## **Begründung zur strukturierten Datenbankarchitektur für ein hybrides Dashboard im Gemüseverarbeitungsbetrieb**

## **Zielsetzung des Systems**

Im Rahmen eines intelligenten Datenverarbeitungssystems für einen Gemüseverarbeitungs- und Verpackungsbetrieb bestehen folgende Ziele:

1. **Echtzeitüberwachung** von Sensorwerten (z. B. Waagen an Verpackungslinien).
2. **Vorverarbeitung** und Bereinigung der Rohdaten für statistische Analyse.
3. **Berechnung und Speicherung von Kennzahlen (KPIs)** wie Befüllzeit, Abweichung, Leistung pro Schicht, Genauigkeit, etc.
4. **Vergleichbarkeit zwischen Schichten, Mitarbeitern, Produktionslinien**.
5. **Langfristige Analyse** und **Vorhersagefunktionen** (z. B. Saisonalität, Produktionsplanung).
6. **Visualisierung in einem hybriden Dashboard** (z. B. Grafana), das Daten aus mehreren Quellen integriert: Echtzeit, aggregierte Daten, manuelle Eingaben.

## **Warum eine strukturierte Datenbank erforderlich ist**

### **Vorteile strukturierter Datenbanken:**

| **Aspekt** | **Grund** |
| --- | --- |
| **Datenintegrität & Konsistenz** | Eindeutige Typen, Relationen, Validierung von Eingaben |
| **Automatisierbarkeit** | Leichte Nutzung mit ETL-Pipelines, Airflow, Pandas, DBT etc. |
| **SQL-basierte Analyse** | Aggregationen, Zeitreihenvergleiche, Joins, Indikatorberechnungen |
| **Visualisierungskompatibilität** | Direkte Integration mit BI-Tools wie Grafana, Metabase, Power BI |
| **Langfristige Skalierbarkeit** | Erweitern um neue Sensoren, Schichten, Produktionslinien |
| **Vorhersage-Vorbereitung** | Saubere Datenbasis für Machine-Learning-Modelle und Prognosen |

## 

## **Technologien & Ressourcen für die Umsetzung**

| **Komponente** | **Empfohlene Lösung / Technologie** |
| --- | --- |
| **Relationale Datenbank** | PostgreSQL oder ClickHouse (hohe Leistung für KPIs) |
| **ETL-Verarbeitung** | Python (Pandas), Apache Airflow, DBT |
| **Datenvisualisierung** | Grafana (hybrides Dashboard, viele Datenquellen) |
| **Sensorintegration** | Telegraf, MQTT, CSV-Importer, API-Konnektoren |
| **Containerisierung (lokal)** | Docker, Docker Compose |
| **Skalierbarkeit (optional)** | Kubernetes (z. B. k3s für On-Prem-Umgebungen) |

## **Hybrides Dashboard: warum notwendig?**

Ein hybrides Dashboard integriert verschiedene Datenquellen:

* Live-Daten von Sensoren (aus InfluxDB oder direkt aus sensor\_messungen)
* Aggregierte KPIs aus relationaler Datenbank (kpi\_berichte, container\_befuellung, etc.)
* Manuelle Eingaben von Bedienern (aus manuelle\_eingaben)
* Prognose-Outputs aus ML- oder Statistik-Skripten

Nur mit einer **strukturierten, relationalen Datenbank** lässt sich dieses Modell wartbar, erweiterbar und effizient automatisieren.

## **Fazit**

Für eine belastbare und skalierbare Architektur, die sowohl **Echtzeitüberwachung** als auch **historische Analyse, KPI-Auswertung und Prognosefunktionen** bietet, ist eine **strukturierte relationale Datenbank unverzichtbar**. Sie stellt das **Rückgrat** der Datenverarbeitung und Visualisierung in einem professionellen Produktionsumfeld dar.

MongoDB als unstrukturierte Alternative bietet Flexibilität und vereinfacht das schnelle Ingesten „angereicherter“ Daten, führt jedoch zu Daten­duplikation, erschwert Integritäts­garantien und Reporting. In einem industriellen Gemüseverarbeitungsbetrieb, in dem KPI-Vergleiche zwischen Schichten, strenge Validierung und eine enge BI-Integration entscheidend sind, bleibt eine relationale Datenbank die vorhersehbarere und kosteneffizientere Wahl.

