Xafron Documentation

Data Quality Detection System

Generated on: 8/7/2025

README

Data Quality Detection System Documentation

Welcome to the comprehensive documentation for the Data Quality Detection System.

📚 Documentation Structure

Getting Started

- **Installation Guide** - System requirements and setup instructions
- **Quick Start Tutorial** - Run your first detection in minutes
- **Basic Usage Guide** - Common workflows and commands

User Guides

- **Running Detection** - Detailed guide for running data quality checks
- **Analyzing Results** - Understanding and interpreting detection output
- **Performance Optimization** - Tuning detection accuracy and speed

Reference Documentation

- **CLI Reference** - Complete command-line interface documentation
- **Configuration Reference** - Brand configs, thresholds, and settings
- **API Interfaces** - Core interfaces for extending the system

Architecture & Design

- **System Overview** - High-level architecture and design principles
- **Detection Methods** - Deep dive into detection algorithms

Development

- **Adding New Fields** - Extend the system with custom field types
- **Contributing Guide** - Development setup and contribution process

Deployment

- **Deployment Examples** - Production deployment configurations

🚀 Quick Links

- **First time?** Start with the Installation Guide and Quick Start Tutorial
- **Need help with commands?** Check the CLI Reference
- **Customizing for your data?** See Configuration Reference
- **Want to contribute?** Read our Contributing Guide

📖 Learning Path

1. **Setup**: Installation → Quick Start
2. **Usage**: Basic Usage → Running Detection
3. **Analysis**: Analyzing Results → Optimization
4. **Customization**: Configuration → Adding Fields

🔍 Key Concepts

- **Detection Methods**: Validation, Pattern-based, ML-based, and LLM-based anomaly detection
- **Field Mapping**: Configurable mapping between your data columns and standard fields
- **Weighted Combination**: Optimized blending of multiple detection methods
- **Error Injection**: Synthetic error generation for testing and evaluation

📊 System Capabilities

- ✅ Multi-method anomaly detection
- ✅ Configurable detection thresholds
- ✅ Performance evaluation and optimization
- ✅ Interactive result visualization
- ✅ Extensible architecture
- ✅ GPU acceleration support

For the latest updates and source code, visit the project repository.

---

getting-started > README

Getting Started with Data Quality Detection System

Welcome! This guide will help you navigate the documentation and get up to speed with the Data Quality Detection System.

🎯 Learning Paths

Path 1: Quick Start (30 minutes)

Goal: Get the system running and see results quickly

1. **Installation Guide** - Set up your environment
2. **Quick Start Tutorial** - Run your first detection
3. **Basic Usage Guide** - Learn common commands

Path 2: Data Analyst (2-3 hours)

Goal: Learn to run detections and analyze results effectively

1. Complete the Quick Start path above
2. **Running Detection** - Master detection options
3. **Analyzing Results** - Understand output files
4. **CLI Reference** - Explore available commands

Path 3: System Administrator (4-5 hours)

Goal: Configure and optimize the system for your organization

1. Complete the Data Analyst path above
2. **Configuration Reference** - Set up brand configs
3. **Performance Optimization** - Tune for accuracy
4. **Deployment Examples** - Production deployment

Path 4: Developer (Full Day)

Goal: Extend the system with custom fields and methods

1. Complete the System Administrator path above
2. **System Architecture** - Understand the design

3. **Detection Methods** - Deep dive into algorithms
4. **Adding Fields** - Extend functionality
5. **API Interfaces** - Implement custom components
6. **Contributing Guide** - Join development

📋 Prerequisites by Role

All Users

- Basic command line familiarity
- Understanding of CSV data format
- Python environment basics

Data Analysts

- Data quality concepts
- Basic statistics understanding
- Spreadsheet software experience

System Administrators

- JSON configuration files
- Environment variables
- System resource management

Developers

- Python programming
- Object-oriented concepts
- Git version control

🔍 Common Tasks

"I want to..."

Run a quick test
→ Start with Quick Start Tutorial

Process my company's data
→ Read Configuration Reference to set up brand mapping

Improve detection accuracy
→ Follow Performance Optimization

Add a new field type
→ Study Adding Fields

Deploy to production
→ Review Deployment Examples

Understand the output
→ Check Analyzing Results

💡 Tips for Success

1. **Start Simple**: Run the demo with sample data before using your own
2. **Iterate**: Begin with basic detection, then add methods incrementally
3. **Monitor Performance**: Use evaluation tools to measure accuracy
4. **Ask Questions**: Check existing documentation before implementing custom solutions
5. **Version Control**: Keep your configurations in version control

🚀 Next Steps

Choose your learning path above based on your role and goals. Each path builds on the previous one, ensuring you have the foundation needed for more advanced topics.

Remember: The system is designed to be modular and extensible. Start with what you need today, and expand as your requirements grow.

---

getting-started > installation

Installation Guide

This guide will walk you through the installation process for the Data Quality Detection System.

Prerequisites

Before installing the system, ensure you have the following prerequisites:

System Requirements

- **Operating System**: Linux, macOS, or Windows with WSL
- **Python**: Version 3.8 or higher
- **Memory**: Minimum 8GB RAM (16GB+ recommended for ML/LLM models)
- **Storage**: At least 5GB free disk space for models and data
- **GPU** (Optional): CUDA-capable GPU for accelerated ML/LLM processing

Software Dependencies

- Git (for cloning the repository)
- Python pip package manager

- Virtual environment tool (venv, conda, or virtualenv)

Installation Steps

1. Clone the Repository

```bash
git clone
cd   # The actual directory name will depend on your repository
```

2. Create a Virtual Environment

It's recommended to use a virtual environment to avoid dependency conflicts:

```bash
# Using venv (built-in Python module
python -m venv venv
# Activate the virtual environment
# On Linux/macOS:
source venv/bin/activate
# On Windows:
venv\Scripts\activate
# Alternative: Using conda
conda create -n data-quality python=3.8
conda activate data-quality
```

3. Install Core Dependencies

Install the required Python packages:

```
bash
pip install -r requirements.txt
```

The core dependencies include:
- `pandas` : Data manipulation and analysis
- `numpy` : Numerical computations
- `scikit-learn` : Machine learning utilities
- `sentence-transformers` : ML-based detection models
- `torch` : Deep learning framework
- `transformers` : Hugging Face transformers library
- `datasets` : Dataset loading and processing
- `accelerate` : Hardware-accelerated training
- `matplotlib` & `seaborn` : Visualization tools
- `evaluate` : Model evaluation metrics

4. Install Development Dependencies (Optional)

If you plan to contribute or modify the code:

```
bash
pip install -r requirements-dev.txt
```

This includes:
- `pre-commit` : Git hooks for code quality
- `flake8` and extensions: Comprehensive code linting
- `black` : Code formatting
- `isort` : Import sorting
- `mypy` : Type checking
- `pytest` : Testing framework
- `bandit` : Security scanning
- `sphinx` : Documentation generation

5. Install Pre-commit Hooks (Optional)

To ensure code quality on every commit:

```
bash
pre-commit install
```

This sets up automatic code quality checks that run before each commit. To run the checks manually:

```
bash
pre-commit run --all-files
```

GPU Support Setup

For faster ML and LLM model processing:

NVIDIA GPU with CUDA

1. Install CUDA Toolkit (11.7 or higher)
2. Install cuDNN (compatible with your CUDA version)
3. Install PyTorch with CUDA support:

```
bash
```

For CUDA 11.7

```
pip install torch torchvision torchaudio --index-url https://downlo
```

For CUDA 11.8

```
pip install torch torchvision torchaudio --index-url https://downlo
```

Verify GPU Installation

```
python
import torch
print(f"CUDA available: {torch.cuda.is_available()}")
print(f"CUDA device: {torch.cuda.get_device_name(0) if torch.cuda.i
```

Model Downloads

Some detection methods require pre-trained models:

ML-Based Detection Models

The system will automatically download required models on first use:
- Sentence transformer models (~400MB each)
- Custom fine-tuned models (stored in `anomaly_detectors/ml_based/models/`)

LLM-Based Detection Models

For LLM detection, models are downloaded on demand:
- Base transformer models (~1-2GB)
- Fine-tuned models (stored in `anomaly_detectors/llm_based/models/`)

Configuration Setup

1. Brand Configuration

Create or modify brand configuration files:

bash

Create a new brand configuration

```
cp brand_configs/esqualo.json brand_configs/your_brand.json
```

Edit with your brand's field mapp

```
vim brand_configs/your_brand.json
```

2. Environment Variables (Optional)

You can set environment variables to control system behavior:

bash

GPU configuration

```
export CUDA_VISIBLE_DEVICES=0
```

For persistent settings, add to y

```
echo 'export CUDA_VISIBLE_DEVICES=0' >> ~/.bashrc
```

Note: The system doesn't currently use a .env file. Environment variables must be set in your shell or system environment.

Verification

Verify your installation by running a simple detection demo:

```bash
# Run a simple detection demo
python main.py single-demo --help
```

Troubleshooting

Common Issues

1. **ImportError: No module named 'X'**
- Ensure virtual environment is activated
- Run `pip install -r requirements.txt` again

2. **CUDA out of memory**
- Reduce batch size in configuration
- Use CPU mode: `--device cpu`

3. **Model download failures**
- Check internet connection
- Manually download models to the models directory

4. **Permission denied errors**
- Ensure write permissions for output directories
- Run with appropriate user permissions

Getting Help

- Review error logs in the output directory
- Submit issues on the project repository

---

getting-started > basic-usage

Basic Usage Guide

This guide covers the fundamental usage patterns and workflows for the Data Quality Detection System.

Available Commands

The system provides six main commands through `main.py`:

- **single-demo**: Run detection on a single CSV file with comprehensive reporting
- **multi-eval**: Evaluate detection performance with systematic testing

- **ml-train**: Train ML models for anomaly detection
- **llm-train**: Train language models for semantic detection
- **analyze-column**: Deep analysis of a specific data column
- **ml-curves**: Generate performance curves to find optimal thresholds

Core Concepts

Detection Workflow

```
 mermaid
flowchart LR
    A[Input Data] --> B[Field Mapping]
    B --> C[Detection Methods]
    C --> D[Result Aggregation]
    D --> E[Report Generation]
    E --> F[Output Files]
```

Key Components

1. **Data Input**: CSV files with structured data
2. **Field Mapping**: Maps your columns to standard fields
3. **Detection Methods**: Multiple approaches to find issues
4. **Results**: Comprehensive reports with confidence scores
5. **Visualization**: Interactive HTML viewer

Command Structure

The basic command structure is:

```
 bash
python main.py  [options]
```

For example, to run detection on your data:

```
 bash
python main.py single-demo --data-file your_data.csv
```

For detailed options and configurations, see the Running Detection Guide.

Error Injection for Testing

The system can inject synthetic errors to evaluate detection performance:

```
bash
```

Test with 20% synthetic errors

```
python main.py single-demo \
    --data-file clean_data.csv \
    --injection-intensity 0.2
```

For production use without synthetic errors, set `--injection-intensity 0.0`.

Detection Methods

1. Validation (Rule-Based)

- **Use Case**: Format validation, business rules
- **Confidence**: 100%
- **Speed**: Fast

Example:

```
bash
python main.py single-demo \
    --data-file your_data.csv \
    --enable-validation \
    --validation-threshold 0.0
```

2. Pattern-Based Detection

- **Use Case**: Anomaly detection based on known patterns
- **Confidence**: 70-80%
- **Speed**: Fast

Example:

```
bash
python main.py single-demo \
    --data-file your_data.csv \
    --enable-pattern \
    --anomaly-threshold 0.7
```

3. ML-Based Detection

- **Use Case**: Semantic similarity anomalies
- **Confidence**: Configurable

- **Speed**: Medium
- **Requirement**: Trained models

Example:

```
bash
python main.py single-demo \
    --data-file your_data.csv \
    --enable-ml \
    --ml-threshold 0.7
```

4. LLM-Based Detection

- **Use Case**: Complex semantic understanding
- **Confidence**: Configurable
- **Speed**: Slower
- **Requirement**: Language models

Example:

```
bash
python main.py single-demo \
    --data-file your_data.csv \
    --enable-llm \
    --llm-threshold 0.6 \
    --llm-few-shot-examples
```

Threshold Configuration

Adjust detection sensitivity per method:

```
bash
python main.py single-demo \
    --data-file your_data.csv \
    --validation-threshold 0.0 \
    --anomaly-threshold 0.7 \
    --ml-threshold 0.8 \
    --llm-threshold 0.6
```

Threshold Guidelines

- **Lower values**: More sensitive (more detections)
- **Higher values**: Less sensitive (fewer detections)
- **0.0**: Detect everything (validation only)
- **1.0**: Detect nothing

Field Selection

Core Fields Only

Process only essential fields to save memory:

```bash
python main.py single-demo \
    --data-file your_data.csv \
    --core-fields-only
```

Core fields typically include:
- material
- color_name
- category
- size
- care_instructions

