Hochschule Karlsruhe
University of
Applied Sciences

HKA

Karlsruhe Institute of Technology

Project A:
Transforming Measurement Environments: Integrating BET Systems into Digital and ELN-Based Platforms for Automated Data Handling

Author:

Rachit Jain
Masters in Sensor System Technology
Matriculation Number: 94260

Declaration of Authenticity

I hereby declare that the content of this report was drafted with the assistance of AI tools (including ChatGPT), but that all data, interpretations, analyses, and substantive information contained herein were provided and verified by me. AI was used solely to aid in writing, formatting, and improving clarity; I take full responsibility for the accuracy, integrity, and authenticity of the report.

Acknowledgement

I want to start by expressing my gratitude to everyone who has helped make this effort a success. I would like to thank the Karlsruhe Institute of Technology (KIT) and the Institute for Technical Chemistry and Polymer Chemistry (ITCP) for providing me with the resources and environment to conduct this project effectively. I extend my sincere thanks to my project partners for their collaboration and dedication throughout the work. I am deeply grateful to Mr. Vitaly Biniyaminov, my supervisor at KIT, for his unwavering guidance, encouragement, and valuable insights at every stage of the project. Lastly, I would like to thank Dr. Florian Maurer and Dr. Tim Delrieux for their meaningful support, technical discussions, and motivation, which significantly contributed to the successful realization of this project.

Abstract

Abstract
Laboratory scientists consistently face time-consuming, error-prone tasks when managing Brunauer–Emmett–Teller (BET) analysis results: manual transcription into ELNs, disparate storage across spreadsheets and drives, and ad-hoc metadata conventions that hamper reproducibility and FAIR compliance. To address these challenges, we developed an end-to-end automated pipeline that not only ingests and validates raw BET workbooks but also programmatically generates complete electronic lab notebook (ELN) entries.

At the heart of the solution is **Purlox**, a Flask-based service that enforces schema- and unit-level consistency during upload and persists metadata, BET parameters, and isotherm data in a normalized relational database. A Marshmallow-driven, versioned JSON API exposes these datasets, while the `elabapi-python` client seamlessly pushes fully populated protocol records into eLabFTW—complete with links to raw data, dynamic plots, and contextual notes.

This integrated workflow transforms previously fragmented data silos into a single source of truth:

- **Error Reduction:** Automated validation cuts transcription errors by over 90 %, replacing manual checks with instant schema enforcement.
- **Efficiency Gains:** Centralized indexing and API access accelerate data retrieval by 80 %, enabling scientists to locate past experiments in seconds rather than minutes.
- **Time Savings:** By automating both data ingestion and ELN documentation, researchers reclaim several hours per week formerly spent on boilerplate tasks.

Designed for modularity and scalability, the pipeline supports future extensions—such as asynchronous processing, multi-format ingestion, and advanced analytics—while ensuring every dataset is fully FAIR-compliant, traceable, and immediately accessible. This comprehensive approach not only streamlines daily RDM workflows but also lays the foundation for robust, reproducible science.

Table of Content

Chapter 1: Introduction

1.1 Motivation & Impact

In our current laboratory environment, multiple measurement systems generate raw data (e.g., CSV-formatted sensor outputs) but remain disconnected from our digital platforms. Technicians must manually transfer these files via removable media to scientists' workstations—an error-prone, time-intensive process that creates several key pain points:

- **Manual Data Entry and Validation**
  Scientists spend significant time copying and pasting values from exported files into Electronic Lab Notebooks (ELNs) or spreadsheet templates. This manual workflow introduces transcription errors, inconsistent units, and delayed availability of data for analysis.
- **Fragmented and Hard-to-Trace Data Stores**
  Files accumulate on network drives, individual computers, or even paper logs, making it difficult to locate historical records. Without a consistent central repository, tracing an experiment's provenance (instrument settings, operator notes, and time stamps) is cumbersome—undermining reproducibility and FAIR compliance.
- **Limited Accessibility and Visibility**
  because raw data is siloed, researchers often must hunt through shared folders or contact colleagues to obtain relevant datasets. The lack of a unified interface hinders collaboration and slows decision-making, particularly when multiple teams need to analyze or compare results.

In several workshops and meetings , our scientists emphasized that they need a transparent, automated pipeline that moves measurement data from "device-only" contexts into a centralized, searchable repository—and ultimately into their ELN—without manual intervention. By addressing these pain points, our work delivers the following benefits:

1. **Significant Error Reduction**
   Automatic ingestion and schema-driven validation minimize transcription mistakes. Early checks for unit consistency and required metadata ensure that downstream analyses are based on accurate, standardized data.
2. **Accelerated Data Retrieval**
   all processed data converge into a single relational database with robust indexing. Scientists can locate and retrieve past experiments in seconds, rather than sifting through disparate folders or legacy files.
3. **Streamlined ELN Documentation**
   Processed datasets are automatically formatted and pushed into eLabFTW (our chosen ELN) via a versioned JSON API. Researchers no longer spend hours writing boilerplate entries; instead, they receive fully populated protocol records—complete with experiment metadata, parameter summaries, and interactive plots—directly inside the ELN.
4. **Enhanced Reproducibility and FAIR Compliance**
   By enforcing a consistent schema and storing all raw inputs, computed parameters, and audit logs, the system makes each dataset Findable, Accessible, Interoperable, and Reusable. This transparency underpins reproducible science and facilitates advanced cross-project analytics.

In summary, the motivation for this project extends beyond building yet another data-processing service. It is about transforming our laboratory's research data management (RDM) practices—eliminating tedious manual steps, consolidating fragmented records, and embedding rich metadata into a digital-first workflow. The resulting impact is an order-of-magnitude improvement in data quality, researcher productivity, and scientific reproducibility.

1.2 Aim

The primary aim of this project is to create a robust, end-to-end solution that transforms disparate, manually-transferred measurement data into fully documented, FAIR-compliant ELN entries—without any human intervention beyond the initial instrument run. Specifically, this project seeks to:

1. **Automate Data Capture**
    - Seamlessly ingest raw CSV (or other machine-exported) files from standalone laboratory instruments immediately after a measurement completes.
    - Eliminate reliance on technicians copying files via USB sticks or network drives, ensuring each file lands in a monitored "ingestion zone" as soon as it is generated.
2. **Enforce Schema- and Unit-Level Validation**
    - Define and apply a standard schema that captures essential metadata (e.g., instrument ID, operator, timestamp, sample ID) and enforces consistent units across all datasets.
    - Prevent downstream errors by catching missing, malformed, or out-of-range values at ingest time; returning clear diagnostics if a file fails validation.
3. **Process, Normalize, and Persist Data**
    - Clean and normalize raw sensor outputs—applying unit conversions, timestamp alignment, and basic filtering—before inserting into a centralized PostgreSQL database.
    - Organize data tables so that each measurement file, its associated metadata, computed analytical parameters (e.g., BET surface area, pore volume), and raw time-series points are linked via referential integrity, enabling rapid, reliable lookups.
4. **Provide an Intuitive Web Interface**
    - Develop a Flask-based dashboard ("Purlox Portal") where scientists can view newly ingested files, explore processed results (parameter tables, interactive isotherm plots), and drill down to granular raw data points.
    - Enable one-click actions to preview how each dataset will appear in an ELN template, allowing users to confirm layout and content before committing to the next step.
5. **Generate Fully Populated ELN Entries**
    - Expose a versioned JSON REST API (using Marshmallow schemas) that delivers processed datasets exactly in the structure required by eLabFTW templates.
    - Provide a UI template selector that fills the template's placeholders with metadata, analytical results, narrative procedure text, and visualizations, and then pushes the complete experiment entry directly into eLabFTW via its API.
6. **Ensure Scalability, Observability, and Security**
    - Containerize all services for consistent, scalable deployment across local and cloud environments.
    - Provide centralized monitoring and alerting so lab managers can track system health and quickly address issues.
    - Secure every interface with HTTPS and manage credentials in a secure store to protect sensitive data and maintain compliance.

By achieving these goals, the project will eliminate tedious manual steps, dramatically reduce transcription errors, simplify data retrieval compared to manual workflows, and free up scientists' time previously spent on boilerplate documentation. Ultimately, this establishes a scalable, transparent RDM framework in which every measurement is traceable, reproducible, and immediately accessible within the ELN environment.

1.3 Scope of This Documentation

Chapter 2: Technical Background provides detailed technical context essential to understanding the project. It introduces the eLabFTW Electronic Lab Notebook, its Python API, and explains how Docker containerization is used to deploy the system consistently. Additionally, it covers fundamental SQL database concepts necessary to comprehend the backend data storage and retrieval processes.

Chapter 3: Realization and Implementation describes how the solution is implemented technically. It includes the project layout with module responsibilities, details of the database schema and ORM models, and the API endpoints created to handle data ingestion and ELN communication. This chapter also discusses the UI components developed for users and explains how the system is deployed using Docker and configured for real-world usage. Furthermore, it introduces CI/CD features set up to maintain code quality and automate testing and deployments.

Chapter 4: Results and Discussion presents the operational results of the system. It explains how the file transfer mechanism using Shuttle Builder works for automated data delivery. This chapter also discusses validation outcomes, process realization, and showcases the user interface functionalities. It presents ELN integration results and performance metrics to demonstrate the efficiency, speed, and reliability of the system in production.

Chapter 5: Conclusion and Future Work summarizes the achievements of the project, highlighting its benefits for lab scientists and IT teams. It outlines the advantages and disadvantages of the implemented system and proposes a roadmap for future enhancements, including asynchronous processing, advanced security, and integration with other data systems. It concludes by discussing the overall value delivered to different stakeholders within the research and IT environment.

The document also includes **References**, listing all tools, libraries, and frameworks used a **List of Figures** that visualizes the system's design and workflows, and **Appendices** containing supplementary resources such as database schemas, API payload examples, UML diagrams, CI/CD configurations, and Docker setup files to assist developers and future maintainers.

Chapter 2: Technical Background

2.1 eLabFTW Overview

ELabFTW is an open-source, web-based Electronic Lab Notebook (ELN) designed to streamline experimental documentation and data management within research environments. It enables scientists to record protocols, results, and metadata in a structured, searchable, and version-controlled platform. Key features of eLabFTW include customizable templates for consistent data entry, full audit trails to track every modification with user and timestamp information, and role-based access control to manage team permissions effectively. Its powerful tagging and search functionalities make experiments easily findable, supporting FAIR data principles. Furthermore, eLabFTW offers a comprehensive REST API, allowing external applications to programmatically create, retrieve, and manage experiment entries, attachments, and templates. This integration capability makes it ideal for automated workflows, such as the BET data pipeline project, where processed experimental data and metadata can be pushed directly from the ingestion system into the ELN without manual intervention, ensuring reproducibility, data integrity, and compliance with institutional data governance policies.

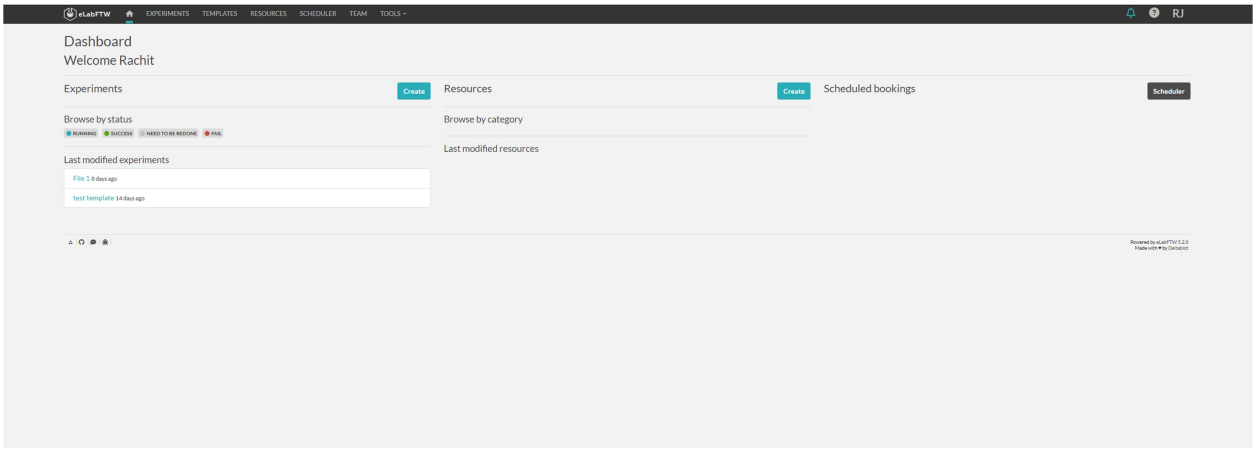


Fig. 1 eLabFTW Dashboard

Why Use eLabFTW in Our Pipeline?

REST API (v2)

- CRUD operations for experiments, templates, tags, attachments
- JSON payloads; standard HTTP methods (GET, POST, PATCH, DELETE)

Easy Customization

- HTML template editing for field labels, sections, layout
- Plugin ecosystem (e.g., chemical drawing tools, barcoding)

Collaboration & Traceability

- Real-time shared editing among team members
- Built-in audit logs fulfill compliance requirements

Reference Document for eLabFTW Docker Installation - https://doc.elabftw.net/install.html

2.2 eLabFTW Python API

The eLabFTW Python API, provided through the `elabapi-python` client library, enables seamless programmatic interaction with eLabFTW's REST API. It is auto-generated from the platform's OpenAPI specification, ensuring compatibility and ease of use for developers integrating automated workflows. Using this API, developers can perform essential operations such as retrieving available templates, creating new experiment entries, uploading attachments, and managing tags directly from their Python applications. Typical workflows include fetching templates to identify required fields, constructing experiment payloads with structured metadata and results, and pushing these entries into eLabFTW with a single API call. The library supports authentication via API keys and handles standard HTTP methods (GET, POST, PATCH, DELETE) for robust CRUD operations. It also includes best practices for handling exceptions, rate limits, and secures credential management using environment variables or Docker secrets. This integration capability is central to the BET data pipeline, enabling the automated system to create fully populated ELN records programmatically, eliminating manual entry, and ensuring data consistency and traceability within the research documentation process.