Bei instabilen Sensor-Datenformaten und hohen Anforderungen an horizontale Skalierung verdient MongoDB jedoch als Ergänzung oder für die Speicherung der „rohen“ Daten Beachtung.

## **Struktur der Datenbank für dein Projekt**

<https://dbdiagram.io/d/DB_DC_v_1-67b718e5263d6cf9a0db8388>

### 

### **Hinweis zur Datenverarbeitung und Interpretation der Tabellenstruktur**

Die Datenbankstruktur, insbesondere die Tabelle sensor\_messungen, ist so aufgebaut, dass sie alle notwendigen Informationen für spätere KPI-Berechnungen und Berichte abbilden kann. Die Tabelle enthält bereits vorbereitete Felder wie *schicht\_id*, *mitarbeiter\_id,* *container\_id* und *befuellung\_id*, die zur Verknüpfung mit anderen Tabellen dienen.

Allerdings ist bei der praktischen Datenverarbeitung – insbesondere beim Import von Rohdaten (z. B. aus CSV-Dateien von Waagen oder Messsystemen) – zu beachten, dass diese Felder im Ursprung der Daten nicht vorhanden sind.

Daher muss im Rahmen der Datenintegration und ETL-Verarbeitung Folgendes berücksichtigt werden:

1. Zeitliche Einordnung (timestamp-basierte Zuordnung):  
     
    Die Spalte zeitstempel muss verwendet werden, um den zugehörigen Schichtzeitraum in der Tabelle schichten zu finden. Daraus ergibt sich die schicht\_id.
2. Zuordnung des Mitarbeiters:  
     
    Sobald eine Schicht identifiziert ist, kann der verantwortliche Mitarbeiter (mitarbeiter\_id) aus dieser Schicht automatisch übernommen werden.
3. Verknüpfung mit Befüllzyklen (Container):  
     
    Jede Messung muss anhand des Zeitpunkts einem Befüllzyklus (container\_befuellung) zugeordnet werden. Daraus resultieren befuellung\_id und container\_id.
4. Erforderliche Datenanreicherung:  
     
    Dieses Vorgehen nennt man „Anreicherung der Rohdaten“ – also das Hinzufügen von Kontextinformationen, bevor die Daten in die Datenbank geladen werden. Nur so ist sichergestellt, dass alle Tabellen richtig verbunden sind und KPI-Berechnungen technisch möglich und korrekt sind.

### **Haupttabellen:**

### **1. sensor\_messungen – Rohdaten von Waagen**

CREATE TABLE sensor\_messungen (

messung\_id SERIAL PRIMARY KEY,

sensor\_id VARCHAR(50),

unix\_timestamp BIGINT,

zeitstempel TIMESTAMP,

gewicht\_gramm NUMERIC(10, 2),

gewicht\_diff NUMERIC(10, 2),

ist\_neues\_objekt BOOLEAN,

standort VARCHAR(100),

schicht\_id INT REFERENCES schichten(schicht\_id),

container\_id VARCHAR(50),

mitarbeiter\_id INT REFERENCES mitarbeiter(mitarbeiter\_id),

befuellung\_id INT REFERENCES container\_befuellung(befuellung\_id)

);

**Berechnete KPIs aus dieser Tabelle**:

1. Befüllzeit eines Containers *(zeitstempel)*
2. Anzahl Einheiten pro Container *(ist\_neues\_objekt)*
3. Durchschnittliches Stückgewicht *(gewicht\_gramm, anzahl\_objekte)*
4. Zeitabstand zwischen den Einheiten *(zeitstempel)*
5. Minimales Stückgewicht *(gewicht\_gramm)*
6. Maximales Stückgewicht *(gewicht\_gramm)*
7. Gesamtanzahl der Einheiten pro Schicht *(ist\_neues\_objekt)*
8. Produktivität pro Einheit *(ist\_neues\_objekt, zeitstempel)*
9. Produktivität pro Gewicht *(gewicht\_gramm, zeitstempel)*
10. Gewichtsstreuung *(gewicht\_gramm)*
11. Produktionsverlust in kg *(gewicht\_gramm)*
12. Produktionsverlust in % *(gewicht\_gramm)*
13. Verlust pro Palette *(gewicht\_gramm)*
14. Effizienz pro Mitarbeiter *(gewicht\_gramm, mitarbeiter\_id)*
15. Ø Bearbeitungszeit pro Artikel *(zeitstempel, gewicht\_gramm)*
16. Qualitätsrate *(gewicht\_gramm, zielgewicht\_gramm aus system\_parameter)*

