TicketSense User Manual

1. Introduction

I developed **TicketSense** as a comprehensive full-stack application to address the "Final Interview Challenge". My goal was to digitize the Singapore 4D and TOTO lottery experience by combining Optical Character Recognition (OCR) for automated ticket scanning with advanced AI-driven predictive modeling.

Key Features I Implemented

- **Smart Scanning:** I integrated Google Gemini's multimodal capabilities to extract numbers from ticket images and verify them against official results.
- **Automated Verification:** I designed a serverless workflow that automatically fetches results and notifies users of wins/losses immediately after a draw.
- **Predictive AI:** I implemented three distinct analytical models (Statistical, Machine Learning, and Deep Learning) to forecast potential outcomes.

2. System Architecture

I chose a decoupled architecture to ensure reliability and automation without requiring manual intervention.

My Automation Pipeline (GitHub Actions)

Instead of relying on traditional cron jobs that require a constantly running server, I utilized **GitHub Actions** to achieve serverless reliability.

1. **Workflow A (The Scraper):** I configured this workflow to trigger via a CRON schedule (e.g., Mon/Thu at 6:35 PM). It executes my custom Python scraper to fetch the latest draw results from official sources and updates my Supabase database.
2. **Workflow B (The Checker):** I set up this dependent workflow to trigger automatically upon the successful completion of Workflow A. It scans all "Pending" tickets in the database, compares them against the newly fetched results, and dispatches notifications to users if they have won.

3. Installation & Setup

Prerequisites

To run the system I built, your environment needs:

- **Python 3.11+** – For the FastAPI backend
- **Node.js 18+** – For the React frontend
- **Git** – For cloning the repository

Environment Configuration

I designed the application to use environment variables for security. You must configure these before running the app.

1. Backend Config:

Navigate to the backend/ folder and create a file named .env:

- # backend/.env
- SUPABASE_URL="your_supabase_project_url"
- SUPABASE_SERVICE_ROLE_KEY="your_supabase_service_role_key"
- GEMINI_API_KEY="your_google_gemini_api_key"

2. Frontend Config:

Navigate to the frontend/ folder and create a file named .env:

- # frontend/.env
- VITE_SUPABASE_URL=" "
- VITE_SUPABASE_PUBLISHABLE_DEFAULT_KEY=" "
- VITE_API_URL="http://localhost:8000"

---

4. Running the Application

You'll need two terminal windows to run the backend and frontend services simultaneously.

1. Start Backend Server

Open a terminal and navigate to the backend directory:

```
cd backend
```

Create and activate a virtual environment:

- python -m venv myvenv
- **On Windows:**
- myvenv\Scripts\activate
- **On macOS/Linux:**
- source myvenv/bin/activate**.**

Install Python dependencies:

```
pip install -r requirements.txt
```

Start the FastAPI server with uvicorn:

```
uvicorn app.main:app --reload
```

2. Start Frontend Server

Open a new terminal and navigate to the frontend directory:

```
cd frontend
```

Install Node.js dependencies:

```
npm install
```

Start the Vite development server:

```
npm run dev
```

Access Points

Once both services are running:

- **Frontend Web Application:** http://localhost:5173
- **Backend API:** http://localhost:8000

3. Checking core functionality

- You can upload the images of tickets I have stored in the GitHub repo under the folder called **Sample_Images.** I have already included 2 losing tickets as well
- Open up a third terminal and navigate to the backend using `cd backend`
- Run this command in the terminal: `python scripts/check_and_notify.py`
- You can also nav to here and run the Github Action by clicking on **"Run workflow"** instead of entering the command in the terminal
- It will take a minute or two for the script to finish running. Once its done navigate to the notification section of the web app
- You can observe the winning and losing notifications alongside the winning amount.

- **NOTE**: If the web app were to be deployed, functionality of the script would change to only check the latest draw. However, since we are using older tickets, I have modified it to check previous draws as well, resulting in the script taking a slightly longer time to execute.

5. Data & Results

To ensure the accuracy of my predictive models and result verification, I established a rigorous data acquisition and validation pipeline.

Data Acquisition (Web Scraping)

I engineered a custom Python scraping engine to build the historical dataset required for this project.

- **Source:** I targeted the official **Singapore Pools website** as the primary source of truth to ensure absolute data integrity.
- **Methodology:** I utilized `requests` and `BeautifulSoup` to systematically traverse historical draw archives, extracting winning numbers, draw dates, and prize structures for both 4D and TOTO.
- **Automation:** As detailed in the architecture section, this scraper runs automatically to keep the dataset synchronized with real-world events.

Data Validation (OCR to Database)

I implemented strict **Pydantic models** to act as a validation gateway between the raw OCR output and my database.