Advanced Options

Weighted Combination

Use optimized weights for better accuracy:

```bash
python main.py single-demo \
    --data-file your_data.csv \
    --use-weighted-combination \
    --weights-file detection_weights.json
```

Generate Weights

Create optimized weights based on performance:

```bash
python main.py single-demo \
    --data-file your_data.csv \
    --injection-intensity 0.2 \
    --generate-weights \
    --weights-output-file custom_weights.json
```

LLM Context Enhancement

Provide context for better LLM detection:

```bash
python main.py single-demo \
    --data-file your_data.csv \
    --enable-llm \
    --llm-temporal-column date_created \
    --llm-context-columns category,brand,season
```

Output Files

After running detection, you'll find the following files in your output directory:

```
output_dir/
├── report.json                    # Detailed results
├── viewer_report.json            # Web viewer format
├── anomaly_summary.csv           # Summary CSV
├── sample_with_errors.csv        # Data with injections
├── sample_with_results.csv       # Results per row
└── confusion_matrix/             # Performance visuals
    ├── overall_matrix.png
    ├── per_field_matrix.png
    └── summary_visual.png
```

The output directory contains:
- **report.json**: Complete detection results in JSON format
- **viewer_report.json**: Formatted for the HTML viewer
- **anomaly_summary.csv**: CSV summary of all detected anomalies
- **sample_with_errors.csv**: Your data with synthetic errors injected (if using evaluation mode)
- **sample_with_results.csv**: Original data with detection results added
- **confusion_matrix/**: Folder containing performance visualization images

Practical Examples

1. Pre-Import Validation

```bash
python main.py single-demo \
    --data-file import_batch.csv \
    --enable-validation \
    --validation-threshold 0.0 \
    --injection-intensity 0.0 \
    --output-dir validation_results
```

2. Anomaly Detection

```bash
python main.py single-demo \
    --data-file historical_data.csv \
    --enable-pattern \
    --enable-ml \
    --anomaly-threshold 0.8 \
    --ml-threshold 0.75 \
    --injection-intensity 0.0
```

3. Full System Test

```bash
python main.py single-demo \
    --data-file test_data.csv \
    --injection-intensity 0.3 \
    --max-issues-per-row 2 \
    --generate-weights \
    --output-dir test_results
```

4. Production Monitoring

```bash
python main.py single-demo \
    --data-file daily_data.csv \
    --injection-intensity 0.0 \
    --use-weighted-combination \
    --weights-file config/production_weights.json \
    --core-fields-only \
    --output-dir monitoring/$(date +%Y%m%d)
```

Performance Tips

Memory Optimization

- Use `--core-fields-only` for large files
- Process in batches for very large datasets
- Disable memory-intensive methods (LLM) if needed

Speed Optimization

- Use only required detection methods
- Increase thresholds to reduce processing
- Use validation-only for quick checks

Accuracy Optimization

- Generate and use weighted combinations
- Fine-tune thresholds based on your data
- Train custom ML models for your fields

---

getting-started > quick-start

Quick Start Guide

Welcome to the Data Quality Monitoring System! This guide will help you run your first detection in minutes.

Prerequisites

Before starting, ensure you have completed the Installation Guide.

Your First Detection Run

The easiest way to start is with the single sample demo:

```
 bash
python main.py single-demo \
    --data-file data/sample_data.csv \
    --output-dir results/quick_start
```

This will:
1. Load your data
2. Inject synthetic anomalies for testing
3. Run all detection methods
4. Generate comprehensive reports

Understanding the Output

Console Output

You'll see real-time progress:

```
🔍 Processing sample: quick_start_sample
📊 Total fields to check: 15
━━━━━━━━━━━━━━━━━━━━ 100% 15/15
✅ Detection complete!
```

Generated Files

Check your output directory:
- `report.json` - Detailed detection results
- `viewer_report.json` - Formatted for the web viewer
- `anomaly_summary.csv` - Summary of all detections
- `confusion_matrix/` - Performance visualizations

Web Viewer

1. Open `single_sample_multi_field_demo/data_quality_viewer.html` in your browser
2. Upload the generated CSV and JSON files
3. Explore interactive visualizations

Try Different Detection Methods

Now that you've seen the basic demo, try running with specific detection methods:

bash

Fast validation only

```
python main.py single-demo \
    --data-file data/sample_data.csv \
    --enable-validation
```

Or try pattern-based detection

```
python main.py single-demo \
    --data-file data/sample_data.csv \
    --enable-pattern
```

What's Next?

Now that you've run your first detection:
- Learn more workflows in the Basic Usage Guide
- Understand your data with the Data Analysis Guide
- Configure detection for your needs in the Configuration Reference

---

architecture > overview

System Architecture Overview

The Data Quality Detection System is built on a modular, extensible architecture that enables multiple detection methods to work together seamlessly. This document provides a comprehensive overview of the system's architecture, design principles, and key components.

Design Principles

1. Modularity

Each detection method is self-contained and implements common interfaces, allowing new methods to be added without modifying existing code.

2. Extensibility

The system is designed to be easily extended with new fields, detection methods, and output formats through configuration and plugins.

3. Performance

Sequential processing, model caching, and GPU acceleration ensure efficient resource usage even with large datasets.

4. Flexibility

Configurable thresholds, weights, and field mappings allow the system to adapt to different domains and use cases.

High-Level Architecture

```mermaid
graph TB
    subgraph "User Interface"
        UI1[CLI Tools]:::ui
        UI2[HTML Viewer]:::ui
    end

    subgraph "Entry Points"
        EP1[single_sample_demo]:::entry
        EP2[multi_sample_evaluation]:::entry
        EP3[ml_curve_generator]:::entry
        EP4[analyze_column]:::entry
    end

    subgraph "Orchestration"
        O1[ComprehensiveFieldDetector]:::orchestrator
        O2[ConsolidatedReporter]:::orchestrator
        O3[ConfusionMatrixAnalyzer]:::orchestrator
    end

    subgraph "Detection Engine"
        DE1[Validation Engine]:::validator
        DE2[Pattern Detector]:::detector
        DE3[ML Detector]:::ml
        DE4[LLM Detector]:::llm
    end

    subgraph "Core Services"
        CS1[Field Mapper]:::service
        CS2[Brand Config]:::service
        CS3[Error Injector]:::service
        CS4[Model Cache]:::service
    end

    subgraph "Data Storage"
        DS1[(CSV Data)]:::storage
        DS2[(JSON Configs)]:::storage
        DS3[(ML Models)]:::storage
        DS4[(Detection Results)]:::storage
    end

    UI1 --> EP1 & EP2 & EP3 & EP4
    EP1 & EP2 --> O1
    O1 --> DE1 & DE2 & DE3 & DE4
    DE1 & DE2 & DE3 & DE4 --> CS1 & CS2 & CS3 & CS4
    CS1 & CS2 & CS3 & CS4 --> DS1 & DS2 & DS3
    O1 --> O2 & O3
    O2 & O3 --> DS4
    UI2 --> DS4
```

```
    classDef ui fill:#64b5f6,stroke:#1565c0,stroke-width:2px,color:
    classDef entry fill:#81c784,stroke:#388e3c,stroke-width:2px,col
    classDef orchestrator fill:#ba68c8,stroke:#6a1b9a,stroke-width:
    classDef validator fill:#4fc3f7,stroke:#0288d1,stroke-width:2px
    classDef detector fill:#ffb74d,stroke:#f57c00,stroke-width:2px,
    classDef ml fill:#ff8a65,stroke:#d84315,stroke-width:2px,color:
    classDef llm fill:#f06292,stroke:#c2185b,stroke-width:2px,color
    classDef service fill:#fff176,stroke:#f9a825,stroke-width:2px,c
    classDef storage fill:#b0bec5,stroke:#455a64,stroke-width:2px,c
```

Layer Architecture

The system is organized into distinct layers, each with specific responsibilities:

1. Entry Points Layer

This layer provides various ways to interact with the system:

- **Demo Scripts**: Quick demonstration and testing
- **Evaluation Tools**: Performance measurement and comparison
- **Utility Scripts**: Data analysis and preparation

2. Orchestration Layer

Coordinates the detection workflow:

- **ComprehensiveFieldDetector**: Manages detection across all fields and methods
- **Evaluator**: Handles performance evaluation and metrics

3. Detection Methods Layer

Implements the core detection algorithms:

```
mermaid
graph LR
    subgraph "Detection Methods"
        V[Validation
100% Confidence]:::validator
        P[Pattern-Based
70-80% Confidence]:::pattern
        M[ML-Based
Configurable]:::ml
        L[LLM-Based
Configurable]:::llm
    end

    V --> VR[Rule Engine]:::engine
    P --> PR[Pattern Matcher]:::engine
    M --> MR[Similarity Engine]:::engine
    L --> LR[Language Model]:::engine

    classDef validator fill:#4fc3f7,stroke:#0288d1,stroke-width:2px
    classDef pattern fill:#ffb74d,stroke:#f57c00,stroke-width:2px,c
    classDef ml fill:#ff8a65,stroke:#d84315,stroke-width:2px,color:
    classDef llm fill:#f06292,stroke:#c2185b,stroke-width:2px,color
    classDef engine fill:#b39ddb,stroke:#512da8,stroke-width:2px,co
```

4. Core Services Layer

Provides shared functionality:

- **FieldMapper**: Translates between standard fields and column names
- **BrandConfig**: Manages brand-specific configurations
- **ErrorInjector**: Generates synthetic errors for testing
- **Reporters**: Formats and outputs detection results

5. Data Layer

Handles all data storage and retrieval:

- **Input Data**: CSV files with structured data
- **Configuration**: JSON files for settings and rules
- **Models**: Trained ML/LLM models
- **Results**: Detection reports and analyzed data

Component Interactions

```
 mermaid
sequenceDiagram
    participant User
    participant CLI
    participant Orchestrator
    participant Detector
    participant Reporter
    participant Storage

    User->>CLI: Run detection command
    CLI->>Orchestrator: Initialize with config
    Orchestrator->>Storage: Load data

    loop For each field
        Orchestrator->>Detector: Detect anomalies
        Detector->>Storage: Load model/rules
        Detector-->>Orchestrator: Return results
    end

    Orchestrator->>Reporter: Generate report
    Reporter->>Storage: Save results
    Reporter-->>User: Display summary
```

Detection Flow

The system processes data through a well-defined flow:

```
 mermaid
flowchart TD
    A[Input Data] --> B{Field Mapping}
    B --> C[Field Selection]
    C --> D{Detection Method}

    D --> E[Validation]
    D --> F[Pattern Detection]
    D --> G[ML Detection]
    D --> H[LLM Detection]

    E & F & G & H --> I[Result Aggregation]
    I --> J{Combination Strategy}

    J --> K[Priority-Based]
    J --> L[Weighted Average]

    K & L --> M[Final Results]
    M --> N[Report Generation]
    N --> O[Output Files]
```

Memory Management

The system implements several strategies for efficient memory usage:

Sequential Processing

Fields are processed one at a time to minimize memory footprint:

```
 python
for field in fields:
    results = detect_field(field)
    save_results(results)
    clear_cache()
```

Model Caching

Models are loaded once and reused:

```
 mermaid
graph LR
    A[First Request] --> B{Model in Cache?}
    B -->|No| C[Load Model]
    C --> D[Add to Cache]
    D --> E[Use Model]
    B -->|Yes| E
    E --> F[Return Results]
```

Batch Processing

Data is processed in configurable batches to balance memory and performance.

Scalability Considerations

Horizontal Scaling

- Field-level parallelization
- Independent detection methods
- Distributed processing support

Vertical Scaling

- GPU acceleration for ML/LLM
- Optimized algorithms
- Efficient data structures

Security Architecture

```
mermaid
graph TB
    subgraph "Security Layers"
        S1[Input Validation]
        S2[Access Control]
        S3[Data Sanitization]
        S4[Output Filtering]
    end

    I[Input] --> S1
    S1 --> S2
    S2 --> P[Processing]
    P --> S3
    S3 --> S4
    S4 --> O[Output]
```

Extension Points

The architecture provides several extension points for customization:

1. **New Detection Methods**: Implement `AnomalyDetectorInterface`
2. **Custom Validators**: Implement `ValidatorInterface`
3. **Output Formats**: Implement `ReporterInterface`
4. **Field Types**: Add configuration and rules
5. **Brand Support**: Add brand configuration files

Performance Optimization

The system includes several performance optimizations:

- **Lazy Loading**: Models loaded only when needed
- **Result Caching**: Avoid redundant computations
- **Parallel Processing**: Multi-threading for independent operations
- **GPU Acceleration**: CUDA support for ML operations

Data Flow

The system follows a pipeline architecture for processing data:

Processing Pipeline

1. **Input Stage**: Load data from CSV files or DataFrames
2. **Preprocessing**: Map columns to standard fields using brand configuration
3. **Validation**: Apply rule-based validators to each field
4. **Detection**: Run pattern-based, ML, and LLM detection methods

5. **Aggregation**: Combine results from all methods
6. **Reporting**: Generate reports in multiple formats (JSON, CSV, HTML)

Parallel Processing

The system optimizes performance through parallelization:
- Fields are processed sequentially to manage memory
- Detection methods run in parallel for each field
- Results are aggregated after all methods complete

Component Details

Validators

Field-specific validators implement rule-based checks:
- Material composition validation
- Color name standardization
- Size format verification
- Category hierarchy validation
- Care instruction compliance

Each validator implements the `ValidatorInterface` for consistency.

