ChatGPT

Logistics Operations Health Assessment & Feedback System

Abstract: We design a system to measure and improve logistics operations using hierarchical surveys and AI-driven feedback. The system supports multi-tier roles (Admin, Zone/Region/City/Branch Supervisors) with strict access controls. Managers complete a structured health-assessment survey (based on a standard logistics scorecard) that yields numeric KPIs and an overall "health" score. Each submission triggers calls to large language models (e.g. GPT-4, Google Gemini) that generate actionable feedback at the question, category, and overall levels. A built-in feedback loop prompts users to plan and update improvement tasks (planned/completed/pending) and then re-assess, enabling continuous improvement. The front-end is built in Streamlit (forms, dashboards, feedback views) with secure authentication and auto-logout. The back-end uses a relational database to store users, responses, scores, AI feedback, and task updates. We outline RBAC roles, survey scoring logic, AI integration design, UI/UX features, database schema, security measures, and deployment considerations.

User Roles and Access Control

- **Roles & Hierarchy:** We define five roles: **Admin**, **Zone Supervisor**, **Region Supervisor**, **City Supervisor**, and **Branch Supervisor**. Admin has full visibility across all data; each subordinate role is restricted to its own area of responsibility (e.g. an East Zone Supervisor sees only the East Zone's data). This employs a hierarchical Role-Based Access Control (RBAC) model [1] [2]. In hierarchical RBAC, senior roles inherit permissions of more junior roles; for example a Zone Supervisor can view data for any branches within their zone, while a Branch Supervisor sees only their branch [1] [2].

- **Authentication:** Each user logs in with a unique employee ID and password. The system uses secure authentication (e.g. via a library like Streamlit-Authenticator) and issues a session token. Passwords are stored only in hashed form [3] (using modern schemes like bcrypt or Argon2 [4]). On login, the user's role is checked and a user session is established with role information. All UI and data queries are filtered according to this role – for example, database queries include the user's zone/region branch ID, ensuring row-level data segregation [5]. This prevents, say, a City Supervisor from viewing a city outside their jurisdiction.

- **Role-Based Queries:** At the database level, we enforce access via role-based filtering or row-level security. For instance, each survey response record is tagged with branch/city/region IDs, and SQL policies (or application logic) ensure that users can only SELECT or UPDATE rows matching their permitted hierarchy [5]. Thus, a Region Supervisor's queries automatically filter out data from other regions, and the Admin account bypasses all filters.

Survey Framework and Scoring

- **Hierarchical Survey Design:** The survey is structured by organizational level (Zone, Region, City, Branch). At each level there are predefined *categories* (e.g. Operational Efficiency, Safety, Customer Service) with fixed weights (summing to 100%). Under each category are multiple objective *questions* yielding quantitative metrics (e.g. on-time delivery %, inventory accuracy %).

Category examples include Operational Throughput, Quality/Accuracy, Safety/Compliance, Customer Service, and Financial Performance [6] [7].

- **Question Scoring:** Each question has a formula that maps the raw input (actual KPI vs. target) to a 0–100 score. "Higher-is-better" metrics use (Actual/Target)*100 (capped at 100), while "lower-is-better" use (Target/Actual)*100 [8]. For example, if target lead time is 24 hours but actual is 30 hours, the score is (24/30)*100≈80. All scores are floored at 0 and capped at 100. The survey system automatically computes each question score after submission.

- **Weighted Aggregation:** Within each category, each question has a predefined weight (e.g. 2% of total if 50 questions). We compute the *category score* as the weighted average of its question scores [8]. The final **Health Score** is the weighted sum of all questions across categories (equivalently, the weighted average across categories). In practice this is identical to the Balanced Scorecard methodology: multiply each question score by its weight and sum to 100 [8]. For example, if two questions have weights 70% and 30% of a category and scores 80 and 90, the category score is 0.7*80* + *0.3*90 = 83 [8].

- **Implementation Notes:** The survey questions, target values, and weight parameters are stored in database tables (e.g. `Questions`, `Categories`). When a user fills the form, the app looks up each question's formula and weight, computes scores, and writes results to a `Responses` or `Scores` table. The system supports different sets of questions per hierarchy level (zone vs region vs city, etc.), with each level's questions defined in the "Logistics Operations Health Assessment Framework" reference.

AI-Powered Feedback Integration

- **Overview:** Each submitted survey triggers calls to a generative AI model (e.g. OpenAI's GPT, Google's Gemini, or a comparable API). We send the user's responses and computed scores via an API to the AI, with a prompt engineered to generate helpful feedback. The model returns textual recommendations at three granularity levels: (a) *Per-question* (e.g. specific suggestions for improving on-time delivery), (b) *Per-category* (summary for broader areas like "Operational Efficiency"), and (c) *Overall survey* (strategic advice for overall health improvement).

