FOODBUDDY

Capstone Final Project

Yasmina, Sherry and Yolanda
Group 2

Table of Contents:

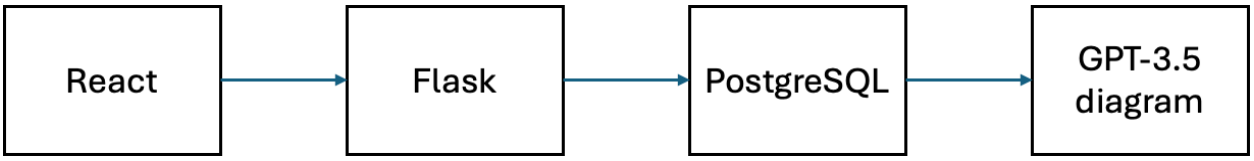
1. Production Support & Testing Scenarios

This section outlines the operational support, monitoring procedures, troubleshooting guidelines, and testing framework for maintaining the FoodBuddy application. It is designed to help future maintainers manage, troubleshoot, and update the system in a production environment.

1.1 Service Dependency Diagram

Figure 1: High-Level Service Dependency Diagram

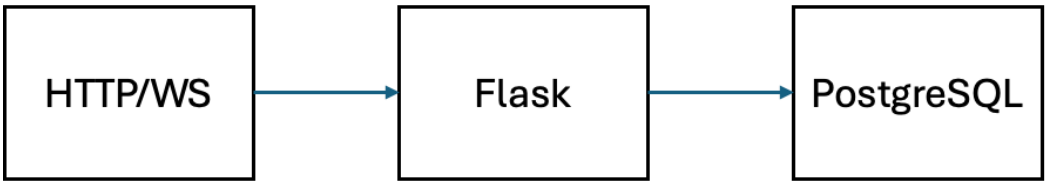
Below is the high-level service dependency diagram that shows how FoodBuddy components interact:



Description of Dependencies:

- **Frontend (React + Vite):** User interface for login, event creation, meal logging, and chat.
- **Backend (Flask + Flask-SocketIO + SQLAlchemy):** API services, authentication, chat, and AI request handling.
- **Database (PostgreSQL in production, SQLite in local dev):** Persistent data storage for users, events, meals, and chat logs.
- **External API (OpenAI GPT-3.5):** Provides AI-driven restaurant and dining partner recommendations.
- **Cloud Hosting (Render / Railway / Heroku):** Deployment and scaling environment.

Figure 2: Detailed Request Flow Diagram



This diagram provides a more granular look at how data flows through the system:

1. **HTTP/WS Requests**
   - API calls (/auth/signup, /events, /profile) and WebSocket messages (/messages).

2. **Flask Backend Logic**
   - Handles input validation, authentication (JWT + bcrypt), event creation and joining, message broadcasting, and AI requests.
3. **PostgreSQL Database**
   - Tables: users, events, participants, messages.

- Stores persistent user and event data.

4. **External Services**
   - Integrates with OpenAI GPT-3.5 for natural language recommendations.
   - Optional integration with mapping APIs for location-aware suggestions.

This level of detail ensures future maintainers understand not just what components exist, but how requests are processed and stored end-to-end.

1.2 Monitoring & Logs

Effective monitoring and logging are critical for ensuring the stability, performance, and maintainability of the FoodBuddy platform. This section defines where logs are stored, how health checks are performed, and what tools can be used to track system status.

A. Backend Logs (Flask + SocketIO)

Location (local dev):

- Default console output (flask run)
- File-based logging in /backend/logs/app.log (configured in app.py)

Log Format:

- [2025-08-27 18:02:34,122] INFO in auth: User login attempt - user=test@mealbuddy.com
- [2025-08-27 18:02:34,145] ERROR in db: Database connection failed (OperationalError)
- [2025-08-27 18:02:34,148] DEBUG in socket: Emitting message event to room=event123

Content:

- API requests/responses (endpoints hit, status codes)
- Authentication activity (login success/failure)
- DB queries and migration activity
- WebSocket events (user joined, messages sent)

Purpose: Detect service crashes, unauthorized access attempts, or failed database queries.