Anomaly Detectors

Three types of anomaly detection:

1. **Pattern-Based**: Uses regex patterns and known value lists
2. **ML-Based**: Employs sentence transformers for semantic similarity
3. **LLM-Based**: Leverages language models for context understanding

All detectors implement the `AnomalyDetectorInterface`.

Orchestration

The `ComprehensiveFieldDetector` coordinates the detection process:
- Manages field mapping and configuration
- Runs detection methods in parallel
- Aggregates and weights results
- Handles error injection for testing

Reporting

Multiple output formats are supported:
- JSON reports with detailed metrics
- CSV summaries for spreadsheet analysis
- HTML viewer for interactive exploration
- Confusion matrices for performance visualization

---

architecture > detection-methods

Detection Methods Architecture

This document provides a comprehensive overview of the detection methods, their theoretical foundations, and implementation details.

Overview

The Data Quality Detection System employs a multi-layered approach to anomaly detection, combining deterministic rule-based validation with advanced machine learning methods. Each detection method addresses different types of data quality issues:

1. **Rule-Based Validation** - Fast, deterministic checks for format violations and business rules
2. **Pattern-Based Detection** - JSON-configured pattern matching for known formats
3. **ML-Based Detection** - Machine learning models for semantic anomalies
4. **LLM-Based Detection** - Large language models for complex linguistic patterns

Detection Philosophy

Our approach uses progressive confidence levels based on the type of anomaly:

- **Deterministic Errors**: Format violations, business rule breaches → Rule-based (100% confidence)
- **Pattern Anomalies**: Known pattern mismatches → Pattern detection (70-80% confidence)
- **Semantic Anomalies**: Contextual inconsistencies → ML detection (60-75% confidence)
- **Complex Linguistic Errors**: Language violations → LLM detection (50-70% confidence)

Method Comparison

Method	Confidence	Training Required	Speed	Best Use Cases
Validation	100%	No	~1ms/record	Format errors, business rules
Pattern-Based	70-80%	No	~5ms/record	Known patterns, regex validation
ML-Based	60-75%	Yes	~20ms/record	Semantic consistency
LLM-Based	50-70%	Yes	~100ms/record	Complex linguistic patterns

Architecture Principles

Modularity

Each detection method implements the `AnomalyDetectorInterface`:

```
python
class AnomalyDetectorInterface(ABC):
    @abstractmethod
    def _detect_anomaly(self, value: Any, context: Dict[str, Any] =
        pass

    def learn_patterns(self, df: pd.DataFrame, column_name: str) ->
        pass

    def bulk_detect(self, df: pd.DataFrame, column_name: str, batch
        pass
```

Performance Optimization

The detection pipeline includes several optimizations:

- **Parallel Processing**: Multi-core CPU utilization
- **Batch Processing**: GPU-optimized batch operations
- **Caching**: Pattern and model caching
- **Early Exit**: Fast-fail on obvious anomalies

Scalability

The architecture supports:

- Horizontal scaling through parallel workers
- Vertical scaling with GPU acceleration
- Batch processing for large datasets
- Distributed execution (future)

Detection Pipeline

1. Data Preprocessing

All detection methods share common preprocessing:

python

Data normalization

Missing value handling

Type conversion

Feature extraction

2. Method Selection

The system can apply methods in sequence or parallel:

- **Sequential**: Rule → Pattern → ML → LLM
- **Parallel**: All methods simultaneously
- **Adaptive**: Based on data characteristics

3. Result Aggregation

Multiple detection results are combined:

- **Union**: Any method flags as anomaly
- **Intersection**: All methods agree
- **Weighted**: Confidence-based voting
- **Hierarchical**: Priority-based selection

Rule-Based Validation

Architecture

```
mermaid
flowchart LR
    I[Input]:::input --> V[Validator]:::process
    V --> RE[Rule Engine]:::process
    RE --> R[Result]:::output

    FR[Field Rules]:::rules --> V

    classDef input fill:#81c784,stroke:#388e3c,stroke-width:2px,col
    classDef process fill:#64b5f6,stroke:#1565c0,stroke-width:2px,c
    classDef output fill:#ce93d8,stroke:#6a1b9a,stroke-width:2px,co
    classDef rules fill:#ffb74d,stroke:#f57c00,stroke-width:2px,col
```

Components

- **Validators**: Field-specific validation classes
- **Rule Engine**: Executes validation rules
- **Rule Repository**: Stores validation rules

Characteristics

- Deterministic results
- Fast execution
- No training required
- Limited to known patterns

Pattern-Based Detection

Architecture

```
mermaid
flowchart LR
    I[Input]:::input --> FE[Feature Extraction]:::process
    FE --> PM[Pattern Matcher]:::process
    PM --> AS[Anomaly Score]:::output

    PDB[Pattern DB]:::data --> FE
    SM[Statistical Models]:::data --> PM

    classDef input fill:#81c784,stroke:#388e3c,stroke-width:2px,col
    classDef process fill:#64b5f6,stroke:#1565c0,stroke-width:2px,c
    classDef output fill:#ce93d8,stroke:#6a1b9a,stroke-width:2px,co
    classDef data fill:#fff176,stroke:#f9a825,stroke-width:2px,colo
```

Components

- **Feature Extractors**: Convert data to features
- **Pattern Database**: Stores known patterns

- **Statistical Models**: Distribution analysis
- **Anomaly Scorer**: Calculates deviation scores

Techniques

- Regular expression matching
- Known value lookup
- JSON-configured pattern rules
- Format validation

ML-Based Detection

Architecture

```
 mermaid
flowchart TB
    TD[Training Data]:::train --> MT[Model Training]:::process
    MT --> TM[Trained Model]:::model
    MT --> RC[Reference Centroid]:::model

    ID[Input Data]:::input --> FE[Feature Engineering]:::process
    FE --> P[Similarity Calculation]:::process
    TM --> P
    RC --> P
    P --> AS[Anomaly Score]:::output

    classDef train fill:#a5d6a7,stroke:#388e3c,stroke-width:2px,col
    classDef input fill:#81c784,stroke:#388e3c,stroke-width:2px,col
    classDef process fill:#64b5f6,stroke:#1565c0,stroke-width:2px,c
    classDef model fill:#ffcc80,stroke:#ef6c00,stroke-width:2px,col
    classDef output fill:#ce93d8,stroke:#6a1b9a,stroke-width:2px,co
```

Components

- **Model Registry**: Stores trained models
- **Feature Pipeline**: Standardized feature extraction
- **Model Ensemble**: Multiple model combination
- **GPU Manager**: Handles GPU allocation

Model Types

The ML-based detection uses:

- **Sentence Transformers**: For text embedding and similarity
- **Reference Centroids**: Pre-computed centers of normality for each field
- **Cosine Similarity**: For anomaly scoring against centroids
- **Triplet Loss Training**: For learning semantic representations
- **Field-specific Models**: Different transformer models per field type

GPU Acceleration

The ML pipeline includes GPU optimizations:

```python
# Automatic batch sizing
optimal_batch_size = get_optimal_batch_size()

# GPU memory management
with gpu_context():
    predictions = model.predict_batch(data)
```

LLM-Based Detection

Architecture

```mermaid
flowchart LR
    I[Input Text]:::input --> T[Tokenization]:::process
    T --> LM[Language Model]:::model
    LM --> PS[Probability Scoring]:::process
    PS --> AS[Anomaly Score]:::output

    TD[Training Data]:::train --> FT[Fine-tuning]:::process
    FT --> LM

    classDef input fill:#81c784,stroke:#388e3c,stroke-width:2px,col
    classDef process fill:#64b5f6,stroke:#1565c0,stroke-width:2px,
    classDef output fill:#ce93d8,stroke:#6a1b9a,stroke-width:2px,co
    classDef train fill:#a5d6a7,stroke:#388e3c,stroke-width:2px,col
    classDef model fill:#ffcc80,stroke:#ef6c00,stroke-width:2px,col
```

Components

- **Fine-tuned Language Models**: Field-specific masked language models
- **Tokenizer**: Text preprocessing and token probability calculation
- **Probability Scorer**: Converts token probabilities to anomaly scores
- **Dynamic Context Encoder**: Optional contextual information integration

Features

- Domain-specific language modeling
- Token probability-based anomaly scoring

- Fine-tuned understanding of field patterns
- Optional temporal and categorical context

Integration Patterns

The system supports multiple integration approaches:

Sequential Processing

Detection methods can be run in sequence, with early exit on first detection:
- Fast rule-based checks first
- Pattern-based analysis using JSON configuration rules
- ML models for complex patterns
- LLM for difficult cases (if enabled)

Parallel Processing

Multiple detectors can run simultaneously:
- All methods process the same data in parallel
- Results are aggregated based on configuration
- Supports different aggregation strategies (union, intersection, voting)

Performance Characteristics

Latency Comparison

Method	Single Record	1K Records	100K Records
Rule-Based	<1ms	~10ms	~1s
Pattern-Based	~5ms	~50ms	~5s
ML-Based (CPU)	~10ms	~100ms	~10s
ML-Based (GPU)	~50ms	~200ms	~2s
LLM-Based	~500ms	~5s	~500s

Accuracy Trade-offs

- **Rule-Based**: High precision, low recall
- **Pattern-Based**: Balanced precision/recall
- **ML-Based**: High recall, tunable precision
- **LLM-Based**: High accuracy, expensive

Configuration

Method Selection

```json
{
  "detection_methods": {
    "rule_based": {
      "enabled": true,
      "priority": 1
    },
    "pattern_based": {
      "enabled": true,
      "priority": 2
    },
    "ml_based": {
      "enabled": true,
      "priority": 3,
      "use_gpu": true
    },
    "llm_based": {
      "enabled": false,
      "priority": 4
    }
  }
}
```

Threshold Configuration

Each method supports configurable thresholds:

```json
{
  "thresholds": {
    "pattern_based": {
      "statistical_outlier": 3.0,
      "frequency_threshold": 0.01
    },
    "ml_based": {
      "anomaly_score": 0.7,
      "confidence_threshold": 0.8
    }
  }
}
```

Theoretical Foundations

Rule-Based Validation

Theory: Grounded in formal logic and domain expertise, providing deterministic results based on predefined constraints.

Key Concepts:
- Boolean logic for constraint checking
- Domain-specific business rules
- Format validation using regular expressions
- Hierarchical rule application

Example Rules:
- Material must contain percentage and fiber name
- Color names must be from approved list
- Sizes must follow standard format (S, M, L, XL)

Pattern-Based Detection

Theory: Statistical pattern recognition combined with rule-based matching to identify anomalies that deviate from expected patterns.

Key Concepts:
- Frequency analysis for rare values
- Pattern matching using regex
- Statistical outlier detection
- Known value whitelisting

Mathematical Foundation:
- Z-score for statistical outliers: `z = (x - μ) / σ`
- Frequency threshold: Values appearing < 1% are flagged
- Pattern confidence: Based on match percentage

ML-Based Detection

Theory: Uses deep learning embeddings to capture semantic meaning and identify anomalies through vector similarity.

Key Concepts:
- Sentence transformers for text embedding
- Centroid-based anomaly detection
- Cosine similarity for semantic comparison
- Triplet loss for model training

Mathematical Foundation:
- Embedding generation: `e = transformer(text)`
- Centroid calculation: `c = mean(embeddings)`
- Anomaly score: `score = 1 - cosine_similarity(e, c)`
- Threshold: Typically 0.7-0.8 based on validation

LLM-Based Detection

Theory: Leverages large language models to understand context and identify complex linguistic anomalies.

Key Concepts:
- Contextual understanding using transformers
- Probability-based anomaly scoring
- Few-shot learning for adaptation
- Instruction-following for specific checks

Mathematical Foundation:
- Log probability: `log P(text|context)`
- Perplexity: `exp(-1/n * Σ log P(xi|x`
`- Anomaly threshold: Based on probability distribution`

Progressive Detection Flow

```
Input → Validation (100% confidence)
  ↓ Pass
Pattern Detection (80% confidence)
  ↓ Uncertain
ML Detection (75% confidence)
  ↓ Still uncertain
LLM Detection (70% confidence)
  ↓
Final Decision with Weighted Confidence
```

Best Practices

Method Selection

```
- Use validation for critical business rules
- Apply pattern detection for known formats
- Enable ML for semantic consistency
- Reserve LLM for complex cases
```

Performance Optimization

```
- Run methods in parallel when possible
- Cache ML model embeddings
- Batch LLM requests
- Use GPU acceleration for ML/LLM
```

Accuracy Tuning

```
- Start with conservative thresholds
- Use evaluation mode to measure performance
- Generate optimized weights from results
- Adjust thresholds based on false positive rates
```

Best Practices

1. **Start Simple**: Begin with rule-based validation
2. **Add Complexity Gradually**: Layer detection methods
3. **Monitor Performance**: Track latency and accuracy

4. **Tune Thresholds**: Adjust based on feedback
5. **Cache Results**: Avoid redundant computations
6. **Parallelize**: Use all available cores/GPUs

Future Enhancements

- **Online Learning**: Continuous model updates
- **Federated Detection**: Distributed anomaly detection
- **Active Learning**: Human-in-the-loop improvements
- **Multi-modal Detection**: Combining structured and unstructured data

---

user-guides > running-detection

Running Detection

This guide covers how to run data quality detection on your datasets using various methods and configurations.