### **2. container\_befuellung – Einzelne Befüllzyklen**

CREATE TABLE container\_befuellung (

befuellung\_id SERIAL PRIMARY KEY,

start\_zeit TIMESTAMP,

ende\_zeit TIMESTAMP,

zielgewicht\_gramm NUMERIC(10,2),

endgewicht\_gramm NUMERIC(10,2),

abweichung\_gramm NUMERIC(10,2),

container\_id VARCHAR(50),

anzahl\_objekte INT,

ist\_abgeschlossen BOOLEAN,

schicht\_id INT REFERENCES schichten(schicht\_id)

);

**Berechnete KPIs aus dieser Tabelle**:

1. Befüllzeit eines Containers *(start\_zeit, ende\_zeit)*
2. Abweichung vom Zielgewicht *(zielgewicht\_gramm, endgewicht\_gramm)*
3. Anzahl befüllter Container pro Schicht *(befuellung\_id, ist\_abgeschlossen)*
4. Anzahl Einheiten pro Container *(anzahl\_objekte)*
5. Durchschnittliches Stückgewicht *(endgewicht\_gramm, anzahl\_objekte)*
6. Zeit zwischen Containerbefüllungen *(ende\_zeit)*
7. Durchschnittlicher Leerlauf *(start\_zeit, ende\_zeit)*
8. Gesamtanzahl der Container pro Schicht *(befuellung\_id)*
9. Gesamtgewicht der produzierten Ware *(endgewicht\_gramm)*
10. Produktivität pro Gewicht *(endgewicht\_gramm, start\_zeit, ende\_zeit)*
11. IST pro Palette *(endgewicht\_gramm)*
12. Kosten pro Produktart *(endgewicht\_gramm, parameter\_context aus system\_parameter)*

### **3. schichten – Schichtinformationen**

CREATE TABLE schichten (

schicht\_id SERIAL PRIMARY KEY,

mitarbeiter\_id INT REFERENCES mitarbeiter(mitarbeiter\_id),

beginn TIMESTAMP,

ende TIMESTAMP,

ist\_nachtschicht BOOLEAN

);

**Berechnete KPIs aus dieser Tabelle**: 3. Anzahl befüllter Container *(schicht\_id)* 12. Anzahl Einheiten *(schicht\_id)* 13. Anzahl Container *(schicht\_id)* 14. Gesamtgewicht *(schicht\_id)* 15. Produktivität pro Einheit *(beginn, ende, schicht\_id)* 16. Produktivität pro Gewicht *(beginn, ende, schicht\_id)* 29. Stunden pro Schicht *(beginn, ende)* 37. Leistungsvergleich Früh / Spät *(ist\_nachtschicht)*

### **4. mitarbeiter – Mitarbeiterdaten**

CREATE TABLE mitarbeiter (

mitarbeiter\_id SERIAL PRIMARY KEY,

name VARCHAR(100),

rolle VARCHAR(50),

aktiv BOOLEAN,

eingestellt\_am DATE

);

**Berechnete KPIs aus dieser Tabelle**: 30. Anzahl Mitarbeiter pro Schicht *(mitarbeiter\_id in schicht\_parameter)* 31. Personalkosten pro Schicht *(mitarbeiter\_id, stundenlohn aus system\_parameter)* 36. Effizienz pro Mitarbeiter *(mitarbeiter\_id)* 37. Leistungsvergleich Früh / Spät *(mitarbeiter\_id in schichten)*

