Tools and Technologies

- **Frontend:** React (v18+), React Router, and Material UI (MUI v5) for UI components. We will use MUI DataGrid/X for tables and MUI X Charts for analytics (the community version is MIT-licensed and free). We can scaffold from open-source MUI admin templates (e.g. Devias Kit, Berry) that include dashboards, forms, tables, charts and profile pages. Other JS libraries include Axios or `fetch` for REST calls, a form library like React Hook Form or Formik, and testing tools (Jest + React Testing Library). Linting/formatting will use ESLint and Prettier (with a standard config).
- **Backend:** Python 3.10+, FastAPI (ASGI framework), and Uvicorn/Gunicorn as the server. Use SQLModel (or SQLAlchemy + Pydantic) for the ORM layer, which supports MySQL out of the box [4]. MySQL will be accessed via a driver like `mysqlclient` or PyMySQL. For authentication, use FastAPI's OAuth2 utilities: store hashed passwords (with Passlib) and issue JWTs (`python-jose`). We will handle file parsing with Python libraries: Pandas/openpyxl for Excel/CSV, PyPDF2 or PDFMiner for PDFs, and python-docx for Word. NLP/AI prototyping can use Hugging Face's Transformers (e.g. GPT-2/GPT-J) or the Hugging Face Inference API (free-tier) to score text. Results will be saved in MySQL. Additional tools include Alembic for DB migrations, and logging via Python's `logging`. For testing, use Pytest (with `pytest-asyncio`) and HTTPX for API tests.
- **DevOps/CI:** Git with GitHub, with GitHub Actions for CI/CD. Workflows will lint and test both back-end and front-end on each push/PR. Use `pre-commit` hooks to enforce style (Black, isort, flake8, mypy) before commits. The back-end GitHub workflow will install Python dependencies, run Flake8 linting and Pytest as shown below. The front-end job will run `npm install`, ESLint, and Jest tests. For containerization, we can use Docker (on-prem) so each service is deployable easily on local servers. In the future, we can set up a simple deployment action to build the Docker image and push to an on-prem registry or server.

Frontend Architecture & Structure

The React app will implement **role-based views** (auditor, reviewer, admin). We store the user's role and JWT in a global context or state (via React Context or Redux) after login, then use React Router to guard routes. For example, Admin routes (e.g. user management, global reports) are only accessible to admin roles; auditors get a limited dashboard. Typical pages/components will include:

- **Login/Signup page:** Local authentication (email/password). Later this will switch to corporate SSO (OIDC/SAML).

- **Dashboards:** Multiple dashboards (by role) showing summary stats. Use MUI Grid/Card components and charts for visuals. MUI X Charts or other React chart libraries can render time-series or pie charts for analytics.

- **Upload Form:** A page with a file upload form (Excel/CSV/PDF/Word). Use an `<input type="file">` component; on submit, post via Axios to the FastAPI `/upload` endpoint. We may wrap this in a React component that shows progress.

- **Issue Viewer/Report Page:** After processing, auditors and reviewers can view detected issues and scores. Implement a table (MUI DataGrid) listing parsed fields, AI scores, and any flags. Include filtering and sorting. Provide buttons to export or generate detailed reports.

- **Analytics Page:** Charts summarizing results over time or by category (e.g. average ESG score per department). Use MUI Charts as noted.

Each page will import reusable components from a `components/` folder. We'll have a `services/api.js` (or similar) where all API calls (GET/POST) are defined. Role-based access in the UI is enforced by wrapping routes: e.g. a custom `<PrivateRoute>` component checks the user's JWT and role before rendering, redirecting to login otherwise.

Backend Architecture & Structure

We will organize the FastAPI app by feature modules (domain-oriented) following best practices. For example:

```
backend/app/
├── main.py            # creates FastAPI app, includes routers
├── database.py        # SQLModel engine, SessionLocal, Base
├── models.py          # SQLModel ORM classes (or separated per module)
├── schemas.py         # Pydantic models for requests/responses
├── auth.py            # auth routes (login, user management)
├── routers/
│   ├── users.py       # endpoints for user CRUD, role assignments
│   ├── checklists.py  # endpoints for managing ESG checklist templates
│   ├── submissions.py # endpoints to handle file uploads and submissions
│   ├── results.py     # endpoints to view NLP scores and issues
│   └── reports.py     # endpoints to generate/export reports
├── services/          # business logic (file parsing, NLP scoring)
│   ├── file_processor.py
│   └── ai_client.py
├── utils.py           # utility functions (e.g. file type detection)
└── dependencies.py    # common FastAPI dependencies (e.g. get_db, auth guards)
```