Basic Detection

The simplest way to run detection is using the demo command:

```bash
python main.py single-demo --data-file your_data.csv
```

This runs detection with default settings on your entire dataset.

Configuring Detection Methods

You can enable specific detection methods:

```bash
python main.py single-demo \
    --data-file your_data.csv \
    --enable-validation \    # Rule-based validation
    --enable-pattern \       # Pattern-based anomaly detection
    --enable-ml \           # Machine learning detection
    --enable-llm            # Language model detection
```

Setting Detection Thresholds

Adjust sensitivity for each detection method:

```
bash
python main.py single-demo \
    --data-file your_data.csv \
    --validation-threshold 0.0 \    # Most strict (0.0)
    --anomaly-threshold 0.7 \       # Medium confidence
    --ml-threshold 0.7 \            # Medium confidence
    --llm-threshold 0.6             # Slightly more lenient
```

Threshold Guidelines

- **0.0**: Highest confidence, fewest false positives
- **0.5**: Balanced detection
- **0.8**: More sensitive, may have more false positives
- **1.0**: Detect everything (not recommended)

Working with Large Datasets

For large datasets, use these strategies:

1. Sample Processing

To process a sample of your data, first create a subset:

```
bash
```

Create a sample file

```
head -n 1000 large_data.csv > sample_data.csv
```

Run detection on the sample

```
python main.py single-demo \
    --data-file sample_data.csv
```

2. Core Fields Only

```
bash
python main.py single-demo \
    --data-file large_data.csv \
    --core-fields-only
```

3. Specific Fields

Note: The single-demo command processes all configured fields. To process specific fields only, you can modify your brand configuration or use the multi-eval command for field-specific evaluation.

Output Options

Specify Output Directory

```bash
python main.py single-demo \
    --data-file your_data.csv \
    --output-dir results/2024-01-detection
```

Output Files

The single-demo command automatically generates:
- JSON reports (report.json, viewer_report.json)
- CSV summaries (anomaly_summary.csv)
- Result files with detection information
- Confusion matrix visualizations (if evaluation mode)

Using Weighted Combination

For optimized detection based on historical performance:

```bash
python main.py single-demo \
    --data-file your_data.csv \
    --use-weighted-combination \
    --weights-file detection_weights.json
```

Injection Testing

Test detection performance with synthetic errors:

```bash
python main.py single-demo \
    --data-file clean_data.csv \
    --injection-intensity 0.2    # Inject errors in 20% of data
```

Note: Error injection is randomized. The exact errors will vary between runs.

Viewing Results

After detection completes:

1. **HTML Viewer**: Open `single_sample_multi_field_demo/data_quality_viewer.html` in your browser
2. **CSV Results**: Review the generated CSV files in your output directory
3. **JSON Report**: Detailed metrics in the report.json file

Example Workflows

Quick Quality Check

bash

Fast check with pattern detection

```
python main.py single-demo \
    --data-file daily_upload.csv \
    --enable-pattern
```

Full Production Run

bash

Comprehensive detection with all

```
python main.py single-demo \
    --data-file production_data.csv \
    --enable-validation \
    --enable-pattern \
    --enable-ml \
    --validation-threshold 0.0 \
    --anomaly-threshold 0.6 \
    --ml-threshold 0.7 \
    --output-dir results/production_$(date +%Y%m%d)
```

Testing New Configuration

bash

Test with injection to validate c

```
python main.py single-demo \
    --data-file test_data.csv \
    --injection-intensity 0.3 \
    --enable-validation \
    --enable-pattern
```

Troubleshooting

Out of Memory Errors

- Use `--core-fields-only`
- Create a smaller sample file first
- Disable ML/LLM detection
- Process in batches

Slow Performance

- Enable GPU if available
- Use `--enable-pattern` only for quick checks
- Use `--core-fields-only` to process fewer fields

No Detections Found

- Check field mappings in brand configuration
- Lower detection thresholds
- Verify data format matches expectations

---

user-guides > analyzing-results

Analyzing Results

This guide covers how to analyze your data before detection and interpret the results afterwards.

Before Detection: Understanding Your Data

Explore Data Structure

Before running detection, understand your data:

bash

View the first few rows

```
head -n 10 your_data.csv
```

Count records

```
wc -l your_data.csv
```

Analyze Individual Columns

Use the `analyze-column` command to understand field characteristics:

```bash
python main.py analyze-column your_data.csv column_name
```

This shows:
- Unique value distribution
- Common patterns
- Potential anomalies
- Recommended detection methods

Identify Key Fields

Focus on fields that are:
- Critical for business operations
- Prone to quality issues
- Used in downstream processes

After Detection: Interpreting Results

Result Files Overview

After running detection, you'll find in your output directory:

- `report.json` - Comprehensive detection results
- `viewer_report.json` - Formatted for the web viewer
- `anomaly_summary.csv` - Summary of all detected anomalies
- `sample_with_errors.csv` - Data with injected errors (if using evaluation mode)
- `sample_with_results.csv` - Original data with detection results
- `confusion_matrix/` - Performance visualization images

Using the HTML Viewer

The interactive viewer is the easiest way to explore results:

1. Open `single_sample_multi_field_demo/data_quality_viewer.html` in your browser
2. Upload files from your output directory:
- CSV file (anomaly_summary.csv or sample_with_results.csv)
- JSON report (viewer_report.json)
3. Use the interface to:
- Filter by confidence level
- Sort by different criteria
- View detailed explanations
- Export filtered results

Understanding the CSV Output

The anomaly_summary.csv contains:

- `row_index` - Row number in original data
- `column_name` - Field where anomaly was detected
- `detection_method` - Which method found the anomaly
- `error_type` - Type of issue detected
- `confidence` - Detection confidence (0-1)
- `details` - Additional information
- `error_data` - The problematic value

Interpreting Confidence Scores

- **0.8-1.0**: High confidence - likely real issues
- **0.5-0.8**: Medium confidence - review recommended
- **0.0-0.5**: Low confidence - possible false positives

Reading the JSON Report

The report.json contains:
- Detection summary statistics
- Performance metrics (if evaluation mode)
- Field-by-field breakdown
- Method-specific results

Key sections:
- `summary`: Overall detection statistics
- `field_results`: Results per field
- `metrics`: Performance metrics (precision, recall, F1)

Analysis Workflow

1. Quick Overview

```python
import json
import pandas as pd
```

Load results

```
with open('report.json', 'r') as f:
    report = json.load(f)
```

Check summary

```
print(f"Total anomalies: {report['summary'].get('total_anomalies',
if 'metrics' in report:
    print(f"Precision: {report['metrics'].get('precision', 0):.2f}'
    print(f"Recall: {report['metrics'].get('recall', 0):.2f}")
```

2. Deep Dive Analysis

```python
```

Load anomaly details

```
df = pd.read_csv('anomaly_summary.csv')
```

Analyze by field

```
field_counts = df['column_name'].value_counts()
print("Issues by field:")
print(field_counts)
```

High confidence issues

```
high_conf = df[df['confidence'] > 0.8]
print(f"\nHigh confidence issues: {len(high_conf)}")
```

3. Pattern Analysis

Group issues to find patterns:

```python
# Group by error type
error_patterns = df.groupby(['column_name', 'error_type']).size()
print("\nError patterns:")
print(error_patterns.sort_values(ascending=False).head(10))
```

Best Practices

For Data Analysis

- Start with critical business fields
- Use analyze-column before full detection
- Document expected patterns

For Result Interpretation

- Focus on high-confidence detections first
- Look for patterns across multiple records
- Consider business context when reviewing
- Export and share findings with stakeholders

Troubleshooting

No Results Generated

- Check that detection methods were enabled
- Verify data file format is correct
- Review console output for errors

Too Many False Positives

- Increase detection thresholds
- Review and update validation rules
- Consider training custom ML models

Missing Expected Issues

- Lower detection thresholds
- Enable additional detection methods

```
- Check field mappings in brand configuration
```

---

user-guides > optimization

Performance Optimization Guide

This guide covers evaluating detection performance and optimizing the system for your specific data.

Performance Evaluation

Understanding Metrics

The system tracks key performance metrics:

- **Precision**: How many detections were correct (true positives / all positives)
- **Recall**: How many errors were caught (true positives / all actual errors)
- **F1 Score**: Harmonic mean of precision and recall
- **Detection Rate**: Percentage of records flagged
- **Confidence Distribution**: Spread of confidence scores

Running Evaluation

Basic Evaluation

Use multi-eval for systematic performance testing:

```
bash
python main.py multi-eval your_data.csv \
    --field material \
    --num-samples 100 \
    --output-dir evaluation_results
```

Field-Specific Evaluation

Test specific fields with different detectors:

```
bash
python main.py multi-eval your_data.csv \
    --field color_name \
    --ml-detector \
    --run all \
    --num-samples 50
```

Synthetic Error Testing

Multi-eval automatically injects errors for evaluation:

```bash
python main.py multi-eval clean_data.csv \
    --field material \
    --error-probability 0.2 \
    --max-errors 3 \
    --num-samples 100
```

Analyzing Evaluation Results

Review the generated reports to understand:
- Detection accuracy per field
- Method effectiveness comparison
- Error type distribution
- Confidence score calibration

Weighted Combination Optimization

Understanding Weighted Combination

The system can combine detection methods using optimized weights based on their effectiveness for specific fields. This improves accuracy by relying more on methods that perform well for particular data types.

Generating Detection Weights

After evaluation, generate optimized weights:

```bash
python single_sample_multi_field_demo/generate_detection_weights.py
    -i evaluation_results/report.json \
    -o detection_weights.json
```

Weight File Structure

```
json
{
  "field_weights": {
    "material": {
      "validation": 0.45,
      "pattern_based": 0.35,
      "ml_based": 0.15,
      "llm_based": 0.05
    },
    "color_name": {
      "validation": 0.30,
      "pattern_based": 0.25,
      "ml_based": 0.35,
      "llm_based": 0.10
    }
  },
  "default_weights": {
    "validation": 0.40,
    "pattern_based": 0.30,
    "ml_based": 0.20,
    "llm_based": 0.10
  }
}
```

Using Optimized Weights

Apply weights in detection:

```
bash
python main.py single-demo \
    --data-file your_data.csv \
    --use-weighted-combination \
    --weights-file detection_weights.json
```

Threshold Optimization

Finding Optimal Thresholds

Use ML curves to find the best thresholds:

```
bash
python main.py ml-curves your_data.csv \
    --fields material color_name \
    --output-dir threshold_analysis
```

This generates:
- Precision-recall curves

- F1 score vs threshold plots
- Recommended threshold values

Applying Optimized Thresholds

```
bash
python main.py single-demo \
    --data-file your_data.csv \
    --validation-threshold 0.0 \
    --anomaly-threshold 0.75 \
    --ml-threshold 0.82 \
    --llm-threshold 0.65
```

Performance Tuning Strategies

1. Speed Optimization

For faster processing:
- Use `--core-fields-only` for essential fields
- Disable expensive methods (LLM) when not needed
- Process in batches for large datasets
- Enable GPU acceleration for ML/LLM

2. Accuracy Optimization

For better detection:
- Generate and use weighted combinations
- Fine-tune thresholds per field
- Train custom ML models
- Update validation rules regularly

3. Memory Optimization

For large datasets:
- Process fields sequentially
- Reduce batch sizes
- Use validation-only for initial screening
- Split data into smaller chunks

Continuous Improvement Workflow

1. Baseline Establishment

bash

Initial evaluation

```
python main.py multi-eval baseline_data.csv \
    --field all \
    --num-samples 200 \
    --output-dir baseline_results
```

2. Weight Generation

bash

Generate optimized weights

```
python single_sample_multi_field_demo/generate_detection_weights.py
    -i baseline_results/report.json \
    -o weights_v1.json
```

3. Performance Monitoring

bash

Regular evaluation with weights

```
python main.py single-demo \
    --data-file daily_data.csv \
    --use-weighted-combination \
    --weights-file weights_v1.json \
    --output-dir monitoring/$(date +%Y%m%d)
```

4. Periodic Re-optimization

Re-evaluate and update weights quarterly or when:
- Data patterns change significantly
- New fields are added
- Detection accuracy drops
- Business requirements change

Best Practices

For Evaluation

```
- Use representative data samples
- Test with realistic error rates
- Evaluate all critical fields
- Document baseline performance
```

For Optimization

```
- Start with default weights
- Optimize incrementally
- Validate improvements
- Keep historical weights for comparison
```

For Production

```
- Monitor detection rates
- Track false positive trends
- Update weights periodically
- Maintain separate weights for different data types
```

Troubleshooting Performance Issues

Low Precision (Too Many False Positives)

```
- Increase detection thresholds
- Review and update validation rules
- Reduce weights for noisy methods
- Consider field-specific thresholds
```

Low Recall (Missing Real Errors)

```
- Lower detection thresholds
- Enable additional detection methods
- Increase weights for effective methods
- Add more validation rules
```

Slow Processing

```
- Profile to identify bottlenecks
- Disable unnecessary methods
- Optimize batch sizes
- Consider parallel processing
```

Unstable Results

```
- Check for data quality issues
- Verify consistent preprocessing
- Use larger evaluation samples
- Average results across multiple runs
```

---

reference > cli

CLI Reference

This document provides a comprehensive reference for all command-line interfaces in the Data Quality Detection System.