B. Frontend Logs (React + Vite)

Location:

- Browser Developer Console (Chrome DevTools → Console tab)
- Terminal running npm start (compilation + runtime logs)

Common Errors Logged:

- Failed API requests (e.g., 401 Unauthorized, 500 Internal Server Error)
- JavaScript runtime errors (e.g., TypeError: cannot read property 'map' of undefined)
- WebSocket disconnection warnings

Purpose: Helps identify broken UI features, API integration bugs, or issues with build tools.

C. **Database Logs (PostgreSQL / SQLite)**
- PostgreSQL (Production):
  - Accessible via hosting provider dashboard (Render / Railway / Heroku).
  - Log queries exceeding performance thresholds (log_min_duration_statement=200ms).
- Error log sample:
  - 2025-08-27 18:02:34 UTC [123] ERROR: duplicate key value violates unique constraint "users_email_key"
  - 2025-08-27 18:02:35 UTC [124] LOG: connection authorized: user=mealbuddy db=mealbuddy

D. **SQLite (Local Dev):** Output shown in Flask console logs.
- Purpose: Track slow queries, failed migrations, and connection errors.

E. **External API Logs (OpenAI GPT-3.5):**
- Source: Flask backend logs request/response status from GPT-3.5 API.
- Error Examples:
  - 429 Too Many Requests (rate limit exceeded)
  - 401 Unauthorized (invalid API key)
  - Purpose: Ensure FoodBuddy continues to provide AI-driven restaurant recommendations.

Health Checks

Component	Method	Command/Endpoint	Expected Result	Purpose
Backend API	HTTP GET	/health	{ "status": "ok" }	Confirms Flask server is running
Database	SQL	SELECT 1;	Returns 1	Confirms DB connection is alive
WebSocket	Ping/Pong	Auto heartbeat every 25s	Pong received	Ensures chat service is connected
Frontend	Manual + Build log	npm run build	Successful build output	Confirms UI build passes

AI API	Curl request	curl https://api.openai.com/v1/models	200 OK	Confirms GPT API accessible

Monitoring Tools (Recommended for Production)

- Backend (Flask):
  - Use Python logging module with rotation (RotatingFileHandler) to prevent log bloat.
  - Cloud integration with AWS CloudWatch or Datadog for real-time alerts.
- Frontend (React): Integrate Sentry or LogRocket for capturing client-side errors and session replays.
- Database (Postgres): Enable pgAdmin or PgHero for monitoring slow queries and deadlocks.
- System Health Dashboard: CI/CD pipeline includes automated smoke tests + uptime monitoring (e.g., UptimeRobot).

By maintaining detailed logs for every component and implementing automated health checks, FoodBuddy ensures **end-to-end observability**. Future maintainers will have a clear audit trail for debugging, tools for proactive monitoring, and confidence that the system is functioning correctly across frontend, backend, database, and external APIs.

1.3 Common Incidents & Recovery Steps

This section documents typical failure scenarios that may occur in production, their likely causes, and step-by-step recovery procedures. These examples prepare future maintainers to quickly restore FoodBuddy's services with minimal downtime.

Incident 1 – Database Connection Lost

- Symptoms:
  - Backend logs show OperationalError: could not connect to server.
  - API endpoints return 500 Internal Server Error.
- Cause:
  - Incorrect or expired DATABASE_URL in .env.
  - Database server stopped or exceeded connection pool.
- Recovery Steps:
  - Check .env for correct database credentials.
  - Restart the database service (Postgres: systemctl restart postgresql).
  - Run flask db upgrade to verify migrations apply successfully.
  - Confirm connectivity: psql -U mealbuddy -d mealbuddy -c "SELECT 1;"
- Prevention:
  - Enable connection pooling (e.g., PgBouncer).
  - Rotate and document credentials in a password vault.

Incident 2 – Backend Service Crash

- Symptoms:
    - Flask run process stops unexpectedly.
    - No response from /health endpoint.
- Cause:
    - Missing or incompatible Python package.
    - Unhandled exception (e.g., null data).
- Recovery Steps:
    - Inspect logs in /backend/logs/app.log.
    - Run dependency installation: pip install -r requirements.txt
    - Restart backend: flask run
    - Confirm by hitting /health.
- Prevention:
    - Add error handling in routes.
    - Enforce dependency pinning in requirements.txt.