### **5. manuelle\_eingaben – Benutzereingaben**

CREATE TABLE manuelle\_eingaben (

eingabe\_id SERIAL PRIMARY KEY,

zeitstempel TIMESTAMP,

mitarbeiter\_id INT REFERENCES mitarbeiter(mitarbeiter\_id),

typ VARCHAR(50),

kommentar TEXT

);

**Berechnete KPIs aus dieser Tabelle**: 19. Sortierqualität *(typ, kommentar)* 39. Qualitätsindikation pro Charge *(kommentar)*

### **6. kpi\_berichte – Berechnete KPIs für Auswertung**

CREATE TABLE kpi\_berichte (

bericht\_id SERIAL PRIMARY KEY,

zeitstempel TIMESTAMP,

schicht\_id INT REFERENCES schichten(schicht\_id),

kpi\_name VARCHAR(100),

einheit VARCHAR(20),

wert NUMERIC(12, 2)

);

**Nutzung dieser Tabelle**: Diese Tabelle speichert ausschließlich die Ergebnisse der KPI-Berechnungen.

**Berechnungsfrequenz**:

* Automatisch nach jeder Schicht (Trigger auf Schichtende)
* Regelmäßig per Zeitplan (z. B. alle 60 Minuten via Cronjob)
* Echtzeit-Trigger bei bestimmten Ereignissen (z. B. Alert)

**Umsetzungsvorschläge**:

* ETL-Skripte in Python (z. B. Airflow DAGs)
* Speicherung per INSERT INTO kpi\_berichte ...
* Nutzung in Reporting-Tools (z. B. Grafana)

### **7. alerts\_log – Warn- und Fehlerprotokolle**

CREATE TABLE alerts\_log (

alert\_id SERIAL PRIMARY KEY,

timestamp TIMESTAMP,

sensor\_id VARCHAR(50),

alert\_type VARCHAR(50),

description TEXT,

messung\_id INT REFERENCES sensor\_messungen(messung\_id),

schicht\_id INT REFERENCES schichten(schicht\_id),

mitarbeiter\_id INT REFERENCES mitarbeiter(mitarbeiter\_id)

);

**Berechnete KPIs aus dieser Tabelle**: 40. Verfügbarkeit *(timestamp, alert\_type)*

### **8. system\_parameter – Historische Systemparameter**

CREATE TABLE system\_parameter (

parameter\_id SERIAL PRIMARY KEY,

name VARCHAR(100),

wert NUMERIC(10,2),

einheit VARCHAR(20),

gueltig\_ab DATE,

gueltig\_bis DATE,

erstellt\_am TIMESTAMP,

kommentar TEXT,

parameter\_context VARCHAR(50),

schicht\_id INT REFERENCES schichten(schicht\_id),

mitarbeiter\_id INT REFERENCES mitarbeiter(mitarbeiter\_id)

);

**Berechnete KPIs aus dieser Tabelle**: 2. Abweichung vom Zielgewicht *(zielgewicht\_gramm als Parameter)* 25. Abweichung zum Soll-Verlust *(soll\_verlust als Parameter)* 28. Ø Mitarbeiterkosten pro Stunde *(stundenlohn als Parameter)* 31. Personalkosten pro Schicht *(stundenlohn, mitarbeiter\_id)* 35. Kosten pro Produktart *(kosten/kg oder €/stück)*

### **9. schicht\_parameter – Manuelle Parameter pro Schicht**

CREATE TABLE schicht\_parameter (

eintrag\_id SERIAL PRIMARY KEY,

schicht\_id INT REFERENCES schichten(schicht\_id),

parameter\_name VARCHAR(50),

wert NUMERIC(10,2),

einheit VARCHAR(10),

kommentar TEXT

);

**Berechnete KPIs aus dieser Tabelle**: 30. Anzahl Mitarbeiter pro Schicht *(wert wenn parameter\_name = 'anzahl\_mitarbeiter')* 31. Personalkosten pro Schicht *(wert \* zeit \* stundenlohn)*

