Student Performance Project — Detailed Technical Report

Report date: *30 October 2025*

Table of contents

---

1. Executive summary

This project implements a full data pipeline and analytics product for monitoring and predicting student performance. The system ingests multiple CSV sources, persists them in

PostgreSQL, provides interactive functionality and programmatic analysis via a Streamlit-based interface (the original REST API was replaced by Streamlit), runs predictive models to flag students at risk, and presents interactive charts in a dashboard application. The delivered artifacts include raw & final datasets, ETL scripts, the database creation SQL, ML model code & documentation, dashboard code, and supporting visuals.

---

2. Project scope & objectives

- Centralize student data (demographics, attendance, study hours, sports participation, grades).
- Provide interactive analytics for administrators and teachers.
- Build a predictive model to identify students at risk.
- Replace the original REST API with a Streamlit application that serves both interactive UI and programmatic analysis endpoints (suitable for internal dashboards).
- Deliver production-ready artifacts and recommendations for deployment & governance.

---

3. End-to-end architecture (high-level)

1. **Data sources**: DATA.csv, Final_df.csv, other CSVs, screenshots/figures for documentation.
2. **ETL**: import_csv_to_postgres.py — cleans and loads CSVs into PostgreSQL.
3. **Storage**: PostgreSQL database created with Creation_Queries.sql and modeled in ER diagram.png.
4. **Application / API**: Streamlit app (replacing prior student_api.py) that queries DB, exposes interactive controls, displays responses and saved reports.
5. **Dashboard**: student_dashboard_app.py (may be combined with Streamlit app or kept as a separate visualization app) shows charts and KPIs. app_pro.py is the production integrator.
6. **ML**: student_performance_model (1).py trains model(s); trained model artifacts persisted (joblib/pickle).
7. **Monitoring & Delivery**: Docker + CI/CD recommended; scheduled ETL; logging & alerting.

---

4. Detailed step-by-step product flow

4.1 Data sources and schema

Files & typical columns (as found in repository):

- DATA.csv — raw records. Columns typically include: student_id, student_name, age, gender, grade_level, study_hours, attendance_rate, sports_participation (bool or category), final_score, date (if time series).
- Final_df.csv — merged & cleaned dataset used for analytics and ML. Contains engineered columns (e.g., attendance_flag, study_bin, performance_label).
- label_mappings.json — mapping for categorical encodings (e.g., {"Low":0,"Medium":1,"High":2}).

4.2 Data cleaning & preprocessing (ETL)

Primary steps performed (and to verify in code):

1. **Load CSVs** using pandas.read_csv() with explicit dtype for critical columns (IDs, dates).
2. **Trim & normalize column names** (lowercase, underscores).
3. **Handle duplicates**: df.drop_duplicates(subset=['student_id','date']).
4. **Missing values**: Strategy per column:
    - For numeric columns (study_hours, attendance_rate): fill with median or model-based imputation.
    - For categorical columns: fill with "Unknown" or mode, or maintain NaN for special handling.
5. **Type conversions & parsing**: parse date strings with pd.to_datetime().
6. **Feature engineering**:
    - Binning study hours: study_bin = cut(study_hours, bins=[0,1,3,6,inf], labels=[...]).
    - Attendance flag: attendance_rate < 0.75 => at_risk_attendance.
    - Sports participation: binary 0/1 or categories (none / occasional / regular).
7. **Label creation** (if classification): performance_label from final_score thresholds.

ETL best-practices to add:

- Add a small data_validation step (use pandera or great_expectations) to assert schema and ranges.
- Add logging & metrics for rows processed, rows rejected, and parsing errors.

4.3 Database design & creation (PostgreSQL)

From Creation_Queries.sql and ER diagram.png:

- Typical tables:
    - students(student_id PK, name, dob, gender, grade_level, ...)
    - attendance(attendance_id PK, student_id FK, date, present boolean, attendance_rate)
    - activity(sports_id PK, student_id FK, sport_type, participation_level)
    - scores(score_id PK, student_id FK, exam_id, score, performance_label)
    - meta_tables for lookups (grades, classes).
- Indexes: CREATE INDEX idx_student_id ON attendance(student_id); CREATE INDEX idx_date ON attendance(date);
- Foreign keys enforce referential integrity.