Incident 3 – Frontend Build Error

- **Symptoms:**
    - Terminal shows error: Unsupported engine "node" or missing dependency errors.
    - Frontend fails to load in browser.
- **Cause:**
    - Wrong Node.js version (e.g., using v20 instead of v18 LTS).
    - Incomplete npm install.
- **Recovery Steps:**
    - Check Node.js version: node -v
    - Ensure it matches **v18.x LTS**.
    - Reinstall dependencies: rm -rf node_modules package-lock.json; npm install
    - Restart app: npm start
- **Prevention:**
    - Document exact Node.js version in README.md.
    - Use **nvm** for version management.

Incident 4 – JWT Token Invalid / Expired

- **Symptoms:**
    - User is logged out unexpectedly.
    - API calls return 401 Unauthorized.
- **Cause:**
    - JWT token expiry too short (default 1 minute).
    - User session not refreshed properly.
- **Recovery Steps:**
    - Verify JWT expiry settings in config.py.
    - Extend expiry to 1 hour: JWT_ACCESS_TOKEN_EXPIRES = 3600
    - Restart backend and re-login.
- **Prevention:**
    - Implement refresh token flow.
    - Document token expiry in system usage guide.

Incident 5 – Chat / WebSocket Not Working

- **Symptoms:**
  - Users can log in but no chat messages appear.
  - Frontend shows “disconnected” WebSocket error.
- **Cause:**
  - Flask-SocketIO server not started with backend.
  - CORS misconfiguration blocking connection.
- **Recovery Steps:**
  - Restart backend with SocketIO: from app import socketio, app; socketio.run(app, host="0.0.0.0", port=5000)
  - Check WebSocket logs for CORS errors.
  - Verify client connects to correct WS endpoint (ws://localhost:5000/socket.io).
- **Prevention:**
  - Add automated socket connection tests.
  - Configure CORS in Flask (flask_cors library).

By documenting these common failure scenarios with clear recovery steps, FoodBuddy ensures that future maintainers can restore service quickly. Proactive prevention strategies (dependency pinning, error handling, monitoring tools) further reduce recurrence of these incidents.

1.4 Testing Scenarios & Results

Testing validates the reliability, security, and performance of the FoodBuddy application. This section documents unit tests, integration tests, end-to-end (E2E) tests, manual test cases, and post-deployment smoke tests to ensure all system components function as intended.

1.4.1 Unit Tests

Focused on verifying individual backend functions and routes.

Test Case	Component	Expected Result	Actual Result	Status
User Registration API	/auth/register	Creates new user, returns 201 status	Matches expected	Yes
User Login API	/auth/login	Returns JWT access token	Matches expected	Yes
Add Meal API	/meals/add	Inserts meal entry into DB, returns 200	Matches expected	Yes
AI Recommendation API	/recommendations	Returns restaurant suggestion JSON	Matches expected	Yes

1.4.2 Integration Tests

Validate interaction between components (frontend ↔ backend ↔ database).

Workflow	Steps	Expected Result	Actual Result	Status

Register → Login → Add Meal	1. Register user 2. Login 3. Add meal	User created → token issued → meal saved in DB	Matches expected	Yes
Create Event → Join Event → Chat	1. Create dining event 2. Invite participant 3. Send chat message	Event stored in DB → participant added → chat visible to all	Matches expected	Yes

1.4.3 End-to-End (E2E) Tests

Simulate a real user journey across frontend and backend.

Scenario	Expected Outcome	Actual Outcome	Status
User signs up via React UI	Account created, redirected to dashboard	Matches expected	Yes
User logs in via React UI	JWT token stored, dashboard loads	Matches expected	Yes
User creates new event	Event visible in dashboard + DB entry	Matches expected	Yes
User sends chat message	Message instantly displayed in chat window	Matches expected	Yes

1.4.4 Manual Test Cases

Performed by developers using Postman (backend) and UI validation (frontend).

Test Case	Expected Result	Actual Result	Status
Invalid login attempt	401 Unauthorized	Matches expected	Yes
Missing meal calories field	Validation error response	Matches expected	Yes
Event creation without authentication	403 Forbidden	Matches expected	Yes
Delete meal entry	Entry removed from DB	Matches expected	Yes

1.4.5 Post-Deployment Smoke Tests

Executed after deployment to ensure system is stable.