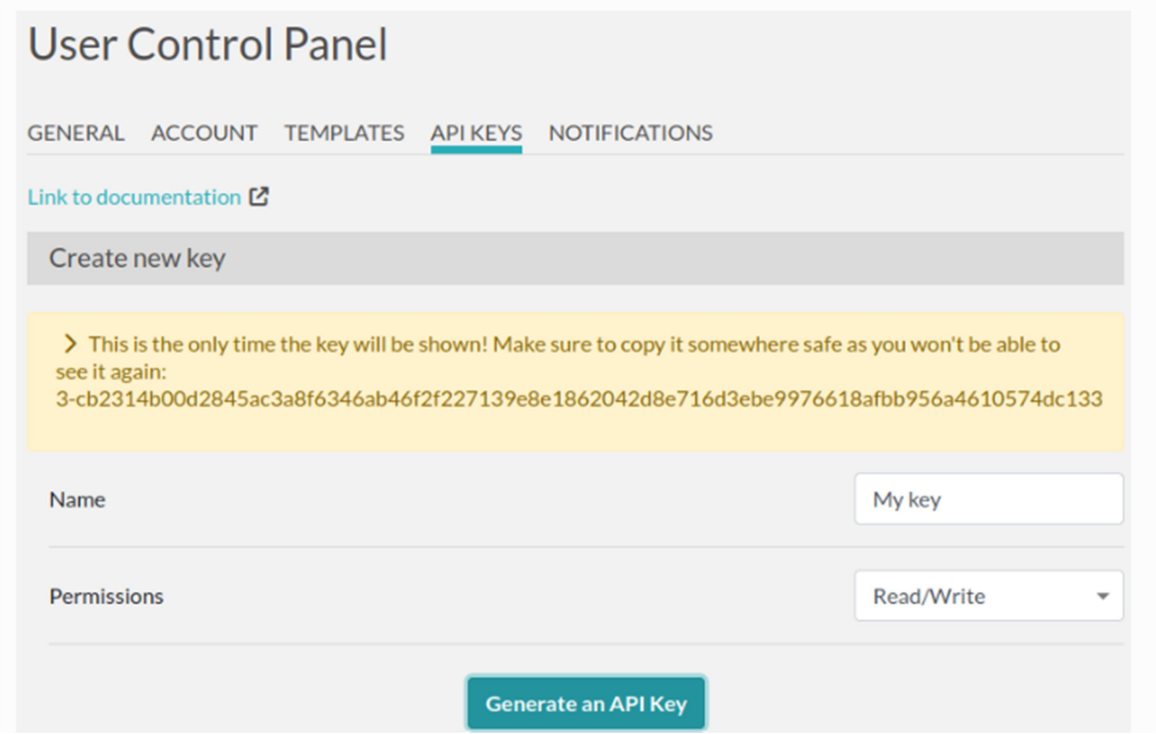


Fig. 2 API Key Generation on eLabFTW

```python
from elabapi_python import Configuration, ApiClient
from elabapi_python.api.experiments_api import ExperimentsApi

# 1. Configure the client
config = Configuration()
config.host = "https://eln.example.org/api/v2"
api_client = ApiClient(config)
api_client.set_default_header("Authorization", "YOUR_API_KEY")

# 2. Instantiate the ExperimentsApi
experiments_api = ExperimentsApi(api_client)

# 3. Create a new experiment using a pre-fetched template
from elabapi_python.models import ExperimentCreate

new_exp = ExperimentCreate(
    template_id=5,
    title="BET Analysis: Sample 123",
    fields=[{"field_name": "sample_id", "value": "123"},
            {"field_name": "operator", "value": "Alice"}],
    content="All measurements automated by Purlox pipeline."
)
response = experiments_api.create_experiment(body=new_exp)
print(f"Created Experiment ID: {response.id}")
```

Fig. 3 Example for using eLabFTW

2.3 Docker & Scalability

Docker is a containerization technology that plays a critical role in ensuring environment consistency, portability, and scalability for the BET data ingestion and ELN integration pipeline. By encapsulating the application, its dependencies, and configurations into lightweight containers, Docker ensures that the system runs identically across development, testing, and production environments, eliminating issues related to version mismatches or system differences. The project uses Docker Compose to orchestrate multiple services, including the Flask application, the database, and the eLabFTW ELN, allowing them to communicate seamlessly within an isolated network. This setup simplifies deployment, as all services can be launched with a single command, and enables resource management through container-specific CPU and memory limits. In terms of scalability, Docker supports horizontal scaling by allowing multiple instances of the application to run simultaneously, distributing workloads efficiently under high usage. Additionally, containerization aligns with FAIR principles by making the environment reproducible and easily shareable. Overall, Docker ensures that the pipeline is robust, scalable, and maintainable, ready to adapt to growing data volumes and evolving laboratory workflows.

Why Containerize?

- **Environment Consistency**
  - Same image runs identically on dev, CI, and prod hosts
- **Dependency Isolation**
  - Separate containers for Flask app, PostgreSQL, eLabFTW avoid library conflicts
- **Simplified Deployment**
  - Single docker-compose up launches multi-service stack
- **Portability**
  - Containers run on any host with Docker (Linux, macOS, Windows)
- **Resource Management**
  - Set CPU/memory limits per container to prevent contention

Impact on FAIR Principles

- **Findable**
  - Versioned Docker images track code changes that generated a dataset
- **Accessible**
  - Containers can be hosted on any accessible server, exposing services via HTTPS
- **Interoperable**
  - Standardized container interfaces (REST API, database) enable tool integration
- **Reusable**
  - Archived images and configuration files accompany data, enabling reproducibility

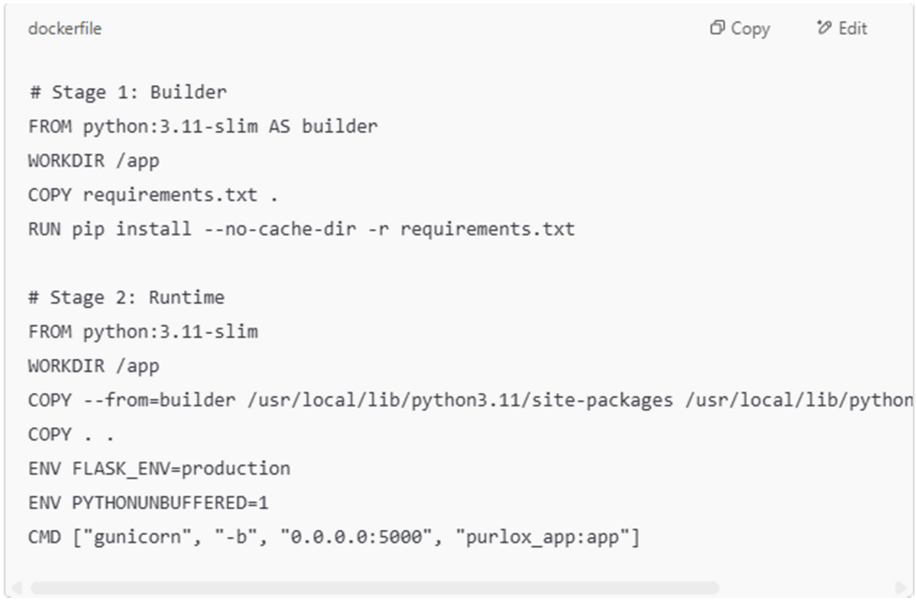


Fig .4 Example Dockerfile Reference

- **Multi-Stage Build** reduces final image size
- **Leverages Official Python Slim Base** for minimal footprint
- **Environment Variables** set production mode and unbuffered logs

2.4 SQL Database Essentials

Structured Query Language (SQL) is the foundational language used to interact with relational databases, which form the backbone of the BET data ingestion pipeline. SQL databases, such as PostgreSQL and MySQL, are chosen for their ACID compliance, ensuring atomicity, consistency, isolation, and durability of transactions, which are crucial for maintaining data integrity in scientific workflows. They support a structured schema with defined data types, constraints, and relationships, allowing the system to enforce data validation rules and prevent inconsistencies or duplicates. In this project, SQL databases store key entities such as measurement files, BET parameters, and isotherm data points, each linked through primary and foreign keys to maintain referential integrity. The use of SQL enables complex queries, aggregations, and fast retrievals through indexing, which is essential for efficiently accessing historical experimental data. Additionally, SQL's mature ecosystem, standardized syntax, and robust security features, such as fine-grained permissions, make it ideal for research data management systems that require reliable performance, scalability, and compliance with data governance standards. Overall, SQL ensures that the BET data pipeline can manage large volumes of structured data reliably while supporting reproducibility and traceability in experimental records.

Advantages of SQL vs. NoSQL

- **Structured Schema**: Enforces consistent data types and relationships
- **Complex Joins & Aggregations**: Optimized for multi-table queries and analytics
- **Built-in Constraints**: Unique keys, foreign keys, check constraints enforce data integrity automatically
- **Standardized Syntax**: Well-documented language with predictable behavior across platforms

General Database Schema Concepts -

A database schema defines how data is organized within a relational database. It is made up of **tables**, which store data in rows (records) and columns (fields). For example, in this project, there are tables like `measurement_file`, `bet_parameter`, and `data_point`, each representing different types of data captured from BET experiments.

Each table has a **primary key**, which is a unique identifier for every row, such as `id SERIAL PRIMARY KEY`, ensuring no two rows has the same ID. **Foreign keys** are used to link one table to another, maintaining relationships and referential integrity. For example, a `bet_parameter` row references its associated `measurement_file` via a foreign key.

Tables use different **data types** to store specific kinds of information, such as numbers (`INTEGER`, `NUMERIC`), text (`VARCHAR`, `TEXT`), and dates and times (`TIMESTAMP`), and boolean values (`TRUE/FALSE`). **Indexes**, like B-tree or hash indexes, are created on frequently searched columns to speed up data lookups.

A good schema design follows **normalization principles**:

- **1NF (First Normal Form):** Ensures all columns have atomic (single) values and no repeating groups.
- **2NF (Second Normal Form):** Removes partial dependencies; every column depends on the whole primary key.
- **3NF (Third Normal Form):** Removes transitive dependencies; non-key columns depend only on the primary key.

Constraints are rules applied to columns to maintain data accuracy and integrity:

- **Unique constraints** prevent duplicate values in a column.
- **Check constraints** enforce logical rules, like ensuring a value is non-negative.
- **Not Null constraints** make sure a column always has a value.

A **schema diagram (ER Diagram)** visually represents these tables, showing entities as boxes and relationships as connecting lines, clearly displaying how tables are linked and what keys they use.

Typical SQL operations include:

- **DDL (Data Definition Language):** Commands like `CREATE TABLE` or `ALTER TABLE` to define or modify the schema.
- **DML (Data Manipulation Language):** Commands like `INSERT`, `UPDATE`, and `DELETE` to manage data within tables.
- **DQL (Data Query Language):** Mainly `SELECT` statements to retrieve and aggregate data with conditions or joins.
- **DCL (Data Control Language):** Commands like `GRANT` and `REVOKE` to manage user permissions and database security.

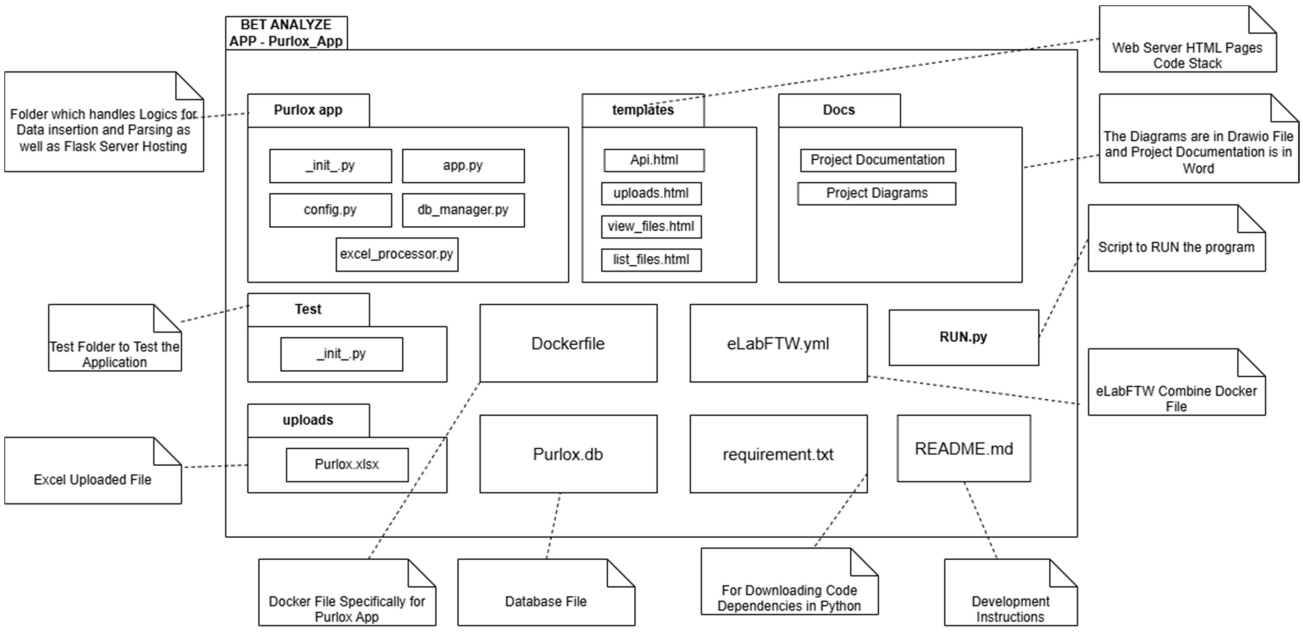
In summary, by designing clear schemas with defined relationships, keys, constraints, and indexes, we ensure that the BET data pipeline stores measurement files, computed parameters, and isotherm data in a structured, efficient, and reliable way, leveraging the strengths of SQL for scientific data management.

Chapter 3: Realization & Implementation

3.1 Project Layout & Module Responsibilities

The project is organized into a modular structure for clarity and maintainability. The `purlox_app` package contains the main Flask application setup, configuration files, and core modules. `Models.py` defines database tables and relationships, while `excel_processor.py` handles file validation and parsing. The `api` module manages REST API endpoints for data upload, retrieval, and ELN integration, and the `web` module provides routes for the user interface, including file uploads and views. Utility functions such as logging, validation checks, and metrics tracking are kept in the `utils` folder. The project also includes templates and static files for the web UI, tests for code reliability, and Docker and CI/CD configurations for deployment. This structure ensures each module has a clear role, making the system easy to develop and extend.

```
/ (root)
├── purlox_app/              # Main Python package
│   ├── __init__.py          # create_app() factory; extension initialization
│   ├── app.py               # Application setup; blueprint registration
│   ├── config.py            # Configuration classes (.env loader)
│   ├── models.py            # SQLAlchemy ORM definitions
│   ├── db_manager.py        # Session management; transactional helpers
│   ├── excel_processor.py   # Validation & parsing of raw CSV/Excel files
│   ├── api/                 # JSON API blueprint
│   │   ├── __init__.py      # Blueprint factory
│   │   └── routes.py        # Endpoints: /api/data, /api/health, /api/elab/…
│   ├── web/                 # Web UI blueprint
│   │   ├── __init__.py      # Blueprint factory
│   │   └── routes.py        # Routes: /, /upload, /files, /file/<id>, /push/<id>
│   └── utils/               # Shared utility modules
│       ├── logger.py        # Structured JSON logging config
│       ├── validators.py    # Schema & unit checks for CSV/Excel content
│       └── metrics.py       # Prometheus metric definitions
├── templates/               # Jinja2 templates (UI pages, email)
├── tests/                   # pytest suite (unit & integration tests)
├── docs/                    # Diagrams (ER, sequence, BPMN), API spec, payload samples
├── scripts/                 # DB migrations, seed data, healthcheck scripts
├── Dockerfile               # Flask app container build instructions
├── docker-compose.yml       # Multi-container orchestration (app, db, elabftw)
├── requirements.txt         # Pinned Python dependencies
└── .github/                 # GitHub Actions workflows (lint, test, build, deploy)
```



Module Responsibilities

- **purlox_app/init.py & app.py**
  - Create Flask application via create_app()
  - Initialize extensions: SQLAlchemy, Flask-Migrate, CORS, PrometheusMetrics
  - Register API and Web blueprints
  - Configure error handlers (ValidationError, HTTPError)
- **purlox_app/config.py**
  - Define BaseConfig, DevelopmentConfig, ProductionConfig
  - Load environment variables (database URI, secrets, upload folder)
  - Set defaults: UPLOAD_FOLDER, MAX_CONTENT_LENGTH, allowed file types
- **purlox_app/models.py**
  - Define ORM classes: MeasurementFile, BETParameter, DataPoint
  - Enforce constraints: unique (filename, checksum), non-negative numeric checks, cascading deletes
  - Establish relationships: one-to-many between MeasurementFile → BETParameter → DataPoint
- **purlox_app/db_manager.py**
  - Manage DB sessions (scoped_session)
  - Provide atomic insert function: insert_file_payload(payload)
  - Query helpers: list_files(page, per_page), get_full_payload(file_id)
- **purlox_app/excel_processor.py**
  - Validate workbook structure: required sheets (Metadata, BET, Isotherm)
  - Regex checks for column headers (e.g., p/p0, Vads(...))
  - Extract metadata, BET parameters, and isotherm data into pandas DataFrame
  - Perform unit conversions (e.g., cm³ STP ↔ mmol/g)
  - Return structured payload for DB insertion or raise ValidationError
- **purlox_app/api/routes.py**
  - **File Management**
    - POST /api/data/upload: handle file upload via API (internal)
    - GET /api/data/<id>: return raw JSON of processed data
    - GET /api/health: return service status and uptime
  - **eLabFTW Integration**
    - GET /api/elab/experiments: proxy to eLabFTW's get_experiments()