- *Figure: Example summary dashboard showing AI-generated overall feedback and analytics.* The AI is instructed to analyze the numeric responses against industry best practices (citing logistics KPIs) and output actionable advice. For example, if on-time delivery is low, the AI might suggest process optimizations or equipment investments. If inventory accuracy is high, it might commend that area. Research shows generative AI can efficiently distill survey data into concise, actionable insights [9]. In our system, we format the input prompt to include the survey context and reference industry benchmarks (from the Health Assessment Framework sources), and parse the model's JSON/text response into structured feedback items.

- **Feedback Storage:** The AI's responses are saved in the backend (e.g. in a table `AI_Feedback` linked to each survey submission). Each feedback entry records the level (question/category/overall), the associated question or category ID (if applicable), and the feedback text. This allows the UI to display feedback contextually. For traceability, we may also log the model version and prompt used.

- **Examples:** Using generative AI for surveys is gaining traction. For instance, tools can automatically generate concise summaries of survey findings and recommend specific

improvements based on respondent feedback [9]. In our logistics context, the AI might note patterns across questions (e.g. "Multiple efficiency metrics fell below target; consider cross-training staff and optimizing routes.").

Progress Tracking Loop

- **Action Items Prompt:** After initial feedback is given, the system enters a continuous improvement loop. Upon subsequent logins or on a dashboard page, the user is prompted with *action items* derived from past feedback. Each AI recommendation can be turned into a task (e.g. "Reduce lead time by 10%"). The user marks each task's status: *Planned*, *Completed*, or *Pending*.

- **User Updates:** The user periodically updates this action-tracking page. When tasks are completed, they indicate so. The system may allow adding notes or next steps. This process mirrors a Plan–Do–Check–Act (PDCA) cycle: the survey ("Check") identifies gaps, planning/implementation occurs, then re-survey ("Act") to measure improvement.

- **Iterative Surveys:** When the user re-fills the survey form (monthly or on schedule), their new responses produce updated scores. The system again calls the AI for fresh feedback based on the latest state. Over time, a historical log of survey scores and task statuses builds up. This iterative loop encourages continuous improvement: if a prior suggestion was marked "Completed" but the metric still lags, the new AI analysis will reinforce urgency or offer refined advice.

- **Tracking Table:** In the database, we maintain a table `Tasks` or `ActionItems` with columns (task_id, survey_id, description, status, updated_at). Linking tasks to the originating question or category can help trace effectiveness. A dashboard can chart task completion trends over time alongside score improvements.

Front-End & User Interface

- **Streamlit App Structure:** The UI is built in Python using Streamlit. On startup, users are presented with a **Login form** (e.g. via `stauth.login`) requiring their employee ID and password [10] [11]. Successful authentication sets `st.session_state` variables for the user's name, role, and region. A logout button is available (calling `stauth.logout()`), and the app enforces an inactivity timeout (auto-logout) to close stale sessions [12]. Users can also go to a "Change Password" page, entering old and new passwords; this triggers a backend update (with hashing) in the credentials store.

- **Survey Form UI:** The main interface uses Streamlit forms (`st.form`) to present the hierarchical survey questions. Questions are grouped by category. Input widgets may include numeric inputs or sliders for percentage metrics. For example, "On-time delivery rate (%)" might use `st.number_input(min_value=0, max_value=100)`. When the user submits, the app computes scores and saves responses.

- **Dashboards and Feedback View:** After submission (or on a summary page), the app shows the **results dashboard**. Key features include:

- **Scorecards:** Display the category-wise scores (e.g. as a bar chart or colored metrics).
- **Historical Trends:** Line charts of past overall scores per month.

- **AI Feedback Tabs:** Using `st.tabs` or expanders, the AI feedback is displayed: one section lists question-specific tips, another shows category-level insights, and a top section gives overall recommendations.

- **Action Items Panel:** A checklist or table of the current action items (planned/completed/pending), possibly with buttons or radio inputs to update status.

- **Interactive Elements:** Streamlit's session state and rerun mechanism enable interactive workflows. For example, marking a task "Completed" can update its status in the database immediately. We can also use `st.experimental_memo` or caching to speed up repeated queries for static data (like question definitions).

- **Auto Logout:** Per NIST guidelines, sessions should idle timeout after ~1 hour [12]. We implement this by tracking last activity and forcing re-authentication after the limit. Streamlit-Authenticator's cookie settings support expiration, or we can use a background timer in Streamlit to clear the session.