Main Entry Point

The system provides a unified entry point through `main.py`:

```
bash
python main.py [command] [options]
```

Available Commands

```
mermaid
graph TD
    A[main.py] --> B[single-demo]
    A --> C[multi-eval]
    A --> D[ml-train]
    A --> E[llm-train]
    A --> F[analyze-column]
    A --> G[ml-curves]
```

Command Reference

single-demo

```
Run single sample demonstration with comprehensive detection.
```

```
bash
python main.py single-demo [options]
```

Required Arguments
- `--data-file PATH`: Path to input CSV file

Optional Arguments

Output Options:
- `--output-dir PATH`: Output directory (default: `demo_results`)

Detection Methods:
- `--enable-validation`: Enable validation detection
- `--enable-pattern`: Enable pattern-based detection
- `--enable-ml`: Enable ML-based detection
- `--enable-llm`: Enable LLM-based detection

Note: If no detection methods are explicitly enabled, all available methods run by default.

Thresholds:
- `--validation-threshold FLOAT`: Validation threshold (default: 0.0)
- `--anomaly-threshold FLOAT`: Pattern anomaly threshold (default: 0.7)
- `--ml-threshold FLOAT`: ML threshold (default: 0.7)
- `--llm-threshold FLOAT`: LLM threshold (default: 0.6)

Error Injection:
- `--injection-intensity FLOAT`: Probability of injecting issues per cell (default: 0.2)
- `--max-issues-per-row INT`: Maximum fields to corrupt per row (default: 2)

LLM Options:
- `--llm-few-shot-examples`: Enable few-shot examples for LLM
- `--llm-temporal-column STR`: Column with temporal info for LLM
- `--llm-context-columns STR`: Comma-separated context columns

Combination Strategy:
- `--use-weighted-combination`: Use weighted score combination
- `--weights-file PATH`: Path to detection weights JSON (default: detection_weights.json)
- `--generate-weights`: Generate weights after completion
- `--weights-output-file PATH`: Output for generated weights
- `--baseline-weight FLOAT`: Weight for poor performing methods (default: 0.1)

Field Selection:
- `--core-fields-only`: Process only core fields (material, color_name, category, size, care_instructions)

Examples

bash

Basic usage

```
python main.py single-demo --data-file data/products.csv
```

With error injection for testing

```
python main.py single-demo \
    --data-file data/products.csv \
    --injection-intensity 0.2 \
    --output-dir results/test
```

Specific methods and thresholds

```
python main.py single-demo \
    --data-file data/products.csv \
    --enable-validation \
    --enable-ml \
    --ml-threshold 0.8
```

Using weighted combination

```
python main.py single-demo \
    --data-file data/products.csv \
    --use-weighted-combination \
    --weights-file config/detection_weights.json
```

`multi-eval`

`Run evaluation across multiple samples for performance analysis.`

```
bash
python main.py multi-eval [options]
```

Arguments

Required:
- `--field FIELD`: Target field to validate (e.g., 'material', 'care_instructions')

Optional:
- `--validator STR`: Validator name (defaults to field name)
- `--anomaly-detector STR`: Anomaly detector name (defaults to validator name)
- `--ml-detector`: Enable ML-based detection
- `--llm-detector`: Enable LLM-based detection
- `--run CHOICE`: What to run: validation, anomaly, ml, llm, both, all (default: both)

Sampling:
- `--num-samples INT`: Number of samples to generate (default: 32)
- `--max-errors INT`: Max errors per sample (default: 3)
- `--error-probability FLOAT`: Error injection probability (default: 0.1)

Output:
- `--output-dir PATH`: Results directory (default: evaluation_results)
- `--ignore-errors ERROR [ERROR ...]`: Error rules to ignore
- `--ignore-fp`: Ignore false positives in evaluation

Thresholds:
- `--validation-threshold FLOAT`: Validation threshold (default: 0.0)
- `--anomaly-threshold FLOAT`: Anomaly threshold (default: 0.7)
- `--ml-threshold FLOAT`: ML threshold (default: 0.7)
- `--llm-threshold FLOAT`: LLM threshold (default: 0.6)
- `--high-confidence-threshold FLOAT`: High confidence threshold (default: 0.8)

Performance:
- `--batch-size INT`: Batch size (default: auto)
- `--max-workers INT`: Parallel workers (default: 7)

Examples

```
bash
```

Basic evaluation

```
python main.py multi-eval \
    --field material \
    --num-samples 50
```

Full evaluation with all detectors

```
python main.py multi-eval \
    --field care_instructions \
    --run all \
    --ml-detector \
    --llm-detector \
    --num-samples 100
```

`ml-train`

`Train ML-based anomaly detection models or run anomaly checks.`

```bash
python main.py ml-train [options]
```

Arguments

Options:
- `--use-hp-search`: Use recall-focused hyperparameter search
- `--hp-trials INT`: Number of hyperparameter search trials (default: 15)
- `--fields FIELD [FIELD ...]`: Fields to include in training (default: all)
- `--check-anomalies FIELD`: Run anomaly check on given field
- `--threshold FLOAT`: Similarity threshold for anomaly detection (default: 0.6)
- `--output PATH`: Output CSV file for anomaly check results

Note: This command primarily manages pre-trained models and hyperparameter search rather than training from scratch.

Examples

```
bash
```

Run hyperparameter search

```
python main.py ml-train \
    --use-hp-search \
    --fields material color_name
```

Check anomalies in a field

```
python main.py ml-train \
    --check-anomalies material \
    --threshold 0.7 \
    --output anomaly_results.csv
```

analyze-column

`Analyze a specific column in a CSV file.`

```
bash
python main.py analyze-column CSV_FILE [FIELD_NAME]
```

Arguments

Positional:
- `CSV_FILE`: Path to the CSV file to analyze
- `FIELD_NAME`: Name of the field to analyze (default: color_name)

Note: Brand configuration is managed through the brand_configs directory, not command-line arguments.

Examples

bash

Analyze default column (color_name

python main.py analyze-column data/products.csv

Analyze specific column

python main.py analyze-column data/products.csv material

ml-curves

`Generate precision-recall and ROC curves for ML-based and LLM-based anomaly detection.`

```
bash
python main.py ml-curves DATA_FILE [options]
```

Arguments

Positional:
- `DATA_FILE`: Path to the CSV data file

Optional:
- `--detection-type {ml,llm}`: Type of detection to evaluate (default: ml)
- `--fields FIELD [FIELD ...]`: Specific fields to generate curves for (default: all available)
- `--output-dir PATH`: Output directory for curves (default: detection_curves)
- `--thresholds FLOAT [FLOAT ...]`: Specific thresholds to test (default: ML=0.1-0.95, LLM=-0.5-0.1)

Examples

bash

Generate ML curves for all fields

```
python main.py ml-curves data/products.csv
```

Generate LLM curves for specific

```
python main.py ml-curves data/products.csv \
    --detection-type llm \
    --fields material color_name \
    --output-dir llm_curves
```

Test specific thresholds

```
python main.py ml-curves data/products.csv \
    --thresholds 0.5 0.6 0.7 0.8 0.9
```

Global Options

Note: The current implementation has limited global options. Most configuration is done through command-specific arguments or configuration files.

Practical Examples

Detection Workflow

bash

1. Analyze your data first

```
python main.py analyze-column products.csv material
```

2. Run detection with appropriate

```
python main.py single-demo --data-file products.csv --enable-valida
```

3. Evaluate performance

```
python main.py multi-eval products.csv --field material --num-sampl
```

4. Generate optimized weights

```
python single_sample_multi_field_demo/generate_detection_weights.py
```

5. Use optimized configuration

```
python main.py single-demo --data-file products.csv --use-weighted-
```

Training Custom Models

bash

Train ML models for specific fiel

python main.py ml-train training_data.csv --fields "material color_

Train with hyperparameter optimiz

python main.py ml-train training_data.csv --use-hp-search --hp-tria

Generate performance curves

python main.py ml-curves test_data.csv --fields material --output-c

Configuration Files

Command Arguments from File

Note: The --args-file option is not currently implemented. To reuse command configurations, consider using shell scripts or aliases.

Output Formats

Standard Output Structure

All commands create consistent output structure:

```
output_dir/
├── detection_report.json    # Detailed results
├── summary_report.txt       # Human-readable summary
├── metrics.json            # Performance metrics
├── sample_with_results.csv # Data with detection results
└── logs/
    └── detection.log       # Detailed logs
```

JSON Report Format

```
json
{
    "metadata": {
        "timestamp": "2024-01-15T10:30:00Z",
        "version": "1.0",
        "command": "single-demo",
        "parameters": {...}
    },
    "summary": {
        "total_records": 1000,
        "errors_detected": 150,
        "detection_methods": ["validation", "ml"]
    },
    "results": {...}
}
```

Exit Codes

Code	Meaning
0	Success
1	General error
2	Invalid arguments
3	File not found
4	Configuration error
5	Model/resource error

Advanced Usage

Piping and Chaining

bash

Analyze column and pipe to detecti

```
python main.py analyze-column --data-file data.csv --column materia
python main.py single-demo --data-file data.csv --fields material
```

Chain multiple evaluations

```
for intensity in 0.1 0.2 0.3; do
    python main.py multi-eval \
        --data-file data.csv \
        --injection-intensity $intensity \
        --output-dir results/intensity_$intensity
done
```

Batch Processing

bash

Process multiple files

```
for file in data/*.csv; do
    python main.py single-demo \
        --data-file "$file" \
        --output-dir "results/$(basename $file .csv)"
done
```

Integration Examples

```bash
# Cron job for daily monitoring

0 2 * * * /usr/bin/python /app/main.py single-demo \
    --data-file /data/daily_export.csv \
    --output-dir /reports/$(date +\%Y\%m\%d) \
    --email-report admin@example.com

# CI/CD pipeline integration

python main.py multi-eval \
    --data-file $CI_DATA_FILE \
    --threshold-config $CI_THRESHOLD_CONFIG \
    --fail-on-degradation
```

Troubleshooting

Common Issues

1. **Command not found**

```bash
# Ensure you're in the project directory
cd /path/to/detection-system
python main.py --help
```

2. **Module import errors**

```bash
# Activate virtual environment
source venv/bin/activate
# Reinstall dependencies
pip install -r requirements.txt
```

3. **Memory errors**

```
bash
  # Reduce batch size
  python main.py single-demo --batch-size 50
  # Process sample
  python main.py single-demo --sample-size 1000
```

Debug Mode

Enable detailed debugging:

```
bash
python main.py single-demo \
    --data-file data.csv \
    --debug \
    --verbose \
    --log-file debug.log
```

---

reference > configuration

Configuration Reference

This reference covers all configuration options for the Data Quality Detection System.

Brand Configuration

Brand configurations define how the system maps your data columns to standard fields and sets brand-specific parameters.

