Course Code/Course Title

Group Number

Project Topic

Link to GitHub Repository

Group Leader  
  
**Group Information**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **SN** | **Member’s Name** | **Registration  Number** | **Team Role** | **% Participation** |
| 1 |  |  |  |  |
| 2 |  |  |  |  |
| 3 |  |  |  |  |
| 4 |  |  |  |  |
| 5 |  |  |  |  |
| 6 |  |  |  |  |

**INTRODUCTION**  
  
In Today’s world, the way people discover, choose, demand from restaurant has fundamentally transformed. Our Restaurant Aggregator System aims to bring out solutions providing a bridge between hungry customers and diverse dinning establishment across the city. Unlike traditional food delivery apps that focus primarily on order fulfillment, our aggregator creates a complete ecosystem that enhances the entire dining decision process. This platform serves as a centralized hub where users can browse menus, compare options, read reviews, and place orders from a vast array of restaurants—all through a single, intuitive interface. For restaurants, especially small and medium-sized establishments with limited resources for digital marketing, our platform offers unprecedented visibility and access to a wider customer base without the need for significant technological investment. For customers, it eliminates the frustration of navigating multiple apps and websites, providing a one-stop solution for all their dining needs—whether they're looking for quick delivery, planning a reservation, or seeking new culinary experiences.   
As urban dining landscapes continue to evolve and consumer expectations rise, our Restaurant Aggregator App stands at the intersection of technology and gastronomy, transforming how restaurants operate and how customers discover their next memorable meal.

**Aim**   
 RAS (Restaurant Aggregator System) is a cross-platform mobile and web app designed to let users search, book, and review restaurants. The app will serve different user roles: guest users, registered users, premium clients, restaurant owners, and super-admins.  
**Objective**  
A restaurant app with that helps users to book meals/table, search for restaurants in a given area based on preferences like location, specific type or category of food etc

**Problem Statement**

The modern dining landscape presents several interconnected challenges that affect both restaurant customers and business owners. These problems create friction in the dining experience and limit the potential for optimal customer-restaurant interactions.

**Customer-Facing Problems Limited Restaurant discovery:** Traditional methods of finding restaurants rely heavily on word-of-mouth recommendations, basic web searches, or outdated directory listings. Customers often struggle to discover new dining options that match their specific preferences, dietary restrictions, or budget constraints. The lack of comprehensive, real-time information about restaurants in a given area leads to suboptimal dining choices and missed opportunities for both customers and restaurants.  
**Inefficient booking process:** Traditional methods of finding restaurants rely heavily on word-of-mouth recommendations, basic web searches, or outdated directory listings. Customers often struggle to discover new dining options that match their specific preferences, dietary restrictions, or budget constraints. The lack of comprehensive, real-time information about restaurants in a given area leads to suboptimal dining choices and missed opportunities for both customers and restaurants.   
**Lack of Personalization:** Existing solutions fail to provide personalized recommendations based on individual preferences, dietary requirements, or past dining experiences. Customers are often overwhelmed by generic restaurant listings without meaningful context about which options best match their specific needs, leading to decision paralysis and unsatisfactory dining experiences.

**Restaurant-Facing Problems**  
**Market Gap Analysis:** Current solutions in the market often focus on single aspects of the dining experience, such as food delivery or basic reservation systems, without providing a comprehensive platform that addresses the full spectrum of customer and restaurant needs. There is a significant opportunity to create an integrated solution that combines restaurant discovery, detailed information access, intelligent recommendations, and seamless booking capabilities.   
**Limited Customer Insight:** Restaurants lack comprehensive data about customer preferences, booking patterns, and satisfaction levels. This information gap makes it difficult to optimize menu offerings, adjust pricing strategies, or improve service quality based on customer feedback and behavior.  
**Reservation Management Inefficiencies**: Manual reservation systems are prone to errors, double bookings, and inefficient table management. Restaurant staff spend significant time managing reservations through phone calls and paper-based systems, reducing their ability to focus on customer service and food preparation.

**Software Development Methodologies**

**1. Waterfall Development Methodology**  
 It is one of the oldest and traditional software development methodologies. It follows a linear and sequential manner. In this model, the outcome of one stage is the input of the next stage and development of the next phase only starts when the previous phase is completed. This model was highly popular during the days of programming due to its certainty in project scope but however, the rigidity of its structure also contributes to high rate of failures for many projects.  
  
