty = abs(cvs['pulp_density'] - optimal_density) * 2.0

    return quality + penalty + density_penalty
```

Issue 3: Poor Model Performance on New Data

Problem: Models work well on training data but poorly on new conditions.

Solutions:

```
# 1. Check for data distribution shift
def check_data_drift(old_data, new_data, features):
    """Check if new data distribution differs from training data"""

    for feature in features:
        old_mean = old_data[feature].mean()
```

```
        new_mean = new_data[feature].mean()
        old_std = old_data[feature].std()

        # Simple drift detection
        z_score = abs(new_mean - old_mean) / old_std

        if z_score > 2.0:  # More than 2 standard deviations
            print(f"⚠️ Significant drift detected in {feature}")
            print(f"   Old mean: {old_mean:.2f}, New mean: {new_mean:.2f}")

# 2. Expand training data range
def expand_training_data():
    """Include more diverse operating conditions in training"""

    # Look for gaps in your training data
    for mv in MVs:
        min_val, max_val = MV_BOUNDS[mv]
        actual_min = df_clean[mv].min()
        actual_max = df_clean[mv].max()

        coverage = (actual_max - actual_min) / (max_val - min_val)
        print(f"{mv} coverage: {coverage:.1%}")

        if coverage < 0.8:  # Less than 80% coverage
            print(f"⚠️ Consider collecting data in range {actual_max:.1f}-
{max_val:.1f}")

expand_training_data()
```

Issue 4: Optimization Takes Too Long

Problem: Bayesian optimization is slow with many trials.

Solutions:

```
# 1. Use parallel optimization
def parallel_optimization():
    """Run optimization with multiple workers"""

    study = optuna.create_study(direction='minimize')

    # Use multiple processes
    study.optimize(objective_func, n_trials=1000, n_jobs=4)  # 4 parallel workers

    return study

# 2. Use early stopping
def optimization_with_early_stopping():
    """Stop optimization when no improvement for N trials"""

    class EarlyStoppingCallback:
```

```python
    def __init__(self, patience=100):
        self.patience = patience
        self.best_value = float('inf')
        self.trials_without_improvement = 0

    def __call__(self, study, trial):
        if trial.value < self.best_value:
            self.best_value = trial.value
            self.trials_without_improvement = 0
        else:
            self.trials_without_improvement += 1

        if self.trials_without_improvement >= self.patience:
            study.stop()

study = optuna.create_study(direction='minimize')
callback = EarlyStoppingCallback(patience=150)

study.optimize(objective_func, n_trials=2000, callbacks=[callback])

return study
```

---

14. Expected Results and Success Metrics

Success Criteria

- **Model Accuracy**: All process models should achieve $R^2 > 0.8$
- **Quality Model**: $R^2 > 0.85$ for +200 μm prediction
- **Optimization Improvement**: At least 10% reduction in +200 μm fraction
- **Operational Feasibility**: Optimal settings within equipment constraints
- **Consistency**: Similar results across multiple optimization runs

Performance Tracking

```python
def track_optimization_performance():
    """Track key performance indicators"""

    kpis = {
        'baseline_plus200': df_clean[target].mean(),
        'baseline_std': df_clean[target].std(),
        'optimized_plus200': optimal_quality,
```

```
            'improvement_percent': ((df_clean[target].mean() - optimal_quality) /
                                   df_clean[target].mean()) * 100,
            'model_accuracies': {
                'power_model_r2': r2_score(y_test1, model1.predict(X_test1)),
                'density_model_r2': r2_score(y_test2, model2.predict(X_test2)),
                'flow_model_r2': r2_score(y_test3, model3.predict(X_test3)),
                'pressure_model_r2': r2_score(y_test4, model4.predict(X_test4)),
                'quality_model_r2': r2_score(y_test_q, quality_model.predict(X_test_q))
            }
        }

    print("Optimization Performance Summary:")
    print(f"Baseline +200μm: {kpis['baseline_plus200']:.2f}% ±
{kpis['baseline_std']:.2f}%")
    print(f"Optimized +200μm: {kpis['optimized_plus200']:.2f}%")
    print(f"Improvement: {kpis['improvement_percent']:.1f}%")
    print("\nModel Accuracies:")
    for model, r2 in kpis['model_accuracies'].items():
        print(f"  {model}: {r2:.3f}")

    return kpis

performance_metrics = track_optimization_performance()
```

---

15. Complete Implementation Code

Step 15.1: Master Script

```
#!/usr/bin/env python3
"""
Ball Mill Optimization - Complete Implementation
"""

import pandas as pd
import numpy as np
import xgboost as xgb
import optuna
import joblib
from sklearn.model_selection import train_test_split
from sklearn.metrics import r2_score, mean_squared_error
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings('ignore')

class BallMillOptimizer:
    """Complete ball mill optimization system"""

    def __init__(self):
```

```python
        self.models = {}
        self.scalers = {}
        self.bounds = {}
        self.constraints = {}

    def load_data(self, filepath):
        """Load and prepare process data"""
        self.df = pd.read_csv(filepath)
        print(f"Loaded {len(self.df)} records")

        # Define variable categories
        self.MVs = ['ore_feed_rate', 'mill_water_flow', 'sump_water_flow',
'ball_dosage']
        self.CVs = ['motor_power', 'pulp_density', 'pulp_flow',
'hydrocyclone_pressure']
        self.DVs = ['ore_hardness', 'grindability_index']  # Adjust to your data
        self.target = 'plus_200_micron_percentage'

        return self.df

    def prepare_data(self):
        """Clean and prepare data for modeling"""
        # Remove outliers and missing values
        self.df_clean = self.df.dropna()

        # Feature engineering
        self.df_clean['water_to_ore_ratio'] = (
            (self.df_clean['mill_water_flow'] + self.df_clean['sump_water_flow']) /
            self.df_clean['ore_feed_rate']
        )

        print(f"Data prepared: {len(self.df_clean)} clean records")

    def train_all_models(self):
        """Train all process and quality models"""

        # Train process models (MV → CV)
        self._train_process_models()

        # Train quality model (CV → Quality)
        self._train_quality_model()

        print("All models trained successfully!")

    def _train_process_models(self):
        """Train the four process models"""

        # Model 1: All MVs → Motor Power
        X1 = self.df_clean[self.MVs]
        y1 = self.df_clean['motor_power']
        X_train1, X_test1, y_train1, y_test1 = train_test_split(X1, y1,
test_size=0.2, random_state=42)

        self.models['power'] = xgb.XGBRegressor(n_estimators=200, max_depth=6,
random_state=42)
        self.models['power'].fit(X_train1, y_train1)
```

```
        # Evaluate
        y_pred1 = self.models['power'].predict(X_test1)
        print(f"Power Model R²: {r2_score(y_test1, y_pred1):.4f}")

        # Models 2,3,4: MVs (excluding balls) → Density, Flow, Pressure
        for cv_name, cv_col in [('density', 'pulp_density'),
                                ('flow', 'pulp_flow'),
                                ('pressure', 'hydrocyclone_pressure')]:

            X = self.df_clean[['ore_feed_rate', 'mill_water_flow',
'sump_water_flow']]
            y = self.df_clean[cv_col]
            X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

            self.models[cv_name] = xgb.XGBRegressor(n_estimators=200, max_depth=6,
random_state=42)
            self.models[cv_name].fit(X_train, y_train)

            y_pred = self.models[cv_name].predict(X_test)
            print(f"{cv_name.title()} Model R²: {r2_score(y_test, y_pred):.4f}")

    def _train_quality_model(self):
        """Train the main quality model using real CVs"""

        X_quality = self.df_clean[self.CVs]
        if hasattr(self, 'DVs') and self.DVs:
            X_quality = pd.concat([X_quality, self.df_clean[self.DVs]], axis=1)

        y_quality = self.df_clean[self.target]

        X_train_q, X_test_q, y_train_q, y_test_q = train_test_split(
            X_quality, y_quality, test_size=0.2, random_state=42
        )

        self.models['quality'] = xgb.XGBRegressor(
            n_estimators=500,
            max_depth=8,
            learning_rate=0.05,
            subsample=0.8,
            colsample_bytree=0.8,
            reg_alpha=0.1,
            reg_lambda=0.1,
            random_state=42
        )

        self.models['quality'].fit(X_train_q, y_train_q)

        # Evaluate
        y_pred_q = self.models['quality'].predict(X_test_q)
        quality_r2 = r2_score(y_test_q, y_pred_q)
        print(f"Quality Model R²: {quality_r2:.4f}")

        # Store for later reference
        self.quality_r2 = quality_r2

    def predict_cvs_from_mvs(self, ore_feed, mill_water, sump_water, ball_dosage):
```

```
        """Predict all CVs from given MVs"""

        mvs_full = np.array([[ore_feed, mill_water, sump_water, ball_dosage]])
        mvs_partial = np.array([[ore_feed, mill_water, sump_water]])

        predicted_cvs = {
            'motor_power': self.models['power'].predict(mvs_full)[0],
            'pulp_density': self.models['density'].predict(mvs_partial)[0],
            'pulp_flow': self.models['flow'].predict(mvs_partial)[0],
            'hydrocyclone_pressure': self.models['pressure'].predict(mvs_partial)
[0]
        }

        return predicted_cvs

    def optimize(self, n_trials=1000, current_ore_quality=None):
        """Run Bayesian optimization to find optimal MVs"""

        def objective(trial):
            # Sample MVs
            ore_feed = trial.suggest_float('ore_feed_rate', 50, 150)
            mill_water = trial.suggest_float('mill_water_flow', 10, 50)
            sump_water = trial.suggest_float('sump_water_flow', 5, 30)
            ball_dosage = trial.suggest_float('ball_dosage', 0.5, 2.0)

            # Predict CVs
            predicted_cvs = self.predict_cvs_from_mvs(ore_feed, mill_water,
sump_water, ball_dosage)

            # Check constraints
            constraints_met = all(
                CV_CONSTRAINTS[cv_name][0] <= cv_value <= CV_CONSTRAINTS[cv_name]
[1]
                for cv_name, cv_value in predicted_cvs.items()
            )

            if not constraints_met:
                return 100.0  # High penalty

            # Predict quality
            cv_array = np.array(list(predicted_cvs.values()))
            if current_ore_quality is not None:
                cv_array = np.concatenate([cv_array, current_ore_quality])

            predicted_quality = self.models['quality'].predict([cv_array])[0]

            return predicted_quality

        # Run optimization
        study = optuna.create_study(direction='minimize')
        study.optimize(objective, n_trials=n_trials)

        self.optimal_params = study.best_params
        self.optimal_quality = study.best_value

        print(f"Optimization completed!")
        print(f"Best +200μm fraction: {self.optimal_quality:.2f}%")
```