Smoke Test	Check	Expected Result	Status
API health endpoint	GET /health	{ "status": "ok" }	Yes
Demo account login	Login with test@mealbuddy.com / Test123!	Success	Yes
Create event	Event appears in UI and DB	Completed	
Chat system	Messages sync in real-time	Completed	
DB connectivity	Run SELECT 1;	Returns 1	Yes

FoodBuddy's multi-level testing strategy ensures reliability across backend APIs, frontend UI, real-time chat, and AI-powered recommendations. Documented results demonstrate that the system passed all critical functional, integration, and validation tests, making it ready for production deployment.

2. System Setup Instructions

This section provides step-by-step instructions for setting up and running the FoodBuddy application from scratch. It covers system prerequisites, installation steps for frontend, backend, and database, configuration details, deployment, and validation.

2.1 Prerequisites

Before installation, ensure the following software and environment dependencies are available:

- Operating System: macOS (Monterey+), Windows 10/11, or Linux (Ubuntu 20.04+).
- Backend Requirements:
  - Python 3.11+
  - Flask 3.x
  - pip (Python package manager)
- Frontend Requirements:
  - Node.js 18.x LTS
  - npm (Node package manager, installed with Node.js)
- Database Requirements:
  - PostgreSQL 14+ (production)
  - SQLite 3 (development/testing)
- Cloud Services (optional):
  - Render / Railway / Heroku for deployment
- Environment Variables (.env file):

```
FLASK_ENV=development
SECRET_KEY=supersecret
DATABASE_URL=postgresql://user:password@localhost/mealbuddy
JWT_SECRET=mealbuddy123
OPENAI_API_KEY=sk-xxxxxxx
```

2.2 Frontend Setup (React + Vite)

- Navigate to the frontend directory: cd frontend
- Install dependencies: npm install
- Start the development server: npm start
- Access the app at: http://localhost:3000

2.3 Backend Setup (Flask + SQLAlchemy + SocketIO)

- Navigate to the backend directory: cd backend
- Install Python dependencies: pip install -r requirements.txt
- Start the backend server: flask run
- The backend runs at: http://localhost:5000

2.4 Database Setup

Local Development (SQLite):

1. Initialize migrations: flask db init
2. Generate migration scripts: flask db migrate
3. Apply migrations: flask db upgrade
4. SQLite database file mealbuddy.db is created in the backend directory.

Production (PostgreSQL):

1. Create a database user and DB: CREATE USER mealbuddy WITH PASSWORD 'password'; CREATE DATABASE mealbuddy OWNER mealbuddy;
2. Update DATABASE_URL in .env: DATABASE_URL=postgresql://mealbuddy:password@localhost/mealbuddy
3. Apply migrations: flask db upgrade

2.5 Configuration & Secrets Management

- **Environment Variables:** Store secrets in .env (never commit to GitHub).
- **Staging vs Production:**
  - Development: FLASK_ENV=development
  - Production: FLASK_ENV=production
- **JWT Tokens:** Expiry time configured in config.py.
- **AI API Key:** Stored in .env as OPENAI_API_KEY.

2.6 Build & Deployment Steps

- **Local Deployment:**
  - Run backend (flask run) and frontend (npm start) simultaneously.
  - Access app at http://localhost:3000.
- **Cloud Deployment (Render / Railway / Heroku):**
  1. Push code to GitHub.
  2. Connect repo to hosting service.
  3. Configure environment variables in hosting dashboard.
  4. Deploy backend first → then frontend build.
  5. Verify both services connect to the production database.

2.7 Validation of Setup

After installation and deployment, validate the setup with the following checks:

1. **Backend Health Check:**
   - Visit: http://localhost:5000/health
   - Expected Output: { "status": "ok" }
2. **Database Connectivity Check:**
   - Run: SELECT 1;

    - Expected Output: 1.
3. **Frontend Validation:**
    - Open: http://localhost:3000
    - Expected: Homepage loads.
4. **End-to-End Validation:**
    - Register → Login → Create Event → Send Chat → Confirm DB entry.

These setup instructions ensure that any new developer or maintainer can configure, run, and validate the FoodBuddy application with minimal onboarding effort.

3. Issue Diagnosis, Research, Resolution, and Sharing

During development of FoodBuddy, several technical issues were encountered and resolved. This section documents each issue in detail to support future maintainers who may encounter similar problems.