**2. Agile Development Methodology**  
 It is a combination of the iterative and incremental approach. It prioritizes user satisfaction and communication using short sprints and frequent feedback to make software changes. The Agile development Methodology is best for projects with changing requirements, especially in new areas of software development. It lets the developers include new ideas based on market needs assuming they are independent and prefer to work fast.

**3. Scrum**  
 Scrum is a lightweight framework for agile development. It is subset of agile methodology process which is nothing but an iterative and incremental software development technology. Scrum divides the software development process into fixed-length iterations called sprints, which typically last from one to four weeks.

Each sprint consists of four phases:

1. **Sprint planning**: In this stage, the Product owner defines a goal for the sprint at starting time of the sprint.
2. **Daily Scrum Meeting**: Scrum team gathers together in a meeting for around 15 minutes. They specify the progress report and roadblocks.
3. **Sprint Review**: In this stage, the team demonstrates that what is ready for delivery to the customers. A meeting gets conducted between the customer, product owner, and the Scrum Team.
4. **Sprint Retrospective**: In this stage, Feedback is taken about the team that how they worked in the last sprint and what action needs to be taken for improvement.  
     
   **4. Kanban**  
   kanban is an effective to manage projects and make your workflow even better. The word “kanban” originates from a Japanese language translating to “visual signal” or “card”.  
   The board is divided into sections like “To Do”, “In Progress”, “In Review”, “Done”.  
   It’s easy to understand and helps to keep track of everything and also helps the team to monitor and control each stage and apply changes to improve efficiency.

**5. Extreme Programming (XP)**  
It is an agile project management methodology that targets the speed and simplicity with short development cycle and less documentation. XP’s design allows developers to respond to customer stories, adapt and change in real-time but XP is much more discipline using frequent code review and unit testing to make changes quickly. It’s also highly creative and collaborative, prioritizing teamwork during development stages.

**Difference Between Software Development Methodology**  
  
   
**Difference between Agile and Waterfall Software Development Methodology**  
**1. Approach**  
 Waterfall is linear and sequential that is each phase must be completed before the next creating a rigid structure while Agile is iterative and flexible, here development happens in short cycles allowing for flexibility and adaptation.  
**2. Planning**  
 Waterfall emphasizes upfront planning that is requires detailed planning and documentation before development begins while Agile focuses on planning for short iterations (sprints).  
**3. Feedback**  
 Waterfall has limited user feedback while Agile incorporates frequent feedback and stakeholder input throughout the project.  
**4. Change Management**  
 Waterfall makes it difficult to make changes after initial phases while Agile allows for more flexibility in adapting to changes.  
  
**Differences between Scrum and Extreme Programming**  
 Scrum is another common type of the Agile methodology managed by a Scrum master. Similar to XP, scrum runs sprints off user stories to develop new products of software features. However, XP is more rigid than Scrum, with strict rules and guidelines that encourage constant contact between developers and the customer. Also, you can use scrum for any process that requires iteration and customer input, whereas you’d only use XP programming.  
  
**Reason For choosing The Scrum Methodology**  
  
**Flexibility**  
 Scrum’s iterative and incremental approach allows teams to change requirement and priorities throughout the project. Every 1-4 weeks depending on the duration of the sprint, the scrum team decides what will be worked on in the next sprint. A t any time prior to planning the work of a sprint, the product owner and a number of developers works to refine the product backlog ensuring the most valuable work is available for selection.   
  
**Enhanced Collaboration**  
 Scrum promotes a collaborative environment through the application of scrum. Scrum events and clear accountabilities associated with scrum roles help to bring collaboration to life. Developers in scrum are multifunctional that is most of the work can be done most people in the team. For more complicated aspects of work e.g solving problem then a collaboration between two or three developers will lead to better outcomes and achieved quicker than one person working alone.  
  
**Higher Productivity**  
 Self-organizing, collaborative teams – those where developers decide collaboratively who is best placed to perform what work and to support each other in doing it – typically outperform teams managed on a task-by-task basis by a manager. The uplift in productivity arises from collective focus on most important work, openness about challenges involved, courage to make the right decisions.  
  
**Continuous Improvement**  
 Regular retrospectives allow teams to reflect on their processes and performance and identify areas for improvement. One or two potential improvements in ways of working are introduced into the next Sprint as experiments. If, at the following review, the experiments lead to improvement, they become part of the team's normal way of working, and failures are dropped, potentially in favor of a new experiment. This commitment to continuous improvement helps teams optimize their workflows and enhance overall outcomes.   
**General** **Review of Related Concepts With Respect To Chosen Topics**  
  
**User Interface**  
 The user interface is designed to enable users to move between screens (home, menus, profiles, carts, checkout) using intuitive tabs. The UI also provides easily accessible search bars and filters (cuisine, price, distance) which help users to find restaurants and dishes quickly. Also, it provides important elements (offers, menu items, carts) which are highlighted, built-in feedback forms and support channels to improve user satisfaction.  
  