In this structure, each `routers/*.py` uses a FastAPI `APIRouter`. For example, `routers/users.py` might start with `router = APIRouter(prefix="/users", tags=["users"])` and define endpoints. We then include these routers in `main.py` (e.g. `app.include_router(users.router)`), as shown in the FastAPI docs [8]. The `services/` folder contains functions for heavier tasks: e.g. `file_processor.parse_excel()`, `ai_client.score_text()`. This separation lets routers stay thin (handling HTTP/request) and business logic stay in reusable services. The `database.py` sets up the MySQL engine (e.g. via `create_engine("mysql://...")`) and a `get_db()` dependency that yields sessions.

FastAPI Data Flow and AI Integration

FastAPI endpoints will accept JSON or multipart/form-data, interact with MySQL, and invoke AI modules as needed. For example, a POST `/upload` endpoint will receive uploaded files via FastAPI's `UploadFile` type. We'll read the file (e.g. save to disk or parse in memory with Pandas/OpenPyXL or PyPDF2/python-docx), extract the relevant text/data, and create a new submission record in MySQL.

After storing the raw data, the endpoint can call our NLP service: for instance, a function `score = ai_client.score_text(extracted_text)` which sends the text to an NLP model (via Hugging Face Inference API or a local transformer). The AI response (a score or validation message) is then saved back in the database (e.g. in a `results` table or JSON column) linked to that submission. We may run this synchronously or as a FastAPI `BackgroundTasks` job or via Celery if processing is slow.

When serving data to the frontend, FastAPI will use Pydantic schemas to serialize DB objects. For export endpoints (downloadable reports), we can leverage FastAPI's `StreamingResponse`. For example, to export a report as CSV or Excel, we build a Pandas DataFrame and return it as a file stream with appropriate headers. (This pattern is documented: e.g. using `StreamingResponse(df.to_csv(), media_type="text/csv")` or writing to an `io.BytesIO` and returning as XLSX.)

Role-based security in the API will mirror the UI. We will use OAuth2Bearer and write dependencies (`Depends`) to load the current user and verify their role (e.g. raise HTTP 403 if an auditor tries to hit an admin-only endpoint). This way, even if a user crafted a manual API call, they'll be blocked if not authorized.

Database Schema Design

We will use a normalized schema for users/roles and for checklists. At a minimum:
- **users**: (id PK, username, email, password_hash, role_id FK → roles.id, created_at, etc.)
- **roles**: (id PK, role_name, description) – e.g. "admin", "auditor", "reviewer". We link each user to one or more roles. For full flexibility (e.g. later adding granular permissions), we can implement many-to-many bridging tables: `user_roles (user_id, role_id)` and `role_permissions (role_id,permission_id)`. Permission entries could enumerate actions/screens (e.g. `view_dashboard`, `export_report`). This RBAC design is recommended for maintainability.
- **checklists**: (id, title, description) – master list of ESG checklist templates.
- **checklist_items**: (id, checklist_id FK, question_text, weight, category) – items or questions within a checklist.
- **submissions**: (id, checklist_id FK, user_id FK, submitted_at, status) – represents an uploaded form instance.
- **files**: (id, submission_id FK, filename, file_type, content or filepath, uploaded_at) – original uploaded files.
- **results**: (id, submission_id FK, score, analysis_timestamp) – stores the NLP scoring outcome. Additional JSON fields can record details or flags.
- **issues**: (id, submission_id FK, item_id FK, issue_description, severity) – any specific issues found (e.g. missing answer, anomaly).

Each table will have appropriate indexes (e.g. on foreign keys) and use InnoDB with constraints. For quick analytics, we may add summary fields (e.g. a submission's overall score) to the `submissions` table to avoid expensive joins. Timestamps and user references allow audit-trail queries.