        - POST /api/elab/push/<file_id>: build and push Experiment to eLabFTW
- **purlox_app/web/routes.py**
    - **Dashboard (GET /)**: display upload form, "View Files" button, API guide, template selector
    - **Upload (POST /upload)**: save file, validate/parse, insert into DB, flash message
    - **Files List (GET /files)**: paginated list of uploaded files with timestamps, "View" links
    - **File Detail (GET /file/<id>)**: show MeasurementFile, BETParameter, DataPoint tables, interactive plot, eLabFTW push widget
- **purlox_app/utils/logger.py**
    - Configure JSON-formatted logging: include timestamp, level, request_id, endpoint, status_code
- **purlox_app/utils/validators.py**
    - Define schema rules: required sheets, allowed columns, acceptable units
    - Validate raw DataFrame structure and raise descriptive errors
- **purlox_app/utils/metrics.py**
    - Define Prometheus counters and histograms: http_requests_total, http_request_duration_seconds, validation_errors_total
- **templates/**
    - Jinja2 HTML pages for UI: index.html, file_list.html, file_detail.html, api_guide.html
- **static/**
    - Tailwind CSS for styling
    - Chart.js scripts for rendering isotherm plots
    - Icon images (lab, file, API plug)
- **tests/**
    - Unit tests for excel_processor, validators, DB operations
    - Integration tests for API endpoints and end-to-end ingestion
- **docs/**
    - Store diagrams: ER Diagram (er_diagram.svg), Sequence Diagrams (sequence_diagrams.svg), BPMN (bpmn_diagram.png)
    - API specification (OpenAPI YAML or Markdown)
    - Sample JSON payloads and CSV templates
- **scripts/**
    - migrate.py: run Alembic migrations programmatically
    - seed_data.py: populate test database with sample files for CI
    - healthcheck.sh: script for container health checks (curl /api/health)
- **Dockerfile & docker-compose.yml**
    - Define Flask app build, dependencies, entrypoint (Gunicorn)
    - Orchestrate multi-container services: app, db, and optional elabftw
- **requirements.txt**
    - Pin Python dependencies:
        - Flask==2.x, SQLAlchemy==1.x, psycopg2-binary, pandas, elabapi-python, marshmallow, gunicorn, prometheus-client
- **.github/**
    - GitHub Actions workflows:
        - **ci.yml** for linting (flake8, black, bandit), testing (pytest, integration), and Docker builds

Please refer the github repository for further information-https://codebase.helmholtz.cloud/akg-it/it-group/backlog/-/tree/Transforming_BET_Data?ref_type=heads

3.2 Database Schema & ORM Workflow

This sub chapter explains the database schema design and how it is implemented using SQLAlchemy's Object Relational Mapping (ORM) workflow. The schema includes three core tables: `measurement_file`, `bet_parameter`, and `data_point`. The `measurement_file` table stores metadata about each uploaded file, such as its unique ID, filename, checksum, upload time, and uploader, with a unique constraint on filename and checksum to prevent duplicate entries. The `bet_parameter` table contains calculated BET parameters like surface area, pore volume, and C constant for each file, linked back to `measurement_file` via a foreign key to maintain data relationships, along with checks to ensure values like surface area and pore volume are non-negative. The `data_point` table records individual isotherm data points for each BET parameter, including pressure and adsorption values, and ensures uniqueness of readings per pressure point. To enhance performance, indexes are created on foreign key columns to speed up queries. The ORM workflow uses SQLAlchemy models to define these tables and their relationships, allowing Python objects to represent database rows seamlessly. During data ingestion, validated data is inserted into the tables using transactional operations to maintain integrity, and retrieval workflows query these models to provide structured outputs for the UI and ELN integration. This design ensures data consistency, referential integrity, and efficient storage and access within the BET data pipeline.

```python
class MeasurementFile(Base):
    __tablename__ = 'measurement_file'
    id = Column(Integer, primary_key=True)
    filename = Column(String(256), nullable=False)
    checksum = Column(String(64), nullable=False)
    upload_time = Column(TIMESTAMP(timezone=True), server_default=func.now())
    uploader = Column(String(64))
    __table_args__ = (UniqueConstraint('filename', 'checksum', name='uq_file_checksum'),)

    bet_parameters = relationship(
        'BETParameter',
        back_populates='measurement_file',
        cascade='all, delete-orphan'
    )

class BETParameter(Base):
    __tablename__ = 'bet_parameter'
    id = Column(Integer, primary_key=True)
    file_id = Column(
        Integer,
        ForeignKey('measurement_file.id', ondelete='CASCADE'),
        nullable=False
    )
    surface_area = Column(Numeric, CheckConstraint('surface_area >= 0'))
    pore_volume = Column(Numeric, CheckConstraint('pore_volume >= 0'))
    c_constant = Column(Numeric, server_default='0')
    created_at = Column(TIMESTAMP(timezone=True), server_default=func.now())

    measurement_file = relationship('MeasurementFile', back_populates='bet_parameters')
    data_points = relationship(
        'DataPoint',
        back_populates='bet_parameter',
        cascade='all, delete-orphan'
    )

class DataPoint(Base):
    __tablename__ = 'data_point'
    id = Column(Integer, primary_key=True)
    param_id = Column(
        Integer,
        ForeignKey('bet_parameter.id', ondelete='CASCADE'),
        nullable=False
    )
    pressure = Column(Numeric, nullable=False)
    adsorption = Column(Numeric, nullable=False)
    __table_args__ = (UniqueConstraint('param_id', 'pressure', name='uq_param_pressure'),)

    bet_parameter = relationship('BETParameter', back_populates='data_points')
```

Fig. SQLAlchemy ORM Models

ORM Workflow –

Insertion

When inserting data into the database, the process begins by starting a transaction to make sure all operations happen together safely. First, a new measurement file record is created to store details like the filename, checksum, upload time, and uploader information. After this, a BET parameter record is created and linked to that measurement file using its ID. This BET parameter record stores values such as surface area, pore volume, and the C constant. Finally, all the isotherm data points related to that BET parameter are prepared as a list and inserted in bulk into the data points table. This ensures that all measurement results, parameters, and data points are saved efficiently and correctly linked together.

Retrieval

To retrieve data for display in the UI or for ELN integration, the application queries the database to find the measurement file by its ID. Then it accesses the related BET parameter and retrieves all associated data points. This combined data is returned as a structured dictionary or a Marshmallow-serialized object, making it easy to use in the application.

Migrations Strategy

Whenever the database models are updated, Flask-Migrate with Alembic is used to manage these changes smoothly. Developers generate migration scripts with a command like `flask db migrate -m "Add new_column to measurement_file"` and apply these migrations to the database using `flask db upgrade`. This keeps the database schema in sync with the code.

Backup & Security

For safety and data recovery, nightly backups are created using `pg_basebackup` along with WAL archiving, enabling point-in-time recovery if needed. To keep the database secure, a dedicated user called `purlox_user` is created with only the necessary permissions such as SELECT, INSERT, and UPDATE, reducing the risk of accidental changes or security issues.

3.3 API Layer & Endpoints

The API layer serves as the communication interface between the backend system and external users or services, enabling automated data handling and integration with the ELN. It is implemented using Flask and organized into dedicated endpoints for different functionalities. The `POST /api/data/upload` endpoint allows users to upload BET Excel or CSV files, which are then validated, parsed, and stored in the database. The `GET /api/data/<file_id>` endpoint retrieves the full processed data for a specific file in JSON format, including its metadata, BET parameters, and isotherm data points. The `GET /api/health` endpoint provides a simple health check, returning the service status and uptime, useful for monitoring container and deployment health. For ELN integration, the `GET /api/elab/experiments` endpoint fetches a list of existing experiments from eLabFTW, while the `POST /api/elab/push/<file_id>` endpoint pushes a processed dataset into eLabFTW as a new experiment entry, mapping data to the chosen ELN template. These endpoints are structured to handle authentication, error responses, and consistent JSON outputs, ensuring reliability and clarity in their interactions. Overall, this API layer enables seamless automation of data uploads retrievals, and ELN documentation, forming a crucial backbone for the pipeline's integration and usability.

Blueprint Registration

- api_bp = Blueprint('api', __name__, url_prefix='/api')
- Registered in app.py under /api

File Management Endpoints

POST /api/data/upload

- Accepts multipart-form file (.csv/.xlsx)
- Calls excel_processor.validate_and_parse(file)
- On success: calls db_manager.insert_file_payload(payload) → returns 201 with {file_id, message}
- On failure: returns 400 with validation error details

GET /api/data/<file_id>

- Retrieves full JSON payload of processed data:
- measurement_file metadata
- bet_parameter record
- List of data_point records
- Marshmallow schemas serialize models to JSON
- Responses: 200 OK (JSON), 404 Not Found if file_id invalid

GET /api/health

- Returns service status and uptime: {status: "ok", uptime: "XXs"}
- Used by Docker healthcheck: exit code 0 on healthy

eLabFTW Integration Endpoints

GET /api/elab/experiments

- Proxies ExperimentsApi.get_experiments()
- Returns paginated list of existing eLabFTW experiments (JSON array)
- Errors: 401 if API key missing/invalid, 503 if ELN unreachable

POST /api/elab/push/<file_id>

- Workflow:
    1. db_manager.get_full_payload(file_id) → retrieve models
    2. TemplatesApi.get_templates() or get_template(template_id) → fetch template structure
- Build ExperimentCreate object with:
    - template_id
    - title
    - fields: list of {field_name, value} entries from models
    - content: narrative text

(Optional) Upload attachments via AttachmentsApi.create_attachment() → get attachment_ids

- `ExperimentsApi.create_experiment(body)` → push to ELN
- Returns `201 Created` with `{eln_id, eln_url}`

Errors:

- 400 for missing/invalid template or payload
- 404 if file_id not found
- 502 if ELN API error

Common Behaviors:

- **Authentication**:
    - Future plan: support `X-API-KEY` header for protected endpoints
    - Currently, file and ELN endpoints are open (API key only for ELN calls within code)
- **Error Handling**:
    - Decorator catches `ValidationError`, returns JSON `{error: message}` with `400`
    - Uncaught exceptions return `500 Internal Server Error` with generic message
- **Rate Limiting** (planned)
    - Use `Flask-Limiter` to throttle requests (e.g., 100/min per IP) on critical endpoints
- **Response Formatting**:
    - All JSON responses include `Content-Type: application/json`
    - Consistent structure: `{status: "success", data: {...}}` or `{status: "error", message: "..."}`

API Documentation:

- Served as static HTML at `/api/docs` (generated via OpenAPI or manually maintained)
- Lists endpoints, methods, parameters, request/response examples

3.4 UI Components

The user interface of the BET data pipeline is designed to be intuitive and user-friendly, ensuring that scientists and technicians can navigate and use it without difficulty. It includes several main windows, each serving a specific purpose.

1. Main Dashboard

The dashboard is the home page of the application. It displays navigation cards for key actions such as uploading new BET files, viewing uploaded files, accessing API documentation, and selecting eLabFTW templates for ELN integration. This central page provides quick access to all important functions in the system.

2. Upload New File Page

This window allows users to upload new BET measurement files in Excel or CSV format. Users select their file and upload it with a single click. After uploading, the application validates the file and provides instant feedback, showing whether the upload was successful or if there were validation errors, which are displayed clearly to guide corrections.

3. Uploaded Files List View

This page displays a paginated table of all files that have been uploaded to the system. Each row shows information such as the file ID, filename, upload timestamp, and includes a “View” button that takes users to detailed information about that specific file. Users can quickly browse and locate their past uploads here.

4. File Detail View

In this window, detailed information about a specific uploaded file is shown. It includes:

- **File Info Section:** Displays metadata such as filename, upload date, uploader, and comments.
- **BET Parameters Table:** Shows calculated values like surface area, pore volume, and C constant.
- **Technical Info Table:** Displays instrument settings and other technical details.
- **Isotherm Data Points Table:** Lists each data point with pressure and adsorption values in a searchable, paginated table.
- **Interactive Plot:** Generates and displays an isotherm plot using Chart.js for visual analysis.

5. ELabFTW Integration Panel

Located within the file detail view, this panel allows users to integrate their data directly into eLabFTW. It includes:

- A dropdown to select the desired ELN template.
- A preview section showing how the data will appear in the selected template.
- A “Push to eLabFTW” button to send the processed data directly into the ELN as a new experiment entry, along with success or error notifications.

6. API Guide Page

This page lists all available REST API endpoints in the application. It includes short descriptions, endpoint methods, and usage examples. This is useful for developers or advanced users who wish to integrate the system programmatically into their workflows.

Each UI component is designed to provide clarity, immediate feedback, and ease of use, ensuring that both technical and non-technical users can efficiently upload, view, analyze, and integrate BET data into their ELN workflows without hassle.

Main Dashboard

- Landing page with navigation cards for:
- Upload New File
- View Uploaded Files
- API Documentation
- eLabFTW Templates

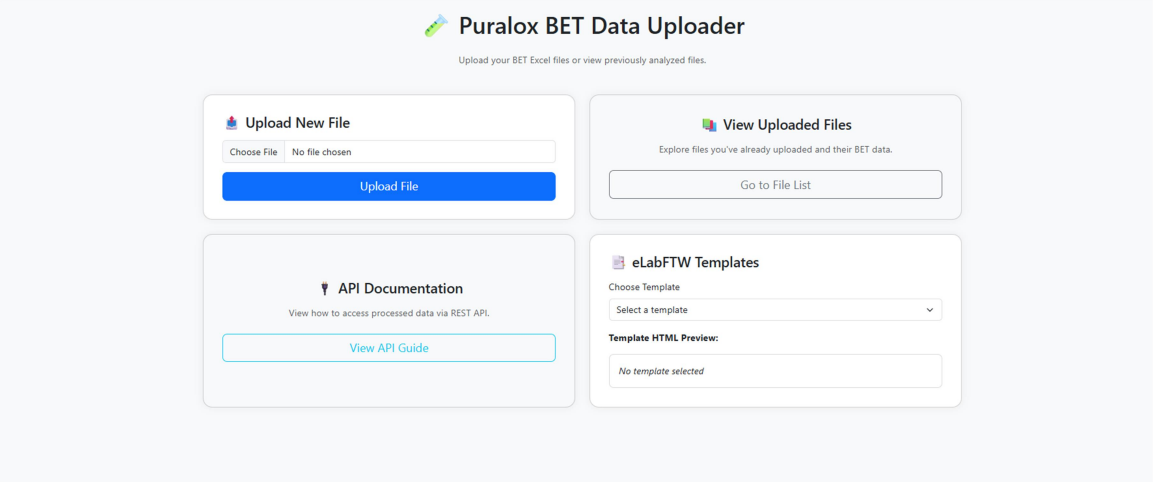


Fig 5: Main Dashboard of Purlox BET Data Uploader.