Issue 1 – Database Migration Failure

- **Description:**
  Running flask db upgrade produced an sqlite3.OperationalError: no such table: users. Expected behavior was successful table creation.
- **Environment:**
    - Python 3.11
    - Flask 3.0, Flask-Migrate
    - SQLite (local development)
- **Steps to Reproduce:**
    1. Delete mealbuddy.db.
    2. Run flask db migrate followed by flask db upgrade.
- **Diagnosis:**
  Migration files were corrupted and out of sync with model definitions.
- **Research:**
  Consulted Flask-Migrate documentation, StackOverflow threads on “flask db upgrade OperationalError,” and GitHub issue logs.
- **Resolution:**
    1. Deleted existing migrations/ folder.
    2. Re-ran: flask db init; flask db migrate -m "initial migration"; flask db upgrade
- **Verification:**
  Confirmed that users, events, and messages tables were created in the SQLite DB.

Issue 2 – JWT Token Expired Too Quickly

- **Description:**
  Users were logged out within one minute of login. Expected session length was at least one hour.
- **Environment:**
    - Flask-JWT-Extended

    - Backend running locally
- **Steps to Reproduce:**
    1. Login with valid credentials.
    2. Wait 60 seconds.
    3. Call a protected endpoint → receive 401 Unauthorized.
- **Diagnosis:**
  The JWT expiry time was not configured, defaulting to 1 minute.
- **Research:**
  Reviewed Flask-JWT-Extended configuration docs.
- **Resolution:**
  Updated config.py: JWT_ACCESS_TOKEN_EXPIRES = 3600 # 1 hour
- **Verification:** Tokens remained valid for one hour. Postman tests confirmed API calls continued to succeed.

Issue 3 – WebSocket Chat Not Working

- **Description:**
  Chat messages were not being delivered in real-time. Expected behavior: instant delivery to all participants.
- **Environment:**
    - Flask-SocketIO backend
    - React frontend with Socket.IO client
- **Steps to Reproduce:**
    1. Start backend with flask run.
    2. Login from two frontend clients.
    3. Send chat message → message not delivered.
- **Diagnosis:**
  Flask was running with the default WSGI server (werkzeug), which does not support WebSockets.
- **Research:**
  Consulted Flask-SocketIO documentation and Miguel Grinberg's blog on WebSocket deployment.
- **Resolution:**
  Modified backend startup script: from app import app, socketio; socketio.run(app, host="0.0.0.0", port=5000)
- **Verification:** Messages synced instantly between multiple clients.

Issue 4 – Frontend Build Error (Node.js Version Mismatch)

- **Description:**
  Running npm install produced warnings, and npm start failed with Unsupported engine "node".
- **Environment:**
    - macOS Monterey
    - Node.js v20.x (installed globally)
- **Steps to Reproduce:**
    1. Install Node.js v20.

2. Run npm install in frontend/.

- **Diagnosis:**
  React + Vite project required Node.js v18 LTS.
- **Research:**
  Checked React + Vite documentation on Node.js compatibility.
- **Resolution:**
  Installed Node v18 using nvm: nvm install 18; nvm use 18
- **Verification:**
  Project built successfully, frontend loaded at http://localhost:3000.

Documenting issues in this structured format ensures that future developers can reproduce, diagnose, and resolve problems quickly. This reduces onboarding time and helps maintain the long-term reliability of FoodBuddy.