```python
        print("Optimal parameters:")
        for param, value in self.optimal_params.items():
            print(f"  {param}: {value:.2f}")

        return study

# Usage example
if __name__ == "__main__":
    # Initialize optimizer
    optimizer = BallMillOptimizer()

    # Load and prepare data
    optimizer.load_data('your_ball_mill_data.csv')
    optimizer.prepare_data()

    # Train models
    optimizer.train_all_models()

    # Run optimization
    study = optimizer.optimize(n_trials=1000)

    # Analyze results
    print(f"\nPotential improvement: {optimizer.optimal_quality:.2f}% +200μm
fraction")
```

---

16. Next Steps and Recommendations

Phase 1: Model Development (Weeks 1-2)

1. **Data preparation**: Clean historical data and engineer features
2. **Model training**: Build and validate all 5 models
3. **Initial testing**: Validate complete prediction chain

Phase 2: Optimization Setup (Week 3)

1. **Constraint definition**: Set realistic bounds and constraints
2. **Objective function**: Implement and test optimization objective
3. **Small-scale testing**: Run optimization on subset of conditions

Phase 3: Implementation (Weeks 4-6)

1. **Gradual rollout**: Implement changes in small steps

2. **Performance monitoring**: Track actual vs. predicted results
3. **Model updates**: Retrain models as needed based on new data

Phase 4: Continuous Improvement (Ongoing)

1. **Regular validation**: Monthly model performance checks
2. **Constraint updates**: Adjust bounds based on operational experience
3. **Advanced features**: Add seasonal effects, equipment wear models

---

17. Key Success Factors

Technical Factors

- **Data Quality**: Ensure sensor data is reliable and well-calibrated
- **Model Validation**: Thoroughly test models before implementation
- **Constraint Accuracy**: Realistic bounds prevent unsafe operation
- **Gradual Implementation**: Small changes reduce risk

Operational Factors

- **Operator Training**: Ensure operators understand new setpoints
- **Process Monitoring**: Watch for unexpected behavior during changes
- **Backup Plans**: Have procedures to return to previous settings
- **Regular Reviews**: Weekly assessment of optimization performance

Risk Mitigation

- **Model Uncertainty**: Use confidence intervals for predictions
- **Equipment Protection**: Hard constraints on motor power and pressure
- **Process Stability**: Avoid rapid changes in setpoints
- **Quality Assurance**: Continuous monitoring of +200 μm measurements

---

18. Advanced Topics

Step 18.1: Uncertainty Quantification

Add confidence intervals to your predictions:

```
from sklearn.ensemble import RandomForestRegressor
import scipy.stats as stats

class UncertaintyQuantification:
    """Add uncertainty estimates to predictions"""

    def __init__(self, n_bootstrap=100):
        self.n_bootstrap = n_bootstrap
        self.bootstrap_models = {}

    def train_bootstrap_models(self, X, y, model_name):
        """Train multiple models on bootstrap samples"""

        bootstrap_models = []

        for i in range(self.n_bootstrap):
            # Create bootstrap sample
            indices = np.random.choice(len(X), len(X), replace=True)
            X_boot = X.iloc[indices] if hasattr(X, 'iloc') else X[indices]
            y_boot = y.iloc[indices] if hasattr(y, 'iloc') else y[indices]

            # Train model on bootstrap sample
            model = xgb.XGBRegressor(n_estimators=100, max_depth=6, random_state=i)
            model.fit(X_boot, y_boot)
            bootstrap_models.append(model)

        self.bootstrap_models[model_name] = bootstrap_models

    def predict_with_uncertainty(self, X, model_name):
        """Predict with confidence intervals"""

        predictions = []
        for model in self.bootstrap_models[model_name]:
            pred = model.predict(X)
            predictions.append(pred)

        predictions = np.array(predictions)

        mean_pred = np.mean(predictions, axis=0)
        std_pred = np.std(predictions, axis=0)

        # 95% confidence intervals
        ci_lower = mean_pred - 1.96 * std_pred
        ci_upper = mean_pred + 1.96 * std_pred
```

```
        return mean_pred, ci_lower, ci_upper

# Example usage
uncertainty_estimator = UncertaintyQuantification()

# Train uncertainty models for quality prediction
X_quality = df_clean[CVs]
y_quality = df_clean[target]
uncertainty_estimator.train_bootstrap_models(X_quality, y_quality, 'quality')

# Make prediction with uncertainty
test_cvs = X_quality.iloc[:5]  # First 5 samples
mean_pred, ci_lower, ci_upper =
uncertainty_estimator.predict_with_uncertainty(test_cvs, 'quality')

print("Predictions with 95% Confidence Intervals:")
for i in range(len(mean_pred)):
    print(f"Sample {i+1}: {mean_pred[i]:.2f}% [{ci_lower[i]:.2f}%,
{ci_upper[i]:.2f}%]")
```

Step 18.2: Online Learning and Model Updates

Implement automatic model updates as new data becomes available:

```
class OnlineLearningSystem:
    """System for continuous model improvement"""

    def __init__(self, optimizer):
        self.optimizer = optimizer
        self.update_frequency = 168  # Hours (1 week)
        self.performance_threshold = 0.05  # 5% R² decrease triggers update

    def should_update_model(self, model_name, new_data):
        """Check if model needs updating based on recent performance"""

        if model_name == 'quality':
            X_new = new_data[self.optimizer.CVs]
            y_new = new_data[self.optimizer.target]
            y_pred = self.optimizer.models['quality'].predict(X_new)
        else:
            # Handle process models
            if model_name == 'power':
                X_new = new_data[self.optimizer.MVs]
                y_new = new_data['motor_power']
            elif model_name == 'density':
                X_new = new_data[['ore_feed_rate', 'mill_water_flow',
'sump_water_flow']]
                y_new = new_data['pulp_density']
            # ... similar for flow and pressure

            y_pred = self.optimizer.models[model_name].predict(X_new)
```

```
        # Calculate current performance
        current_r2 = r2_score(y_new, y_pred)

        # Compare with original performance (you need to store this during
training)
        original_r2 = getattr(self.optimizer, f'{model_name}_r2', 0.9)  # Default
if not stored

        performance_drop = original_r2 - current_r2

        return performance_drop > self.performance_threshold

    def update_model(self, model_name, new_data, combine_ratio=0.3):
        """Update specific model with new data"""

        print(f"Updating {model_name} model...")

        # Combine old and new data (weighted toward new data)
        if model_name == 'quality':
            X_old = self.optimizer.df_clean[self.optimizer.CVs]
            y_old = self.optimizer.df_clean[self.optimizer.target]
            X_new = new_data[self.optimizer.CVs]
            y_new = new_data[self.optimizer.target]
        # ... handle other models similarly

        # Create combined dataset
        n_old_samples = int(len(X_old) * (1 - combine_ratio))
        n_new_samples = len(X_new)

        X_old_sample = X_old.sample(n_old_samples, random_state=42)
        y_old_sample = y_old.loc[X_old_sample.index]

        X_combined = pd.concat([X_old_sample, X_new])
        y_combined = pd.concat([y_old_sample, y_new])

        # Retrain model
        self.optimizer.models[model_name].fit(X_combined, y_combined)

        # Validate updated model
        y_pred_new = self.optimizer.models[model_name].predict(X_new)
        new_r2 = r2_score(y_new, y_pred_new)

        print(f"Updated {model_name} model R²: {new_r2:.4f}")

        # Save updated model
        joblib.dump(self.optimizer.models[model_name],
f'{model_name}_model_updated.pkl')

# Usage
online_system = OnlineLearningSystem(optimizer)

# Check if update needed (would run periodically)
new_week_data = df_clean.tail(168)  # Last week of hourly data
if online_system.should_update_model('quality', new_week_data):
    online_system.update_model('quality', new_week_data)
```

Step 18.3: Advanced Constraint Handling

Implement dynamic constraints that adapt to operating conditions:

```
class DynamicConstraints:
    """Handle time-varying and condition-dependent constraints"""

    def __init__(self):
        self.seasonal_factors = {}
        self.equipment_wear_models = {}

    def calculate_dynamic_bounds(self, current_conditions):
        """Calculate bounds based on current operating conditions"""

        # Example: Adjust power limits based on mill liner wear
        liner_wear_factor = current_conditions.get('liner_wear_percent', 0) / 100
        max_power_adjusted = 1200 * (1 - 0.2 * liner_wear_factor)  # Reduce max power as liner wears

        # Example: Adjust water limits based on ore moisture
        ore_moisture = current_conditions.get('ore_moisture_percent', 8)
        water_adjustment = (ore_moisture - 8) * 0.1  # Adjust water for moisture variation

        dynamic_bounds = {
            'ore_feed_rate': (50, 150),
            'mill_water_flow': (max(10, 30 - water_adjustment), min(50, 40 + water_adjustment)),
            'sump_water_flow': (5, 30),
            'ball_dosage': (0.5, 2.0)
        }

        dynamic_constraints = {
            'motor_power': (500, max_power_adjusted),
            'pulp_density': (1.2, 1.6),
            'pulp_flow': (80, 200),
            'hydrocyclone_pressure': (1.0, 3.0)
        }

        return dynamic_bounds, dynamic_constraints

# Integration with optimizer
def optimize_with_dynamic_constraints(optimizer, current_conditions, n_trials=1000):
    """Run optimization with dynamic constraints"""

    constraint_handler = DynamicConstraints()
    dynamic_bounds, dynamic_constraints = constraint_handler.calculate_dynamic_bounds(current_conditions)

    def dynamic_objective(trial):
        # Use dynamic bounds for sampling
        ore_feed = trial.suggest_float('ore_feed_rate', *dynamic_bounds['ore_feed_rate'])
```

```
        mill_water = trial.suggest_float('mill_water_flow',
*dynamic_bounds['mill_water_flow'])
        sump_water = trial.suggest_float('sump_water_flow',
*dynamic_bounds['sump_water_flow'])
        ball_dosage = trial.suggest_float('ball_dosage',
*dynamic_bounds['ball_dosage'])

        # Predict and check dynamic constraints
        predicted_cvs = optimizer.predict_cvs_from_mvs(ore_feed, mill_water,
sump_water, ball_dosage)

        constraints_met = all(
            dynamic_constraints[cv_name][0] <= cv_value <=
dynamic_constraints[cv_name][1]
            for cv_name, cv_value in predicted_cvs.items()
        )

        if not constraints_met:
            return 100.0

        # Predict quality
        cv_array = np.array(list(predicted_cvs.values()))
        predicted_quality = optimizer.models['quality'].predict([cv_array])[0]

        return predicted_quality

    study = optuna.create_study(direction='minimize')
    study.optimize(dynamic_objective, n_trials=n_trials)

    return study
```

---

19. Deployment and Production Monitoring

Step 19.1: Production Deployment Script

```
class ProductionDeployment:
    """Handle production deployment of optimized settings"""

    def __init__(self, optimizer):
        self.optimizer = optimizer
        self.deployment_log = []

    def validate_before_deployment(self, optimal_mvs):
        """Final validation before implementing optimal settings"""

        # Predict expected outcomes
```

```
        predicted_quality, predicted_cvs, feasible = predict_quality_from_mvs(
            optimal_mvs['ore_feed_rate'],
            optimal_mvs['mill_water_flow'],
            optimal_mvs['sump_water_flow'],
            optimal_mvs['ball_dosage']
        )

        validation_checks = {
            'feasible': feasible,
            'quality_improvement': predicted_quality < df_clean[target].mean(),
            'power_reasonable': 500 <= predicted_cvs['motor_power'] <= 1200,
            'density_reasonable': 1.2 <= predicted_cvs['pulp_density'] <= 1.6,
            'flow_reasonable': 80 <= predicted_cvs['pulp_flow'] <= 200,
            'pressure_reasonable': 1.0 <= predicted_cvs['hydrocyclone_pressure'] <=
3.0
        }

        all_checks_passed = all(validation_checks.values())

        print("Pre-deployment Validation:")
        for check, passed in validation_checks.items():
            status = "✅ PASS" if passed else "❌ FAIL"
            print(f"  {check}: {status}")

        if all_checks_passed:
            print("\n🚀 All validations passed - Ready for deployment!")
        else:
            print("\n⚠️ Some validations failed - Review before deployment")

        return all_checks_passed, validation_checks

    def create_deployment_schedule(self, current_mvs, optimal_mvs,
transition_hours=24):
        """Create hour-by-hour deployment schedule"""

        schedule = []
        n_steps = transition_hours

        for hour in range(n_steps + 1):
            alpha = hour / n_steps

            step_mvs = {}
            for mv in ['ore_feed_rate', 'mill_water_flow', 'sump_water_flow',
'ball_dosage']:
                current_val = current_mvs.get(mv, optimal_mvs[mv])
                step_mvs[mv] = current_val + alpha * (optimal_mvs[mv] -
current_val)