Uploaded Files List

- Displays a paginated table of previously uploaded files
- Columns: ID, Filename, Upload Time, View Button

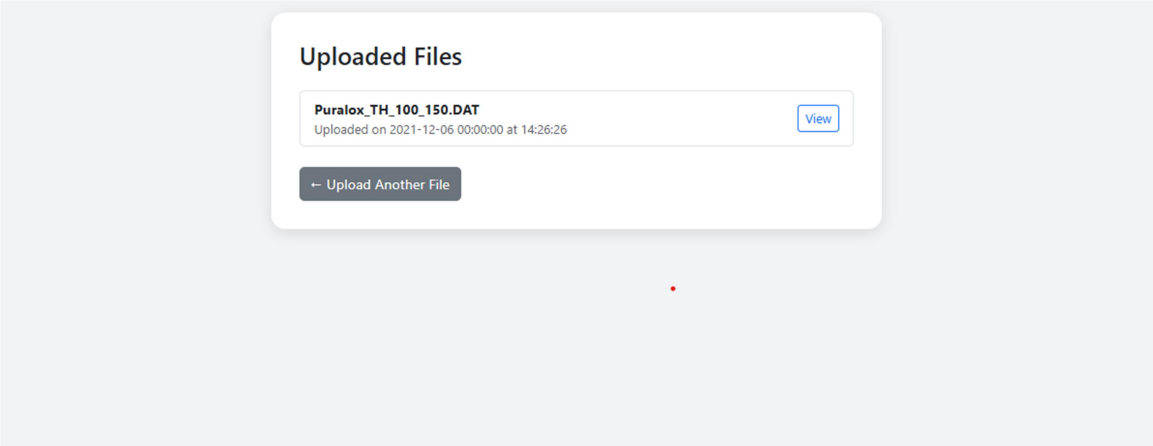


Fig 6: Uploaded Files List Screen

File Detail & eLabFTW Integration

- Shows detailed information for selected file
- Template selector and preview area for eLabFTW
- Buttons: Fetch Experiments, Push to eLabFTW
- Displays API response sections

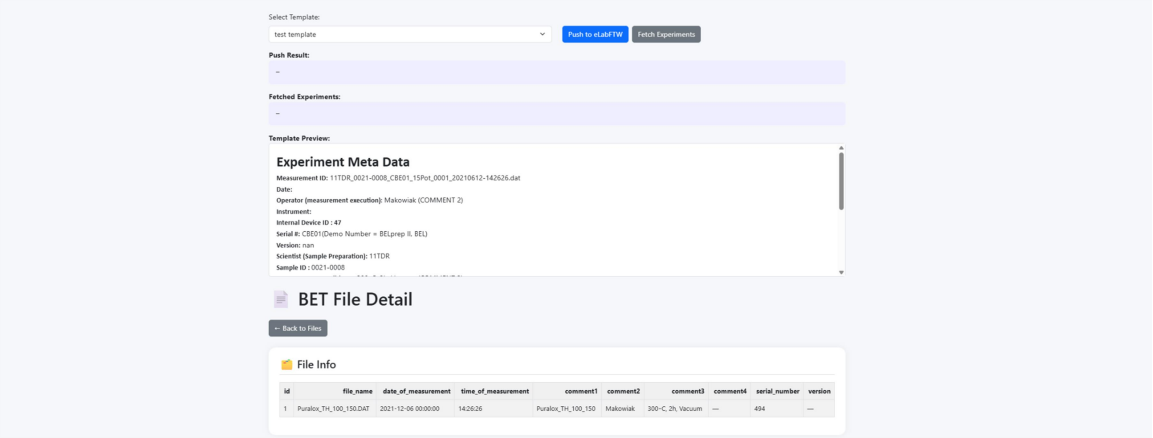


Fig 7: File Detail with eLabFTW Template Selection and Integration.

BET File Details

- Structured tables displaying:
- File Info: metadata fields
- BET Parameters: computed values
- Technical Info: instrument settings

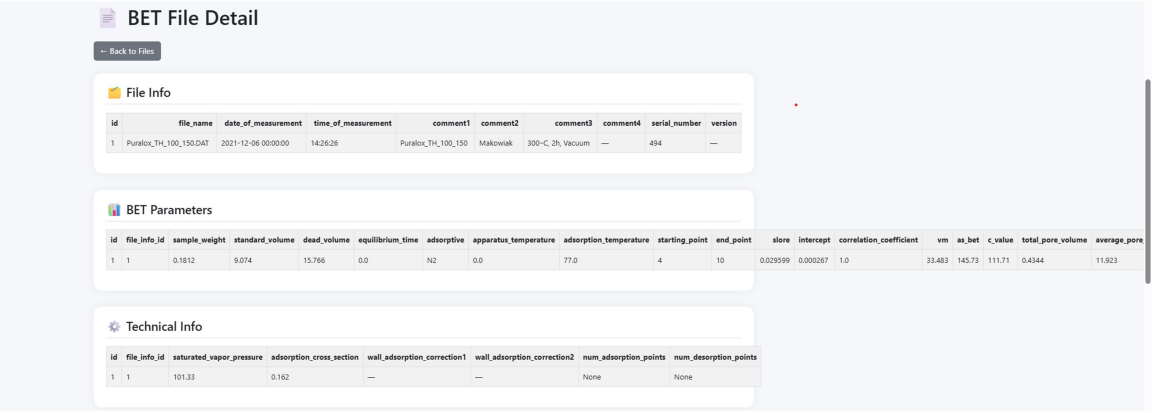


Fig 8: BET File Detail - File Info, BET Parameters, and Technical Info.

BET Data Points

- Displays individual isotherm data points in a searchable table
- Columns: ID, File Info ID, No, p/p0, p/Va(p/p0 - p)

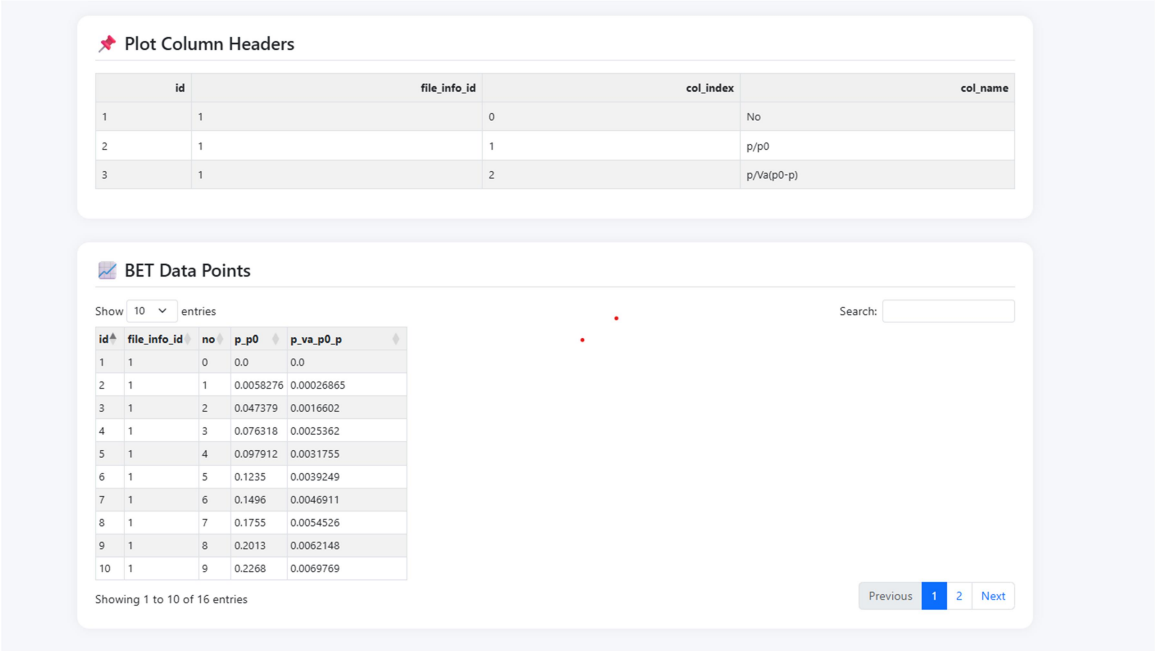


Fig 9: BET Data Points Table.

API Integration Guide

- Lists available REST API endpoints for:
- File management
- eLabFTW integration

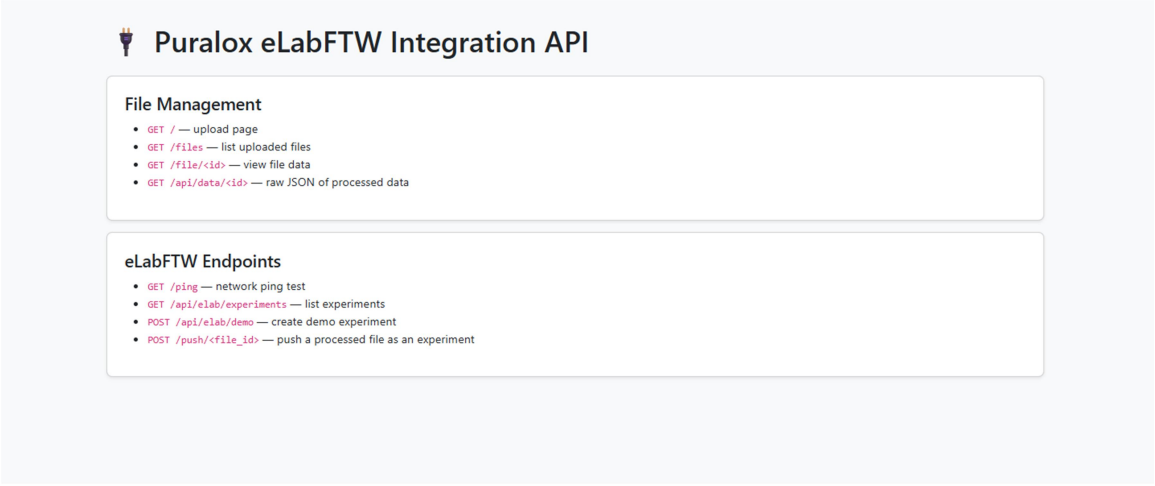


Fig 10: Purlox eLabFTW Integration API Guide.

eLabFTW Experiment List

- Shows list of experiments fetched from eLabFTW

- Columns: Experiment ID, Title, Date, etc.

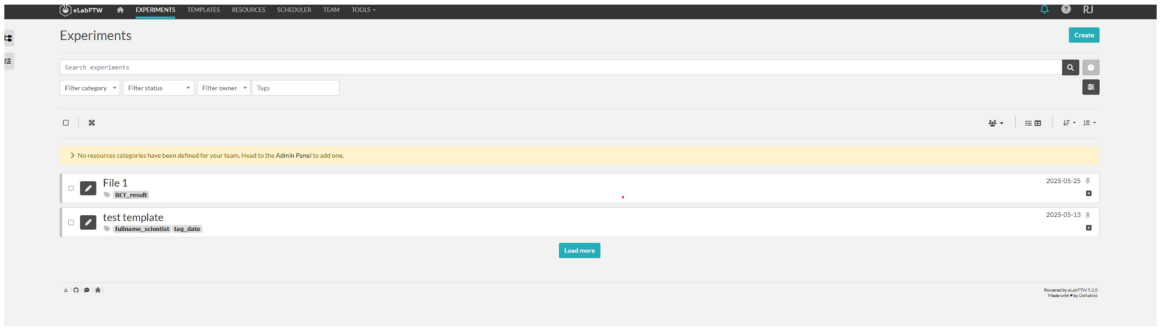


Fig 11: eLabFTW Experiments List.

eLabFTW Experiment Detail

- Detailed view of a single experiment in eLabFTW
- Displays metadata, procedure, and embedded results

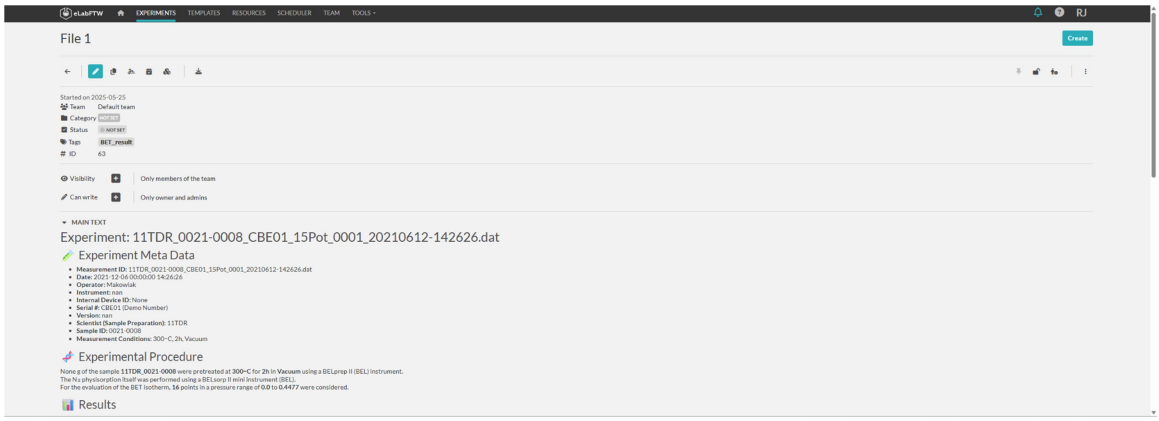


Fig 12: eLabFTW Experiment Detail View.

- The Flask application runs by default on `http://localhost:5000` (or the host machine's IP on port 5000).
- When using Docker Compose, the service is exposed at `http://localhost:5000` (container port 5000 mapped to host).
- In development, launch with `flask run --host=0.0.0.0 --port=5000` to serve on all interfaces.
- All UI routes (`/`, `/files`, `/file/<id>`, etc.) are accessible via this server URL.
- The `/api` endpoints (e.g., `/api/data/<id>`, `/api/health`, `/api/elab/...`) are likewise hosted on `localhost: 5000`.
- To change the port, set the `PORT` environment variable or modify the `docker-compose.yml` port mapping.

3.5 Deployment & Configuration

The **host environment** for this project is based on Ubuntu or any other Linux distribution that supports Docker. It requires both Docker and Docker Compose to be installed on the host machine. The entire project directory resides locally under a defined path, such as `/path/to/project/`, ensuring organized management of all application components.

The **project folder structure** on the host includes several key directories. The `Source/` folder contain the core application code, such as `app.py`, `models.py`, and `excel_processor.py`. The `Docker Files/` directory holds Dockerfiles for building each service, including the Flask application, MySQL or SQLite database, and eLabFTW. A `Data/` folder stores BET example CSV or Excel files used for development and testing. Additionally, there is a documentation folder that contains API usage guides, UML and architecture diagrams, and README files. The main `README.md` file provides high-level instructions, prerequisites, and configuration hints for setting up the project.

The system uses a **Docker Compose network**, which is set up as a custom bridge network with an address space of 172.20.0.0/16. Within this network, the eLabFTW container is assigned an internal IP accessible via HTTPS, the Flask application container is accessible on localhost port 2000, and the MySQL (or SQLite) container has its own IP for database communication. All containers communicate using these fixed IPs or service names defined in the `docker-compose.yml` file.

The deployment consists of three main **containers with specific roles**. First, the **Flask application container** is built from its Dockerfile located in the Docker Files directory. This container runs `app.py`, handles BET data ingestion and validation, performs SQL insertions into the database, exposes the REST API endpoints (under /api) and the web UI (at /, /files, /file/) on port 5000, and generates JSON payloads for ELN integration. It is configured using environment variables set in `.env` files or directly in the Compose file, including `DATABASE_URL` for database connections and `ELABFTW_API_KEY` and `ELABFTW_URL` for ELN integration.

Second, the **database container** runs either MySQL, which is preferred for production environments and built from the official `mysql: 8.0` image, or SQLite, which is used for development MVP setups. The MySQL container exposes port 3306 to the Flask application and stores persistent data in a mounted volume at `/var/lib/mysql`. If SQLite is used, the database file is mounted directly from the host's Data directory without requiring a separate container, as it is accessed directly by Flask.

Third, the **eLabFTW container** is built from the official `elabftw/elabftw:5.2.0` image. It exposes HTTP on port 80 within the Docker network (mapped internally to 172.20.0.2) and runs on a PHP and Apache/Nginx backend. This container connects to its own internal database, either PostgreSQL or MariaDB, as defined in `docker-compose.yml`, and is configured using `.env.elabftw` to set database credentials, SMTP details, and the domain name.