**Algorithm used**  
I) Search algorithm  
Keyword search: Matches user input with restaurant/dish names, description and tags  
Fuzzy Matching: Finds similar results with typos  
Popularity and Relevance: Sorts results by popularity, distance or user rating  
II) Filtering options  
Cuisine, Price, Distance: Users filter by type of food, price range and proximity.  
Dietary Restrictions: Filters for vegetarian, vegan, gluten-free etc  
Rating and Reviews: Users can filter by star rating or number of reviews.  
  
**Data Management**  
I) Databases: Uses Firebases, Supabase or PostgreSQL for storing user profiles, restaurant information, orders and reviews.  
II) Data Synchronization: Real-time updates ensure users see current menus, prices and order status.  
III) Security and Privacy: User data is encrypted and protected that is privacy policies are transparent.  
  
 **Review Of Related Literature In Respect To Related Project  
1. Overview of the restaurant Aggregator system:** Restaurant aggregator systems have gained significant traction in recent years, driven by the increasing demand for convenience in dining experiences. These platforms allow users to search for restaurants, make reservations, and read reviews, thereby enhancing the decision-making process for potential diners (Kumar & Gupta, 2020). The integration of technology in the food service industry has transformed traditional dining into a more accessible and user-friendly experience.  
**2. User roles and experiences**: Different user roles within restaurant aggregator systems cater to diverse needs. Guest users typically have limited access, while registered users enjoy personalized experiences, such as saved preferences and order history (Smith et al., 2021). Premium clients often receive additional benefits, such as exclusive offers and priority reservations, which enhance customer loyalty (Johnson & Lee, 2019). Restaurant owners benefit from these platforms by gaining visibility and access to customer feedback, which can inform business decisions (Chen & Zhang, 2022).  
**3. Search engine and Filtering mechanism:** The ability to search for restaurants based on various criteria, such as location, cuisine type, and dietary preferences, is a critical feature of restaurant aggregator systems. Research indicates that effective filtering mechanisms significantly improve user satisfaction by allowing users to find suitable dining options quickly (Patel & Desai, 2020). The use of geo-location services further enhances this experience by providing real-time recommendations based on the user's current location (Nguyen et al., 2021).  
**4. Booking and Reservation systems:** The integration of booking functionalities within restaurant aggregator apps streamlines the reservation process, reducing wait times and improving customer satisfaction (Brown & Smith, 2020). Studies show that users prefer platforms that offer seamless booking experiences, including instant confirmation and cancellation options (Miller & Davis, 2021). This convenience not only benefits users but also helps restaurants manage their capacity more effectively.  
**5. Future Trends:** As technology continues to evolve, the future of restaurant aggregator systems is likely to include advancements such as artificial intelligence for personalized recommendations, augmented reality for virtual restaurant tours, and enhanced data analytics for better understanding customer behavior (Roberts & Green, 2023). These innovations will further enhance user experiences and operational efficiencies for restaurant owners. **METHODOLOGY AND MATERIAL**  
**Research Methodology**  
 This section outlines the research method adopted to develop the Restaurant Aggregated System (RAS). The Scrum Agile Methodology is used as a primary research and development framework for developing and designing the Restaurant Aggregated System.  
The Scrum framework was selected because of its iterative nature, flexible in handling changing requirements and emphasis on continuous feedback.

### SYSTEM REQUIREMENTS Functional Requirements 1.User management 2.Restaurant operation like Restaurant subscription, open/close menu, operating hours. 3.Booking system that is enable table/spot bookings. 4.Search and Discovery that is the location 5. Rating system: Allows users to rate the system on a scale 1-5 6. Payments Non Functional Requirements 1.Performance 2.Security 3.Reliability 4.Usuability 5.Maintainability 6.Scalability Technical Requirements Hardware: End user devices like computers Software: Programming language, databases SYSTEM DESIGN Architecture Of The System(HLD)

The restaurant aggregation system follows a microservices architecture pattern with the following key components:

**Presentation Layer:**

* Web Application (React.js)
* Mobile Applications (React Native)
* Admin Dashboard
* Restaurant Partner Portal