            # Predict outcomes for this step
            quality, cvs, feasible = predict_quality_from_mvs(**step_mvs)

            schedule.append({
                'hour': hour,
                'ore_feed_rate': step_mvs['ore_feed_rate'],
                'mill_water_flow': step_mvs['mill_water_flow'],
                'sump_water_flow': step_mvs['sump_water_flow'],
                'ball_dosage': step_mvs['ball_dosage'],
```

```
                'predicted_quality': quality,
                'predicted_power': cvs['motor_power'],
                'feasible': feasible
            })

        schedule_df = pd.DataFrame(schedule)
        print("24-Hour Deployment Schedule Created")
        print(schedule_df.round(2))

        return schedule_df

    def log_deployment_step(self, hour, actual_mvs, actual_cvs, actual_quality):
        """Log each deployment step for tracking"""

        self.deployment_log.append({
            'timestamp': pd.Timestamp.now(),
            'hour': hour,
            'actual_mvs': actual_mvs,
            'actual_cvs': actual_cvs,
            'actual_quality': actual_quality
        })

        print(f"Hour {hour} logged - Quality: {actual_quality:.2f}%")

# Example deployment
deployment_system = ProductionDeployment(optimizer)
ready_to_deploy, checks =
deployment_system.validate_before_deployment(optimal_params)

if ready_to_deploy:
    current_operation = df_clean[MVs].tail(10).mean().to_dict()
    deployment_schedule = deployment_system.create_deployment_schedule(
        current_operation, optimal_params, transition_hours=24
    )
```

Step 19.2: Real-Time Performance Dashboard

```
import plotly.graph_objects as go
from plotly.subplots import make_subplots
import plotly.express as px

class RealTimeDashboard:
    """Create real-time monitoring dashboard"""

    def __init__(self):
        self.performance_data = []

    def update_dashboard(self, current_mvs, current_cvs, current_quality,
predicted_quality):
        """Update dashboard with new measurements"""

        # Add new data point
```

```
        self.performance_data.append({
            'timestamp': pd.Timestamp.now(),
            'actual_quality': current_quality,
            'predicted_quality': predicted_quality,
            'prediction_error': abs(current_quality - predicted_quality),
            'ore_feed': current_mvs['ore_feed_rate'],
            'motor_power': current_cvs['motor_power'],
            'pulp_density': current_cvs['pulp_density']
        })

        # Keep only last 100 points for display
        if len(self.performance_data) > 100:
            self.performance_data = self.performance_data[-100:]

    def create_dashboard(self):
        """Create interactive dashboard"""

        df_dash = pd.DataFrame(self.performance_data)

        # Create subplots
        fig = make_subplots(
            rows=3, cols=2,
            subplot_titles=['Quality Tracking', 'Prediction Accuracy',
                          'Ore Feed Rate', 'Motor Power',
                          'Pulp Density', 'Overall Performance'],
            specs=[[{"secondary_y": False}, {"secondary_y": False}],
                   [{"secondary_y": False}, {"secondary_y": False}],
                   [{"secondary_y": False}, {"secondary_y": False}]]
        )

        # Quality tracking
        fig.add_trace(
            go.Scatter(x=df_dash['timestamp'], y=df_dash['actual_quality'],
                      name='Actual Quality', line=dict(color='blue')),
            row=1, col=1
        )
        fig.add_trace(
            go.Scatter(x=df_dash['timestamp'], y=df_dash['predicted_quality'],
                      name='Predicted Quality', line=dict(color='red',
dash='dash')),
            row=1, col=1
        )

        # Prediction error
        fig.add_trace(
            go.Scatter(x=df_dash['timestamp'], y=df_dash['prediction_error'],
                      name='Prediction Error', line=dict(color='orange')),
            row=1, col=2
        )

        # Process variables
        fig.add_trace(
            go.Scatter(x=df_dash['timestamp'], y=df_dash['ore_feed'],
                      name='Ore Feed Rate', line=dict(color='green')),
            row=2, col=1
        )
```

```
        fig.add_trace(
            go.Scatter(x=df_dash['timestamp'], y=df_dash['motor_power'],
                      name='Motor Power', line=dict(color='purple')),
            row=2, col=2
        )

        fig.add_trace(
            go.Scatter(x=df_dash['timestamp'], y=df_dash['pulp_density'],
                      name='Pulp Density', line=dict(color='brown')),
            row=3, col=1
        )

        # Overall performance metric
        rolling_quality = df_dash['actual_quality'].rolling(window=10).mean()
        fig.add_trace(
            go.Scatter(x=df_dash['timestamp'], y=rolling_quality,
                      name='10-Point Average Quality', line=dict(color='black')),
            row=3, col=2
        )

        fig.update_layout(height=800, title="Ball Mill Optimization Dashboard")
        fig.show()

        return fig

# Initialize dashboard
dashboard = RealTimeDashboard()

# Example of updating dashboard (would be called periodically)
def simulate_real_time_update():
    """Simulate real-time dashboard updates"""

    for i in range(20):  # Simulate 20 time points
        # Simulate current measurements (replace with actual sensor readings)
        current_mvs = {
            'ore_feed_rate': optimal_params['ore_feed_rate'] + np.random.normal(0,
5),
            'mill_water_flow': optimal_params['mill_water_flow'] +
np.random.normal(0, 2),
            'sump_water_flow': optimal_params['sump_water_flow'] +
np.random.normal(0, 1),
            'ball_dosage': optimal_params['ball_dosage'] + np.random.normal(0, 0.1)
        }

        # Predict what we expect
        predicted_quality, predicted_cvs, _ =
predict_quality_from_mvs(**current_mvs)

        # Simulate actual measurements (with some noise)
        actual_quality = predicted_quality + np.random.normal(0, 1.0)
        actual_cvs = {k: v + np.random.normal(0, v*0.05) for k, v in
predicted_cvs.items()}

        # Update dashboard
        dashboard.update_dashboard(current_mvs, actual_cvs, actual_quality,
predicted_quality)
```

```
    # Run simulation
    simulate_real_time_update()
    dashboard_fig = dashboard.create_dashboard()
```

---

20. Quality Assurance and Testing

Step 20.1: Model Testing Framework

```
class ModelTestingSuite:
    """Comprehensive testing for all models"""

    def __init__(self, optimizer):
        self.optimizer = optimizer
        self.test_results = {}

    def test_model_consistency(self):
        """Test if models give consistent results across multiple runs"""

        # Fixed test inputs
        test_mvs = np.array([[100, 25, 15, 1.0]])  # Standard operating point

        # Run prediction multiple times
        predictions = []
        for _ in range(10):
            pred_quality, pred_cvs, _ = predict_quality_from_mvs(*test_mvs[0])
            predictions.append(pred_quality)

        consistency_std = np.std(predictions)

        print(f"Model Consistency Test:")
        print(f"  Standard deviation across 10 runs: {consistency_std:.4f}%")
        print(f"  Consistency: {'✅ GOOD' if consistency_std < 0.1 else '⚠️
CHECK'}")

        return consistency_std

    def test_edge_cases(self):
        """Test model behavior at operating boundaries"""

        edge_cases = [
            {'name': 'Minimum Feed', 'mvs': [50, 25, 15, 1.0]},
            {'name': 'Maximum Feed', 'mvs': [150, 25, 15, 1.0]},
            {'name': 'Minimum Water', 'mvs': [100, 10, 5, 1.0]},
            {'name': 'Maximum Water', 'mvs': [100, 50, 30, 1.0]},
            {'name': 'Minimum Balls', 'mvs': [100, 25, 15, 0.5]},
            {'name': 'Maximum Balls', 'mvs': [100, 25, 15, 2.0]}
        ]

        print("Edge Case Testing:")
```

```
        for case in edge_cases:
            quality, cvs, feasible = predict_quality_from_mvs(*case['mvs'])

            print(f"  {case['name']}:")
            print(f"    Quality: {quality:.2f}%, Feasible: {feasible}")
            print(f"    Power: {cvs['motor_power']:.0f} kW, Density:
{cvs['pulp_density']:.2f} kg/L")

    def test_physical_relationships(self):
        """Test if models respect known physical relationships"""

        # Test 1: Higher ore feed should increase motor power
        low_feed_power = predict_quality_from_mvs(80, 25, 15, 1.0)[1]
['motor_power']
        high_feed_power = predict_quality_from_mvs(120, 25, 15, 1.0)[1]
['motor_power']

        feed_power_test = high_feed_power > low_feed_power
        print(f"Feed-Power Relationship: {'✅ PASS' if feed_power_test else '❌
FAIL'}")
        print(f"  Low feed power: {low_feed_power:.0f} kW")
        print(f"  High feed power: {high_feed_power:.0f} kW")

        # Test 2: More water should decrease pulp density
        low_water_density = predict_quality_from_mvs(100, 15, 10, 1.0)[1]
['pulp_density']
        high_water_density = predict_quality_from_mvs(100, 35, 25, 1.0)[1]
['pulp_density']

        water_density_test = high_water_density < low_water_density
        print(f"Water-Density Relationship: {'✅ PASS' if water_density_test else
'❌ FAIL'}")
        print(f"  Low water density: {low_water_density:.2f} kg/L")
        print(f"  High water density: {high_water_density:.2f} kg/L")

        # Test 3: More balls should increase motor power
        low_balls_power = predict_quality_from_mvs(100, 25, 15, 0.8)[1]
['motor_power']
        high_balls_power = predict_quality_from_mvs(100, 25, 15, 1.5)[1]
['motor_power']

        balls_power_test = high_balls_power > low_balls_power
        print(f"Balls-Power Relationship: {'✅ PASS' if balls_power_test else '❌
FAIL'}")
        print(f"  Low balls power: {low_balls_power:.0f} kW")
        print(f"  High balls power: {high_balls_power:.0f} kW")

        return all([feed_power_test, water_density_test, balls_power_test])

# Run comprehensive testing
testing_suite = ModelTestingSuite(optimizer)
consistency_result = testing_suite.test_model_consistency()
testing_suite.test_edge_cases()
physics_check = testing_suite.test_physical_relationships()
```

Step 20.2: Automated Report Generation

```
def generate_optimization_report(optimizer, study, optimal_params):
    """Generate comprehensive optimization report"""

    report = f"""
# Ball Mill Optimization Report
Generated: {pd.Timestamp.now().strftime('%Y-%m-%d %H:%M:%S')}