- **Security Features:** The login form is served over HTTPS (encrypted in transit). Password fields are masked. Streamlit-Authenticator is configured to hash passwords, and we may enforce complexity rules on new passwords. All database calls use parameterized queries to avoid injection.

- **Example UI:** *Figure: Illustrative survey form UI capturing responses (with a completion gauge).* This shows how managers enter KPI values. The layout uses Streamlit's column and expander components to organize a potentially large number of questions without overwhelming the user.

Database Architecture

We employ a relational database (e.g. PostgreSQL or MySQL) to store all data with the following core tables:

- **Users:** (user_id, name, role, zone_id, region_id, city_id, branch_id, password_hash, etc.). Each user's row includes their jurisdiction identifiers to facilitate filtering. The `password_hash` is never stored in plaintext [3].

- **Roles:** (role_id, role_name) — static list (Admin, Zone Sup, etc.).

- **Hierarchy Tables (optional):** We may have reference tables for Zone, Region, City, Branch with foreign keys linking them. For example, a `Branches` table with columns (branch_id, name, city_id, manager_user_id) can relate branches to cities and branch supervisors.

- **Survey Definitions:**

- **Categories:** (category_id, name, weight, level) — defines category names and their weights (e.g. each 20%).

- **Questions:** (question_id, category_id, text, weight, formula, is_lower_better flag). The `formula` field may encode the normalization rule or target metric reference. Each question belongs to one category.

- **Responses:** (response_id, survey_id, question_id, raw_value, score). Records each answered question. `raw_value` is the user's input, `score` is computed 0–100.

- **Surveys:** (survey_id, user_id, role_level, period, overall_score, timestamp). A survey row links to the user who submitted it, the time, and stores the final overall health score. `role_level` indicates whether it was a Zone/Region/City/Branch-level survey.

- **Scores:** (optional) We might store pre-computed category scores per survey for quick reporting: (survey_id, category_id, category_score).

- **AI_Feedback:** (feedback_id, survey_id, level, category_id, question_id, feedback_text). Stores the text returned by AI. `level` indicates "question", "category" or "overall". If level="question", `question_id` links to which question; if level="category", `category_id` is set; if "overall", both IDs are null.

- **Tasks (ActionItems):** (task_id, survey_id, description, status, updated_at). Tracks improvement tasks tied to a specific survey (typically derived from AI feedback). `status` is an enum {Planned, Completed, Pending}.

- **Audit_Log:** (log_id, user_id, action, timestamp, details). Optional table to record key actions (logins, survey submissions, password changes) for compliance.

This normalized schema ensures data integrity and clarity. Relationships use foreign keys (e.g. Responses→Surveys, Questions→Categories). We can visualize the ERD (Figure): users⇄surveys⇄responses⇄questions⇄categories, plus feedback and tasks linked to surveys.

For performance and scalability, we can implement **row-level security** in the DB as an extra safeguard [5]. For example, policies can automatically restrict SELECT/UPDATE to rows where the zone/region matches the user's context, eliminating any risk of a query accidently leaking data across boundaries [5].

Security Measures

- **Session Timeout:** Inactivity logout is enforced as per security best practices. NIST SP 800-63B recommends an idle timeout of ~1 hour or less [12]. Our app implements this by expiring the auth cookie or requiring re-login after 60 minutes without interaction.

- **Password Policies:** We follow NIST guidance: passwords are **never stored in cleartext** [3]. They are salted and hashed using a slow hashing scheme (bcrypt/Argon2) [3] [4]. Hash iterations or cost factors are set high enough to deter brute-force attacks, and the salt length meets guidelines. We also recommend users choose strong, unique passwords and have procedures for secure password resets.

- **Encryption:** All sensitive data is protected in transit and at rest. We enforce HTTPS/TLS for all web traffic. According to OWASP, all sensitive data at rest should be encrypted [13]. The database uses disk encryption (e.g. encrypted tablespace or managed DB encryption). Connection strings/secrets are stored securely (not hard-coded). If handling particularly sensitive PII, we could encrypt specific columns as needed.

- **Access Control and Auditing:** Even inside the DB, privilege separation is applied. Database credentials used by the app have least privilege (only needed schemas/tables). We audit key operations: all admin activities and data changes are logged. Audit logs help meet compliance and trace any unauthorized access.

- **Password Reset/Change:** We enforce controlled reset flows (e.g. email verification or admin approval for resets). When users change passwords, the new password is hashed on-the-fly and replaces the old hash, with no plaintext retention. We may enforce periodic password updates (for example, every 90–180 days) following best practices, though modern guidance focuses more on strength than arbitrary expiration.