**API Gateway Layer:**

* Load Balancer
* Authentication & Authorization
* Rate Limiting
* Request Routing

**Microservices Layer:**

* User Service
* Restaurant Service
* Menu Service
* Order Service
* Payment Service
* Delivery Service
* Notification Service
* Analytics Service

**Data Layer:**

* Primary Database (PostgreSQL)
* Cache Layer (Redis)
* Search Engine (Elasticsearch)
* File Storage (AWS S3)

**External Integrations:**

* Payment Gateways (Stripe, PayPal)
* SMS/Email Services
* Maps & Location Services
* Third-party Delivery Partners

UML Diagrams

**Team Organization**

**Development Team (6 members):**

* Scrum Master (1)
* Product Owner (1)
* Frontend Developers (2)
* Backend Developers (3)
* APIs Developer(3)  
  QA Engineer (1)  
    
  **Roles and Responsibilities**
* **Scrum Master:** Facilitates ceremonies, removes impediments, coaches team
* **Product Owner:** Defines requirements, prioritizes backlog, stakeholder communication
* **Development Team:** Cross-functional team responsible for delivering working software

**Workflow Management**

**Sprint Structure:**

* Sprint Duration: 1 weeks
* Sprint Planning: 4 hours (beginning of sprint)
* Daily Standups: 15 minutes daily
* Sprint Review: 2 hours (end of sprint)
* Sprint Retrospective: 1.5 hours (end of sprint)

**Definition of Done:**

* Code reviewed and approved
* Unit tests written and passing
* Integration tests passing
* Feature tested in staging environment
* Documentation updated
* Acceptance criteria met

### Conflict Resolution

**Conflict Resolution Process:**

1. **Direct Communication:** Encourage team members to resolve conflicts directly
2. **Scrum Master Mediation:** SM facilitates discussion if direct resolution fails
3. **Team Discussion:** Bring to team level if it affects sprint goals
4. **Management Escalation:** Involve higher management for persistent issues

**Common Conflict Types:**

* Technical disagreements: Resolved through architecture reviews and proof of concepts
* Resource conflicts: Addressed through backlog prioritization
* Communication issues: Handled through improved ceremony structures

**Challenges encounter and how you overcame them  
1.Firebase Authentication key exposure  
Problem:** The firebase-key.json contains sensitive private keys and is exposed in the codebase.  
Security Risk: High - Anyone with access to this key can perform unauthorized operations on your Firebase project.  
**Solutions:**  
**bash# Immediate Actions:**  
1. Revoke the current service account key in Firebase Console  
2. Generate a new service account key  
3. Remove firebase-key.json from version control  
4. Add firebase-key.json to .gitignore  
**# Environment Variable Approach:**export FIREBASE\_PRIVATE\_KEY="-----BEGIN PRIVATE KEY-----\n..."  
export FIREBASE\_CLIENT\_EMAIL=[your-service-account@project.iam.gserviceaccount.com](mailto:your-service-account@project.iam.gserviceaccount.com)  
export FIREBASE\_PROJECT\_ID="your-project-id"  
Updated client.py:  
pythonimport os  
import json  
from firebase\_admin import credentials  
**# Use environment variables instead of** file  
private\_key = os.getenv('FIREBASE\_PRIVATE\_KEY').replace('\\n', '\n')  
cred\_dict = {  
 "type": "service\_account",  
 "project\_id": os.getenv('FIREBASE\_PROJECT\_ID'),  
 "private\_key": private\_key,  
 "client\_email": os.getenv('FIREBASE\_CLIENT\_EMAIL'),

# ... other required fields  
}  
cred = credentials.Certificate(cred\_dict)  
**2.** **Inconsistent Database Usage  
Problem:** Code mixes Firebase Realtime Database (rtdb) and Firestore (collection().stream()) operations. **Impact:** API endpoints will fail because they're using wrong database methods. **Solution:** Choose one database type and update all endpoints consistently:For Realtime Database (Current working approach):python# search.py - Fix the search endpoint@router.get("/")def search\_restaurants(q: str = Query(...)): all\_restaurants = rtdb.reference("restaurants").get() or {} results = [] for rid, rdata in all\_restaurants.items(): if q.lower() in rdata.get("name", "").lower(): results.append({\*\*rdata, "id": rid}) return results  
**3.** **Missing Input Validation  
Problem:** No validation for coordinates, menu prices, or service types.  
**Solution:** Add Pydantic validators:  
pythonfrom pydantic import validator

class Restaurant(BaseModel):  
 # ... existing fields

@validator('latitude')  
 def validate\_latitude(cls, v):  
 if not -90 <= v <= 90:  
 raise ValueError('Latitude must be between -90 and 90')  
 return v  
   