## Executive Summary
- **Optimization Objective**: Minimize +200 μm pulp fraction
- **Baseline Performance**: {df_clean[target].mean():.2f}% ±
{df_clean[target].std():.2f}%
- **Optimized Performance**: {optimizer.optimal_quality:.2f}%
- **Expected Improvement**: {((df_clean[target].mean() - optimizer.optimal_quality)
/ df_clean[target].mean() * 100):.1f}%

## Optimal Operating Parameters
"""

    for param, value in optimal_params.items():
        unit_map = {
            'ore_feed_rate': 't/h',
            'mill_water_flow': 'm³/h',
            'sump_water_flow': 'm³/h',
            'ball_dosage': 't/h'
        }
        unit = unit_map.get(param, '')
        report += f"- **{param.replace('_', ' ').title()}**: {value:.2f} {unit}\n"

    # Add predicted process conditions
    opt_quality, opt_cvs, _ = predict_quality_from_mvs(**optimal_params)

    report += f"""
## Predicted Process Conditions
- **Motor Power**: {opt_cvs['motor_power']:.0f} kW
- **Pulp Density**: {opt_cvs['pulp_density']:.2f} kg/L
- **Pulp Flow**: {opt_cvs['pulp_flow']:.0f} m³/h
- **Hydrocyclone Pressure**: {opt_cvs['hydrocyclone_pressure']:.2f} bar

## Model Performance Summary
"""

    for model_name in ['power', 'density', 'flow', 'pressure', 'quality']:
        if model_name in performance_metrics['model_accuracies']:
            r2 = performance_metrics['model_accuracies'][f'{model_name}_model_r2']
            report += f"- **{model_name.title()} Model R²**: {r2:.4f}\n"

    report += f"""
## Risk Assessment
- **Constraint Compliance**: All optimal settings within safe operating limits
- **Model Confidence**: Quality model R² = {optimizer.quality_r2:.4f}
- **Improvement Confidence**: {'High' if optimizer.quality_r2 > 0.85 else 'Medium'
if optimizer.quality_r2 > 0.75 else 'Low'}
```

```
## Implementation Recommendations
1. **Gradual Transition**: Implement changes over 24-48 hours
2. **Close Monitoring**: Track actual vs predicted performance hourly
3. **Rollback Plan**: Return to previous settings if quality degrades
4. **Model Updates**: Retrain models monthly with new data

## Next Steps
1. Validate predictions with plant trial
2. Implement gradual transition plan
3. Setup continuous monitoring system
4. Schedule first model update review
"""

    # Save report
    with
open(f'optimization_report_{pd.Timestamp.now().strftime("%Y%m%d_%H%M")}.md', 'w')
as f:
        f.write(report)

    print("Optimization report generated and saved!")
    return report

# Generate final report
optimization_report = generate_optimization_report(optimizer, optimization_study,
optimal_params)
print(optimization_report[:500] + "...")  # Print first 500 characters
```

21. Long-Term Strategy

Step 21.1: Seasonal and Cyclical Optimization

```
class SeasonalOptimization:
    """Handle seasonal variations in ore properties and market conditions"""

    def __init__(self, optimizer):
        self.optimizer = optimizer
        self.seasonal_models = {}

    def analyze_seasonal_patterns(self, df_with_dates):
        """Analyze how optimal settings vary by season"""

        # Add time features
        df_with_dates['month'] =
pd.to_datetime(df_with_dates['timestamp']).dt.month
        df_with_dates['season'] = df_with_dates['month'].map({
            12: 'Winter', 1: 'Winter', 2: 'Winter',
            3: 'Spring', 4: 'Spring', 5: 'Spring',
            6: 'Summer', 7: 'Summer', 8: 'Summer',
```

```
            9: 'Fall', 10: 'Fall', 11: 'Fall'
        })

        # Analyze seasonal differences
        seasonal_summary = df_with_dates.groupby('season').agg({
            'plus_200_micron_percentage': ['mean', 'std'],
            'ore_hardness': 'mean',
            'motor_power': 'mean',
            'ore_feed_rate': 'mean'
        }).round(2)

        print("Seasonal Analysis:")
        print(seasonal_summary)

        return seasonal_summary

    def optimize_by_season(self, season, ore_quality_for_season):
        """Run optimization for specific season"""

        def seasonal_objective(trial):
            mvs = [
                trial.suggest_float('ore_feed_rate', *MV_BOUNDS['ore_feed_rate']),
                trial.suggest_float('mill_water_flow',
*MV_BOUNDS['mill_water_flow']),
                trial.suggest_float('sump_water_flow',
*MV_BOUNDS['sump_water_flow']),
                trial.suggest_float('ball_dosage', *MV_BOUNDS['ball_dosage'])
            ]

            quality, cvs, feasible = predict_quality_from_mvs(*mvs,
ore_quality_for_season)

            if not feasible:
                return 100.0

            # Add seasonal adjustments
            if season == 'Summer':
                # Higher water costs in summer
                water_penalty = (mvs[1] + mvs[2] - 30) * 0.05 if (mvs[1] + mvs[2])
> 30 else 0
                quality += water_penalty
            elif season == 'Winter':
                # Higher energy costs in winter
                power_penalty = (cvs['motor_power'] - 900) * 0.002 if
cvs['motor_power'] > 900 else 0
                quality += power_penalty

            return quality

        study = optuna.create_study(direction='minimize')
        study.optimize(seasonal_objective, n_trials=500)

        self.seasonal_models[season] = study.best_params

        print(f"Seasonal optimization for {season} completed:")
        print(f"  Best quality: {study.best_value:.2f}%")
        print(f"  Optimal settings: {study.best_params}")
```

```python
        return study

# Example seasonal optimization
seasonal_optimizer = SeasonalOptimization(optimizer)

# You would run this for each season with appropriate ore quality data
# seasonal_study = seasonal_optimizer.optimize_by_season('Summer',
summer_ore_quality)
```

Step 21.2: Economic Optimization

```python
class EconomicOptimization:
    """Incorporate economic factors into optimization"""

    def __init__(self, cost_parameters):
        self.costs = cost_parameters

    def calculate_operating_cost(self, mvs, cvs):
        """Calculate total operating cost per hour"""

        # Energy cost
        energy_cost = cvs['motor_power'] * self.costs['electricity_rate']  # $/kWh

        # Water cost
        water_cost = (mvs['mill_water_flow'] + mvs['sump_water_flow']) *
self.costs['water_rate']  # $/m³

        # Ball consumption cost
        ball_cost = mvs['ball_dosage'] * self.costs['ball_price']  # $/t

        # Reagent cost (simplified - based on ore throughput)
        reagent_cost = mvs['ore_feed_rate'] * self.costs['reagent_rate']  # $/t ore

        total_cost = energy_cost + water_cost + ball_cost + reagent_cost

        return {
            'total_cost': total_cost,
            'energy_cost': energy_cost,
            'water_cost': water_cost,
            'ball_cost': ball_cost,
            'reagent_cost': reagent_cost
        }

    def calculate_revenue_impact(self, quality_improvement, throughput):
        """Calculate revenue impact of quality improvement"""

        # Revenue from reduced reprocessing
        reprocessing_savings = quality_improvement * throughput *
self.costs['reprocessing_cost']  # $/t

        # Revenue from higher recovery in flotation
```

```
        recovery_improvement = quality_improvement * 0.5  # Assume 0.5% recovery
improvement per 1% quality improvement
        additional_revenue = recovery_improvement * throughput *
self.costs['concentrate_value']  # $/t

        total_revenue_impact = reprocessing_savings + additional_revenue

        return total_revenue_impact

    def economic_objective(self, trial):
        """Objective function that considers both quality and economics"""

        # Sample MVs
        mvs = {
            'ore_feed_rate': trial.suggest_float('ore_feed_rate', 50, 150),
            'mill_water_flow': trial.suggest_float('mill_water_flow', 10, 50),
            'sump_water_flow': trial.suggest_float('sump_water_flow', 5, 30),
            'ball_dosage': trial.suggest_float('ball_dosage', 0.5, 2.0)
        }

        # Predict process outcomes
        quality, cvs, feasible = predict_quality_from_mvs(**mvs)

        if not feasible:
            return 10000.0  # High cost penalty

        # Calculate costs
        cost_breakdown = self.calculate_operating_cost(mvs, cvs)

        # Calculate revenue impact
        baseline_quality = df_clean[target].mean()
        quality_improvement = max(0, baseline_quality - quality)  # % improvement
        throughput = mvs['ore_feed_rate']  # t/h

        revenue_impact = self.calculate_revenue_impact(quality_improvement,
throughput)

        # Net economic benefit (negative for maximization)
        net_benefit = revenue_impact - cost_breakdown['total_cost']

        return -net_benefit  # Minimize negative benefit = maximize benefit

# Example cost parameters (adjust to your operation)
cost_params = {
    'electricity_rate': 0.12,      # $/kWh
    'water_rate': 0.50,           # $/m³
    'ball_price': 800,            # $/t
    'reagent_rate': 2.50,         # $/t ore processed
    'reprocessing_cost': 5.00,    # $/t for reprocessing +200μm material
    'concentrate_value': 500      # $/t concentrate
}

economic_optimizer = EconomicOptimization(cost_params)

# Run economic optimization
def run_economic_optimization():
    study = optuna.create_study(direction='minimize')  # Minimize negative profit =
```

```
maximize profit
    study.optimize(economic_optimizer.economic_objective, n_trials=1000)

    print("Economic Optimization Results:")
    print(f"Best economic benefit: ${-study.best_value:.2f}/hour")
    print("Economically optimal parameters:")
    for param, value in study.best_params.items():
        print(f"  {param}: {value:.2f}")

    return study

# Uncomment to run economic optimization
# economic_study = run_economic_optimization()
```

---

22. Documentation and Knowledge Transfer

Step 22.1: Technical Documentation

```
def generate_technical_documentation():
    """Generate comprehensive technical documentation"""

    tech_doc = f"""
# Ball Mill Optimization - Technical Documentation

## Model Architecture

### Process Models (MV → CV)
1. **Power Model**: f(ore_feed, mill_water, sump_water, ball_dosage) → motor_power
   - Purpose: Predict energy consumption
   - Input features: All 4 manipulated variables
   - Algorithm: XGBoost Regressor
   - Performance: R² = {performance_metrics['model_accuracies']
['power_model_r2']:.4f}

2. **Density Model**: f(ore_feed, mill_water, sump_water) → pulp_density
   - Purpose: Predict solid-liquid ratio
   - Input features: Feed rate and water flows (balls excluded)
   - Algorithm: XGBoost Regressor
   - Performance: R² = {performance_metrics['model_accuracies']
['density_model_r2']:.4f}

3. **Flow Model**: f(ore_feed, mill_water, sump_water) → pulp_flow
   - Purpose: Predict volumetric throughput
   - Input features: Feed rate and water flows
   - Algorithm: XGBoost Regressor
   - Performance: R² = {performance_metrics['model_accuracies']
```

```
['flow_model_r2']:.4f}

4. **Pressure Model**: f(ore_feed, mill_water, sump_water) → hydrocyclone_pressure
   - Purpose: Predict hydraulic conditions
   - Input features: Feed rate and water flows
   - Algorithm: XGBoost Regressor
   - Performance: R² = {performance_metrics['model_accuracies']
['pressure_model_r2']:.4f}