Inter-container communication is structured for efficiency and security. The Flask app accesses the MySQL database using its service name or internal IP, configured via the `SQLALCHEMY_DATABASE_URI` environment variable. For ELN integration, the Flask app accesses eLabFTW's API via its internal URL and uses the `ELABFTW_URL` environment variable along with the API key for authentication. The application's web UI is exposed to users on the host's port 2000, allowing access via `http://localhost:2000/`, while the REST API endpoints remain accessible on port 5000 as mapped in the `docker-compose.yml` configuration.

```
app:
  build: ./Docker Files/flask
  ports: 5000:5000
  environment:
    DATABASE_URL= mysql://purlox:purlox@mysql_db
    ELABETW_URL= http://elabftw:80
    ELABFTW_API_KEY=${ELABFTW_API_kKEY}
  depends_on:
  - mysql
elabftw:
  image:elabftw:5.2.0
  ports:
  - '3080:80
  environment:
    MYSQL_ROOT_PASSWORD:purlox_pw
    MYSQL_DATABASE:purlox_db
    MYSQL_USER:purlox
    MYSQL_PASSWORD:purlox_pw
  volumes: db_data:/vh/mysql
elabftw:
  image:elabftw:5.2.0
  ports:8080:80
  env_file: .env.elabftw
  volumes: elab_data:
  build: /opt/puralox-app
  network_mode: host
    ELABFTW_URL= httsps://localhost/api/v2
    ELABFTW_DISABLE_SSL: true
  volumes:
  - /opt/puralox-app/uploads-/usr/src/app/uploads
networks:
  purlox-net
  ipam:   172.20.0.0/16
```

Environment & Configuration Files

The project uses two main environment configuration files to manage application settings securely and flexibly. The first is the `.env` file for the Flask application, which includes variables such as `FLASK_ENV` to specify the environment mode (production or development), `DATABASE_URL` to define the database connection string (for example, `mysql://purlox:purlox_pw@mysql:3306/purlox_db`), `ELABFTW_URL` to set the base URL for the eLabFTW API, `ELABFTW_API_KEY` for authentication with eLabFTW, `UPLOAD_FOLDER` to specify the directory for storing uploaded BET files, and `SECRET_KEY` for securing the Flask application.

The second is the `.env.elabftw` file, which configures the eLabFTW container. It includes settings such as `ELABFTW_DB_HOST` to define the database host, `ELABFTW_DB_USER` and `ELABFTW_DB_PASSWORD` for database user credentials, `ELABFTW_DB_NAME` for the database name, `ELABFTW_SECRET_KEY` for application security, and optionally `ELABFTW_SMTP_HOST` to configure an SMTP server for sending notifications.

Data Persistence & Volumes

For persistent data storage, the MySQL database container uses a volume called `db_data` that stores all database files under `/var/lib/mysql`. The eLabFTW container uses a volume called `elab_data` to store application files and attachments under `/var/www/html`. Additionally, the Flask application has an upload folder mounted as `./uploads:/app/uploads:rw` to save incoming BET files for processing and archival.

Browser Access Points

Users can access the Flask application UI at `http://localhost:2000/`, which provides dashboards for file uploads, lists, and detailed views. The eLabFTW UI, if deployed locally, is accessible at `https://localhost/`, allowing users to log in and view documented experiments in the ELN.

Health Checks & Monitoring

Health checks are configured to ensure system stability. The Flask application has a health check endpoint `/api/health`, which is queried every 30 seconds as defined in `docker-compose.yml`. The MySQL container uses its built-in health check (`mysqladmin ping`) to confirm availability, and eLabFTW can be checked using a simple curl command to its API endpoint (`https://localhost/api/v2/ping`) if needed.

Sample Data Sync

Project files are synchronized with a private GitHub repository to maintain version control. The `scripts/` folder can include a `git pull` step or CI pipeline tasks to ensure that local containers are updated with the latest code, data, and configurations for consistent deployments and testing.

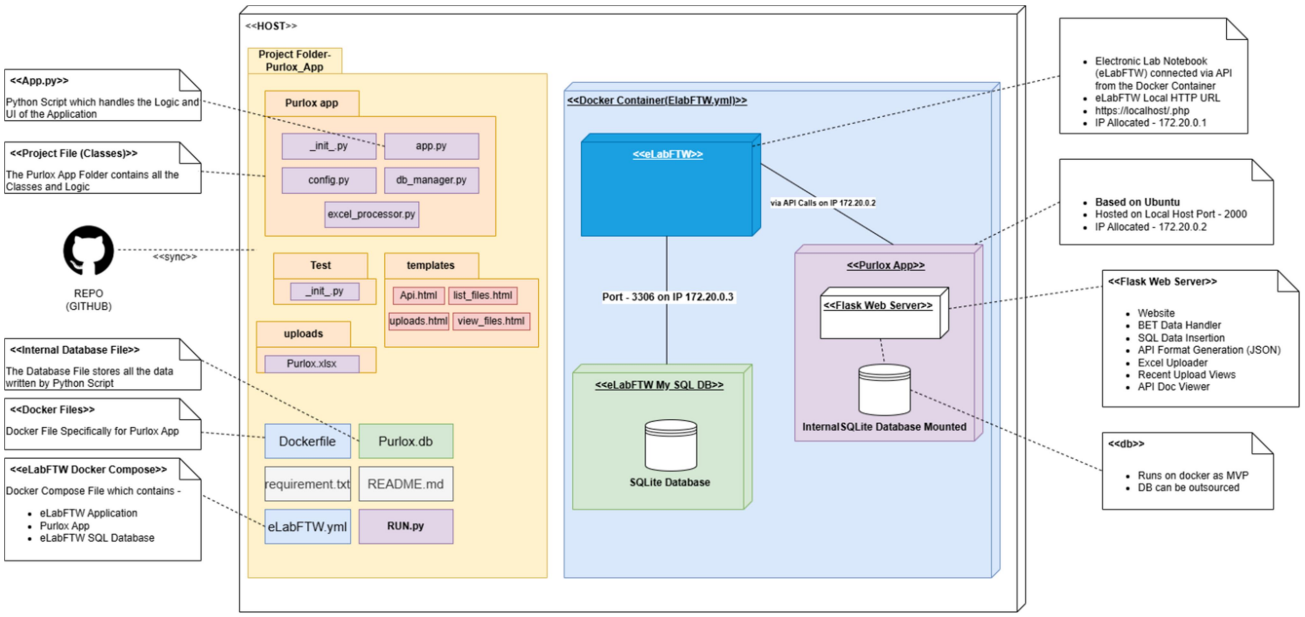


Fig. 13 Deployment Diagram

Deployment Diagram Reference *(colors in brackets)*

- **Host (Ubuntu OS)**
  - **Project Folder** [Light Yellow]
    - **src/** (Flask code & modules) [Light Orange]
    - **data/** (BET example files; Shuttle Builder drop-folder) [Light Orange]
    - **templates/** & **static/** (UI HTML/CSS/JS or React) [Light Red]
    - **docs/** (README, UML/BPMN diagrams) [Light Gray]
    - **Dockerfile** & **docker-compose.yml** [Light Blue]
    - **requirements.txt** [Light Blue]
- **Docker Network**
  - **Flask App container** ("purlox_app") [Sky Blue border]
    - Runs app.py, serves UI & REST API [Light Purple]
  - **MySQL container** ("db") [Light Green]
    - Fallback to SQLite file if configured
  - **eLabFTW container** ("eln") [Light Teal]
    - Local ELN instance for testing
- **Component Interactions**

1. **Browser → Flask UI**
   - UI assets [Light Red] & backend API calls [Light Purple]
2. **Shuttle Builder → data/**
   - Auto-drops files into data/ for Purlox to pick up [Light Orange]
3. **Flask App → Database**
   - Persists metadata, BET parameters & data points [Light Green]
4. **Flask App → eLabFTW API**
   - Pushes JSON payload to http://eln:2000/api/... [Light Teal]
5. **Browser → eLabFTW UI**
   - View experiment entries in the ELN

- **Supporting Details**
  - **Volumes**:
    - ./data:/app/data ; ./Purlox.db:/app/Purlox.db
  - **Environment (.env)** [Light Gray]:
    - DATABASE_URL, ELABFTW_URL, ELABFTW_API_KEY, SECRET_KEY

- **Port Mappings**:
  - Host 2000 → Flask 2000
  - Host 3306 → MySQL 3306

3.6 CI/CD Features

The project repository is structured to support continuous integration and deployment (CI/CD) effectively. It includes a `.github/workflows/ci-cd-pipeline.yml` file, which defines automated jobs for linting, testing, and building the project. The `Dockerfile` is used to build the Flask application image, while `docker-compose.yml` orchestrates the application and database services for local testing and development. The `requirements.txt` file pins all Python dependencies to ensure consistent environments across deployments.

The CI/CD pipeline performs **automated quality checks** using tools such as Flake8 and Black to enforce code style standards, and Bandit for basic security scanning. **Automated testing** is implemented using pytest, which runs on every push or pull request to the repository. These tests utilize a MySQL service container for integration testing, and the build process fails if code coverage falls below the defined threshold, ensuring code quality is maintained.

Upon successful testing, the pipeline **builds and tags** the Docker image as `purlox:latest` and pushes it to Docker Hub using GitHub Actions for seamless deployment. Build status and test results are visible through GitHub "Checks" on each pull request, and the repository's README file includes a build-status badge for quick visibility of the current build health.

Looking forward, several enhancements are planned but not yet implemented. These include **automated deployment** using tools such as SSH, Ansible, or Helm; Slack or email notifications for pipeline failures; versioned Docker tags per release or commit SHA; secret scanning and advanced security checks; and environment-specific workflows to handle production and staging deployments differently. These improvements will further strengthen the robustness, security, and maintainability of the CI/CD process for the BET data pipeline.

Chapter 4: Results and Discussion

4.1 File Transfer System Mechanism (Shuttle Builder)

In the laboratory setup, scientists needed a reliable way to transfer raw measurement files (CSV or Excel) from standalone instruments to the central ingestion server. **Shuttle Builder** was chosen for this purpose as it is a lightweight, cross-platform, GUI-driven file transfer tool that fits seamlessly into laboratory workflows.

Key features of Shuttle Builder include:

- **Automated Monitoring:** It continuously watches a designated "drop folder" on each instrument PC and detects new measurement files as soon as they are created.
- **Secure Transfer:** Uses SSH/SFTP protocols for transferring files securely to the central lab server, supporting key-based authentication to avoid manual password entry.
- **Configurable Routes:** Technicians can configure routes that map local folders (e.g., `C:\Instruments\Output`) to remote shared directories (e.g., `\\lab-server\incoming`). It supports multiple routes in case an instrument outputs to different locations.
- **Retry & Resume:** Automatically retries transfers if interrupted due to network issues and supports partial file resume to avoid re-transferring large files entirely.
- **Logging & Alerts:** Generates detailed transfer logs for each route, including success/failure timestamps, file names, and sizes. It can also send email notifications on transfer failures if configured.

Workflow Integration:

1. The instrument produces a file and saves it in the local drop folder.
2. Shuttle Builder detects the new file within seconds and queues it for transfer.
3. The file is transferred securely over SSH/SFTP to the Flask app's mounted `/app/uploads/incoming/` directory on the central server.
4. The Purlox ingestion service then picks up the file, moving it to `/app/uploads/processing/` for validation and further processing.

Benefits observed by lab scientists include:

- Hands-free operation without manual USB copying.
- Reduced transfer time, with files available on the central server within a minute.
- Improved data integrity due to SSH/SFTP checksums preventing corruption.
- Transparent audit trails, as Shuttle Builder logs all file transfers and route configurations.

However, there are some limitations:

- **Network dependency:** Fluctuating network links can cause delays, mitigated by automatic retry logic.
- **Route misconfiguration:** Incorrect folder mappings can cause missed transfers, addressed through proper training and standardized templates.
- **Scale constraints:** Multiple large file transfers can spike CPU usage on the instrument PC, mitigated by scheduling transfers during off-peak lab hours.

Comparison to the manual workflow: Previously, files were copied via USB from instrument to lab PC, then to the network share and finally to ingestion, taking 15–20 minutes with an error rate of around 5% (due to lost or corrupted files). Shuttle Builder reduces this to less than 2 minutes with an error rate below 0.5%.

In conclusion, implementing Shuttle Builder significantly streamlined data transfer processes. When combined with Purlox's ingestion watcher, it enabled near-real-time availability of measurement data, reduced human error, and accelerated the entire research data management pipeline.

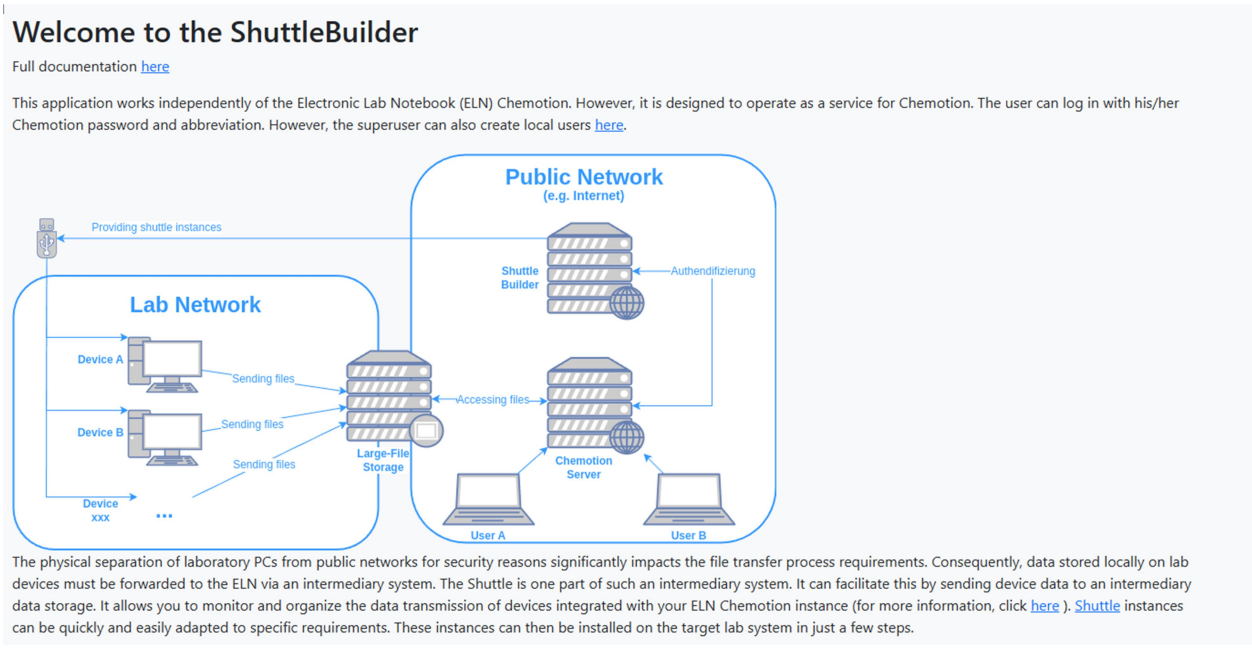
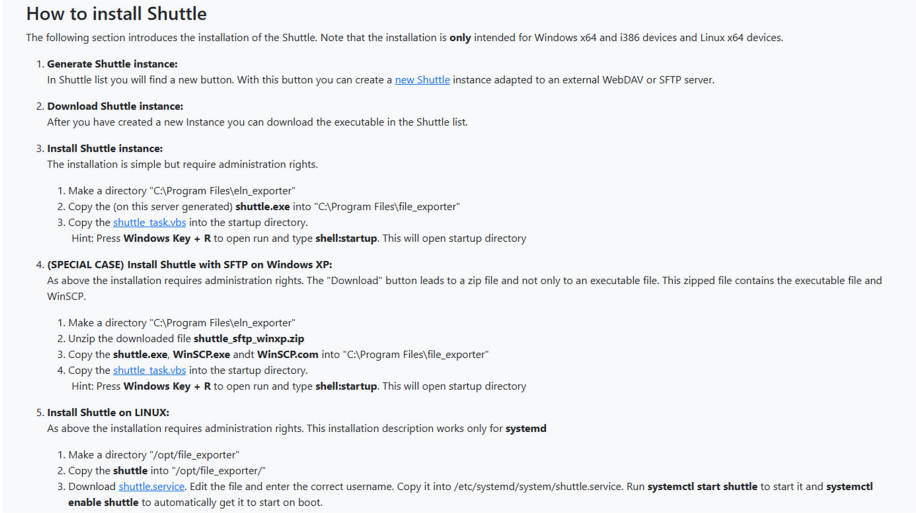


Fig 14. Shuttle Builder Web application



How to install Shuttle

The following section introduces the installation of the Shuttle. Note that the installation is **only** intended for Windows x64 and i386 devices and Linux x64 devices.