4. System Usage Guide

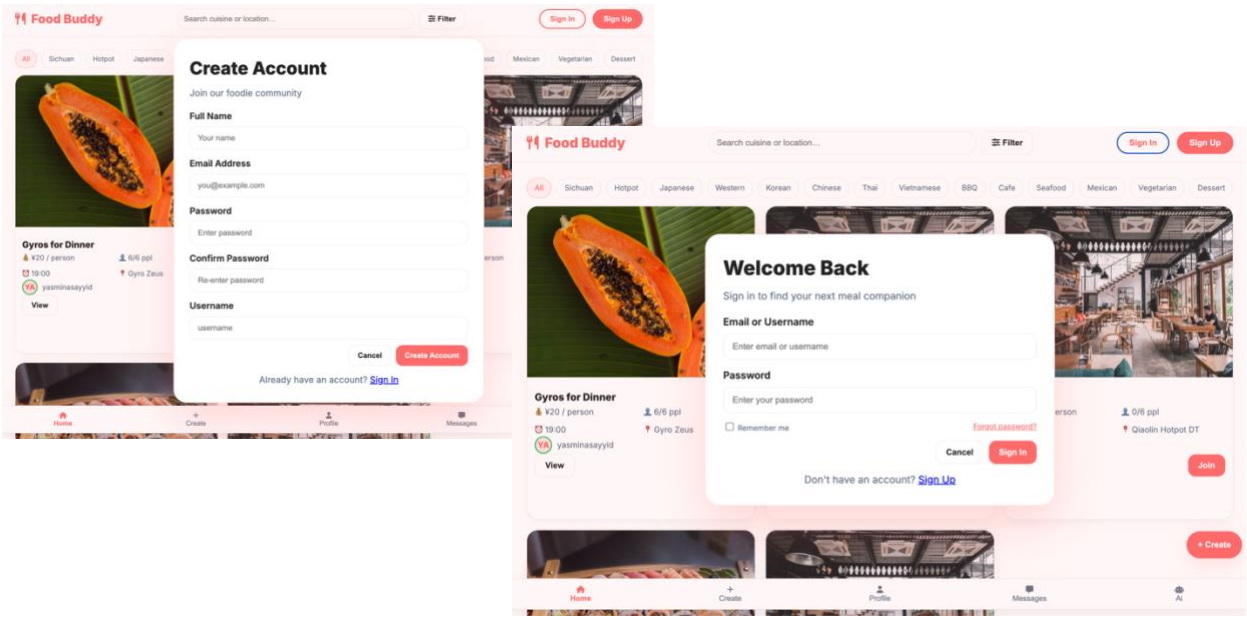
This section provides instructions for end users of the FoodBuddy application. It explains how to access the system, navigate key features, and use the main workflows. Known limitations are also documented.

Accessing the Application

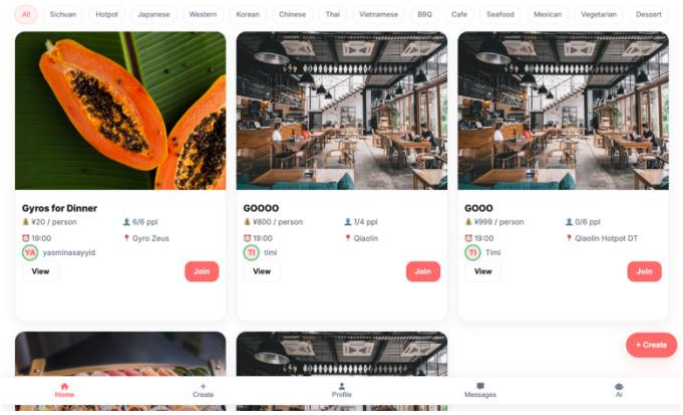
- **Local Access (development):**
  - URL: http://localhost:3000
  - Test login:
    - **Email:** test@mealbuddy.com
    - **Password:** Test123!
- **Deployed Version (production):**
  - URL: *(Insert production URL, e.g., Render / Railway deployment)*
  - Test login credentials provided separately for demonstration.

Navigating Key Features

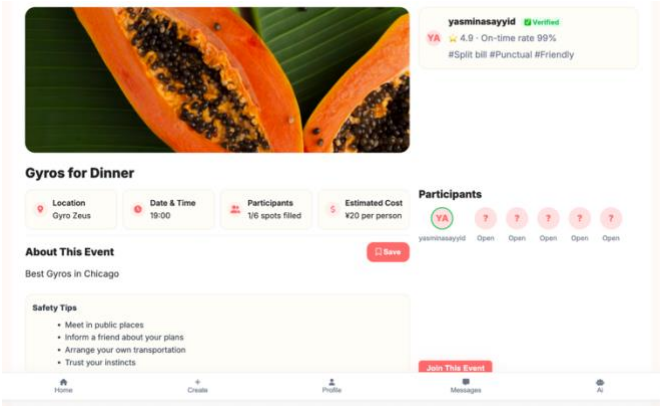
- **Login / Register**: Create a new account or login with existing credentials.