### Quality Model (CV → Quality)
- **Purpose**: Predict +200 μm fraction from process conditions
- **Input features**: Motor power, pulp density, flow, pressure + ore quality
- **Algorithm**: XGBoost Regressor (500 trees, depth 8)
- **Performance**: R² = {performance_metrics['model_accuracies']
['quality_model_r2']:.4f}

## Optimization Strategy
- **Method**: Bayesian Optimization with Optuna
- **Search Space**: 4-dimensional MV space
- **Constraints**: Physical equipment limits and process requirements
- **Objective**: Minimize +200 μm fraction percentage

## Operating Constraints
"""

    for mv, bounds in MV_BOUNDS.items():
        tech_doc += f"- **{mv}**: {bounds[0]} - {bounds[1]}\n"

    tech_doc += "\n### Process Constraints\n"
    for cv, bounds in CV_CONSTRAINTS.items():
        tech_doc += f"- **{cv}**: {bounds[0]} - {bounds[1]}\n"

    tech_doc += f"""
## Expected Results
- **Quality Improvement**: {performance_metrics['improvement_percent']:.1f}%
- **Baseline +200μm**: {performance_metrics['baseline_plus200']:.2f}%
- **Optimized +200μm**: {performance_metrics['optimized_plus200']:.2f}%

## Model Limitations
1. **Temporal Scope**: Models trained on data from [DATE_RANGE]
2. **Operating Scope**: Valid within defined MV bounds only
3. **Equipment Condition**: Assumes current equipment wear state
4. **Ore Quality**: Performance may vary with significantly different ore types

## Maintenance Schedule
- **Daily**: Monitor prediction accuracy
- **Weekly**: Check constraint violations
- **Monthly**: Validate model performance on new data
- **Quarterly**: Retrain models if performance degrades
- **Annually**: Complete model architecture review
"""

    return tech_doc

technical_documentation = generate_technical_documentation()
```

Step 22.2: Operator Guidelines

```
def generate_operator_guidelines():
    """Create practical guidelines for plant operators"""

    operator_guide = f"""
# Ball Mill Optimization - Operator Guidelines

## Quick Reference - Optimal Settings

### Target Setpoints
- **Ore Feed Rate**: {optimal_params['ore_feed_rate']:.1f} t/h
- **Mill Water Flow**: {optimal_params['mill_water_flow']:.1f} m³/h
- **Sump Water Flow**: {optimal_params['sump_water_flow']:.1f} m³/h
- **Ball Dosage**: {optimal_params['ball_dosage']:.2f} t/h

### Expected Process Conditions
- **Motor Power**: {opt_cvs['motor_power']:.0f} kW (normal range:
{opt_cvs['motor_power']*0.95:.0f}-{opt_cvs['motor_power']*1.05:.0f} kW)
- **Pulp Density**: {opt_cvs['pulp_density']:.2f} kg/L (normal range:
{opt_cvs['pulp_density']*0.98:.2f}-{opt_cvs['pulp_density']*1.02:.2f} kg/L)
- **Pulp Flow**: {opt_cvs['pulp_flow']:.0f} m³/h (normal range:
{opt_cvs['pulp_flow']*0.95:.0f}-{opt_cvs['pulp_flow']*1.05:.0f} m³/h)
- **Pressure**: {opt_cvs['hydrocyclone_pressure']:.2f} bar (normal range:
{opt_cvs['hydrocyclone_pressure']*0.9:.2f}-
{opt_cvs['hydrocyclone_pressure']*1.1:.2f} bar)

## Implementation Instructions

### Day 1-2: Gradual Transition
1. **Start slowly**: Change only one parameter at a time
2. **Monitor closely**: Check motor power and density every 30 minutes
3. **Document everything**: Record all changes and observations

### What to Watch For

#### ✅ Good Signs
- Motor power stable within expected range
- Pulp density trending toward target
- No unusual vibrations or sounds
- Hydrocyclone operating smoothly

#### ⚠️ Warning Signs
- Motor power > 1200 kW or < 500 kW
- Pulp density < 1.2 or > 1.6 kg/L
- Unusual mill sounds or vibrations
- Hydrocyclone pressure > 3.0 bar

#### 🚨 Stop and Call Engineer
- Motor overload alarms
- Pump cavitation
- Abnormal mill vibration
- Any safety system activation
```

```
## Troubleshooting

### High Motor Power
**Causes**: Too much ore feed, insufficient water, worn balls
**Actions**:
1. Reduce ore feed by 5-10 t/h
2. Increase mill water by 2-3 m³/h
3. Check ball charge level

### Low Pulp Density
**Causes**: Too much water, low ore feed
**Actions**:
1. Reduce total water by 3-5 m³/h
2. Increase ore feed slightly (2-3 t/h)
3. Check for water leaks

### High +200µm in Samples
**Causes**: Insufficient grinding, wrong operating point
**Actions**:
1. Increase ball dosage by 0.1-0.2 t/h
2. Adjust water/ore ratio
3. Check cyclone operation

## Emergency Procedures

### Immediate Rollback Plan
If optimization settings cause problems:

1. **Stop all changes immediately**
2. **Return to previous settings**:
   - Ore feed: [PREVIOUS_VALUE] t/h
   - Mill water: [PREVIOUS_VALUE] m³/h
   - Sump water: [PREVIOUS_VALUE] m³/h
   - Ball dosage: [PREVIOUS_VALUE] t/h
3. **Monitor for 2 hours** until process stabilizes
4. **Contact process engineer** before attempting changes again

### Contact Information
- **Process Engineer**: [NAME] - [PHONE] - [EMAIL]
- **Shift Supervisor**: [NAME] - [PHONE]
- **Maintenance**: [NAME] - [PHONE]

## Daily Checklist

### Start of Shift
- [ ] Check current setpoints match targets
- [ ] Verify all instruments reading normally
- [ ] Review previous shift notes
- [ ] Check +200µm lab results from previous shift

### Every 2 Hours
- [ ] Record motor power, density, flow, pressure
- [ ] Compare with expected ranges
- [ ] Note any adjustments made
- [ ] Check mill sounds and vibration

### End of Shift
```

```
- [ ] Calculate average +200µm for shift
- [ ] Document any problems or observations
- [ ] Update log with final setpoints
- [ ] Brief incoming shift on status

## Performance Tracking

### Key Metrics to Track
- **Primary**: +200µm percentage (target: < {optimal_quality:.1f}%)
- **Secondary**: Motor power utilization, water consumption
- **Tertiary**: Equipment availability, process stability

### When to Celebrate
- Sustained +200µm < {optimal_quality + 1:.1f}% for 24 hours
- No constraint violations for 48 hours
- Smooth transition completed without issues
"""

    return operator_guide

operator_guidelines = generate_operator_guidelines()
```

---

23. Final Implementation Checklist

Pre-Launch Checklist

Data and Models ✓

- [ ] **Historical data cleaned** and validated (minimum 3 months of stable operation)
- [ ] **All 5 models trained** with $R^2 > 0.8$
- [ ] **Complete chain validated** on held-out test data
- [ ] **Edge cases tested** at operating boundaries
- [ ] **Physical relationships verified** (higher feed → higher power, etc.)

Optimization Setup ✓

- [ ] **Bounds defined** based on equipment specifications
- [ ] **Constraints validated** with process engineers
- [ ] **Objective function tested** manually
- [ ] **Optimization runs** produce consistent results
- [ ] **Optimal solution validated** by process experts

Safety and Risk Management ✓

- ☐ **Emergency procedures** documented and communicated
- ☐ **Rollback plan** tested and approved
- ☐ **Constraint monitoring** system in place
- ☐ **Operator training** completed
- ☐ **Management approval** obtained

Launch Day Protocol

```
def launch_day_protocol():
    """Step-by-step launch day procedure"""

    protocol_steps = [
        {
            'step': 1,
            'action': 'Verify baseline performance',
            'details': 'Record current MVs, CVs, and quality for 2 hours',
            'success_criteria': 'Stable operation, quality within historical range'
        },
        {
            'step': 2,
            'action': 'Implement first change',
            'details': 'Adjust only ore feed rate toward optimal (+/- 5 t/h)',
            'success_criteria': 'Motor power responds as predicted (+/- 50 kW)'
        },
        {
            'step': 3,
            'action': 'Monitor and stabilize',
            'details': 'Wait 1 hour, monitor all CVs',
            'success_criteria': 'All CVs within expected ranges'
        },
        {
            'step': 4,
            'action': 'Implement water adjustments',
            'details': 'Adjust mill and sump water flows toward optimal',
            'success_criteria': 'Density responds as predicted (+/- 0.05 kg/L)'
        },
        {
            'step': 5,
            'action': 'Final ball dosage adjustment',
            'details': 'Adjust ball addition rate toward optimal',
            'success_criteria': 'Power and grinding performance stable'
        },
        {
            'step': 6,
            'action': 'Stabilization period',
            'details': 'Monitor for 4 hours at optimal settings',
            'success_criteria': 'All parameters stable, quality improving'
        }
```

```python
    ]

    print("Launch Day Protocol:")
    for step in protocol_steps:
        print(f"\nStep {step['step']}: {step['action']}")
        print(f"  Details: {step['details']}")
        print(f"  Success: {step['success_criteria']}")

    return protocol_steps

launch_protocol = launch_day_protocol()
```

Post-Launch Monitoring (First 30 Days)

```python
def post_launch_monitoring_plan():
    """30-day monitoring plan after optimization launch"""

    monitoring_plan = {
        'Week 1': {
            'frequency': 'Every 2 hours',
            'focus': 'Process stability and constraint compliance',
            'actions': [
                'Record all MVs and CVs',
                'Compare actual vs predicted values',
                'Document any manual adjustments',
                'Calculate prediction errors'
            ]
        },
        'Week 2': {
            'frequency': 'Every 4 hours',
            'focus': 'Quality trend analysis',
            'actions': [
                'Track +200μm trend',
                'Analyze lab results vs predictions',
                'Identify systematic biases',
                'Adjust models if needed'
            ]
        },
        'Week 3-4': {
            'frequency': 'Every 8 hours',
            'focus': 'Performance validation and fine-tuning',
            'actions': [
                'Calculate cumulative improvement',
                'Validate economic benefits',
                'Document lessons learned',
                'Plan model updates'
            ]
        }
    }

    return monitoring_plan
```

```
monitoring_schedule = post_launch_monitoring_plan()
```

---

24. Troubleshooting Guide

Common Issues and Solutions

Issue: Optimization Suggests Extreme Values

Symptom: Optuna recommends ore feed = 150 t/h (maximum bound)

Diagnosis:

```
def diagnose_extreme_values(study):
    """Analyze why optimization suggests extreme values"""

    trials_df = study.trials_dataframe()

    # Look at parameter distributions for best trials
    best_10_percent = trials_df.nsmallest(len(trials_df)//10, 'value')

    print("Analysis of extreme values:")
    for param in ['params_ore_feed_rate', 'params_mill_water_flow',
                  'params_sump_water_flow', 'params_ball_dosage']:
        param_values = best_10_percent[param]
        print(f"{param}: mean={param_values.mean():.2f}, std=
{param_values.std():.2f}")