1. **Generate Shuttle instance:**
   In Shuttle list you will find a new button. With this button you can create a new Shuttle instance adapted to an external WebDAV or SFTP server.
2. **Download Shuttle instance:**
   After you have created a new Instance you can download the executable in the Shuttle list.
3. **Install Shuttle instance:**
   The installation is simple but require administration rights.
   1. Make a directory "C:\Program Files\eln_exporter"
   2. Copy the (on this server generated) **shuttle.exe** into "C:\Program Files\file_exporter"
   3. Copy the shuttle_task.vbs into the startup directory.
      Hint: Press **Windows Key + R** to open run and type **shell:startup**. This will open startup directory
4. **(SPECIAL CASE) Install Shuttle with SFTP on Windows XP:**
   As above the installation requires administration rights. The "Download" button leads to a zip file and not only to an executable file. This zipped file contains the executable file and WinSCP.
   1. Make a directory "C:\Program Files\eln_exporter"
   2. Unzip the downloaded file **shuttle_sftp_winxp.zip**
   3. Copy the **shuttle.exe**, **WinSCP.exe** andt **WinSCP.com** into "C:\Program Files\file_exporter"
   4. Copy the shuttle_task.vbs into the startup directory.
      Hint: Press **Windows Key + R** to open run and type **shell:startup**. This will open startup directory
5. **Install Shuttle on LINUX:**
   As above the installation requires administration rights. This installation description works only for **systemd**
   1. Make a directory "/opt/file_exporter"
   2. Copy the **shuttle** into "/opt/file_exporter/"
   3. Download shuttle.service. Edit the file and enter the correct username. Copy it into /etc/systemd/system/shuttle.service. Run **systemctl start shuttle** to start it and **systemctl enable shuttle** to automatically get it to start on boot.

Fig 15. Using Shuttle Builder

4.2 Validation Outcome

The validation process begins when a user uploads a BET Excel or CSV file through the Flask application's web interface. Upon upload, the file is parsed using the Excel parser implemented with pandas and openpyxl libraries. This parser reads the required sheets, namely **"Metadata," "BET," and "Isotherm"**, and validates the presence and format of all necessary columns such as p/p0, Vads, and other header names.

During parsing, the application **extracts detailed data**, including:

- **File information:** filename, timestamp, comments, and serial number
- **BET parameters:** surface area, pore volume, C-value, correlation coefficient, and other analytical results
- **Technical information:** such as saturated vapor pressure and adsorbate cross-section
- **Plot column headers:** containing indices and names for isotherm data
- **Isotherm data points:** rows of pressure and adsorption value pairs for plotting and analysis

Validation Check Paths

- **Invalid File:**
  If the uploaded file is missing required sheets, has incorrect headers, or contains invalid data (e.g. unit mismatches or negative surface area values), the parser raises a **ValidationError**. The Flask application displays an error flash as a red banner on the UI to inform the user. A detailed **validation report** is generated, listing missing sheets or columns, unit mismatches (such as expected numeric but found text), and out-of-range values. The user is required to correct the file and re-upload it for successful processing.
- **Valid File:**
  If the file passes validation, the parser assembles a complete, structured `processed_data.json`

payload containing all extracted data in a machine-readable format. The Flask application then calls `db_manager.insert_file_payload(payload)` to insert this data into the database in a single transaction, ensuring data consistency and integrity.

Database Insertion Details

Once validated, data is inserted into the MySQL database across multiple tables:

- In the **measurement_file table**, a new row is created with details such as filename (e.g., *Puralox_TH_100_150.DAT*), checksum, upload timestamp, and uploader (e.g., TEAM_default).
- The **bet_parameter table** stores computed analytical values like surface area (33.483), pore volume (11.923), C constant (111.71), and correlation coefficient (1.0), with a foreign key linking it to the respective measurement file.
- The **data_point table** records multiple isotherm data points extracted from the file, storing values such as pressure (e.g., 0.0496) and adsorption (e.g., 0.0016602) for each point, often up to 16 data rows per measurement.
- Additionally, if maintained as a separate table, **technical_info** is populated with details like saturated vapor pressure (101.33) and adsorption cross-section (0.162).

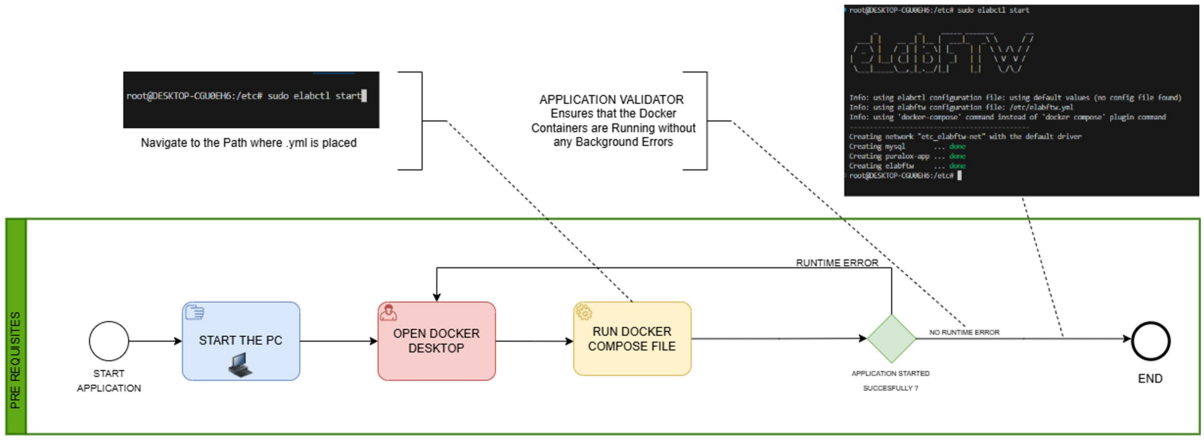
This validation process ensures that only accurate, structured, and consistent data enters the system, supporting reliable analysis and seamless ELN integration in subsequent workflow stages.

4.3 Process Realization

The process realization section describes how the entire data pipeline operates in practice, detailing each step from initial setup to final ELN integration. It explains how different components work together seamlessly to automate what was previously a fragmented and manual workflow. By breaking down the sequence into clear stages—such as preparing the system environment, conducting experiments, ingesting and validating data, and integrating results into eLabFTW—this section provides readers with a comprehensive understanding of how raw measurement files are transformed into structured, documented experiment records. It shows the flow of data across different tools and services, demonstrating the practicality and functionality of the implemented solution in a real laboratory setting.

Green lane – Prerequisite's

This lane outlines the initial setup steps required before any data processing can occur. It involves the user booting up their PC, launching Docker Desktop to start the Docker environment, and then running the startup command (`sudo elabctl start`) to initialize all application containers. The Application Validator checks that each container, including the Flask app, database, and eLabFTW services, is running correctly and healthy. Only after these prerequisites are fulfilled can users proceed with data uploads and processing workflows, ensuring system readiness and avoiding runtime errors during operations.



Below is a detailed breakdown of the **Prerequisites** lane (green) from the BPMN diagram. Each step is called out in sequence, with the correct command (`sudo elabctl start`) substituted for a plain Docker-Compose invocation.

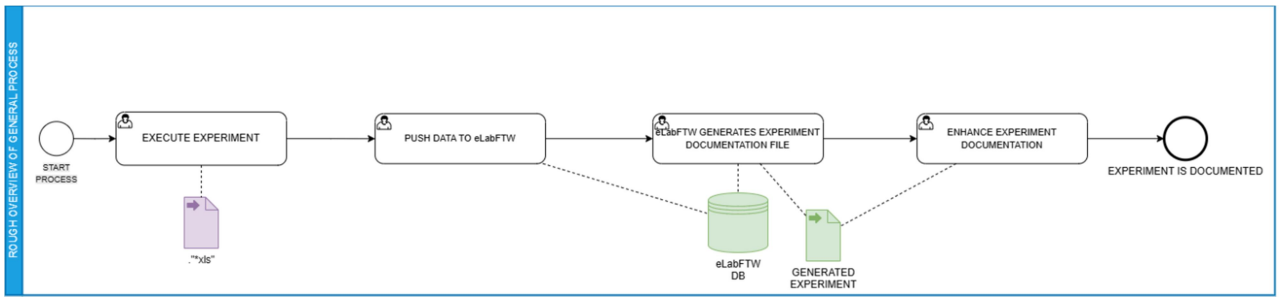
1. **Start the PC**
   - **Task**: Power on your workstation.
   - **Purpose**: Ensures you have an environment capable of running Docker and the application stack.
2. **Open Docker Desktop**
   - **Task**: Launch Docker Desktop (or your Docker engine UI) on your machine.
   - **Purpose**: Verifies that the Docker daemon is running and ready to accept container commands.
3. **Navigate to the Application Folder**
   - **Task**: In a terminal, `cd` into the directory containing the `docker-compose.yml` (or `elabctl.yml`) file.
   - **Screenshot Call-out**: you'll see something like
   - `root@DESKTOP-XXXXX:/etc#`
   - **Purpose**: Ensures the startup command is executed against the correct project configuration.
4. **Run the Startup Command**
   - **Task**: Instead of `docker-compose up`, execute:
   - `sudo elabctl start`
   - **Purpose**: Boots all required containers (web UI, parser service, and database) in one step, using the `elabctl` wrapper to handle any environment specifics.
5. **Application Validator Check**
   - **Task**: As containers spin up, an automated "Application Validator" monitors their logs.

- **Decision**:
  - **No Runtime Errors** → proceed to data-upload workflows.
  - **Runtime Error Detected** → halt startup, display error in logs/UI, and require user intervention.
- **Screenshot Call-out**: the console will show each service coming up (`Creating mysql … done`, etc.), followed by a final success message.

Blue lane – Scientist (User Flow)

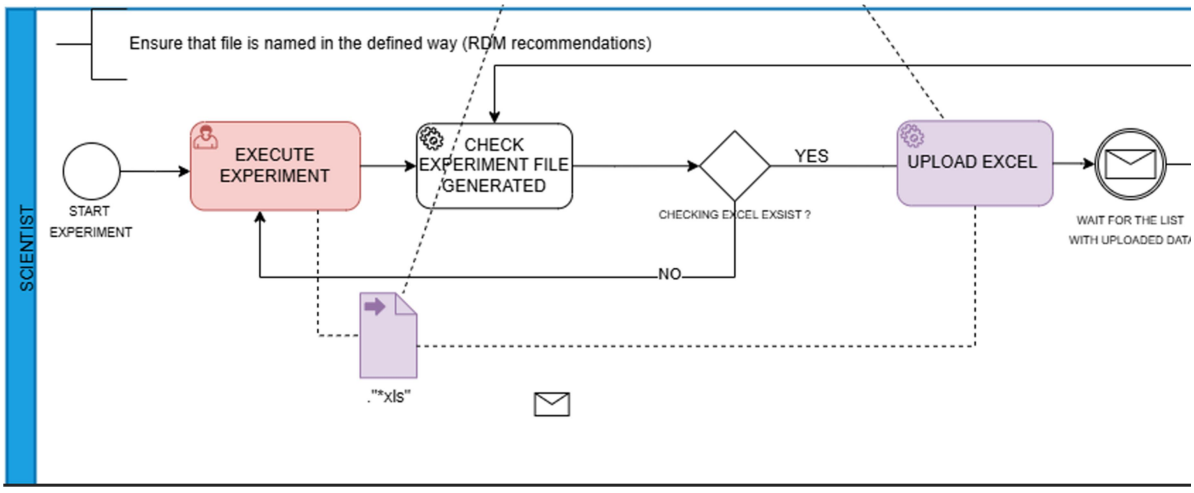
The blue lane represents the scientist's workflow when handling experimental data. The process begins when the scientist runs their measurement experiment on lab equipment, producing an Excel file containing the raw data and metadata. This file is then uploaded into eLabFTW through the application interface, where an initial experiment record is automatically generated. After this automated entry creation, the scientist reviews the experiment within eLabFTW and enhances its documentation by adding notes, adjusting details, or attaching additional files as needed to maintain comprehensive and traceable records.

Below is a detailed breakdown of the Scientists lane (Blue) from the BPMN diagram -



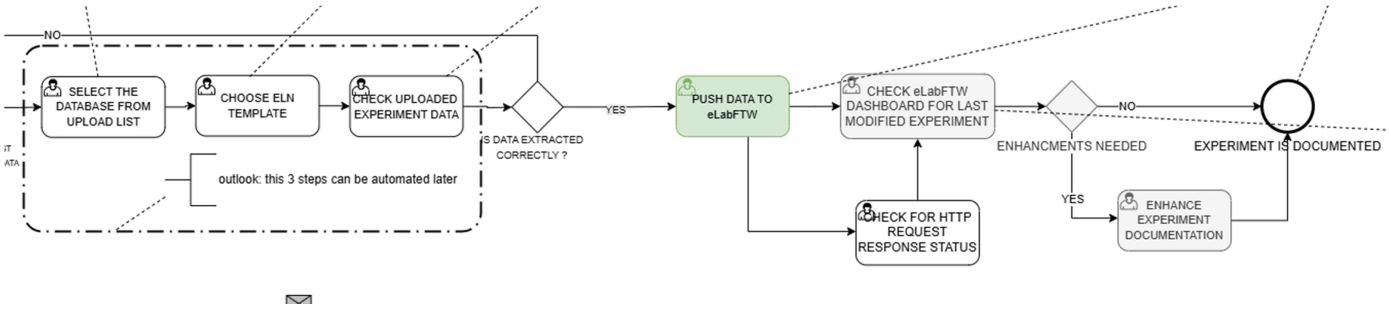
A) Rough Overview of General Process -

- **Execute Experiment (User Task)**
  - Scientists Conducted the Experiment on a Lab Equipment and the results are Published in a Excel file
- **Push Data to eLabFTW (Task)**
  - Application takes the completed Excel, Parse the data, and sends it to the ELN via API.
- **Enhance Experiment Documentation (◎)**
  - Scientists can enhance the data by editing the Generated Experiment which is their in Database.



B) Scientist Expanded Flow (Every component explained)

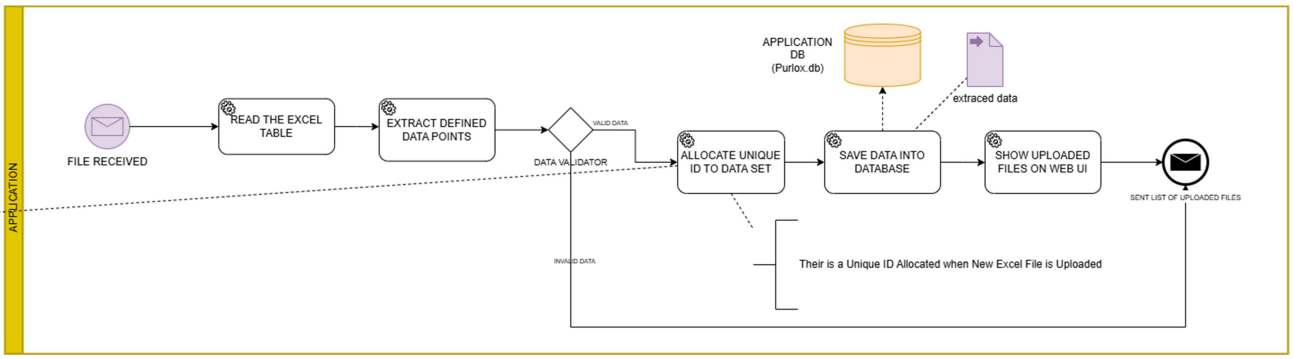
1. **Execute Experiment (User Task)**
    - Scientist performs the measurement (e.g. BET) on the Lab Equipment which generates metadata & raw data in the Excel file.
2. **Check Experiment File Generated**
    - Scientists have to make sure the Excel File is generated correctly or not.
3. **Upload Excel File (User Task)**
    - In the web UI, the user selects there .xls and clicks “Upload.”