## **📊 Abdeckung der wichtigsten KPIs aus dem Notebook:**

| **KPI (aus Notebook)** | **Tabelle(n)** |
| --- | --- |
| Befüllzeit eines Containers | container\_befuellung |
| Abweichung vom Zielgewicht | container\_befuellung |
| Anzahl befüllter Container pro Schicht | container\_befuellung, schichten |
| Anzahl Einheiten pro Container | container\_befuellung |
| Durchschnittliches Gewicht | sensor\_messungen |
| Verteilung der Gewichtsdifferenz | sensor\_messungen |
| Fehler, Kalibrierung, Kommentare | manuelle\_eingaben |
| Mitarbeiterleistung | schichten, kpi\_berichte |

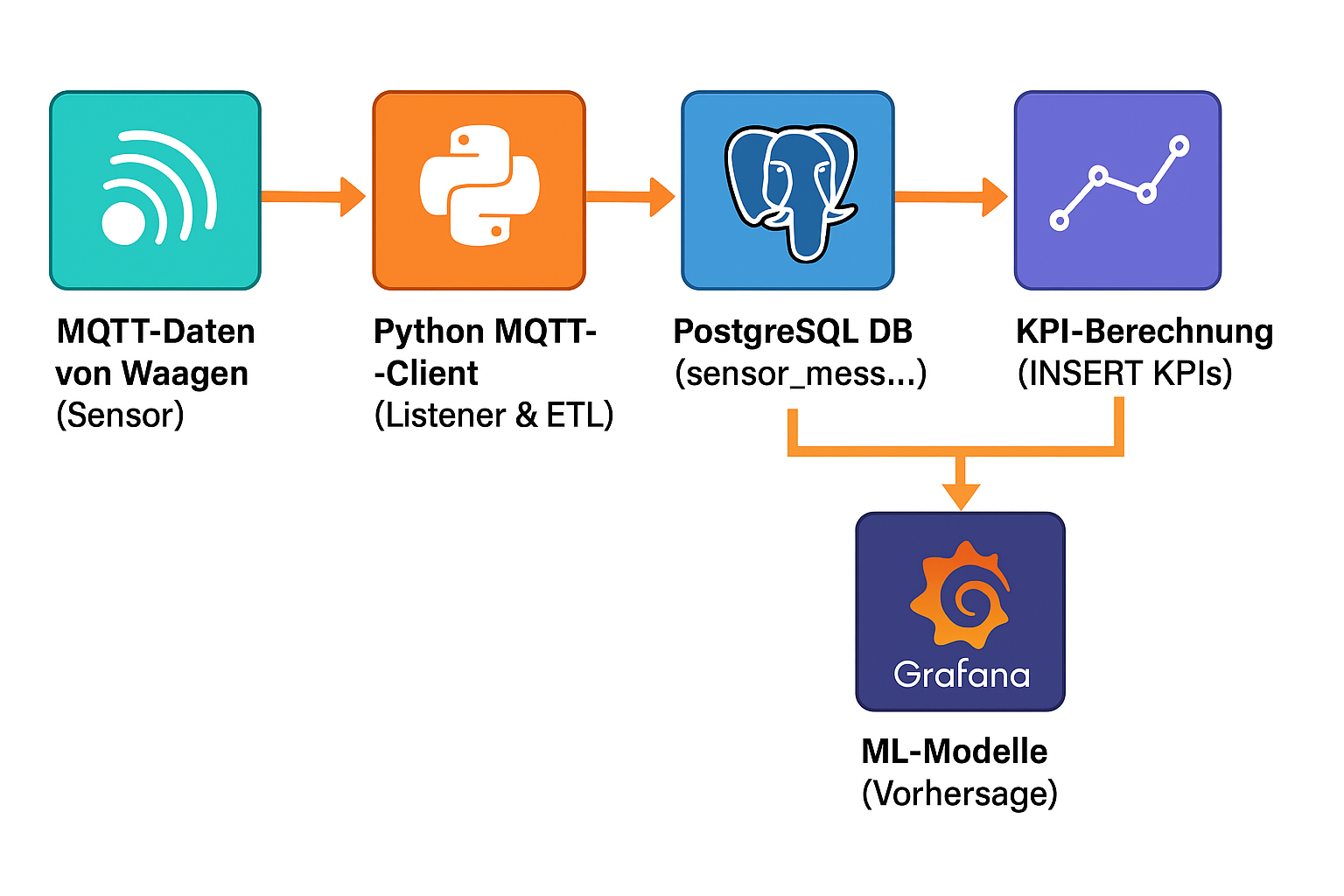
## **Vorteile dieser Struktur:**

* Vollständig relationale Struktur für KPIs
* Einfach in Pandas, SQL oder Airflow zu automatisieren
* Perfekt für Integration in Grafana
* Skalierbar für mehrere Waagen, Containerlinien, Produkte, Schichten