Location

Brand configurations are stored in the `brand_configs/` directory:

```
brand_configs/
├── esqualo.json      # Example brand config
└── your_brand.json   # Your custom brand config
```

Configuration Structure

```
json
{
    "brand_name": "your_brand",

    "field_mappings": {
        "material": "Material_Column",
        "color_name": "Color_Description",
        "category": "Product_Category",
        "size": "Size_Value",
        "product_name": "Product_Name",
        "product_id": "SKU",
        "description": "Long_Description"
    },

    "default_data_path": "data/your_data.csv",

    "custom_thresholds": {
        "validation_threshold": 0.0,
        "anomaly_threshold": 0.7,
        "ml_threshold": 0.7,
        "llm_threshold": 0.6
    }
}
```

Field Mappings

Map your CSV column names to standard field types:

```
json
"field_mappings": {
    "material": "Material_Column",    // Maps 'Material_Column' to
    "color_name": "Color_Description" // Maps 'Color_Description' t
}
```

Standard Fields

The system recognizes these standard field types:

- **Product Attributes**
- `material` - Product material composition
- `color_name` - Color descriptions
- `size` - Size values
- `category` - Product categories
- `subcategory` - Product subcategories

- **Identifiers**
- `product_name` - Product names
- `product_id` - Product IDs/SKUs
- `brand_name` - Brand names

- **Descriptions**
- `description` - Product descriptions
- `care_instructions` - Care/maintenance instructions

- **Business Fields**
- `price` - Price values
- `country` - Country codes/names
- `gender` - Gender classifications

Default Paths

```
json
"default_data_path": "data/products.csv"  // Default input file for
```

Custom Thresholds

Override global detection thresholds:

```
json
"custom_thresholds": {
    "validation_threshold": 0.0,  // 0.0 = strictest
    "anomaly_threshold": 0.7,     // 0.0-1.0 range
    "ml_threshold": 0.7,          // 0.0-1.0 range
    "llm_threshold": 0.6          // 0.0-1.0 range
}
```

Detection Configuration

Validation Rules

Validation rules are defined in JSON files under `validators//error_messages.json`:

```
json
{
    "EMPTY_VALUE": {
        "message": "Material value is empty",
        "severity": "high"
    },
    "INVALID_FORMAT": {
        "message": "Material format is invalid: {details}",
        "severity": "medium"
    }
}
```

Pattern Rules

Pattern-based detection rules are in `anomaly_detectors/pattern_based/rules/.json`:

```json
{
    "known_values": ["cotton", "polyester", "wool", "silk"],
    "patterns": [
        {
            "name": "percentage_pattern",
            "regex": "\\d+%\\s+\\w+",
            "description": "Percentage followed by material"
        }
    ],
    "suspicious_patterns": [
        {
            "pattern": "test|temp|xxx",
            "reason": "Likely test data"
        }
    ]
}
```

ML Model Configuration

ML models are configured during training:

```bash
# Training configuration via CLI
python main.py ml-train data.csv \
    --fields "material color_name" \
    --use-hp-search \
    --hp-trials 20
```

Model artifacts are stored in:
- `anomaly_detectors/ml_based/models//`
- `anomaly_detectors/llm_based/models//`

Environment Variables

Control system behavior with environment variables:

bash

GPU configuration

```
export CUDA_VISIBLE_DEVICES=0
```

Memory limits (not currently impl

```
export MAX_WORKERS=4
export BATCH_SIZE=32
```

Weight Configuration

Optimized detection weights are stored in JSON files:

```json
{
  "field_weights": {
    "material": {
      "validation": 0.45,
      "pattern_based": 0.35,
      "ml_based": 0.15,
      "llm_based": 0.05
    }
  },
  "default_weights": {
    "validation": 0.40,
    "pattern_based": 0.30,
    "ml_based": 0.20,
    "llm_based": 0.10
  }
}
```

Use with: `--use-weighted-combination --weights-file weights.json`

Output Configuration

Directory Structure

Output directories follow this structure:

```
output_dir/
├── report.json                    # Main detection report
├── viewer_report.json            # HTML viewer format
├── anomaly_summary.csv           # CSV summary
├── sample_with_errors.csv        # Data with injected errors
├── sample_with_results.csv       # Original data with results
└── confusion_matrix/             # Performance visualizations
    ├── overall_matrix.png
    └── per_field_matrix.png
```

Report Configuration

Control report generation with CLI flags:
- `--generate-weights` - Generate weight recommendations
- `--core-fields-only` - Process only essential fields
- `--output-dir PATH` - Specify output location

Performance Configuration

Memory Management

- Use `--core-fields-only` to process only essential fields
- Adjust batch sizes for ML/LLM processing
- Process large files in chunks

GPU Configuration

bash

Enable GPU

export CUDA_VISIBLE_DEVICES=0

Disable GPU

python main.py single-demo --device cpu

Parallel Processing

The system automatically parallelizes:
- Field processing within detection methods
- Multiple detection methods (when enabled)

Creating a New Brand Configuration

1. **Copy the template**:

```
bash
  cp brand_configs/esqualo.json brand_configs/new_brand.json
```

2. **Edit field mappings**:

```
json
  {
    "brand_name": "new_brand",
    "field_mappings": {
      "material": "Your_Material_Column",
      "color_name": "Your_Color_Column"
    }
  }
```

3. **Test the configuration**:

```
bash
  python main.py analyze-column your_data.csv Your_Material_Column
```

4. **Run detection**:

```
bash
  python main.py single-demo --data-file your_data.csv
```

Configuration Best Practices

Field Mapping

- Map only columns that exist in your data
- Use exact column names (case-sensitive)
- Start with core fields, add more gradually

Thresholds

- Start with default thresholds
- Adjust based on evaluation results
- Document threshold changes and reasons

File Organization

```
- Keep one config file per brand/dataset
- Use descriptive file names
- Version control configuration files
```

Testing

```
- Test configurations with small data samples first
- Verify field mappings with analyze-column
- Run evaluation to validate thresholds
```

---

reference > interfaces

Core Interfaces API Reference

This document provides comprehensive API documentation for the core interfaces in the Data Quality Detection System. These interfaces define the contracts that all implementations must follow.

Overview

The system uses abstract base classes (ABCs) to define interfaces, ensuring consistency across different implementations. The main interfaces are:

```
 mermaid
classDiagram
    class AnomalyDetectorInterface {
        <>
        +_detect_anomaly(value, context)
        +learn_patterns(df, column_name)
        +get_detector_args()
        +bulk_detect(df, column_name, batch_size, max_workers)
    }

    class ValidatorInterface {
        <>
        +_validate_entry(value)
        +bulk_validate(df, column_name)
    }

    class ReporterInterface {
        <>
        +generate_report(validation_errors, original_df)
    }
```

AnomalyDetectorInterface

Base interface for all anomaly detection methods.

Location

`anomaly_detectors/anomaly_detector_interface.py`

Methods

_detect_anomaly(value: Any, context: Dict[str, Any] = None) -> Optional[AnomalyError]

Contains the specific anomaly detection logic for a single data entry. This method must be implemented by subclasses.

Parameters:
- `value` (Any): The data from the DataFrame column to be checked for anomalies
- `context` (Optional[Dict[str, Any]]): Optional dictionary containing additional context data

Returns:
- None if no anomaly is detected
- An AnomalyError instance if an anomaly is detected

Example:

```
 python
def _detect_anomaly(self, value, context=None):
    if value not in self.known_patterns:
        return AnomalyError(
            anomaly_type="unknown_value",
            probability=0.85,
            anomaly_data={"value": value, "expected": self.known_pa
        )
    return None
```

learn_patterns(df: pd.DataFrame, column_name: str) -> None

Learns normal patterns from the data to establish a baseline for anomaly detection. This is an optional method that anomaly detectors can override.

Parameters:
- `df` (pd.DataFrame): The input DataFrame containing the data to learn from
- `column_name` (str): The name of the column to learn patterns from

Example:

```
python
detector = PatternBasedDetector()
detector.learn_patterns(clean_data, 'material')
```

get_detector_args() -> Dict[str, Any]

Return arguments needed to recreate this detector instance in a worker process.

Returns:
- Dictionary of arguments that can be passed to the constructor

bulk_detect(df: pd.DataFrame, column_name: str, batch_size: Optional[int], max_workers: int) -> List[AnomalyError]

Detects anomalies in a column and returns a list of AnomalyError objects. This method runs the `_detect_anomaly` logic in parallel batches.

Parameters:
- `df` (pd.DataFrame): The input DataFrame containing the data to be analyzed
- `column_name` (str): The name of the column to check for anomalies
- `batch_size` (Optional[int]): Number of rows per batch. If None, automatically calculated
- `max_workers` (int): Number of parallel workers

Returns:
- List[AnomalyError]: A list of AnomalyError instances

Example:

```
python
anomalies = detector.bulk_detect(data_df, 'material', batch_size=1(
```

Implementations

- `PatternBasedDetector`: Rule-based pattern matching
- `MLAnomalyDetector`: Machine learning based detection
- `LLMAnomalyDetector`: Language model based detection

ValidatorInterface

Base interface for field validators that enforce business rules.

Location

`validators/validator_interface.py`

Methods

_validate_entry(value: Any) -> Optional[ValidationError]

Contains the specific validation logic for a single data entry. This method must be implemented by subclasses.

Parameters:
- `value` (Any): The data from the DataFrame column to be validated

Returns:
- None if the value is valid
- A ValidationError instance if the value is invalid

Example:

```
python
validator = MaterialValidator()
errors = validator.validate("", row_index=5)
```

[ValidationError(type='EMPTY_VALU

bulk_validate(df: pd.DataFrame, column_name: str) -> List[ValidationError]

Validates a column and returns a list of ValidationError objects. This method runs the `_validate_entry` logic for each row.

Parameters:
- `df` (pd.DataFrame): The input DataFrame containing the data to be validated
- `column_name` (str): The name of the column to validate within the DataFrame

Returns:
- List[ValidationError]: A list of ValidationError instances with row context

Example:

```
python
errors = validator.bulk_validate(data_df, 'material')
```

ValidationError Class

```
python
@dataclass
class ValidationError:
    error_type: str          # Error code (e.g., 'EMPTY_VALUE')
    probability: float      # Always 1.0 for validators
    row_index: Optional[int] # Row index where error occurred
    column_name: Optional[str] # Column name where error occurred
    error_data: Any         # The actual problematic value
```

Creating Custom Validators

```
python
from validators.validator_interface import ValidatorInterface
from validators.validation_error import ValidationError



class CustomValidator(ValidatorInterface):
    def __init__(self):
        self.field_type = "custom_field"

    def _validate_entry(self, value):
        # Add validation logic
        if not value:
            return ValidationError(
                error_type="MISSING_VALUE",
                probability=1.0
            )
        return None
```

ReporterInterface

Base interface for report generation across different formats.

Location

`validators/reporter_interface.py`

Methods

generate_report(validation_errors: List[ValidationError], original_df: pd.DataFrame) -> List[Dict[str, Any]]

Generates human-readable messages for a list of validation errors.

Parameters:
- `validation_errors` (List[ValidationError]): The list of ValidationError objects produced by a Validator

- `original_df` (pd.DataFrame): The original DataFrame, useful for providing additional context

Returns:
- List[Dict[str, Any]]: A list of dictionaries, where each dictionary contains:
- `row_index`: The integer index of the row containing the error
- `error_data`: The original problematic data
- `display_message`: A human-readable string explaining the error

Example:

```
python
report = reporter.generate_report(validation_errors, data_df)
```

[{"row_index": 5, "error_data": "'

Report Structure

Standard report structure:

```
python
{
    "summary": {
        "total_records": 1000,
        "errors_found": 150,
        "detection_methods": ["validation", "pattern", "ml"],
        "timestamp": "2024-01-01T00:00:00Z"
    },
    "field_results": {
        "material": {
            "errors": 25,
            "error_rate": 0.025,
            "top_errors": [...]
        }
    },
    "detailed_results": [...]
}
```

UnifiedDetectorInterface

High-level interface that combines multiple detection methods.

Location

`multi_sample_evaluation/unified_detection_interface.py`

Methods

detect_issues(df: pd.DataFrame, field_name: str, config: DetectionConfig) -> List[DetectionResult]

Runs detection using configured methods.

Parameters:
- `df` (pd.DataFrame): Input DataFrame
- `field_name` (str): Standard field name to analyze
- `config` (DetectionConfig): Configuration specifying thresholds and enabled methods

Returns:
- List[DetectionResult]: List of all detected issues

Example:

```
 python
config = DetectionConfig(
    validation_threshold=0.0,
    anomaly_threshold=0.7,
    ml_threshold=0.75,
    enable_validation=True,
    enable_anomaly_detection=True,
    enable_ml_detection=True
)
results = detector.detect_issues(data_df, 'material', config)
```

DetectionConfig

Configuration class for controlling detection behavior:

```
 python
@dataclass
class DetectionConfig:
    validation_threshold: float
    anomaly_threshold: float
    ml_threshold: float
    llm_threshold: float = 0.6
    enable_validation: bool = True
    enable_anomaly_detection: bool = True
    enable_ml_detection: bool = True
    enable_llm_detection: bool = False
```

DetectionResult

Unified result format for all detection methods:

```python
@dataclass
class DetectionResult:
    row_index: int
    field_name: str
    detection_type: DetectionType
    error_code: str
    confidence: float
    message: str
    details: Dict[str, Any]
    value: Any
```

Usage Examples

Complete Detection Pipeline

```
python
from anomaly_detectors.ml_based.ml_anomaly_detector import MLAnomal
from validators.material.validate import Validator as MaterialValid
from validators.report import Reporter
```

Initialize components

```
ml_detector = MLAnomalyDetector()
validator = MaterialValidator()
reporter = Reporter('material')
```

Load and prepare data

```
data = pd.read_csv('data.csv')
```

Detect validation errors

```
validation_errors = validator.bulk_validate(data, 'material')
```

Detect ML anomalies

```
ml_anomalies = ml_detector.bulk_detect(data, 'material', batch_size
```

Generate human-readable report

```
validation_report = reporter.generate_report(validation_errors, dat
```

Custom Implementation

```
python
from anomaly_detectors.anomaly_detector_interface import AnomalyDet
from anomaly_detectors.anomaly_error import AnomalyError


class CustomDetector(AnomalyDetectorInterface):
    def __init__(self, threshold=0.8):
        self.threshold = threshold
        self.patterns = {}

    def _detect_anomaly(self, value, context=None):
        # Implement detection logic
        if value not in self.patterns:
            return AnomalyError(
                anomaly_type="unknown_pattern",
                probability=0.85,
                anomaly_data={"value": value}
            )
        return None

    def learn_patterns(self, df, column_name):
        # Implement pattern learning
        self.patterns = set(df[column_name].unique())

    def get_detector_args(self):
        return {"threshold": self.threshold}
```

Best Practices

1. **Interface Compliance**: Always implement all required methods
2. **Error Handling**: Handle edge cases gracefully
3. **Documentation**: Document custom implementations thoroughly
4. **Testing**: Write unit tests for interface implementations
5. **Performance**: Implement bulk methods for efficiency

---

development > adding-fields

Adding New Fields Guide

This guide walks you through the process of adding support for new fields in the Data Quality Detection System. The system's modular architecture makes it straightforward to extend with new field types.