- **Input Validation:** All user inputs (even numeric fields) are validated on the server side. The app uses parameterized queries or ORM bindings to prevent SQL injection. We also constrain inputs to valid ranges (e.g. 0–100 for percentages).

- **Rate Limiting:** To prevent automated attacks on login, we can implement rate limiting or CAPTCHA after repeated failures.

Deployment & Infrastructure

- **Containerization:** The application is containerized using Docker [14]. A Dockerfile installs the necessary Python environment and Streamlit app code. Containerization ensures consistency between environments (dev/test/prod).

- **Hosting Options:** Depending on needs, deployment can target:

- **Cloud Instances:** e.g. AWS EC2 or Google Compute Engine. One can spin up servers, install Docker, and run the container. These can be behind a load balancer if scaling is needed.
- **Kubernetes/ECS/EKS:** For higher scale, deploy on a container orchestration platform. This allows horizontal scaling (multiple Streamlit replicas) and easier rollouts.
- **Streamlit Community Cloud:** For small-scale or rapid prototypes, one can use Streamlit's managed hosting (though it has limitations in customization and is intended for simpler apps) [14].

- **Other PaaS:** Services like Heroku or Azure App Service can run Docker containers and provide managed SSL and scaling.

- **Database:** We use a managed relational database (e.g. AWS RDS, Google Cloud SQL) for high availability and backup support. For multi-zone resilience, replicas or cluster options may be used.

- **Scalability:** The architecture separates the UI from heavy tasks. For example, calls to AI (which may have latency) can be asynchronous: the app enqueues a job or calls the API after saving responses. If very high concurrency is expected, AI calls could be handled by a separate microservice or background worker queue, to avoid blocking the UI thread. Load testing should guide scaling parameters (e.g. number of web replicas). Container orchestrators can auto-scale based on CPU/memory usage or request latency.

- **Data Segregation:** In multi-tenant modes (each zone/region as a "tenant"), scaling can be addressed by sharding or using schema separation as needed [15]. However, since all data is

within one organization, a simpler approach is to run one database with strict access filtering rather than fully separate DBs. Role-based policies in the app and DB ensure logical segregation without requiring multiple instances.

- **CI/CD Pipeline:** We recommend a continuous deployment pipeline. On code commit, automated tests run, then a Docker image is built and pushed to a registry. Deployment tools (e.g. Terraform, Kubernetes manifests) then update the production environment. Secrets (DB credentials, API keys) are injected securely via environment variables or a secret manager.

- **Monitoring:** The deployed system should include monitoring (e.g. Prometheus/Grafana for container health, or cloud provider monitoring) to track uptime, response times, and errors. Alerts can be set up for outages or unusual metrics.

Conclusion

This architecture delivers a secure, role-aware platform for ongoing logistics performance assessment. By combining a structured KPI survey with AI-driven feedback and a built-in improvement loop, we enable managers at every level to get timely, actionable insights into operations. Streamlit's rapid UI development, together with containerized deployment and a robust DB backend, ensure maintainability and scalability. Security is integrated at every layer, following best practices for authentication, encryption, and data isolation. Citations to industry sources demonstrate that our design uses established KPIs and scoring methods [8], proven RBAC security models [1] [2], and innovative AI feedback techniques [9]. This white paper provides both conceptual guidance and implementation specifics (schema design, process flows, and references) to realize the system end-to-end.

---

[1] What Is Role-Based Access Control (RBAC)? A Complete Guide | Frontegg
https://frontegg.com/guides/rbac

[2] Role-Based Access Control (RBAC): A Comprehensive Guide | Pathlock
https://pathlock.com/blog/role-based-access-control-rbac/

[3] [12] NIST Special Publication 800-63B
https://pages.nist.gov/800-63-4/sp800-63b.html

[4] [13] OWASP Top Ten 2017 | A3:2017-Sensitive Data Exposure | OWASP Foundation
https://owasp.org/www-project-top-ten/2017/A3_2017-Sensitive_Data_Exposure

[5] [15] Multi-Tenant Database Design Patterns 2024
https://daily.dev/blog/multi-tenant-database-design-patterns-2024

[6] [7] [8] Logistics Operations Health Assessment Framework.pdf
file://file-YPbPSeXFke8RcQrELn9JVk

[9] Exploring Generative AI-based Surveys | VWO
https://vwo.com/blog/exploring-generative-ai-based-surveys-the-future-of-feedback/

[10] [11] Streamlit-Authenticator, Part 1: Adding an authentication component to your app
https://blog.streamlit.io/streamlit-authenticator-part-1-adding-an-authentication-component-to-your-app/

[14] Deploy Streamlit using Docker - Streamlit Docs
https://docs.streamlit.io/deploy/tutorials/docker