        # Check if values cluster at boundaries
        bounds = MV_BOUNDS[param.replace('params_', '')]
        at_lower = (param_values <= bounds[0] + 0.05 * (bounds[1] -
bounds[0])).sum()
        at_upper = (param_values >= bounds[1] - 0.05 * (bounds[1] -
bounds[0])).sum()

        print(f"  {at_lower} trials at lower bound, {at_upper} trials at upper
bound")

diagnose_extreme_values(optimization_study)
```

Solutions:

1. **Expand training data** at current extreme values
2. **Add soft penalties** for extreme operation

3. **Check model extrapolation** beyond training range
4. **Validate physically** - does extreme value make sense?

Issue: Models Predict Infeasible CV Combinations

Symptom: Predicted CVs violate physical laws or equipment limits

Diagnosis:

```
def diagnose_infeasible_predictions():
    """Check for physically impossible CV combinations"""

    # Test 100 random MV combinations
    n_tests = 100
    infeasible_count = 0

    for _ in range(n_tests):
        # Sample random MVs within bounds
        test_mvs = [
            np.random.uniform(*MV_BOUNDS['ore_feed_rate']),
            np.random.uniform(*MV_BOUNDS['mill_water_flow']),
            np.random.uniform(*MV_BOUNDS['sump_water_flow']),
            np.random.uniform(*MV_BOUNDS['ball_dosage'])
        ]

        _, cvs, feasible = predict_quality_from_mvs(*test_mvs)

        if not feasible:
            infeasible_count += 1
            print(f"Infeasible: {test_mvs} → {cvs}")

    infeasible_rate = infeasible_count / n_tests
    print(f"Infeasible prediction rate: {infeasible_rate:.2%}")

    if infeasible_rate > 0.1:  # More than 10% infeasible
        print("⚠️ High infeasible rate - check model training data coverage")

diagnose_infeasible_predictions()
```

Solutions:

1. **Retrain with more diverse data** covering full operating range
2. **Add physics-based constraints** to model training
3. **Use ensemble methods** to improve prediction reliability
4. **Implement soft constraints** during optimization

Issue: Poor Optimization Convergence

Symptom: Optuna trials show no clear improvement trend

Diagnosis:

```
def diagnose_poor_convergence(study):
    """Analyze optimization convergence issues"""

    trials_df = study.trials_dataframe()

    # Plot convergence
    plt.figure(figsize=(12, 4))

    plt.subplot(1, 2, 1)
    plt.plot(trials_df['number'], trials_df['value'])
    plt.xlabel('Trial Number')
    plt.ylabel('Objective Value')
    plt.title('Convergence Plot')
    plt.grid(True)

    # Plot value distribution
    plt.subplot(1, 2, 2)
    plt.hist(trials_df['value'], bins=50, alpha=0.7)
    plt.xlabel('Objective Value')
    plt.ylabel('Frequency')
    plt.title('Value Distribution')

    plt.tight_layout()
    plt.show()

    # Statistics
    improvement = trials_df['value'].iloc[0] - trials_df['value'].min()
    print(f"Total improvement: {improvement:.4f}")
    print(f"Best value: {trials_df['value'].min():.4f}")
    print(f"Worst value: {trials_df['value'].max():.4f}")

    # Check if stuck in local minimum
    last_100_trials = trials_df.tail(100)
    recent_improvement = last_100_trials['value'].max() - 
last_100_trials['value'].min()
    print(f"Improvement in last 100 trials: {recent_improvement:.4f}")

    if recent_improvement < 0.1:
        print("⚠️ Possible convergence to local minimum")

diagnose_poor_convergence(optimization_study)
```

Solutions:

1. **Increase exploration**: Use different Optuna sampler
2. **Multi-start optimization**: Run multiple studies with different seeds
3. **Expand search space**: Check if bounds are too restrictive

4. **Improve objective function**: Add regularization or smoothing

---

25. Success Metrics and KPIs

Operational KPIs

```
def calculate_operational_kpis(baseline_data, optimized_data):
    """Calculate key performance indicators"""

    kpis = {}

    # Quality KPIs
    kpis['quality_improvement'] = baseline_data[target].mean() -
optimized_data[target].mean()
    kpis['quality_improvement_percent'] = (kpis['quality_improvement'] /
baseline_data[target].mean()) * 100
    kpis['quality_variability_improvement'] = baseline_data[target].std() -
optimized_data[target].std()

    # Efficiency KPIs
    kpis['specific_energy_change'] = (
        (optimized_data['motor_power'] / optimized_data['ore_feed_rate']).mean() -
        (baseline_data['motor_power'] / baseline_data['ore_feed_rate']).mean()
    )

    kpis['water_efficiency_change'] = (
        ((optimized_data['mill_water_flow'] + optimized_data['sump_water_flow']) /
         optimized_data['ore_feed_rate']).mean() -
        ((baseline_data['mill_water_flow'] + baseline_data['sump_water_flow']) /
         baseline_data['ore_feed_rate']).mean()
    )

    # Stability KPIs
    kpis['power_stability'] = optimized_data['motor_power'].std() /
baseline_data['motor_power'].std()
    kpis['density_stability'] = optimized_data['pulp_density'].std() /
baseline_data['pulp_density'].std()

    # Economic KPIs (simplified)
    kpis['estimated_cost_savings_per_hour'] = (
        kpis['quality_improvement'] * optimized_data['ore_feed_rate'].mean() * 5.0
# $5/t improvement
    )

    print("Operational KPIs Summary:")
    print(f"Quality Improvement: {kpis['quality_improvement']:.2f}%
({kpis['quality_improvement_percent']:.1f}%)")
    print(f"Quality Variability: {'Reduced' if
kpis['quality_variability_improvement'] > 0 else 'Increased'} by
```

```
{abs(kpis['quality_variability_improvement']):.2f}%")
    print(f"Specific Energy: {'Reduced' if kpis['specific_energy_change'] < 0 else
'Increased'} by {abs(kpis['specific_energy_change']):.1f} kWh/t")
    print(f"Water Efficiency: {'Improved' if kpis['water_efficiency_change'] < 0
else 'Reduced'} by {abs(kpis['water_efficiency_change']):.2f} m³/t")
    print(f"Estimated Savings:
${kpis['estimated_cost_savings_per_hour']:.0f}/hour")

    return kpis

# Example calculation (you would use actual pre/post optimization data)
baseline_sample = df_clean.sample(1000, random_state=42)
# optimized_sample would be your post-implementation data
# operational_kpis = calculate_operational_kpis(baseline_sample, optimized_sample)
```

Model Performance KPIs

```
def calculate_model_kpis():
    """Calculate model-specific performance indicators"""

    model_kpis = {
        'model_accuracies': {
            'power_model_r2': r2_score(y_test1, model1.predict(X_test1)),
            'density_model_r2': r2_score(y_test2, model2.predict(X_test2)),
            'flow_model_r2': r2_score(y_test3, model3.predict(X_test3)),
            'pressure_model_r2': r2_score(y_test4, model4.predict(X_test4)),
            'quality_model_r2': r2_score(y_test_q, quality_model.predict(X_test_q))
        },
        'prediction_intervals': {
            'quality_mae': np.mean(np.abs(y_test_q -
quality_model.predict(X_test_q))),
            'power_mae': np.mean(np.abs(y_test1 - model1.predict(X_test1))),
            'density_mae': np.mean(np.abs(y_test2 - model2.predict(X_test2)))
        },
        'optimization_metrics': {
            'trials_to_convergence': len(optimization_study.trials),
            'improvement_found': optimization_study.best_value <
df_clean[target].mean(),
            'constraints_satisfied': True  # Would check this during validation
        }
    }

    print("Model Performance KPIs:")
    print("Accuracy (R² scores):")
    for model, r2 in model_kpis['model_accuracies'].items():
        print(f"  {model}: {r2:.4f}")

    print("\nPrediction Accuracy (MAE):")
    for metric, mae in model_kpis['prediction_intervals'].items():
        print(f"  {metric}: {mae:.3f}")

    return model_kpis
```

```
model_performance_kpis = calculate_model_kpis()
```

---

26. Conclusion and Summary

What You Will Achieve

By following this comprehensive plan, you will:

1. **Build a robust multi-model system** that captures the complex relationships in your ball milling process
2. **Optimize directly in the actionable space** (manipulated variables) while leveraging all sensor information
3. **Implement safely and gradually** with proper constraints and monitoring
4. **Achieve sustained quality improvement** with minimal risk to operations
5. **Establish a framework** for continuous optimization and model improvement

Key Success Factors

1. **Data Quality**: The foundation of everything - ensure clean, representative historical data
2. **Model Validation**: Thoroughly test each model before trusting optimization results
3. **Gradual Implementation**: Small, monitored changes reduce operational risk
4. **Continuous Monitoring**: Track performance and update models as needed
5. **Team Collaboration**: Involve process engineers and operators throughout

Expected Timeline

- **Weeks 1-2**: Data preparation and model development
- **Week 3**: Optimization setup and validation
- **Week 4**: Implementation planning and safety checks
- **Weeks 5-6**: Gradual deployment and monitoring
- **Ongoing**: Performance tracking and model updates

Risk Mitigation

The multi-model approach inherently reduces several risks:

- **Model uncertainty**: Multiple models provide cross-validation
- **Operational safety**: Constraints prevent unsafe operation
- **Implementation risk**: Gradual changes allow early detection of issues
- **Business continuity**: Rollback procedures ensure quick recovery

Final Recommendations

1. **Start with data quality assessment** - this is your foundation
2. **Build models incrementally** - validate each step before proceeding
3. **Test extensively offline** before any plant implementation
4. **Engage your operations team** early and throughout the process
5. **Plan for the long term** - this is an ongoing optimization journey, not a one-time project

---

Appendix A: Code Templates

A.1 Complete Training Script Template

```
"""
Ball Mill Optimization - Complete Training Script
Save as: train_ball_mill_models.py
"""

import pandas as pd
import numpy as np
import xgboost as xgb
import optuna
import joblib
from sklearn.model_selection import train_test_split
from sklearn.metrics import r2_score, mean_squared_error
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings('ignore')

def main():
```

```
    """Main training and optimization pipeline"""

    # 1. Load and prepare data
    print("Step 1: Loading data...")
    df = pd.read_csv('your_data.csv')  # Replace with your file

    # Define variables (adjust to your column names)
    MVs = ['ore_feed_rate', 'mill_water_flow', 'sump_water_flow', 'ball_dosage']
    CVs = ['motor_power', 'pulp_density', 'pulp_flow', 'hydrocyclone_pressure']
    DVs = ['ore_hardness', 'grindability_index']  # Optional
    target = 'plus_200_micron_percentage'

    # Clean data
    df_clean = df.dropna()
    print(f"Clean data shape: {df_clean.shape}")

    # 2. Train process models
    print("\nStep 2: Training process models...")
    models = {}

    # Model 1: All MVs → Motor Power
    X1 = df_clean[MVs]
    y1 = df_clean['motor_power']
    X_train1, X_test1, y_train1, y_test1 = train_test_split(X1, y1, test_size=0.2,
random_state=42)

    models['power'] = xgb.XGBRegressor(n_estimators=200, max_depth=6,
random_state=42)
    models['power'].fit(X_train1, y_train1)

    y_pred1 = models['power'].predict(X_test1)
    print(f"Power Model R²: {r2_score(y_test1, y_pred1):.4f}")

    # Models 2,3,4: Partial MVs → Other CVs
    cv_mapping = [
        ('density', 'pulp_density'),
        ('flow', 'pulp_flow'),
        ('pressure', 'hydrocyclone_pressure')
    ]

    for model_name, cv_column in cv_mapping:
        X = df_clean[['ore_feed_rate', 'mill_water_flow', 'sump_water_flow']]
        y = df_clean[cv_column]
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

        models[model_name] = xgb.XGBRegressor(n_estimators=200, max_depth=6,
random_state=42)
        models[model_name].fit(X_train, y_train)

        y_pred = models[model_name].predict(X_test)
        print(f"{model_name.title()} Model R²: {r2_score(y_test, y_pred):.4f}")

    # 3. Train quality model
    print("\nStep 3: Training quality model...")
    X_quality = df_clean[CVs]
    if DVs and all(dv in df_clean.columns for dv in DVs):
```

```
        X_quality = pd.concat([X_quality, df_clean[DVs]], axis=1)

    y_quality = df_clean[target]
    X_train_q, X_test_q, y_train_q, y_test_q = train_test_split(X_quality,
y_quality, test_size=0.2, random_state=42)

    models['quality'] = xgb.XGBRegressor(
        n_estimators=500, max_depth=8, learning_rate=0.05,
        subsample=0.8, colsample_bytree=0.8,
        reg_alpha=0.1, reg_lambda=0.1, random_state=42
    )
    models['quality'].fit(X_train_q, y_train_q)

    y_pred_q = models['quality'].predict(X_test_q)
    quality_r2 = r2_score(y_test_q, y_pred_q)
    print(f"Quality Model R²: {quality_r2:.4f}")

    # 4. Save all models
    print("\nStep 4: Saving models...")
    for name, model in models.items():
        joblib.dump(model, f'model_{name}.pkl')
    print("All models saved!")