4.4 Data ingestion pipeline (import_csv_to_postgres.py)

Flow implemented:

1. Read the CSV into pandas.
2. Run cleaning steps (rename cols, type conversion).
3. Use psycopg2 or sqlalchemy to perform inserts:
    - For performance: COPY is preferred for bulk ingestion from CSV.
    - If incremental loads: use upsert (INSERT ... ON CONFLICT (pk) DO UPDATE ...) to avoid duplicates.
4. Transactions wrap batch inserts to ensure atomicity.
5. Optionally log rows inserted and time taken.

Operational improvements:

- Implement idempotent loads (using a hash of file or ingestion_id) so reruns don't duplicate.
- Add CLI flags --truncate, --batch-size, --dry-run.

4.5 Streamlit-based interface

You indicated: **you changed something and made the API a Streamlit app**. Below I document how that is handled, design trade-offs, and a practical pattern to follow.

4.5.1 Two possible interpretations & the one we use

- **Interpretation A:** Streamlit app which directly queries the DB and exposes interactive pages. This means programmatic REST calls from other services are not available unless specifically added.
- **Interpretation B (hybrid):** Streamlit serves the UI and internally calls the database, while a lightweight API is kept for programmatic access.

Choice & recommendation: Your change implies Interpretation A. That's fine for internal dashboards and human-driven workflows but if external systems (e.g., LMS, automated schedulers) must access the same data, add a small REST layer or expose endpoints from the Streamlit app via an embedded mount. See appendix for hybrid example.

4.5.2 Streamlit design & patterns to implement (detailed)

- **Single-entry Streamlit app** (e.g., dashboard_streamlit.py) that:
    - Connects to PostgreSQL using sqlalchemy (use connection pooling).
    - Caches query results with @st.cache_data(ttl=600) to avoid DB hits for repeated requests.
    - Provides UI controls (filters): class, grade, date range, study_hours bins, attendance thresholds.
    - Displays KPIs, charts, tables, and model predictions.
    - Supports uploading new CSVs for ingestion (secured).
- **Programmatic functions** (non-UI) inside the same codebase for reuse:
    - def get_student(student_id): — queries DB and returns dict.
    - def get_kpi_summary(params): — returns aggregated metrics.
    - def predict_student_risk(student_id_or_df): — loads persisted model and returns risk score(s).

- **Security**:
    - Use Streamlit sharing behind VPN or internal network, or behind an auth reverse-proxy (OAuth or SSO).
    - Add an internal auth layer in Streamlit (e.g., st.session_state['user']) with LDAP / SAML / OIDC integration or a simple token check for small deployments.
- **Performance**:
    - Use @st.cache_data for DB query results and @st.cache_resource for heavy resources (DB engine, ML model).
    - For heavier loads or multiple concurrent users, consider replacing interactive endpoints with an API + frontend separation (Streamlit is not designed for high-concurrency programmatic consumption).

4.5.3 Example Streamlit snippet

Highlights:

- Connect to DB with SQLAlchemy (connection pooled).
- Provide filter UI.
- Display KPIs and charts (Plotly or Matplotlib integrated).
- Load model artifact with joblib.load() and expose predict().

(Full snippet in Appendix B; copy/paste into streamlit_app.py and adapt paths/credentials.)

4.5.4 Implications of using Streamlit

Pros

- Faster development for interactive dashboards.
- One codebase for UI + backend logic — easier to maintain for small teams.
- Built-in UI controls accelerate prototyping.

Cons

- Not a drop-in replacement for REST endpoints if automated integrations exist.
- Scaling: Streamlit doesn't scale like a dedicated API server; concurrency handled by server (e.g., multiple workers, container replicas) but not ideal for heavy programmatic loads.
- Authentication & RBAC patterns are more manual.

4.6 Dashboard app (student_dashboard_app.py & app_pro.py)

Structure

- Data access layer (DAL): functions to fetch KPIs and detailed records.
- Visualization layer: charts (distribution, scatter, bar charts), tables, and filters.
- Interaction: ability to click/click-through to student profile, export CSV/PDF.