Overview

Adding a new field involves:
1. Creating validation rules
2. Defining pattern-based detection rules
3. Training ML models (optional)
4. Configuring field mappings
5. Testing the implementation

```
mermaid
flowchart TD
    A[New Field Request] --> B[Analyze Field Data]
    B --> C[Create Validator]
    B --> D[Define Patterns]
    B --> E[Train ML Model]

    C --> F[Test Validation]
    D --> G[Test Pattern Detection]
    E --> H[Test ML Detection]

    F & G & H --> I[Configure Field Mapping]
    I --> J[Integration Testing]
    J --> K[Deploy Field Support]
```

Step 1: Analyze the Field

Before implementing, understand your field's characteristics:

bash

Analyze field data distribution

```
python analyze_column/analyze_column.py data/sample.csv new_field_r
```

Or use the main entry point

```
python main.py analyze-column data/sample.csv new_field_name
```

Output includes:

- Unique values count

- Top values and frequencies

- Pattern analysis

- Sample values

Step 2: Create a Validator

Validators provide high-confidence error detection through business rules.

2.1 Create Directory Structure

```
 bash
mkdir -p validators/new_field
cd validators/new_field
```

2.2 Implement Validator Class

Create `validators/new_field/validate.py`:

```
python
from validators.validator_interface import ValidatorInterface
from validators.validation_error import ValidationError
import re


class Validator(ValidatorInterface):
    def __init__(self):
        self.field_type = "new_field"
        # Define patterns and rules
        self.valid_pattern = re.compile(r'^[A-Z]{2}\d{4}$')
        self.min_length = 6
        self.max_length = 50

    def _validate_entry(self, value):
        """Validate a single value."""
        # Convert to string for validation
        str_value = str(value).strip()

        # Check for empty values
        if not str_value or str_value.lower() in ['nan', 'none', 'r
            return ValidationError(
                error_type="EMPTY_VALUE",
                probability=1.0
            )

        # Check length constraints
        if len(str_value) < self.min_length:
            return ValidationError(
                error_type="TOO_SHORT",
                probability=1.0
            )

        if len(str_value) > self.max_length:
            return ValidationError(
                error_type="TOO_LONG",
                probability=1.0
            )

        # Check format pattern
        if not self.valid_pattern.match(str_value):
            return ValidationError(
                error_type="INVALID_FORMAT",
                probability=1.0
            )

        # Add custom business logic
        if self._violates_business_rule(str_value):
            return ValidationError(
```

```
                error_type="BUSINESS_RULE_VIOLATION",
                probability=1.0
            )

        return None

    def _violates_business_rule(self, value):
        """Implement custom business logic."""
        # Example: Check against forbidden values
        forbidden = ['XX0000', 'TEST01']
        return value in forbidden
```

2.3 Create Error Messages

Create `validators/new_field/error_messages.json`:

```
json
{
    "EMPTY_VALUE": {
        "message": "Value cannot be empty",
        "description": "This field is required and must contain a v
        "severity": "ERROR",
        "examples": ["", " ", "null", "NaN"]
    },
    "TOO_SHORT": {
        "message": "Value is too short",
        "description": "Value must be at least {min_length} charact
        "severity": "ERROR",
        "examples": ["AB1", "X"]
    },
    "TOO_LONG": {
        "message": "Value exceeds maximum length",
        "description": "Value must not exceed {max_length} characte
        "severity": "WARNING",
        "examples": ["Very long string that exceeds the maximum all
    },
    "INVALID_FORMAT": {
        "message": "Invalid format",
        "description": "Value must match pattern: 2 uppercase lette
        "severity": "ERROR",
        "examples": ["abc123", "12ABCD", "AB12345"]
    },
    "BUSINESS_RULE_VIOLATION": {
        "message": "Business rule violation",
        "description": "Value violates business constraints",
        "severity": "ERROR",
        "examples": ["XX0000", "TEST01"]
    }
}
```

Step 3: Define Pattern-Based Rules

Pattern-based detection identifies anomalies using statistical and rule-based approaches.

3.1 Create Pattern Rules

Create `anomaly_detectors/pattern_based/rules/new_field.json`:

```json
{
    "field_name": "new_field",
    "description": "Pattern rules for new field validation",
    "version": "1.0",

    "known_values": [
        "AB1234", "CD5678", "EF9012",
        "GH3456", "IJ7890", "KL2345"
    ],

    "format_patterns": [
        {
            "name": "standard_format",
            "pattern": "^[A-Z]{2}\\d{4}$",
            "confidence": 0.8,
            "message": "Does not match standard format"
        },
        {
            "name": "legacy_format",
            "pattern": "^\\d{2}[A-Z]{4}$",
            "confidence": 0.7,
            "message": "Matches legacy format (deprecated)"
        }
    ],

    "statistical_rules": {
        "length": {
            "min": 6,
            "max": 10,
            "typical": 6
        },
        "character_distribution": {
            "letters": 0.33,
            "digits": 0.67,
            "special": 0.0
        }
    },

    "validation_rules": [
        {
            "name": "not_empty",
            "type": "not_empty",
            "message": "Value cannot be empty"
        },
        {
            "name": "no_special_chars",
            "type": "regex",
            "pattern": "^[A-Za-z0-9]+$",
            "message": "Contains special characters"
```

```
        }
    ],

    "anomaly_patterns": [
        {
            "name": "suspicious_pattern",
            "pattern": "(00000|11111|99999)",
            "confidence": 0.9,
            "message": "Contains suspicious repeated digits"
        },
        {
            "name": "test_data",
            "pattern": "(TEST|DEMO|SAMPLE)",
            "confidence": 0.95,
            "message": "Appears to be test data"
        }
    ]
}
```

Step 4: Train ML Model (Optional)

For semantic understanding, train an ML model.

4.1 Prepare Training Data

Create a clean dataset with valid examples:

```python
import pandas as pd
```

Load and filter clean data

```
data = pd.read_csv('data/full_dataset.csv')
clean_data = data[data['quality_flag'] == 'clean']
```

Extract field values

```
field_values = clean_data['new_field'].dropna().unique()
```

Save training data

```
pd.DataFrame({'new_field': field_values}).to_csv(
    'data/new_field_training.csv',
    index=False
)
```

4.2 Model Training Configuration

ML models for new fields are trained using the ml-train command. The system will automatically use appropriate settings based on the field type and available data.

4.3 Train the Model

ML models are typically pre-trained or use transfer learning. To configure and test ML detection for your new field:

bash

Run hyperparameter search for the

```
python main.py ml-train \
    --use-hp-search \
    --fields new_field \
    --hp-trials 15
```

Test anomaly detection

```
python main.py ml-train \
    --check-anomalies new_field \
    --threshold 0.75
```

Step 5: Configure Field Mapping

Update brand configuration to include the new field.

5.1 Update Brand Config

Edit `brand_configs/your_brand.json`:

```json
{
    "field_mappings": {
        // ... existing mappings ...
        "new_field": "Your_Column_Name"
    },

    "enabled_fields": [
        // ... existing fields ...
        "new_field"
    ]
}
```

5.2 Register the Field

The new field will be automatically recognized once it's added to the brand configuration and the corresponding validator is created in the `validators/new_field/` directory.

Step 6: Test Implementation

6.1 Manual Testing

Create a test script to verify your validator:

python

test_new_field.py

```
from validators.new_field.validate import Validator
```

Create validator instance

```
validator = Validator()
```

Test valid values

```
valid_values = ['AB1234', 'CD5678', 'EF9012']
print("Testing valid values:")
for value in valid_values:
    error = validator._validate_entry(value)
    print(f"  {value}: {'PASS' if error is None else 'FAIL'}")
```

Test invalid values

```
invalid_values = ['', 'abc123', '123ABC', 'ABCDEF', 'XX0000']
print("\nTesting invalid values:")
for value in invalid_values:
    error = validator._validate_entry(value)
    if error:
        print(f"  {value}: {error.error_type}")
    else:
        print(f"  {value}: Unexpected PASS")
```

Note: The project doesn't currently have a formal test framework. Consider implementing pytest or unittest for automated testing.

6.2 Integration Tests

Test with the complete system:

bash

Test with sample data

```
python main.py single-demo \
    --data-file test_data/new_field_test.csv \
    --enable-validation \
    --enable-pattern \
    --enable-ml \
    --output-dir test_results/new_field
```

Review results in test_results/new

6.3 Performance Testing

bash

Test with larger dataset

```
python main.py multi-eval \
    --input data/full_dataset.csv \
    --field new_field \
    --num-samples 100 \
    --output-dir evaluation_results/new_field
```

Step 7: Documentation

7.1 Update Field Documentation

Create `docs/fields/new_field.md`:

markdown

New Field

Description

Brief description of what this field represents.

Format

- Pattern: ^[A-Z]{2}\d{4}$
- Length: 6 characters
- Example: AB1234

Validation Rules

1. Cannot be empty
2. Must match format pattern
3. Cannot contain special characters
4. Business rule constraints

Common Issues

- Invalid format: Use 2 uppercase letters + 4 digits
- Test data: Remove TEST, DEMO, SAMPLE values
- Legacy format: Update from old format 12ABCD

7.2 Update API Documentation

Add field to API examples and configuration guides.

Best Practices

1. Start Simple

Begin with basic validation rules and gradually add complexity:

python

Phase 1: Basic validation (empty,

Phase 2: Business rules

Phase 3: Pattern-based rules and

Phase 4: ML-based detection

2. Use Existing Patterns

Look for similar fields to reuse patterns:

python

If similar to existing field

from validators.similar_field.validate import Validator as BaseVali

```
class Validator(BaseValidator):
    def __init__(self):
        super().__init__()
        self.field_type = "new_field"
        # Override specific attributes
```

3. Collect Real Data

Use actual data for pattern discovery:

```
bash
```

Analyze real data patterns using t

```
python main.py analyze-column data/your_data.csv new_field
```

4. Progressive Thresholds

Start with conservative thresholds:

```json
{
    "thresholds": {
        "validation": 0.0,    // 100% confidence
        "pattern": 0.8,       // Start high
        "ml": 0.8,           // Start high
        "llm": 0.7           // Adjust based on results
    }
}
```

5. Monitor and Iterate

Track field performance by reviewing the detection results and adjusting thresholds based on false positive/negative rates. Use the evaluation reports generated by multi-eval to understand performance metrics.

Troubleshooting

Common Issues

1. **Import Errors**

```python
  # Ensure __init__.py exists
  touch validators/new_field/__init__.py
```

2. **Pattern Not Matching**

```
python
  # Test patterns independently
  import re
  pattern = re.compile(r'^[A-Z]{2}\d{4}$')
  print(pattern.match('AB1234'))  # Should return match object
```

3. **ML Model Not Loading**

```
bash
  # Check model path
  ls -la models/new_field/
  # Verify model files exist
```

4. **Field Not Detected**

```
bash
  # Check field mapping
  python -c "from common.brand_config import BrandConfig; \
            config = BrandConfig('your_brand'); \
            print(config.get_field_mapping('new_field'))"
```

Checklist

Before deploying a new field:

- [ ] Validator implemented and tested
- [ ] Error messages defined
- [ ] Pattern rules created
- [ ] ML model trained (if applicable)
- [ ] Field mapping configured
- [ ] Unit tests passing
- [ ] Integration tests passing
- [ ] Documentation updated
- [ ] Performance acceptable
- [ ] Code reviewed

---

development > contributing

Contributing Guide

Thank you for your interest in contributing to the Data Quality Detection System! This guide will help you get started with development.