 @validator('longitude')  
 def validate\_longitude(cls, v):  
 if not -180 <= v <= 180:  
 raise ValueError('Longitude must be between -180 and 180')  
 return v

**Security Recommendations**  
Immediate: Revoke and regenerate Firebase service account key  
Environment Variables: Store all sensitive data in environment variables  
API Rate Limiting: Implement rate limiting to prevent abuse  
Input Sanitization: Validate and sanitize all user inputs  
HTTPS Only: Ensure API only accepts HTTPS requests in production  
CORS Configuration: Properly configure CORS for your frontend domains  
  
**Deployment Checklist**Revoke old Firebase keys  
Set up environment variables  
Remove sensitive files from repository  
Implement proper authentication  
Add comprehensive error handling  
Set up monitoring and logging  
Configure rate limiting  
Test all endpoints thoroughly  
**4. Duplicate Rating Endpoints**  
**Problem:** Two different rating systems exist (/ratings/ and /search/rate/).  
**Solution:** Consolidate into one consistent rating system.  
**5. Missing Error Handling**  
**Problem:** Limited error handling for file uploads and Firebase operations.  
**Solution:** Add comprehensive try-catch blocks:  
pythontry:  
 upload\_result = cloudinary.uploader.upload(profile\_pic.file)  
except Exception as e:  
 raise HTTPException(status\_code=500, detail=f"Image upload failed: {str(e)}")  
**6. Cloudinary API Keys Exposed**  
**Problem:** Cloudinary credentials are hardcoded in restaurants.py.  
pythonimport os  
cloudinary.config(  
 cloud\_name=os.getenv('CLOUDINARY\_CLOUD\_NAME'),  
 api\_key=os.getenv('CLOUDINARY\_API\_KEY'),  
 api\_secret=os.getenv('CLOUDINARY\_API\_SECRET')

**Problems of Frontend Development**  
**1.Layout & Styling Issues**  
**Problem**  
Achieving consistent UI across various screen sizes and platforms required extra testing and platform-specific adjustments.  
**Solution**  
We used Flexbox effectively and incorporated tools like Dimensions and SafeAreaView to handle different screen sizes. We also leveraged responsive design libraries like react-native-responsive-screen and conditional styling based on the Platform.

**2. Navigation Complexity**  
**Problem**  
Setting up and managing navigation with React Navigation, especially nested navigators and parameter passing, proved tricky.  
**Solution**  
We carefully followed the documentation for React Navigation, used NavigationContainer and Stack.Navigator properly, and modularized the navigation structure. Debugging with deep linking and using hooks like useNavigation() improved flow management.  
**3. Performance Optimization**  
**Problem**  
Rendering large lists and ensuring smooth UI interactions demanded performance tuning and careful use of optimization techniques.  
**Solution**  
For large lists, we optimized FlatList using props like initialNumToRender, maxToRenderPerBatch, and memoization with React.memo. We also avoided unnecessary re-renders and used lazy loading techniques where needed.  
**4. State Management**   
**Problem**  
As the app scaled, managing state across components became complex, leading us to explore tools like Redux and Context API.  
**Solution**  
We chose appropriate state management tools depending on complexity—Context API for small-scale state and Redux or Zustand for larger apps. Using selectors and modular state slices helped keep the logic clean and efficient.  
**5. Library Compatibility**  
**Problem**  
Some third-party libraries were outdated or incompatible, causing functionality and build issues.  
**Solution**  
We reviewed the library’s GitHub issues, community support, and last update before integrating it. Incompatibilities were resolved by downgrading to stable versions or switching to better-maintained alternatives.  
**6. Debugging Difficulties**  
**Problem**  
Errors, especially from native modules, were often unclear and time-consuming to trace.  
**Solution**  
We relied on tools like React Native Debugger, Flipper, and console.log() tracing to identify and resolve issues. For native errors, checking Xcode/Android Studio logs and using clean builds (npx react-native-clean-project) helped a lot.