- **Structured Extraction:** Instead of relying on unstructured text, I forced the OCR engine to output JSON that strictly conforms to my `TicketCreateData` schema.
- **Type Safety:** My Pydantic layer automatically validates critical fields—ensuring ticket costs are positive floats, draw dates are valid calendar dates, and bet types match specific enums (e.g., "Ordinary", "SystemRoll")—before the data ever touches the database.

Data Storage & Schema

I stored the validated data in a **Supabase (PostgreSQL)** database using a JSONB column structure. This allowed me to handle the variable prize structures while maintaining the query performance of a relational database.

6. Predictive Analysis

For predictive analysis, I designed and implemented three distinct analytical methodologies, progressing from simple statistical observation to complex deep learning.

1. Statistical Baseline: Frequency Analysis

- Concept: This model operates on the Law of Large Numbers, assuming that while lottery draws are theoretically random, physical machines may exhibit minute mechanical biases over short windows.
- Implementation: I analyzed the last 100 draws to calculate a probability distribution for each number. I then ranked these numbers by frequency to identify "Hot" (frequently drawn) and "Cold" (rarely drawn) numbers.
- Role: This serves as the "Common Sense" baseline for users who prefer traditional tracking methods.

2. Machine Learning: XGBoost Classifier

- Concept: I selected XGBoost (Extreme Gradient Boosting) to capture non-linear relationships between external context and draw outcomes. Unlike the statistical model, this looks at when the draw happens.
- Feature Engineering: I transformed raw draw dates into categorical features (Month, Day of Month, Day of Week). The model uses decision trees to determine if specific numbers are more likely to appear on specific days (e.g., "Does the number 7 appear more often on Thursdays in December?").
- Role: This acts as the "Context Aware" model, creating predictions based on calendar patterns rather than just past frequency.

3. Deep Learning: LSTM (Long Short-Term Memory) Network

- Concept: I treated the lottery history as a Time-Series Sequence, similar to stock market data. I used an LSTM, a type of Recurrent Neural Network (RNN) capable of learning order-dependence in sequence prediction problems.
- Architecture:
  - Input Layer: Accepts a sliding window of the past sequence of draws.
  - Hidden Layers: LSTM units with memory cells that retain information about long-term dependencies and trends.
  - Output: A probability score for the next potential winning numbers.
- Role: This is the "Trend Hunter," attempting to detect hidden temporal momentum that static models miss.

My Note on Responsible Presentation:

I added a "Confidence Meter" and explicit disclaimers in the UI to state that lottery draws are stochastic processes. I present these predictions strictly for **educational and exploratory purposes** and do not guarantee financial returns.

---

7. Testing & Troubleshooting

Unit Testing

I adhered to engineering best practices by writing a suite of automated tests covering my API endpoints and validation logic. I did so by thinking of edge cases that my backend might run into. The tests are located under the tests folder in the backend.

To run the tests:

1. Ensure you are in the backend/ directory.
2. Run the pytest command:
3. pytest tests/

Common Troubleshooting.

- **Issue:** *Frontend `.env` changes don't apply*
    - **Fix:** Vite only loads env vars starting with `VITE_`.
    - **Fix:** Restart the dev server (`Ctrl+C`, then `npm run dev`) after editing `frontend/.env`.
- **Issue:** *Supabase errors like "Invalid API key" / "Project not found"*
    - **Fix:** Confirm `SUPABASE_URL` and `SUPABASE_KEY` are correct (no quotes, no trailing spaces).
    - **Fix:** Make sure you're using the correct key type (service key for backend if required).
- **Issue:** *Frontend says "Network Error".*
    - **Fix:** Verify the backend is running at localhost:8000 and that the frontend/.env file is correctly named.
- **Issue:** *`Python` command not found / wrong Python version*
    - **Fix:** Install **Python 3.11+** and verify with `python --version` (or `python3 --version` on macOS).
    - **Fix:** On macOS/Linux, use `python3 -m venv myvenv` if `python` points to Python 2 or isn't mapped.
- **Issue:** *Backend starts, but `.env` values aren't loading*
    - **Fix:** Ensure the `.env` file is inside the backend/ folder (same level as `requirements.txt`).
    - **Fix:** Restart the backend after editing `.env`.

8. Security & Compliance

I built TicketSense with security best practices to satisfy the **"Responsible AI & Privacy"** assessment criteria.

- **Data Minimization:** I ensured that no NRIC, credit card details, or full real names are stored. I only use User IDs (UUIDs) to link tickets to accounts.
- **Secure Storage:** I process images in memory. Row Level Security (RLS) policies are enabled for the database tables as well
- **API Security:** I protected all my endpoints via JWT (JSON Web Token) verification.