Charts included in repository

- Perfomance Distrebution.png — distribution histogram of scores.
- Study_hours Effect on Perfomance.png — scatter or line showing correlation.
- Sport Effect on Performance & Attendance.png — grouped bar or violin plots comparing participants vs non-participants.

Integration

- The Streamlit app is the canonical UI that will load these charts dynamically from the DB or recreate them on the fly. app_pro.py can act as a wrapper that sets environment variables and launches the Streamlit app or bundles dashboard components.

4.7 Machine learning model: development, evaluation, persistence

Files & process

- student_performance_model (1).py — contains the training pipeline:
    - Preprocessing (encoding categorical features via label_mappings.json).
    - Train/test split (e.g., train_test_split(test_size=0.2, stratify=labels)).
    - Model selection (Logistic Regression / RandomForest / XGBoost suggested).
    - Metrics: Accuracy, Precision/Recall/F1 for classification; RMSE or MAE for regression.
    - Save trained models to disk: joblib.dump(model, 'model_student_perf.joblib').
- Machine Learning Model Development.pdf — documents experiments and evaluated metrics. Quote exact numbers from this PDF into the final Word report for accuracy.

Productionization

- Save preprocessing pipeline (scaler, encoders) plus model in a single pipeline object (sklearn.pipeline.Pipeline) to ensure consistent transforms at prediction time.
- Add a predict.py wrapper that loads the pipeline and returns predictions for a single student or batch. This wrapper is callable from Streamlit.

4.8 Visual analytics & interpretation

Key insights to include in the report (derived from visuals):

- **Distribution**: Is performance normally distributed or skewed? Skewness impacts thresholding strategies.
- **Study hours correlation**: Determine correlation coefficient (Pearson/Spearman) between study hours and final score and highlight threshold ranges with noticeable improvement.
- **Sports effect**: Compare mean attendance and mean scores between sports participants and non-participants; test significance (t-test or ANOVA) and report p-values.
- **At-risk profiling**: Combine features (low attendance + low study hours + certain demographic) to produce a risk segment and quantify population size.

Include visual captions and short recommendations under each figure (e.g., “Students with >6 study hours/week show median score increase of X points”).

4.9 Security, logging, testing, CI/CD and deployment

Security

- PII must be encrypted at rest or stored with access controls.
- Use environment variables for DB credentials; do not hardcode secrets.
- Secure Streamlit with authentication (behind reverse-proxy or via SSO).

Logging & Monitoring

- Add structured logs (JSON) for ETL steps and major Streamlit actions (user login, file upload).
- Monitor DB query performance and slow queries; set alerts for ETL failures.

Testing

- Unit tests for ETL functions, model prediction functions, and data validators.

- Integration tests to verify end-to-end flow: CSV -> DB -> dashboard rendering -> model predictions.
- Smoke tests for the Streamlit app endpoints.

CI/CD

- Pipeline steps: linting -> unit tests -> build Docker image -> push to registry -> deploy to staging -> run integration tests -> promote to production.
- Use GitHub Actions/GitLab CI/Jenkins depending on infra.

Deployment recommendations

- Containerize each component:
    - ETL job: small container executed on schedule (cron/airflow).
    - Streamlit app: container with Gunicorn/uvicorn wrapper (or run with streamlit run).
    - Database: managed PostgreSQL (Cloud SQL, RDS) or self-hosted with backups.

---

5. Deliverables

Delivered artifacts (from repo):

- ETL scripts: import_csv_to_postgres.py
- DB schema: Creation_Queries.sql, ER diagram.png
- Datasets: DATA.csv, Final_df.csv
- ML code & docs: student_performance_model (1).py, Machine Learning Model Development.pdf
- Visuals: several PNG/JPG charts
- Dashboard/streamlit code: student_dashboard_app.py, app_pro.py (integrator)
- Project report: Project Report Dashboard.pdf

Suggested immediate next steps (you can ask me to perform any of these):

1. Insert the exact model evaluation numbers (accuracy, F1, etc.) from the PDF into the report.
2. Embed all charts into the Word report and add captions & analysis sentences per figure.
3. Add the full SQL file content as an Appendix in Word.

4. Convert the Streamlit snippet into a production-ready streamlit_app.py, wire to DB, and containerize.
5. Create a minimal FastAPI wrapper if programmatic API endpoints are required.