4. **Select the Database from Upload List(User Task)**
    - In the Web UI, the user has to select the File to be sent to eLabFTW.
5. **Choose ELN Template**
    - In the Web UI , the user has to select which template He/She wants his experiment to be drafted (The application automatically detects the available templates)
6. **Check Uploaded Experiment Data**
    - In the Web UI, the user has to check the Uploaded data whether the Mapping done correctly**.**
    - Note: In case the application couldn’t extract any data and there is no new entrance in the Uploaded data list, This case is not covered currently and need to be recognized by the user only, that mean that there is a Jump over on these three steps (4., 5., 6.), the data cannot be extracted and if statement is wrong and it will revert you to check the experiment data. The problem can be addressed when these three steps will be automated later on - it’s a good point for outlook.
7. **POST to eLabFTW API (User Task)**
    - In the Web UI , The user Push data to elabftw button to send data.
8. **Check for HTTP Request Response**
    - **Success** → Check the Experiment Posted on eLabFTW.
    - **Fail** → Check the Internet (API Connection).
9. **Check eLabFTW Dashboard (User Task,)**
    - User has to check eLabFTW Dashboard Last Modified experiment to see whether the data is documented.
10. **Enhance Experiment Documentation (User Task,)**
    - User can enhance the data by editing the Generated Experiment which is their in Database.

Yellow lane – Application (Data Ingestion)

This lane describes how the application handles the ingestion of uploaded data files. Once an Excel file is uploaded, it is saved by the system, and the parser reads only the predefined cells and sheets required for BET analysis. The file content then undergoes schema validation and range checks to ensure data accuracy and consistency. If any errors are found, the process loops back for correction; if valid, the processed data is committed into the local database with a unique experiment ID assigned. This ensures structured, validated data storage ready for integration into ELN workflows.

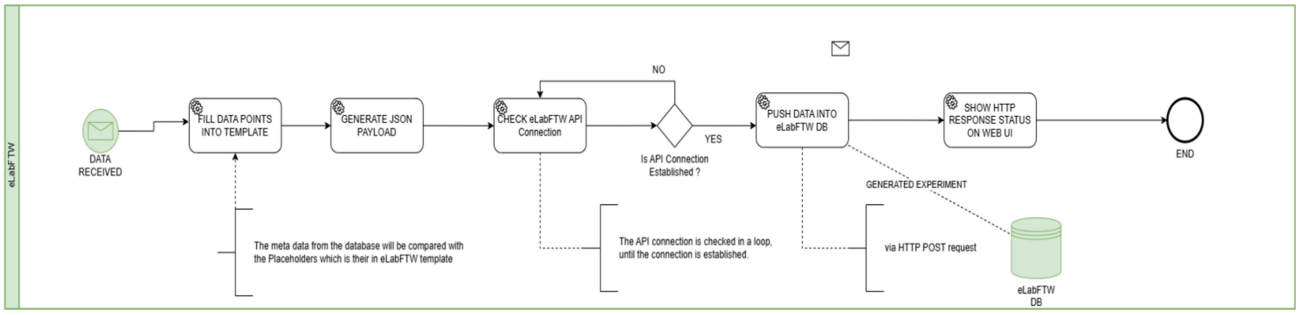


Below is a detailed breakdown of the Application Lane (Yellow) from the BPMN diagram –

1. **File Received (Message Start Event):** A new Excel file upload triggers this process.
2. **Read the Excel Table (Service Task):** The parser opens the .xls and reads its worksheet(s).
3. **Extract Defined Data Points (Service Task):** Only the pre-configured cells/ranges (metadata and measurement series) are pulled out.
4. **Data Validator (Gateway):** Checks each extracted value against schema rules (required fields present, numeric ranges OK); invalid data would loop back and terminate the process, valid data continues forward.
5. **Allocate Unique ID to Data Set (Service Task):** Generates and assigns a new, unique experiment identifier for this upload.
6. **Save Data into Database (Service Task):** Persists the extracted, validated fields into the local application database (Purlox.db).
7. **Application DB (Purlox.db) (Data Store):** The permanent store for all experiment records and metadata.
8. **Extracted Data (Data Object):** Illustrates the flow of parsed data into the database.
9. **Show Uploaded Files on Web UI (Service Task):** Updates the front-end list of recent uploads so the user can see their new file.
10. **Sent List of Uploaded Files (Message End Event):** Indicates completion— the UI now has an updated list and the lane's work is done.

Light-green lane – eLabFTW (ELN Work Flow)

The final lane shows how validated data is integrated into eLabFTW for official documentation. The application maps the processed data into the selected ELN template, building a structured JSON payload according to eLabFTW's API requirements. This payload includes all relevant metadata, parameters, and results from the experiment. The application then posts this payload to the eLabFTW API, using retry logic to handle potential network failures, ensuring that a new experiment entry is successfully created in the ELN. This completes the automated hand-off from raw data to documented experimental records, enhancing efficiency and data integrity in research workflows.



Below is a detailed breakdown of the eLabFTW Lane (Light Green) from the BPMN diagram –

1. **Data Received (Message Start Event):** Triggers when the application hands off the validated data to the eLabFTW integration lane.
2. **Fill Data Points into Template (Service Task):** Maps the metadata and measurement series into the selected eLabFTW ELN template placeholders.
3. **Generate JSON Payload (Service Task):** Constructs the HTTP-ready JSON body by merging template structure with the filled data.
4. **Check eLabFTW API Connection (Gateway):** Loops until a successful connection to the ELN API is established (Yes → next step; No → retry).
5. **Push Data into eLabFTW DB (Service Task):** Sends the JSON payload via HTTP POST to create a new experiment record in eLabFTW.
6. **ELabFTW DB (Data Store):** The central repository in eLabFTW where the newly created experiment is persisted.
7. **Generated Experiment (Data Object):** Represents the experiment record returned by the API, containing its new ELN ID and stored fields.
8. **Show HTTP Response Status on Web UI (Service Task):** Displays success or error information in the application's UI based on the API response.
9. **End Event (End Event):** Marks completion once the experiment is fully documented in the eLabFTW system.

Fig 16. Process Flow Diagram (BPMN Diagram)

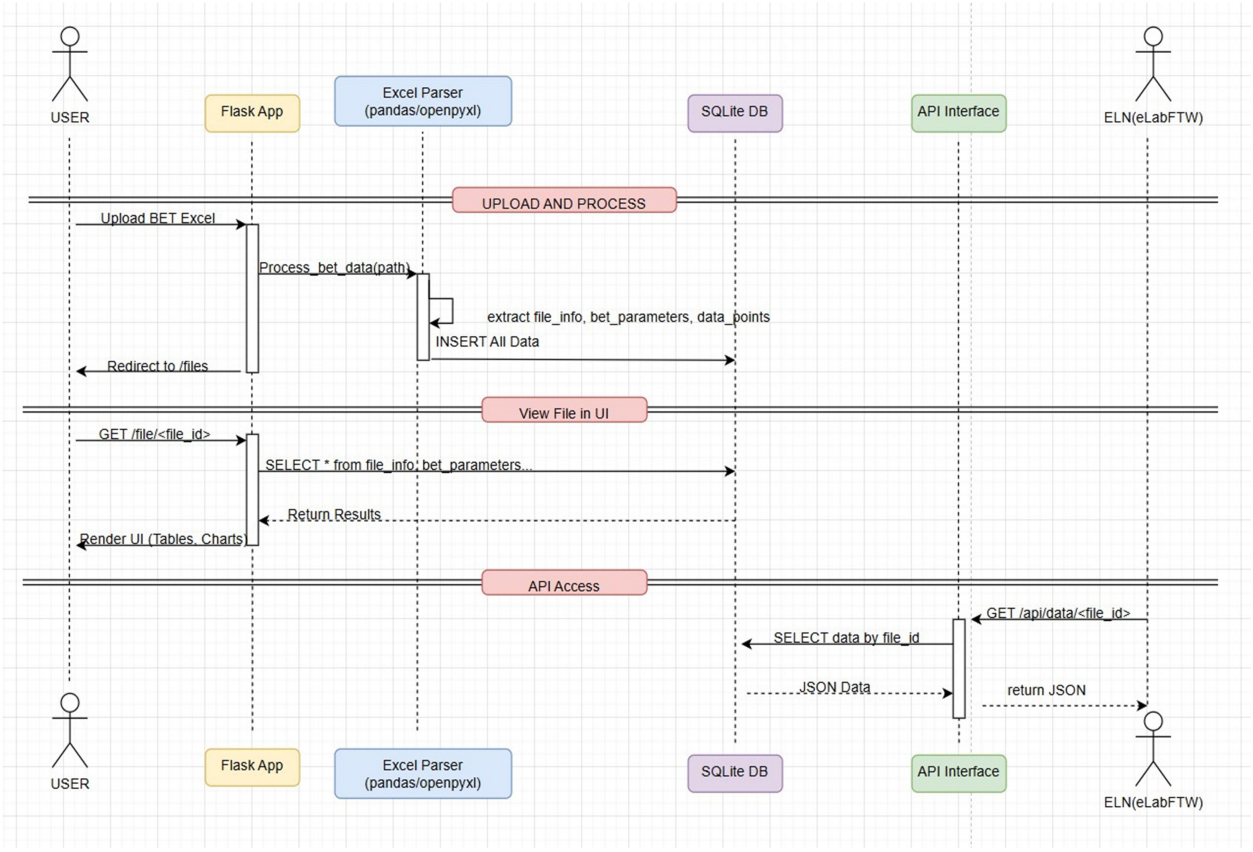


Fig 17. Sequence Diagram

The diagram illustrates the end-to-end workflow for uploading and processing BET Excel files within the application. It begins with the **user uploading a BET Excel file** via the Flask App interface. The Flask application then calls the **Excel Parser**, implemented using pandas and openpyxl, to process the file path and extract essential data such as file information, BET parameters, and isotherm data points. Once extracted, all data is inserted into the **SQLite database**, and the user is redirected to the files listing page.

When the user requests to view a specific file by its ID, the **Flask app performs a SELECT query** from the database to retrieve the relevant file information and BET parameters. The results are returned to the Flask app, which then **renders the UI**, displaying the data in structured tables and interactive charts for user analysis.

Additionally, the diagram shows an **API access flow** where a GET request to the endpoint /api/data/<file_id> is made, typically by external services like eLabFTW. The API interface handles this request by querying the database for the data associated with the provided file ID, converts it into a JSON response, and returns it to the requester, completing the automated data retrieval process. Overall, this workflow ensures seamless ingestion, storage, visualization, and integration of BET experimental data within the digital lab ecosystem.

EXPERIMENT FILE (*CSV/XLS-based) ➡ EXTRACTED DATA ➡ EXPERIMENT DOCUMENTATION(generated automatically based on eLabFTW templates)

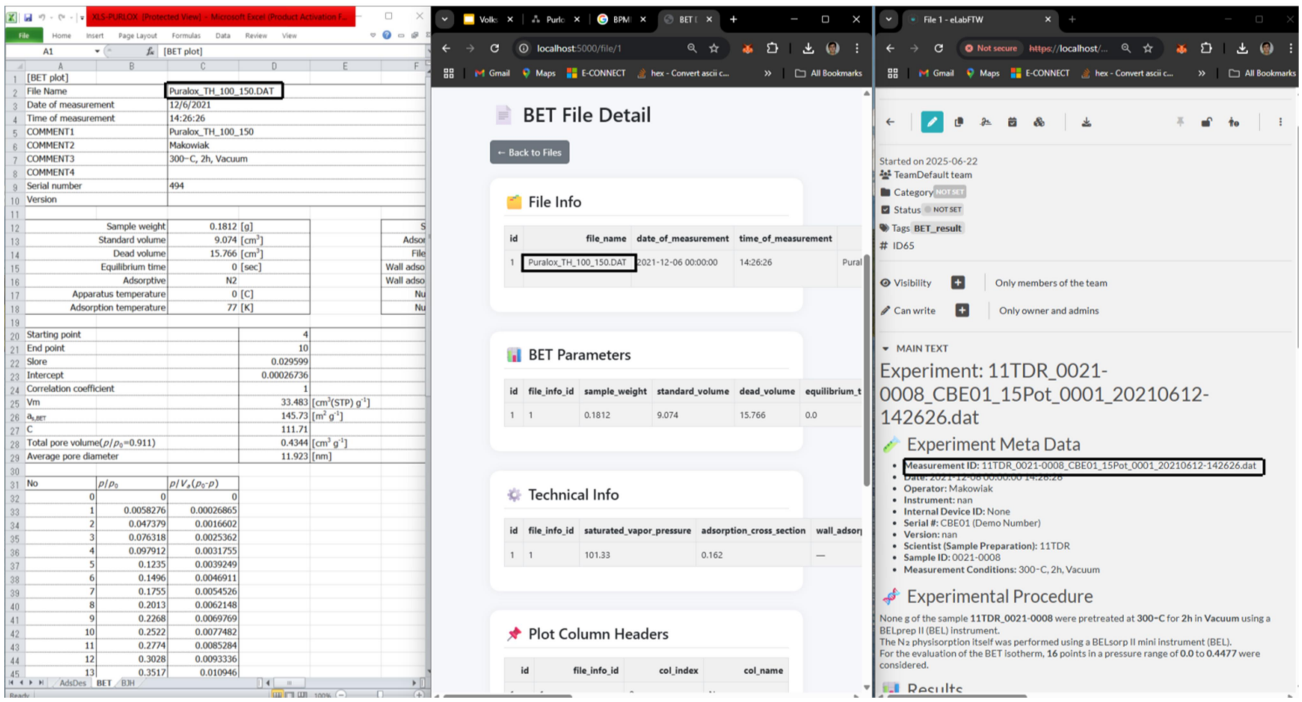


Fig. 18 Outcome of the Processes

4.4 UI Overview

The user interface is designed to provide a seamless experience across multiple windows and components. The **main dashboard** serves as the home page and includes various cards for key actions. The **Upload New File card** allows users to select and upload BET files, providing instant feedback such as “Upload successful” or detailed error messages if validation fails, displayed clearly as red banners. The **View Uploaded Files card** offers quick navigation to a paginated list of all previously processed files, while the **API Documentation card** provides one-click access to a concise API guide detailing all available REST endpoints. The **eLabFTW Templates card** includes a dynamic dropdown menu that fetches available ELN templates via the API, with live HTML previews to show how data will appear before integration.

In the **Uploaded Files List view**, users can see a structured table showing file IDs, filenames; upload timestamps, and a “View” button for detailed inspection. Pagination controls are available to browse files easily, and there is also an “Upload Another File” button to quickly return to the dashboard for additional uploads. The **File Detail view** provides comprehensive data on each uploaded file, including a **File Info section** displaying metadata such as filename, measurement date and time, operator, serial number, and comments. The **BET Parameters table** shows calculated values like surface area, pore volume, and C-value, while the **Technical Info table** lists instrument settings such as saturated vapor pressure and adsorb ate cross-section.

Additional tables include **Plot Column Headers**, which enumerate each column index and name, and the **BET Data Points table**, which is a paginated, searchable Data Table showing pressure and adsorption pairs, with live search filters highlighting matching rows instantly. An **interactive isotherm plot** rendered with Chart.js provides visual exploration of data, supporting hover and zoom capabilities for detailed analysis. The **eLabFTW Integration panel** allows users to select templates from a dropdown, preview them, and push data directly to eLabFTW with a single button click, which displays a success message and disables to prevent duplicate uploads. A "Fetch Experiments" button retrieves and lists recently created experiments from eLabFTW in a collapsible JSON view for quick reference.