    # 5. Run optimization
    print("\nStep 5: Running optimization...")

    # Define bounds and constraints
    MV_BOUNDS = {
        'ore_feed_rate': (50, 150),
        'mill_water_flow': (10, 50),
        'sump_water_flow': (5, 30),
        'ball_dosage': (0.5, 2.0)
    }

    CV_CONSTRAINTS = {
        'motor_power': (500, 1200),
        'pulp_density': (1.2, 1.6),
        'pulp_flow': (80, 200),
        'hydrocyclone_pressure': (1.0, 3.0)
    }

    def predict_cvs_from_mvs(ore_feed, mill_water, sump_water, ball_dosage):
        mvs_full = np.array([[ore_feed, mill_water, sump_water, ball_dosage]])
        mvs_partial = np.array([[ore_feed, mill_water, sump_water]])

        return {
            'motor_power': models['power'].predict(mvs_full)[0],
            'pulp_density': models['density'].predict(mvs_partial)[0],
            'pulp_flow': models['flow'].predict(mvs_partial)[0],
            'hydrocyclone_pressure': models['pressure'].predict(mvs_partial)[0]
        }

    def objective(trial):
        # Sample MVs
        ore_feed = trial.suggest_float('ore_feed_rate',
*MV_BOUNDS['ore_feed_rate'])
        mill_water = trial.suggest_float('mill_water_flow',
```

```
*MV_BOUNDS['mill_water_flow'])
        sump_water = trial.suggest_float('sump_water_flow',
*MV_BOUNDS['sump_water_flow'])
        ball_dosage = trial.suggest_float('ball_dosage', *MV_BOUNDS['ball_dosage'])

        # Predict CVs
        predicted_cvs = predict_cvs_from_mvs(ore_feed, mill_water, sump_water,
ball_dosage)

        # Check constraints
        constraints_met = all(
            CV_CONSTRAINTS[cv_name][0] <= cv_value <= CV_CONSTRAINTS[cv_name][1]
            for cv_name, cv_value in predicted_cvs.items()
        )

        if not constraints_met:
            return 100.0

        # Predict quality
        cv_array = np.array(list(predicted_cvs.values()))
        predicted_quality = models['quality'].predict([cv_array])[0]

        return predicted_quality

    # Run optimization
    study = optuna.create_study(direction='minimize')
    study.optimize(objective, n_trials=1000)

    print(f"\nOptimization Results:")
    print(f"Best +200μm fraction: {study.best_value:.2f}%")
    print(f"Baseline average: {df_clean[target].mean():.2f}%")
    print(f"Improvement: {df_clean[target].mean() - study.best_value:.2f}%")
    print("\nOptimal parameters:")
    for param, value in study.best_params.items():
        print(f"  {param}: {value:.2f}")

    # Save optimization results
    joblib.dump(study, 'optimization_study.pkl')

    return models, study

if __name__ == "__main__":
    models, study = main()
```

A.2 Production Deployment Script Template

```
"""
Ball Mill Optimization - Production Deployment
Save as: deploy_optimization.py
"""

import pandas as pd
```

```python
import numpy as np
import joblib
import time
from datetime import datetime

class ProductionDeployment:
    def __init__(self):
        # Load trained models
        self.models = {
            'power': joblib.load('model_power.pkl'),
            'density': joblib.load('model_density.pkl'),
            'flow': joblib.load('model_flow.pkl'),
            'pressure': joblib.load('model_pressure.pkl'),
            'quality': joblib.load('model_quality.pkl')
        }

        # Load optimization results
        self.study = joblib.load('optimization_study.pkl')
        self.optimal_params = self.study.best_params

        # Initialize logging
        self.deployment_log = []

    def validate_current_state(self):
        """Validate current process state before making changes"""

        # Get current readings (replace with actual sensor interface)
        current_readings = self.get_current_process_readings()

        # Check if current state is stable
        stability_check = self.check_process_stability(current_readings)

        if not stability_check:
            print("⚠️ Process not stable - delay optimization implementation")
            return False

        print("✅ Process stable - ready for optimization")
        return True

    def get_current_process_readings(self):
        """Get current process readings from sensors/DCS"""

        # THIS IS WHERE YOU INTERFACE WITH YOUR ACTUAL SENSORS
        # Replace this simulation with real sensor readings

        current_readings = {
            'ore_feed_rate': 95.0,       # t/h - from belt scale
            'mill_water_flow': 28.0,     # m³/h - from flow meter
            'sump_water_flow': 18.0,     # m³/h - from flow meter
            'ball_dosage': 1.2,          # t/h - from ball feeder
            'motor_power': 820.0,        # kW - from motor
            'pulp_density': 1.38,        # kg/L - from density meter
            'pulp_flow': 115.0,          # m³/h - from flow meter
            'hydrocyclone_pressure': 2.1, # bar - from pressure transmitter
            'timestamp': datetime.now()
        }
```

```
        return current_readings

    def check_process_stability(self, readings, stability_window=30):
        """Check if process has been stable for specified time"""

        # In real implementation, you would check if readings have been
        # within normal ranges for the stability window

        # Simplified check
        stable_conditions = [
            500 <= readings['motor_power'] <= 1200,
            1.2 <= readings['pulp_density'] <= 1.6,
            80 <= readings['pulp_flow'] <= 200,
            1.0 <= readings['hydrocyclone_pressure'] <= 3.0
        ]

        return all(stable_conditions)

    def implement_gradual_change(self, target_params, transition_hours=24):
        """Implement optimization results gradually"""

        current_readings = self.get_current_process_readings()

        print(f"Starting gradual implementation over {transition_hours} hours...")
        print(f"Current settings → Target settings")

        # Calculate hourly increments
        hourly_increments = {}
        for param in ['ore_feed_rate', 'mill_water_flow', 'sump_water_flow',
'ball_dosage']:
            current_val = current_readings[param]
            target_val = target_params[param]
            hourly_increments[param] = (target_val - current_val) /
transition_hours

            print(f"{param}: {current_val:.2f} → {target_val:.2f}
({hourly_increments[param]:+.2f}/hour)")

        # Implement changes hour by hour
        for hour in range(1, transition_hours + 1):
            print(f"\n--- Hour {hour} of {transition_hours} ---")

            # Calculate new setpoints for this hour
            new_setpoints = {}
            for param in hourly_increments:
                current_val = current_readings[param]
                new_setpoints[param] = current_val + (hourly_increments[param] *
hour)

            # Implement new setpoints (THIS IS WHERE YOU SEND COMMANDS TO DCS)
            success = self.send_setpoints_to_dcs(new_setpoints)

            if not success:
                print(f"❌ Failed to implement setpoints at hour {hour}")
                return False

            # Wait and monitor
```

```
            print(f"✅ Setpoints implemented. Monitoring for 1 hour...")

            # Monitor for stability (simplified - you would implement real
monitoring)
            time.sleep(10)  # In real implementation, this would be 3600 seconds (1
hour)

            # Check process response
            new_readings = self.get_current_process_readings()
            stability_ok = self.check_process_stability(new_readings)

            if not stability_ok:
                print(f"⚠️ Process unstable at hour {hour} - stopping
implementation")
                self.emergency_rollback(current_readings)
                return False

            # Log this step
            self.deployment_log.append({
                'hour': hour,
                'setpoints': new_setpoints.copy(),
                'readings': new_readings.copy(),
                'stable': stability_ok
            })

            print(f"Hour {hour} completed successfully")

        print(f"\n🎉 Gradual implementation completed successfully!")
        return True

    def send_setpoints_to_dcs(self, setpoints):
        """Send new setpoints to DCS/PLC system"""

        # THIS IS WHERE YOU INTERFACE WITH YOUR CONTROL SYSTEM
        # Replace with actual DCS communication code

        try:
            # Example: Send to DCS via OPC, Modbus, or other protocol
            # dcs_client.write('ORE_FEED_SETPOINT', setpoints['ore_feed_rate'])
            # dcs_client.write('MILL_WATER_SETPOINT', setpoints['mill_water_flow'])
            # dcs_client.write('SUMP_WATER_SETPOINT', setpoints['sump_water_flow'])
            # dcs_client.write('BALL_DOSAGE_SETPOINT', setpoints['ball_dosage'])

            print(f"Setpoints sent to DCS:")
            for param, value in setpoints.items():
                print(f"  {param}: {value:.2f}")

            return True

        except Exception as e:
            print(f"❌ Error sending setpoints: {str(e)}")
            return False

    def emergency_rollback(self, safe_settings):
        """Emergency rollback to safe operating conditions"""

        print("🚨 EMERGENCY ROLLBACK INITIATED")
```

```python
        rollback_settings = {
            'ore_feed_rate': safe_settings['ore_feed_rate'],
            'mill_water_flow': safe_settings['mill_water_flow'],
            'sump_water_flow': safe_settings['sump_water_flow'],
            'ball_dosage': safe_settings['ball_dosage']
        }

        success = self.send_setpoints_to_dcs(rollback_settings)

        if success:
            print("✅ Rollback completed - process returned to safe state")
        else:
            print("❌ CRITICAL: Rollback failed - manual intervention required")
            # Trigger alarms, notify operators

        return success

    def start_monitoring_mode(self):
        """Start continuous monitoring after implementation"""

        print("Starting continuous monitoring mode...")

        while True:
            try:
                # Get current readings
                readings = self.get_current_process_readings()

                # Predict expected quality
                predicted_cvs = self.predict_cvs_from_mvs(
                    readings['ore_feed_rate'],
                    readings['mill_water_flow'],
                    readings['sump_water_flow'],
                    readings['ball_dosage']
                )

                cv_array = np.array(list(predicted_cvs.values()))
                predicted_quality = self.models['quality'].predict([cv_array])[0]