**7. Cross-Platform Differences**  
**Problem**  
Differences between Android and iOS (like UI behavior and styling) required platform-specific logic.  
**Solution**  
We used the Platform module to write conditional logic and style components differently where needed. Also, we conducted regular testing on both iOS and Android to catch discrepancies early.  
**8. Animation Challenges  
Problem**Creating smooth, complex animations needed deep knowledge of animation libraries and performance considerations.  
**Solution**  
We used react-native-reanimated and react-native-gesture-handler for smooth animations and gestures. Breaking down animations into smaller steps and testing frame rate performance helped ensure responsiveness.  
**9. Testing Effort**  
**Problem**  
UI bugs varied by device and OS, requiring extensive manual testing and some automation setup.  
**Solution**  
We created a manual test checklist covering multiple devices and OS versions. For automation, we integrated Jest for unit testing and explored Detox for end-to-end UI testing.  
**10. Design Constraints**  
**Problem**  
Aligning with both iOS and Android design guidelines while maintaining a unified look added extra complexity.  
**Solution**  
We followed both Material Design (Android) and Human Interface Guidelines (iOS) to ensure native feel. When needed, we customized styles conditionally based on platform to maintain a polished and platform-appropriate look.

**Product Backlog**

Here's the product backlog presented in a structured tabular format:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **PBI**  **ID** | **User Story** | **Description** | **Story Points** | **Sprint** | **Dependencies** |  |
| **Core Platform Functionality** | | | | | | |
| PBI 001 | Browse Nearby Restaurants(Guest) | Implement Restaurant discovery with map/list views | 8 | 1 | None | Map pins with basic info; Cuisine/price filters; Sync between map/list views |
| PBI 002 | Secure Payment Integration | PCI-Compliant payment system | 13 | 1-2 | None | 3+ gateways; Tokenization; Error recovery |
| PBI 003 | Guest Booking System | Identity-Verified bookings without registration | 8 | 2 | PBI-002 | QR confirmation; Payment-on-delivery; Email receipts |
| PBI 004 | Table Booking (Registered) | Filtered Reservations with real-time availability | 8 | 2 | PBI-001 | Cuisine/rating filters; Calendar integration; Notifications |
| **Restaurant Management** | | | | | | |
| PBI 005 | Menu/Subscription Management | Owner portal for menu updates & subscriptions | 13 | 3 | None | Real-time menu editing; Plan management; Access controls |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **PBI ID** | **User Story** | **Description** | **Story Points** | **Sprints** | **Dependencies** | **Acceptance  Criteria** |
| PBI 006 | Account Compliance (Admin) | User account Moderation tools | 8 | 3 | None | Suspension/deletion; Audit logs; Policy enforcement |
| PBI 007 | System Health Monitoring | Performance Tracking dashboard | 8 | 3-4 | None | Uptime stats; Error alerts; Historical reports |
| **User Engagement** | | | | | | |
| PBI 008 | Reviews/Photos (Guest) | User-Generated content system | 5 | 4 | PBI-001 | Star ratings; Photo gallery; Verified submissions |
| PBI 009 | Meal/Restaurant Ratings | Post-visit Feedback mechanism | 5 | 4 | PBI-004 | Separate meal/venue ratings; Editable history |
| PBI 010 | Personalized Recommendations | ML-driven Suggestion engine | 13 | 5 | PBI-009 | Preference-based results; Explainable AI |
| PBI 011 | Targeted Ads (Owners) | Campaign Management system | 8 | 5 | PBI-005 | Geofencing; A/B testing; Performance analytics |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **PBI ID** | **User Story** | **Description** | **Story points** | **Sprints** | **Dependencies** | **Acceptance criteria** |
| PBI 012 | Restaurant Analytics | Owner Performance dashboard | 8 | 5-6 | PBI-009/PBI-011 | Trend visualization; Dish popularity; Data export |
| PBI 013 | Delivery Tracking | Real-time order status | 8 | 6 | PBI-003 | Live GPS; ETA updates; Driver communication |
| PBI 014 | Premium Features | Exclusive offers/pre- orders | 5 | 6 | PBI-004/PBI-010 | Push notifications; Early access; VIP badges |
| PBI 015 | Ad Pricing (Admin) | Campaign Monetization controls | 5 | 6-7 | PBI-011 | CPC adjustments; Fraud detection; Billing sync |
| **Additional Features** | | | | | | |
| PBI 016 | Favorites Sharing (Premium) | Social Wishlist system | 3 | 7 | PBI-004 | Collection management; View-only links; Social integration |
| PBI 017 | Offline Mode | Cached data access | 8 | 7-8 | PBI001/PBI-004 | Local storage; Sync recovery; Queued bookings |