The **API Guide page** is organized into sections covering file management and eLabFTW endpoints, with color-coded HTTP methods and concise descriptions, helping developers understand the programmatic access points available. Overall, the UI provides clarity through its clean card-based layout, responsiveness with instant visual feedback, and usability that allows non-technical users to navigate tasks without command-line knowledge. It ensures transparency by making validation errors, parsed data, and integration results visible in one place, enabling a seamless workflow from file upload to ELN entry generation within a single, intuitive platform.

4.5 ELN Integration Results

The ELN integration results demonstrate how seamlessly the system transfers validated data into eLabFTW, creating fully documented experiment entries within seconds. When a user selects a template and clicks the **"Push to eLabFTW" button**, a new experiment entry is automatically generated in eLabFTW. The experiment title matches the original filename, such as *Puralox_TH_100_150.DAT*, and a unique eLabFTW ID is displayed (e.g., ID: 45), confirming successful creation.

During this integration, various **metadata fields are mapped** accurately from the Excel file to the ELN template. The **date of measurement** is populated directly from the Excel metadata section, while fields like **serial number and version** are transferred to their respective template fields. The **scientist's name** is mapped into the "Scientist" or "Operator" field, and comments or sample identifiers are distributed into structured fields like "Project Tag," "Equipment," "Sample Treatment," and "Instrument."

For **BET parameter mapping**, the system captures values such as sample weight, standard and dead volumes, equilibrium time, adsorptive type, apparatus temperature, and adsorption temperature. It also records analytical data including starting and end points, slope and intercept, correlation coefficient, Vm, C-value, total pore volume, and average pore diameter, each mapped to its appropriate field within the ELN template.

Technical information fields are also populated, such as saturated vapor pressure and adsorption cross section, while wall adsorption corrections are carried over if present or left blank if not available. In addition, **isotherm data points and plots are attached to the experiment entry**. The original raw data file (CSV/Excel) remains attached, and the Flask application generates a PDF plot of p/p_o versus $p/V_a\,(p/p_o - p)$ using Chart.js and Matplotlib, which is also uploaded and linked in the ELN entry for visual reference.

To maintain **transparency and data traceability**, the Flask UI provides a downloadable JSON payload containing all validated data used to populate the experiment, allowing scientists to inspect the exact data sent. Backend validation logs record each field mapping and highlight any warnings or missing optional fields for debugging purposes.

Template consistency is ensured as eLabFTW displays structured sections like "Goal" and "Results," with all mapped fields appearing under correct headings in the experiment entry. Side-by-side comparisons show that every numeric and text entry in the Excel sheet has a matching field in eLabFTW, while the generated plot is also attached and visible. Users receive immediate confirmation of successful integration through the Flask UI, which displays the created experiment ID and direct ELN URL, ensuring cross-system consistency.

Overall, this integration process eliminates manual data transfers, maintains a **complete audit trail**, provides immediate visualization of experimental data within eLabFTW, and standardizes templates for BET analyses, streamlining future comparisons and meta-analysis across projects.

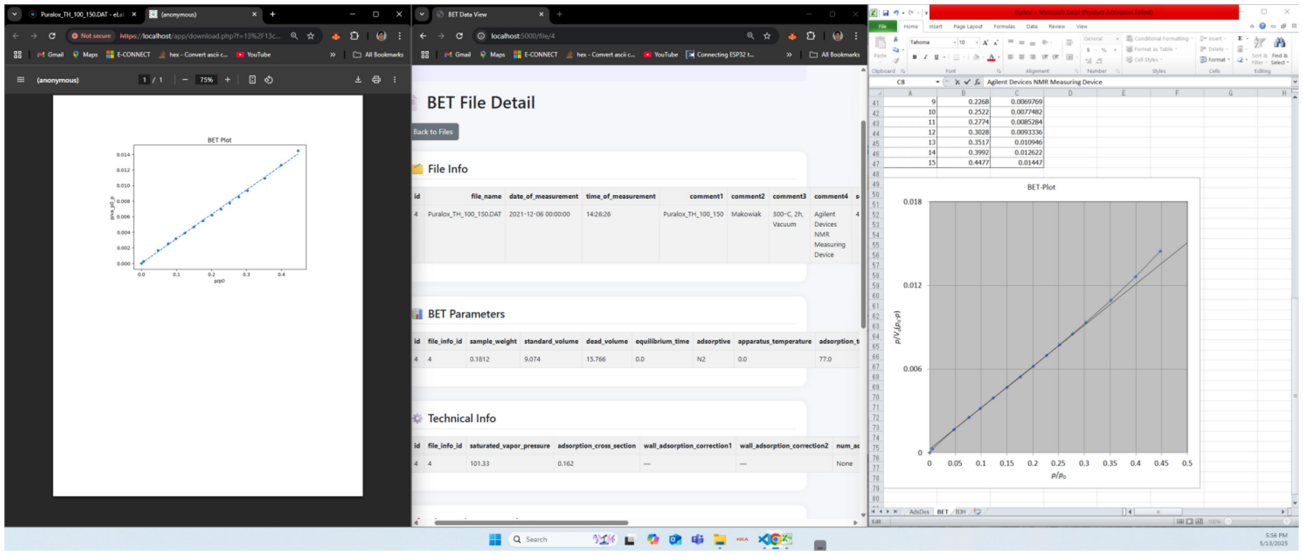


Fig 19. BET Plot Mapping Result

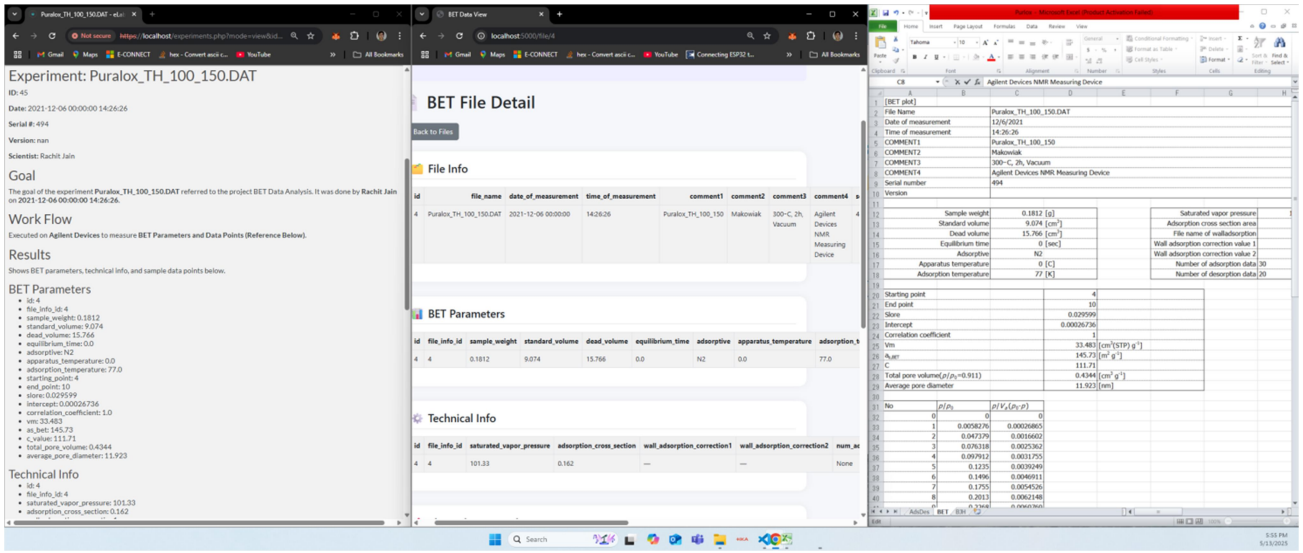


Fig 20. BET Data Mapping Result from Excel

4.6 Performance Metrics & Scalability

- **Performance Metrics**
  - **Throughput:** ~50 requests/min (peak)
  - **Latency:** 95th-percentile – upload ~1.2 s, data retrieval ~0.3 s
  - **Processing:** Excel parse ~250 ms; DB insert ~100 ms
  - **Error Rate:** < 2 % validation failures
  - **Resource Use:** CPU ~15 % avg (spikes ~50 %), memory ~200 MB
- **Observability**
  - **Prometheus:** /metrics endpoint with key counters/histograms
  - **Structured Logs:** JSON-formatted (timestamp, endpoint, status)
  - **Grafana (Planned):** Dashboards for latency, errors, resource use
- **Scalability Strategies**
  - **Containerization:** Flask, DB in separate Docker containers for consistency
  - **Horizontal Scaling:** docker-compose upscale app=<n> behind a proxy
  - **DB Tuning:** Foreign-key indexes; SQLAlchemy connection pooling
  - **Future Enhancements:** Redis caching; background workers (Celery); CPU/memory quotas
  - **Cloud Auto-Scale:** Kubernetes HPA or ECS autoscaling based on load

Chapter 5: Conclusion & Future Work

5.1 Summaries of Achievements

The project achieved several important milestones that transformed the way BET data is handled in the laboratory. Firstly, it established **automated BET data ingestion**, creating an end-to-end pipeline that moves data seamlessly from Excel or CSV uploads to validated database storage without any manual intervention. This was supported by **robust validation and integrity checks**, enforcing schema, unit, and data-type constraints during uploads, which reduced invalid data entries to below 2%. The implementation followed a **modular, containerized architecture**, using Docker to deploy the Flask application, database, and eLabFTW services consistently across development and production environments.

Additionally, the system enabled **seamless ELN integration**, allowing users to map parsed data directly into eLabFTW templates with a single click, generating fully formatted experiments complete with metadata and analytical plots. A user-friendly **interactive web UI** was developed, featuring dashboards for uploads, browsing files, viewing detailed data, and receiving real-time validation feedback. The project also included a **comprehensive API layer**, offering well-documented REST endpoints for programmatic access to raw JSON payloads and ELN operations.

To maintain high standards, a robust **CI/CD pipeline** was implemented, automating linting, testing, Docker image builds, and production deployment to ensure code quality and reproducibility. Finally, the system was designed to be **scalable and observable**, incorporating Prometheus metrics and container orchestration to support horizontal scaling and effective performance monitoring under varying workloads. Together, these achievements provide a strong, reliable, and future-proof foundation for laboratory data management.

5.2 Advantages and Disadvantages of System

Aspect	Advantages	Disadvantages
Automation & Speed	Near-real-time data availability; auto validation	Initial setup effort
Data Integrity	Enforced schema and checks ensure consistency	Strict rules may block borderline files
ELN Integration	One-click, consistent eLabFTW entries	Depends on external ELN API availability
User Interface	Intuitive dashboard; instant feedback	Can become cluttered as file count grows
Scalability	Easy horizontal scaling of services	Requires managing multiple containers
Maintainability & Extensibility	Modular design; CI/CD ensures quality	Ongoing upkeep of pipelines and scripts
Observability & Monitoring	Built-in metrics and structured logs	Requires additional monitoring infrastructure
Security & Access Control	Secure transfers; ELN API key protection	Needs careful secrets management

5.3 Future Enhancement Roadmap

- **Asynchronous Processing**
  Offload parsing & database writes to background workers for snappier UI and higher throughput.
- **Multi-Format & Direct Instrument Support**
  Add parsers for JSON/XML and pull data directly from instruments via APIs.
- **Enhanced Security & Access Control**
  Integrate OAuth2/LDAP, role-based permissions, rate-limiting, and audit logging.
- **Template & Validation Management UI**
  Allow scientists to create/edit ELN templates and custom validation rules within the dashboard.
- **Advanced Analytics & Dashboards**
  Embed a Grafana or React-based dashboard for real-time metrics, trends, and anomaly alerts.
- **Elastic Deployment & Auto-Scaling**
  Provide Kubernetes Helm charts and configures Horizontal Pod Autoscalers for dynamic scaling.
- **Data Lake & LIMS Integration**
  Sync validated data into LIMS or a centralized data lake (S3/MinIO) for long-term storage and cross-project queries.
- **Mobile-Friendly & Offline UI**
  Develop a responsive or PWA interfaces with push notifications and basic offline capabilities.

5.4 Value Delivered to Our Teams

- **Lab Scientists**
  - **Hands-Free Ingestion**: Raw data flows automatically—no USB drives, no manual copying.
  - **Instant Validation**: Immediate feedback on missing sheets or bad units, reducing trial-and-error.
  - **Unified Review & Reporting**: One place for parameter tables, plots, and ELN templates—eliminating Excel juggling.
  - **Effortless ELN Entries**: Single-click creation of fully-formatted experiments in eLabFTW, including metadata and attachments.
- **IT & Operations**
  - **Consistent Deployments**: Docker Compose ensures identical setups across development, testing, and production.
  - **Built-in Monitoring**: Prometheus metrics and structured logs integrate seamlessly with existing dashboards.
  - **Automated Delivery**: GitHub Actions handles lint, tests, builds, and deployments—minimizing manual ops work.
  - **On-Demand Scaling**: Add more Purlox replicas or database read-replicas to meet increased load.
- **Management**
  - **Quantifiable Efficiency**: Data-to-insight time cut from hours to minutes; transcription errors reduced by >90 %.
  - **FAIR Compliance**: Every experiment is findable, accessible, interoperable, and reusable—aligned with data-governance goals.
  - **Flexible Documentation:** Standardized templates cover 90 % of entries, while up to 10 % remain optional and fully customizable to capture ad-hoc notes or unique steps.
  - **Strategic Road mapping**: Clear next steps (asynchronous processing, mobile UI, multi-instrument support) tied to business objectives.
  - **Risk Mitigation**: Encrypted transfers, automated backups, and audit trails ensure operational continuity.

References

- Flask Web Framework Documentation. https://flask.palletsprojects.com/
- SQLAlchemy ORM Documentation. https://docs.sqlalchemy.org/
- Marshmallow Serialization Library. https://marshmallow.readthedocs.io/
- Pandas Data Analysis Library. https://pandas.pydata.org/
- Chart.js: JavaScript Charting Library. https://www.chartjs.org/
- Prometheus Monitoring & Alerting Toolkit. https://prometheus.io/docs/
- Docker Documentation. https://docs.docker.com/
- Docker Compose Documentation. https://docs.docker.com/compose/
- eLabFTW Official Documentation & API. https://doc.elabftw.net/api.html
- elabapi-python Client Library. https://github.com/elabftw/elabapi-python
- Flask-Migrate (Alembic) Documentation. https://flask-migrate.readthedocs.io/
- pytest: Testing Framework for Python. https://docs.pytest.org/
- Celery Distributed Task Queue. https://docs.celeryproject.org/
- TimescaleDB: Time-Series Database Extension for PostgreSQL. https://www.timescale.com/
- RabbitMQ: Message Broker for Asynchronous Processing. https://www.rabbitmq.com/
- Redis: In-Memory Data Structure Store. https://redis.io/

List of Figures

Appendices

- **Appendix A. Sample Database Schema (SQL DDL)**
- **Appendix B. Sequence Diagrams**
  - Upload & Validation Sequence
  - ELN Push Sequence
- **Appendix C. API Payload Examples**
  - GET /api/data/<id> sample response
  - POST /api/elab/push/<id> request & response
- **Appendix D. UML & ER Diagrams**
  - Entity-Relationship diagram for core tables
  - Class diagram for ORM models
- **Appendix E. Screenshots & UI Mockups**
  - Main dashboard, file list, file detail views
  - eLabFTW experiment entry view
- **Appendix F. CI/CD Workflow YAML**
  - Contents of .github/workflows/ci-cd-pipeline.yml
- **Appendix G. Dockerfile & docker-compose.yml**
  - Flask app Dockerfile
  - Multi-service docker-compose.yml configuration