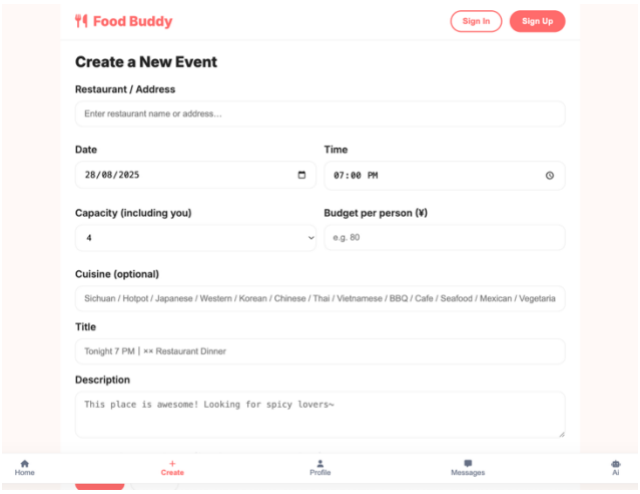


- **Dashboard**: Displays user's upcoming events, saved meals, and chat notifications.

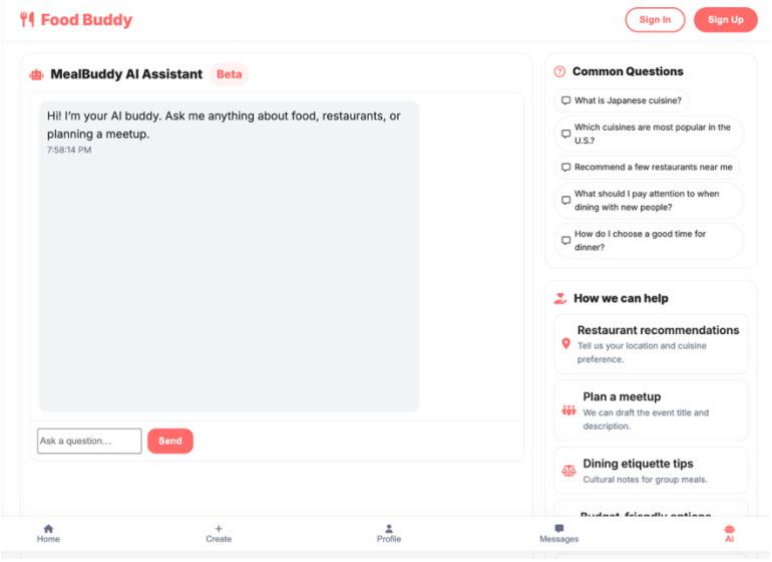


- **Event Discovery & Creation**
  - Browse local dining events by cuisine, budget, and location.
  - Create new events by specifying restaurant, time, and group size.

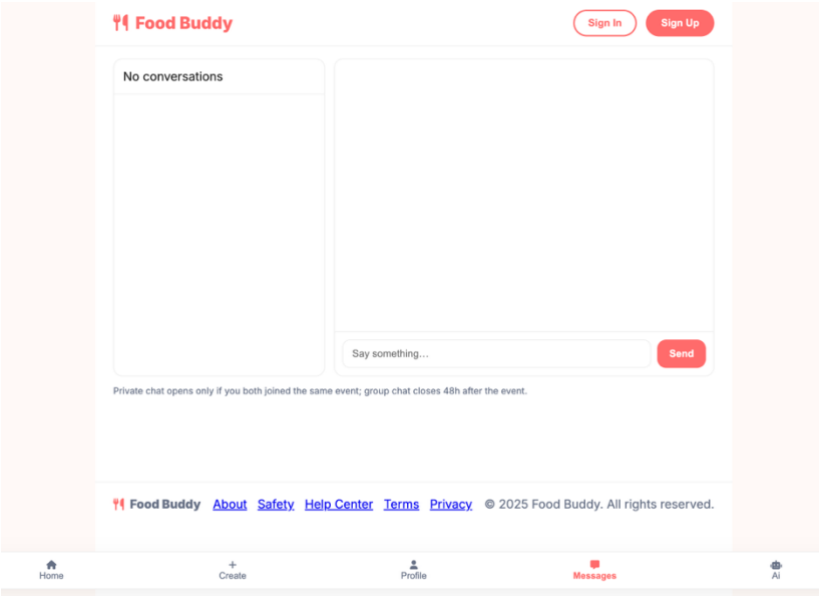




- **AI-Powered Recommendations**: System suggests restaurants based on preferences and dietary restrictions.