Development Setup

Prerequisites

- Python 3.8 or higher
- Git
- Virtual environment tool (venv, conda, virtualenv)
- GPU (optional, for ML/LLM development)

Setting Up Your Development Environment

1. **Fork and Clone the Repository**

```
bash
   git clone https://github.com/your-username/data-quality-detectic
   cd data-quality-detection
```

2. **Create a Virtual Environment**

```
bash
   python -m venv venv
   source venv/bin/activate  # On Windows: venv\Scripts\activate
```

3. **Install Development Dependencies**

```
bash
   pip install -r requirements.txt
   pip install -r requirements-dev.txt
```

4. **Install Pre-commit Hooks**

```
bash
   pre-commit install
```

This installs hooks that automatically check your code before each commit.

Code Quality Standards

Pre-commit Hooks

The project uses pre-commit hooks to maintain code quality. These run automatically before each commit and check for:

- **Import sorting** (isort)
- **Code formatting** (black)
- **Linting** (flake8 with extensions)
- **Type hints** (mypy)
- **Security issues** (bandit)
- **YAML/JSON syntax**
- **Trailing whitespace**
- **File endings**

To run the checks manually:

```
bash
pre-commit run --all-files
```

Code Style

- Follow PEP 8 guidelines
- Use type hints where appropriate
- Write docstrings for all public functions and classes
- Keep functions focused and under 50 lines
- Use meaningful variable and function names

Testing

While the project doesn't currently have a formal test framework, please:

1. **Test your changes manually**:

```
bash
  # Test with sample data
  python main.py single-demo --data-file data/sample.csv
```

2. **Run evaluation mode** to ensure detection accuracy:

```
bash
  python main.py multi-eval data/sample.csv --field your_field
```

3. **Verify no regressions** in existing functionality

Making Contributions

Types of Contributions

- **Bug Fixes**: Fix issues in existing code
- **New Features**: Add new detection methods or fields
- **Documentation**: Improve or add documentation
- **Performance**: Optimize existing code
- **Refactoring**: Improve code structure

Contribution Process

1. **Create an Issue** (optional but recommended)
- Describe what you plan to work on
- Get feedback before starting major work

2. **Create a Feature Branch**

```
bash
  git checkout -b feature/your-feature-name
```

3. **Make Your Changes**
- Write clean, documented code
- Follow the existing code structure
- Update relevant documentation

4. **Run Quality Checks**

```
bash
  pre-commit run --all-files
```

5. **Test Your Changes**

```
bash
  # Run detection on sample data
  python main.py single-demo --data-file your_test_data.csv

  # Run evaluation if applicable
  python main.py multi-eval your_test_data.csv --field affected_fi
```

6. **Commit Your Changes**

```
bash
  git add .
  git commit -m "feat: add support for new field type"
```

Follow conventional commit format:
- `feat:` for new features
- `fix:` for bug fixes
- `docs:` for documentation

- `perf:` for performance improvements
- `refactor:` for code refactoring

7. **Push and Create Pull Request**

```
bash
  git push origin feature/your-feature-name
```

Pull Request Guidelines

- **Title**: Clear, descriptive title
- **Description**: Explain what changes you made and why
- **Testing**: Describe how you tested the changes
- **Documentation**: Note any documentation updates
- **Breaking Changes**: Clearly mark if applicable

Development Guidelines

Adding New Fields

See the Adding Fields Guide for detailed instructions.

Adding Detection Methods

1. Implement the appropriate interface:
- `ValidatorInterface` for rule-based validation
- `AnomalyDetectorInterface` for anomaly detection

2. Add configuration support
3. Update documentation
4. Test thoroughly

Project Structure

Follow the existing structure:

```
project_root/
├── validators/           # Rule-based validators
├── anomaly_detectors/   # Detection method implementations
├── common/              # Shared utilities
├── brand_configs/       # Brand configuration files
└── docs/               # Documentation
```

Common Development Tasks

Running with Debug Mode

```bash
python main.py single-demo --data-file data.csv --debug
```

Analyzing Performance

```bash
python main.py ml-curves data.csv --fields material
```

Training Models

```bash
python main.py ml-train training_data.csv --fields "new_field"
```

Getting Help

- Check existing documentation
- Look at similar implementations in the codebase
- Open an issue for questions
- Reach out to maintainers

Code of Conduct

- Be respectful and inclusive
- Welcome newcomers
- Focus on constructive feedback
- Assume good intentions

Thank you for contributing!

---

deployment > examples

Deployment Examples

This document provides example configurations for deploying the Data Quality Detection System in various environments.

> ⚠️ **IMPORTANT NOTE**: The examples in this document are provided as reference implementations. The current system is a command-line batch processing tool without built-in support for Docker, Kubernetes, or API endpoints. These examples show how you might deploy the system in production environments with custom wrapper scripts and configurations.

Current Deployment Method

The system is designed to run as a batch process:

bash

Basic execution

```
python main.py single-demo --data-file data.csv
```

Scheduled execution with cron

```
0 2   * cd /path/to/project && /path/to/venv/bin/python main.py sir
```

Example Configurations

Example 1: Simple Batch Processing Script

File: `run_detection.sh` (EXAMPLE)

```
bash
#!/bin/bash
```

Example wrapper script for produc

Configuration

```
PROJECT_DIR="/opt/data-quality-detection"
VENV_PATH="$PROJECT_DIR/venv"
DATA_DIR="/data/incoming"
OUTPUT_DIR="/data/results"
LOG_DIR="/var/log/detection"
```

Activate virtual environment

```
source $VENV_PATH/bin/activate
```

Create output directory with times

```
TIMESTAMP=$(date +%Y%m%d_%H%M%S)
RESULT_DIR="$OUTPUT_DIR/$TIMESTAMP"
mkdir -p $RESULT_DIR
```

Run detection

```
cd $PROJECT_DIR
python main.py single-demo \
    --data-file $DATA_DIR/latest.csv \
    --output-dir $RESULT_DIR \
    --enable-validation \
    --enable-pattern \
    --enable-ml \
    2>&1 | tee $LOG_DIR/detection_$TIMESTAMP.log
```

Check exit status

```
if [ $? -eq 0 ]; then
```

```
    echo "Detection completed successfully"
    # Optional: trigger downstream processes
else
    echo "Detection failed"
    # Optional: send alert
fi
```

Example 2: Docker Configuration (NOT IMPLEMENTED)

File: **`Dockerfile`** (EXAMPLE - would need to be created)

```
dockerfile
FROM python:3.8-slim
```

Install system dependencies

```
RUN apt-get update && apt-get install -y \
    build-essential \
    && rm -rf /var/lib/apt/lists/*
```

Set working directory

```
WORKDIR /app
```

Copy requirements

```
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
```

Copy application

```
COPY . .
```

Create directories

```
RUN mkdir -p /app/results /app/logs /app/data
```

Entry point

```
ENTRYPOINT ["python", "main.py"]
```

File: `docker-compose.yml` (EXAMPLE)

```
yaml
version: '3.8'


services:
  detection:
    build: .
    volumes:
      - ./data:/app/data
      - ./results:/app/results
      - ./brand_configs:/app/brand_configs
    environment:
      - CUDA_VISIBLE_DEVICES=0
    command: single-demo --data-file /app/data/input.csv
```

Example 3: Kubernetes CronJob (NOT IMPLEMENTED)

File: **`detection-cronjob.yaml`** (EXAMPLE)

```
yaml
apiVersion: batch/v1
kind: CronJob
metadata:
  name: data-quality-detection
spec:
  schedule: "0 2   *"  # Daily at 2 AM
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: detection
            image: your-registry/data-quality-detection:latest
            command:
              - python
              - main.py
              - single-demo
              - --data-file
              - /data/daily.csv
              - --output-dir
              - /results
            volumeMounts:
            - name: data
              mountPath: /data
            - name: results
              mountPath: /results
            - name: config
              mountPath: /app/brand_configs
            resources:
              requests:
                memory: "4Gi"
                cpu: "2"
              limits:
                memory: "8Gi"
                cpu: "4"
          volumes:
          - name: data
            persistentVolumeClaim:
              claimName: detection-data-pvc
          - name: results
            persistentVolumeClaim:
              claimName: detection-results-pvc
          - name: config
            configMap:
              name: brand-configs
          restartPolicy: OnFailure
```

Example 4: Airflow DAG (INTEGRATION EXAMPLE)

File: `detection_dag.py` (EXAMPLE)

```
python
from datetime import datetime, timedelta
from airflow import DAG
from airflow.operators.bash import BashOperator
from airflow.operators.python import PythonOperator

default_args = {
    'owner': 'data-team',
    'depends_on_past': False,
    'start_date': datetime(2024, 1, 1),
    'email_on_failure': True,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
}

dag = DAG(
    'data_quality_detection',
    default_args=default_args,
    description='Daily data quality detection',
    schedule_interval='0 2   *',
    catchup=False,
)
```

Task 1: Prepare data

```
prepare_data = BashOperator(
    task_id='prepare_data',
    bash_command='''
    # Export data from database to CSV
    psql -h $DB_HOST -U $DB_USER -d $DB_NAME \
        -c "COPY (SELECT * FROM products) TO '/data/daily_export.cs
    ''',
    dag=dag,
)
```

Task 2: Run detection

```
run_detection = BashOperator(
    task_id='run_detection',
    bash_command='''
    cd /opt/data-quality-detection
    source venv/bin/activate
    python main.py single-demo \
        --data-file /data/daily_export.csv \
        --output-dir /results/{{ ds }} \
```

```
        --enable-validation \
        --enable-pattern \
        --enable-ml
    ''',
    dag=dag,
)
```

Task 3: Process results

```
def process_results(context):
    import json
    import pandas as pd

    date = context['ds']
    with open(f'/results/{date}/report.json', 'r') as f:
        report = json.load(f)

    # Extract metrics
    total_anomalies = report['summary']['total_anomalies']

    # Alert if threshold exceeded
    if total_anomalies > 100:
        context['task_instance'].xcom_push(key='alert', value=True)

process_results_task = PythonOperator(
    task_id='process_results',
    python_callable=process_results,
    dag=dag,
)
```

Define task dependencies

```
prepare_data >> run_detection >> process_results_task
```

Example 5: Systemd Service (LINUX DEPLOYMENT)

File: `/etc/systemd/system/detection-monitor.service` (EXAMPLE)

```ini
[Unit]
Description=Data Quality Detection Monitor
After=network.target

[Service]
Type=simple
User=detection
Group=detection
WorkingDirectory=/opt/data-quality-detection
Environment="PATH=/opt/data-quality-detection/venv/bin"
ExecStart=/opt/data-quality-detection/venv/bin/python /opt/data-qua
Restart=on-failure
RestartSec=10


[Install]
WantedBy=multi-user.target
```

File: **`monitor.py`** (EXAMPLE - would need to be created)

```python
#!/usr/bin/env python
"""Example monitoring script that watches for new files and runs de

import os
import time
import subprocess
from pathlib import Path
from datetime import datetime

WATCH_DIR = Path("/data/incoming")
OUTPUT_DIR = Path("/data/results")
PROCESSED_DIR = Path("/data/processed")

def process_file(filepath):
    """Run detection on a single file."""
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    output_path = OUTPUT_DIR / timestamp
    output_path.mkdir(parents=True, exist_ok=True)

    cmd = [
        "python", "main.py", "single-demo",
        "--data-file", str(filepath),
        "--output-dir", str(output_path)
    ]

    result = subprocess.run(cmd, capture_output=True, text=True)

    if result.returncode == 0:
        # Move processed file
        processed_path = PROCESSED_DIR / f"{filepath.stem}_{timesta
        filepath.rename(processed_path)
        print(f"Successfully processed {filepath}")
    else:
        print(f"Error processing {filepath}: {result.stderr}")

def monitor():
    """Monitor directory for new CSV files."""
    processed_files = set()

    while True:
        for filepath in WATCH_DIR.glob("*.csv"):
            if filepath not in processed_files:
                print(f"New file detected: {filepath}")
                process_file(filepath)
                processed_files.add(filepath)

        time.sleep(60)  # Check every minute
```

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

Production Deployment Checklist

When deploying to production, consider:

1. **Environment Setup**
- [ ] Python 3.8+ installed
- [ ] Virtual environment created
- [ ] All dependencies installed
- [ ] GPU drivers (if using ML/LLM)

2. **Configuration**
- [ ] Brand configurations created
- [ ] Thresholds tuned for your data
- [ ] Output directories configured
- [ ] Logging configured

3. **Data Pipeline Integration**
- [ ] Input data format verified
- [ ] Output location accessible
- [ ] Error handling in place
- [ ] Monitoring/alerting configured

4. **Performance**
- [ ] Batch size optimized
- [ ] Memory limits set
- [ ] GPU allocation configured
- [ ] Parallel processing tuned

5. **Security**
- [ ] File permissions set correctly
- [ ] Sensitive data handling reviewed
- [ ] Network access restricted
- [ ] Audit logging enabled

Notes on Examples

- These examples show common deployment patterns
- Adapt them to your specific infrastructure
- Test thoroughly in staging before production
- Monitor resource usage and adjust as needed
- Consider data volume and processing frequency

Remember: The core system is a Python application that processes CSV files. All deployment examples are wrappers around this basic functionality.

End of Xafron Documentation

For the latest version and interactive features, visit: https://docs.xafron.nl