                # Log monitoring data
                monitoring_data = {
                    'timestamp': readings['timestamp'],
                    'actual_mvs': {k: readings[k] for k in ['ore_feed_rate',
'mill_water_flow', 'sump_water_flow', 'ball_dosage']},
                    'actual_cvs': {k: readings[k] for k in ['motor_power',
'pulp_density', 'pulp_flow', 'hydrocyclone_pressure']},
                    'predicted_cvs': predicted_cvs,
                    'predicted_quality': predicted_quality
                }

                self.log_monitoring_data(monitoring_data)

                # Sleep until next monitoring cycle (e.g., every 15 minutes)
                time.sleep(900)  # 15 minutes

            except KeyboardInterrupt:
                print("Monitoring stopped by user")
```

```
                break
        except Exception as e:
            print(f"Monitoring error: {str(e)}")
            time.sleep(60)  # Wait 1 minute before retrying

    def predict_cvs_from_mvs(self, ore_feed, mill_water, sump_water, ball_dosage):
        """Predict CVs from MVs using trained models"""

        mvs_full = np.array([[ore_feed, mill_water, sump_water, ball_dosage]])
        mvs_partial = np.array([[ore_feed, mill_water, sump_water]])

        return {
            'motor_power': self.models['power'].predict(mvs_full)[0],
            'pulp_density': self.models['density'].predict(mvs_partial)[0],
            'pulp_flow': self.models['flow'].predict(mvs_partial)[0],
            'hydrocyclone_pressure': self.models['pressure'].predict(mvs_partial)
[0]
        }

    def log_monitoring_data(self, data):
        """Log monitoring data for performance tracking"""

        # Save to CSV for analysis
        log_df = pd.DataFrame([{
            'timestamp': data['timestamp'],
            'predicted_quality': data['predicted_quality'],
            **data['actual_mvs'],
            **data['actual_cvs'],
            **{f"pred_{k}": v for k, v in data['predicted_cvs'].items()}
        }])

        # Append to log file
        log_df.to_csv('optimization_monitoring.csv', mode='a', header=False,
index=False)

        # Print status
        print(f"[{data['timestamp'].strftime('%H:%M:%S')}] "
              f"Predicted quality: {data['predicted_quality']:.2f}% | "
              f"Power: {data['actual_cvs']['motor_power']:.0f} kW | "
              f"Density: {data['actual_cvs']['pulp_density']:.2f} kg/L")

def run_deployment():
    """Main deployment function"""

    deployment = ProductionDeployment()

    # Step 1: Validate current state
    if not deployment.validate_current_state():
        print("Cannot proceed with deployment - process not ready")
        return

    # Step 2: Show implementation plan
    print(f"\nImplementation Plan:")
    print(f"Target parameters:")
    for param, value in deployment.optimal_params.items():
        print(f"  {param}: {value:.2f}")
```

```
    # Step 3: Get operator confirmation
    proceed = input("\nProceed with gradual implementation? (yes/no): ")
    if proceed.lower() != 'yes':
        print("Implementation cancelled by operator")
        return

    # Step 4: Implement changes
    success = deployment.implement_gradual_change(deployment.optimal_params,
transition_hours=24)

    if success:
        print("\n✅ Implementation successful!")

        # Step 5: Start monitoring
        start_monitoring = input("Start continuous monitoring? (yes/no): ")
        if start_monitoring.lower() == 'yes':
            deployment.start_monitoring_mode()
    else:
        print("\n❌ Implementation failed - check logs and try again")

if __name__ == "__main__":
    run_deployment()
```

A.3 Monitoring and Analysis Script Template

```
"""
Ball Mill Optimization - Monitoring and Analysis
Save as: monitor_optimization.py
"""

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from datetime import datetime, timedelta

class OptimizationMonitor:
    """Monitor and analyze optimization performance"""

    def __init__(self):
        self.monitoring_data = None
        self.baseline_data = None

    def load_monitoring_data(self, filepath='optimization_monitoring.csv'):
        """Load monitoring data from CSV"""

        self.monitoring_data = pd.read_csv(filepath)
        self.monitoring_data['timestamp'] =
pd.to_datetime(self.monitoring_data['timestamp'])

        print(f"Loaded {len(self.monitoring_data)} monitoring records")
        return self.monitoring_data
```

```
    def load_baseline_data(self, filepath='baseline_data.csv'):
        """Load baseline performance data for comparison"""

        self.baseline_data = pd.read_csv(filepath)
        print(f"Loaded {len(self.baseline_data)} baseline records")
        return self.baseline_data

    def analyze_performance(self, analysis_period_hours=168):  # 1 week
        """Analyze optimization performance over specified period"""

        # Get recent data
        cutoff_time = self.monitoring_data['timestamp'].max() -
timedelta(hours=analysis_period_hours)
        recent_data = self.monitoring_data[self.monitoring_data['timestamp'] >=
cutoff_time]

        print(f"\nPerformance Analysis - Last {analysis_period_hours} hours:")
        print(f"Data points analyzed: {len(recent_data)}")

        # Quality performance
        if 'actual_quality' in recent_data.columns:
            actual_quality = recent_data['actual_quality']
            predicted_quality = recent_data['predicted_quality']

            print(f"\nQuality Performance:")
            print(f"  Average actual quality: {actual_quality.mean():.2f}%")
            print(f"  Average predicted quality: {predicted_quality.mean():.2f}%")
            print(f"  Prediction error (MAE): {np.mean(np.abs(actual_quality -
predicted_quality)):.2f}%")

            # Compare with baseline
            if self.baseline_data is not None:
                baseline_quality =
self.baseline_data['plus_200_micron_percentage'].mean()
                improvement = baseline_quality - actual_quality.mean()
                improvement_percent = (improvement / baseline_quality) * 100

                print(f"  Baseline quality: {baseline_quality:.2f}%")
                print(f"  Quality improvement: {improvement:.2f}%
({improvement_percent:.1f}%)")

        # Process stability
        print(f"\nProcess Stability:")
        cv_columns = ['motor_power', 'pulp_density', 'pulp_flow',
'hydrocyclone_pressure']

        for cv in cv_columns:
            if cv in recent_data.columns:
                cv_std = recent_data[cv].std()
                cv_mean = recent_data[cv].mean()
                cv_cv = (cv_std / cv_mean) * 100  # Coefficient of variation

                print(f"  {cv}: μ={cv_mean:.2f}, σ={cv_std:.2f}, CV={cv_cv:.1f}%")

    def create_performance_dashboard(self):
        """Create comprehensive performance dashboard"""
```

```
        if self.monitoring_data is None:
            print("No monitoring data loaded")
            return

        # Setup plotting
        plt.style.use('seaborn-v0_8')
        fig, axes = plt.subplots(3, 2, figsize=(16, 12))
        fig.suptitle('Ball Mill Optimization Performance Dashboard', fontsize=16)

        # 1. Quality trend
        if 'actual_quality' in self.monitoring_data.columns:
            axes[0,0].plot(self.monitoring_data['timestamp'],
                          self.monitoring_data['actual_quality'],
                          label='Actual', alpha=0.7)
            axes[0,0].plot(self.monitoring_data['timestamp'],
                          self.monitoring_data['predicted_quality'],
                          label='Predicted', alpha=0.7)

            if self.baseline_data is not None:
                baseline_mean =
self.baseline_data['plus_200_micron_percentage'].mean()
                axes[0,0].axhline(y=baseline_mean, color='red', linestyle='--',
                                 label=f'Baseline ({baseline_mean:.1f}%)')

            axes[0,0].set_title('+200μm Quality Trend')
            axes[0,0].set_ylabel('Percentage (%)')
            axes[0,0].legend()
            axes[0,0].grid(True)

        # 2. Motor power
        axes[0,1].plot(self.monitoring_data['timestamp'],
                      self.monitoring_data['motor_power'])
        axes[0,1].set_title('Motor Power')
        axes[0,1].set_ylabel('Power (kW)')
        axes[0,1].grid(True)

        # 3. Pulp density
        axes[1,0].plot(self.monitoring_data['timestamp'],
                      self.monitoring_data['pulp_density'])
        axes[1,0].set_title('Pulp Density')
        axes[1,0].set_ylabel('Density (kg/L)')
        axes[1,0].grid(True)

        # 4. Ore feed rate
        axes[1,1].plot(self.monitoring_data['timestamp'],
                      self.monitoring_data['ore_feed_rate'])
        axes[1,1].set_title('Ore Feed Rate')
        axes[1,1].set_ylabel('Feed Rate (t/h)')
        axes[1,1].grid(True)

        # 5. Water flows
        axes[2,0].plot(self.monitoring_data['timestamp'],
                      self.monitoring_data['mill_water_flow'],
                      label='Mill Water')
        axes[2,0].plot(self.monitoring_data['timestamp'],
                      self.monitoring_data['sump_water_flow'],
```

```
                        label='Sump Water')
        axes[2,0].set_title('Water Flows')
        axes[2,0].set_ylabel('Flow (m³/h)')
        axes[2,0].legend()
        axes[2,0].grid(True)

        # 6. Prediction accuracy
        if 'actual_quality' in self.monitoring_data.columns:
            prediction_error = (self.monitoring_data['actual_quality'] -
                              self.monitoring_data['predicted_quality'])
            axes[2,1].plot(self.monitoring_data['timestamp'], prediction_error)
            axes[2,1].axhline(y=0, color='red', linestyle='--')
            axes[2,1].set_title('Prediction Error')
            axes[2,1].set_ylabel('Error (%)')
            axes[2,1].grid(True)

        plt.tight_layout()

plt.savefig(f'performance_dashboard_{datetime.now().strftime("%Y%m%d_%H%M")}.png',
                   dpi=300, bbox_inches='tight')
        plt.show()

    def generate_weekly_report(self):
        """Generate automated weekly performance report"""

        # Analyze last 7 days
        week_ago = self.monitoring_data['timestamp'].max() - timedelta(days=7)
        week_data = self.monitoring_data[self.monitoring_data['timestamp'] >=
week_ago]

        report = f"""
# Weekly Optimization Report
Period: {week_ago.strftime('%Y-%m-%d')} to
{self.monitoring_data['timestamp'].max().strftime('%Y-%m-%d')}

## Summary Statistics
- **Data Points**: {len(week_data)}
- **Average +200μm**: {week_data['predicted_quality'].mean():.2f}%
- **Quality Std Dev**: {week_data['predicted_quality'].std():.2f}%
- **Process Uptime**: {(len(week_data) / (7*24)) * 100:.1f}%

## Process Performance
- **Motor Power**: {week_data['motor_power'].mean():.0f} ±
{week_data['motor_power'].std():.0f} kW
- **Pulp Density**: {week_data['pulp_density'].mean():.2f} ±
{week_data['pulp_density'].std():.2f} kg/L
- **Ore Feed Rate**: {week_data['ore_feed_rate'].mean():.1f} ±
{week_data['ore_feed_rate'].std():.1f} t/h

## Constraint Violations
"""

        # Check for constraint violations
        violations = 0
        if week_data['motor_power'].max() > 1200:
            violations += 1
            report += f"- Motor power exceeded 1200 kW:
```

```
{week_data['motor_power'].max():.0f} kW\n"

        if week_data['pulp_density'].min() < 1.2 or week_data['pulp_density'].max()
> 1.6:
            violations += 1
            report += f"- Density out of range:
{week_data['pulp_density'].min():.2f} - {week_data['pulp_density'].max():.2f}
kg/L\n"

        if violations == 0:
            report += "- No constraint violations detecte
```