Project Structure and Packaging

A suggested directory layout is:

```
project-root/
├── backend/
│   ├── app/
│   │   ├── main.py
│   │   ├── database.py
│   │   ├── models.py
│   │   ├── schemas.py
│   │   ├── auth.py
│   │   ├── routers/
│   │   │   ├── users.py
│   │   │   ├── checklists.py
│   │   │   ├── submissions.py
│   │   │   ├── results.py
│   │   │   └── reports.py
│   │   ├── services/
│   │   │   ├── file_processor.py
│   │   │   └── ai_client.py
│   │   ├── utils.py
│   │   └── dependencies.py
│   ├── tests/               # Pytest suite
│   ├── requirements.txt
│   └── Dockerfile
├── frontend/
│   ├── src/
│   │   ├── components/      # Reusable UI components (forms, tables, charts)
│   │   ├── pages/           # Page-level components (Dashboard, Upload, etc)
│   │   ├── services/        # API calls (using Axios)
│   │   ├── context/         # Auth and role context providers
│   │   └── utils/           # Helper functions (e.g. formatters)
│   ├── public/
│   ├── package.json
│   └── tsconfig.json
├── .github/
│   └── workflows/
│       ├── ci.yml          # run tests and linters
│       └── cd.yml          # deployment steps (e.g. build Docker)
├── .pre-commit-config.yaml
└── README.md
```

This structure keeps frontend and backend code separate but in one repo (alternatively they could be split). Within `backend/app/`, each domain (users, checklists, etc.) has its router and corresponding schemas/models. Tests mirror the structure (e.g. `tests/test_users.py`). Having a single `main.py` with `include_router()` calls is in line with FastAPI's "bigger apps" example. Environment-specific settings (database URL, AI API keys) can be managed via a `.env` file or Pydantic BaseSettings.

DevOps: CI/CD, Linting, Formatting, Testing

We will use GitHub Actions for continuous integration. Key steps include: - **Linting:** In the Python pipeline, run `flake8` (or `ruff`) to enforce PEP8/style rules. (As one CI tutorial notes, "Flake8… scans your FastAPI application for issues and ensures your code meets the required style guide (PEP8)".) In the Node pipeline, run `eslint`.

- **Testing:** Run `pytest` on the backend (unit and integration tests). For React, run `npm test` (Jest) on each push/PR. The CI should fail if any tests fail. ("the pipeline will run all unit tests using PyTest… ensuring no functionality is broken.") We should also generate coverage reports; GitHub Actions can upload them as artifacts or comment on PRs (using a test report action).

- **Pre-commit hooks:** Configure pre-commit to run Black, isort, flake8, and mypy before each commit (for Python) and optionally a linter/formatter hook for JS. This catches errors early.

- **Formatting:** Use Black for Python and Prettier for JS. Adding these to pre-commit (or as Husky hooks) ensures consistent style.

- **CI Configuration:** A typical `ci.yml` workflow will have jobs like: checkout code, set up Python, install deps, run `flake8`, then `pytest` (as shown above). Another job does: set up Node, install npm deps, run `eslint --max-warnings=0` and `npm test`.

- **Deployment:** Since hosting is on-prem, we can implement a simple `cd.yml` that builds the Docker images (frontend and backend), and deploys to the local environment (e.g. pushing the image to an internal registry or invoking a Kubernetes rollout). This can be manual or automated.

AI Model Integration and Future Switching

We should design the AI inference logic to be easily replaceable. For now, we will use a free NLP model via public APIs – for example, the Hugging Face Inference API (with an API token) or a lightweight open-source transformer. Abstract all AI calls behind a service layer: e.g. a class `AIClient` with a method `score_text(text)` that under the hood calls the chosen model/API. The configuration (which model endpoint or key to use) lives in settings. For instance, we can have an environment variable `AI_MODEL_URL` or `HUGGINGFACE_API_KEY`. The backend code will simply call `ai_client = AIClient()` and `ai_client.score_text(data)`.

Later, when switching to the internal "ChatGPT for All" model, we only need to change the `AIClient` implementation or its configuration – we might simply point it at a new URL or load a different local model. Because the frontend only receives scored results (not the model itself), this swap requires **no changes to the frontend or API interface**. The response schema stays the same. In short, by keeping an abstraction (adapter/factory pattern) for the AI engine and using configuration files or environment variables, we can swap out the underlying model without refactoring the FastAPI routes or React components.

Sources: FastAPI's docs demonstrate using `UploadFile` for file uploads and structuring apps with `APIRouter` in sub-packages. SQLModel supports MySQL as noted in the official tutorial. Dashboards built on React/MUI have been widely adopted (see MUI admin templates [3]) and MUI X Charts is MIT-licensed for charting . CI pipelines typically include linting and testing steps (e.g. Flake8 and PyTest) as in recent FastAPI CI examples. Finally, FastAPI can stream CSV/Excel exports using `StreamingResponse` and Pandas,. These practices ensure a robust, maintainable architecture aligned with our requirements.