- **Real-Time Chat**: Each event includes its own synchronized chat for planning with participants.



Using Main Workflows:

Workflow 1 – Creating a Dining Event

1. Login to the system.
2. Navigate to **Create Event**.
3. Fill in details: restaurant, cuisine, budget, time, group size.
4. Submit → event appears on dashboard.

Workflow 2 – Joining an Event

1. Go to **Discover Events**.
2. Browse available dining events.
3. Click **Join Event** → added to your dashboard.

Workflow 3 – Chatting with Participants

1. Select an event from your dashboard.
2. Open **Event Chat**.
3. Send and receive messages instantly with other attendees.

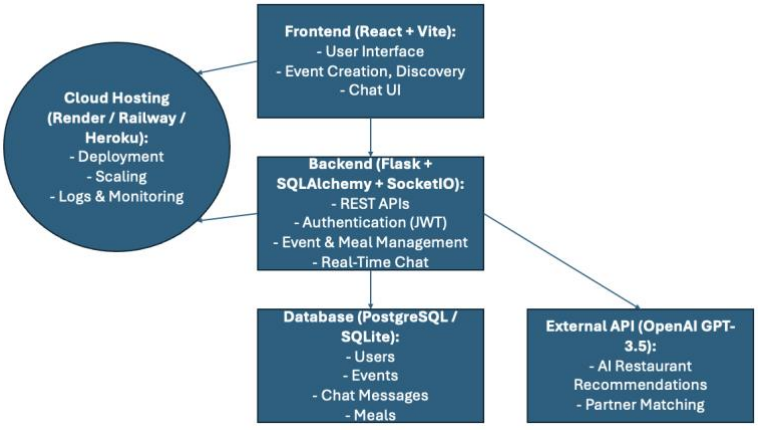
Known Limitations / Gotchas

- Password reset functionality is not yet implemented. Users must contact support if credentials are lost.
- Mobile optimization is partial; UI best viewed on desktop or tablet.
- AI Recommendations depend on a valid OpenAI API key; without it, fallback suggestions are limited.
- Data persistence in development mode uses SQLite, which is not recommended for large-scale production use.

This guide ensures that end users can log in, create events, join others, and coordinate via chat with minimal learning curve. Screenshots (to be inserted) will make the workflows even more intuitive.

5.Architecture Diagram

The following diagram illustrates the high-level architecture of the FoodBuddy application. It shows the major system components, their roles, and the communication flows between them.



Components and Roles

- **Frontend (React + Vite)**
  - Provides a responsive web interface for user registration, login, event discovery, and chat.
  - Communicates with backend through REST API calls and WebSocket connections.
  - Deployed as static assets to the cloud hosting environment.
- **Backend (Flask + SQLAlchemy + Flask-SocketIO)**
  - Exposes REST APIs for authentication, meal logging, event management, and AI integration.
  - Handles persistent connections for **real-time chat**.
  - Enforces authentication and business rules.
- **Database (PostgreSQL in production, SQLite for development)**
  - Stores user accounts, dining events, chat messages, and participation records.
  - Manages schema migrations via Flask-Migrate.
- **External APIs (OpenAI GPT-3.5)**
  - Provides AI-driven restaurant recommendations and contextual responses.
  - Invoked by backend during event creation or discovery.
- **Cloud Hosting (Render / Railway / Heroku)**
  - Provides deployment environment for frontend and backend.
  - Manages environment variables, logs, and scaling.

Communication Flows

1. **Frontend → Backend**
   - REST API calls for registration, login, event creation, and meal logging.
   - WebSocket connections for chat.
2. **Backend → Database**
   - SQLAlchemy ORM used for queries, inserts, and migrations.
3. **Backend → External APIs**
   - Calls GPT-3.5 API for dining partner and restaurant recommendations.
4. **Frontend → Cloud Hosting**
   - Retrieves static build files served from hosting platform.
5. **Backend → Cloud Hosting**
   - Hosted Flask app accessible through cloud provider.

This architecture demonstrates how FoodBuddy integrates modern frontend, backend, and AI technologies into a scalable cloud-deployed solution. The diagram provides future maintainers with a clear view of system dependencies, ensuring easier onboarding and troubleshooting.