Вот обновлённый текст архитектурного описания на немецком языке, с учётом того, что теперь используется MQTT-Streaming von Sensoren вместо CSV-Dateien:

## **Architektur einer vollständigen Echtzeit-Daten-Pipeline mit MQTT**

Diese überarbeitete Architektur beschreibt den Echtzeit-Datenfluss vom Waagensensor über MQTT bis hin zur Visualisierung in Grafana, inklusive KPI-Berechnung und Machine-Learning-Prognosen. Sie ersetzt das vorherige CSV-basierte Szenario.



### **1. Datenquelle: MQTT-Sensoren in Echtzeit**

### **2. Python MQTT Listener + ETL**

Verantwortlich für:

* Verbindung zum MQTT-Broker
* Empfangen und Parsen der MQTT-Nachrichten
* Automatisches Hinzufügen von Kontextinformationen:  
  + schicht\_id → zeitliche Einordnung
  + mitarbeiter\_id → über Schichtzuweisung
  + befuellung\_id, container\_id → über Zeitlogik
* Schreiben der erweiterten Daten in sensor\_messungen (PostgreSQL)

Verwendete Technologien:

* paho-mqtt, pandas, sqlalchemy, psycopg2

### **3. PostgreSQL-Datenbank**

* Zentrale Speicherung der verarbeiteten Sensordaten
* Alle Tabellen gemäß definierter Struktur:  
  + sensor\_messungen, schichten, container\_befuellung, kpi\_berichte, system\_parameter usw.
* Bereit für Abfragen, JOINs, Aggregationen

### **4. KPI-Berechnung**

* Regelmäßiger oder eventbasierter Python-Job
* Liest aktuelle Daten aus sensor\_messungen
* Berechnet relevante KPIs (z. B. Verlust, Stückzahl, Abweichung)
* Speichert Ergebnis in kpi\_berichte

### **5. Visualisierung mit Grafana**

Datenquelle: PostgreSQL direkt verbunden

Dashboards für:

* aktuelle Leistung (KPI live)
* historische Vergleiche
* Visualisierung von Prognosen und Anomalien

**Interaktive Filterung (z. B. nach Schicht, Linie, Produktart)**

### **6. Machine-Learning-Vorhersagen (Optional)**

Modelle trainiert auf historischen Sensordaten + KPI-Zielen

* Vorhersage von Auslastung, Qualität, Verlusten,
* Erkennung von Abweichungen oder ineffizientem Verhalten,
* Vorhersagen speicherbar in ml\_forecast oder kpi\_berichte

## **Komponentenübersicht**

| **Funktion** | **Technologie** |
| --- | --- |
| Streaming | MQTT |
| Listener & ETL | Python + paho-mqtt |
| Speicherung | PostgreSQL |
| Berechnungen & KPIs | Python (pandas, sqlalchemy) |
| Visualisierung | Grafana (PostgreSQL Data Source) |

## Optinal:

| ML-Modellierung | scikit-learn, XGBoost, ggf. PyTorch |
| --- | --- |

## **Erweiterungen möglich:**

* API oder Webinterface für manuelle Eingaben durch Personal
* Zeitgesteuerte Reports (z. B. Tages-KPIs, E-Mail-Versand)
* Automatisiertes Retraining der Modelle bei neuen Mustern

### **Fazit**

Diese Architektur ist speziell für Echtzeitdaten aus Sensoren konzipiert.

Sie unterstützt:

* Sofortige Datenerfassung ohne Dateimanagement
* Vollautomatische KPI-Berechnung und Visualisierung
* Erweiterung um Prognosefunktionen mit ML
* Industrie-taugliche Integration mit hoher Transparenz