Documentation Structure Plan

1. Introduction

1.1 Goal

The primary goal of the RestoSam system is to revolutionize the restaurant dining experience by eliminating traditional pain points such as long waits for service, inefficient table booking, and suboptimal food ordering processes. The system aims to provide a seamless, technology-driven solution that enhances customer satisfaction while improving operational efficiency for restaurants.

1.2 Objectives

The objectives of the RestoSam system include streamlining restaurant operations through automated processes for food ordering, payment processing, and waiter notification systems to reduce wait times and improve service quality. The system aims to enhance customer experience by enabling customers to pre-order food and book tables in advance, view interactive restaurant maps, access multimedia menus, and receive personalized recommendations based on their preferences. Resource management is optimized through systems for automatic order reassignment to minimize food waste and provide real-time table availability management. The system is designed to handle 150+ concurrent users per restaurant during peak hours, with the ability to scale from 1 restaurant at launch to 10 restaurants within several months. Security and compliance are maintained through robust measures and adherence to Russian payment systems and accessibility standards. Finally, the system facilitates business intelligence through analytics dashboards and preference mapping systems to help restaurants make data-driven decisions and provide personalized customer experiences.

---

2. Description of the Model

2.1 Data Ingestion & Integration

The RestoSam system employs a comprehensive data ingestion pipeline that collects data from multiple sources including customer data such as user profiles, order history, preferences, and behavioral patterns collected through mobile and web applications. Restaurant data includes menu items, pricing, availability, table layouts, and operational metrics from restaurant management systems. Payment data consists of transaction records from integrated payment providers such as SBP, Mir, Visa, and other Russian banking systems. Real-time data includes live table availability, order status updates, and waiter notifications. Third-party APIs provide integration with mapping services and social media platforms for enhanced functionality. Data integration is achieved through RESTful APIs and gRPC protocols, ensuring real-time synchronization across all system components.

2.2 Data Storage & Management

The system utilizes a hybrid data storage architecture with PostgreSQL serving as the primary database for structured data including user accounts, orders, restaurant information, and transactional data. MongoDB provides flexible storage for menu items, customer preferences, and multimedia content. Redis serves as the cache layer for high-performance caching of frequently accessed data such as table availability and menu items. Automated backup systems, data validation, and integrity checks ensure data consistency and reliability.

2.3 AI-Driven Insights & Predictive Analytics

The system incorporates machine learning algorithms to provide intelligent recommendations through preference mapping that tracks user behavior to create detailed preference profiles based on order history, ratings, and interaction patterns. The recommendation engine offers AI-powered suggestions for food items based on past orders, dietary preferences, and seasonal availability. Predictive analytics enables forecasting of restaurant occupancy, popular menu items, and optimal staffing levels. Dynamic menu adaptation provides real-time menu updates based on ingredient availability and customer demand patterns.

2.4 Business Intelligence & Visualization

Business intelligence capabilities include comprehensive reporting tools for restaurant administrators to track performance metrics, customer behavior, and operational efficiency. Interactive visualizations display revenue trends, popular dishes, peak hours, and customer demographics. Real-time monitoring provides live dashboards showing current table occupancy, order status, and service performance. Custom reporting enables flexible report generation for business planning and decision-making.

2.5 Security & Compliance

The security framework encompasses multi-factor authentication with support for email/password and social media login through Yandex and Google. Payment security ensures compliance with PCI DSS standards and integration with secure Russian payment systems. Data encryption provides end-to-end encryption for sensitive customer data and payment information. Access control implements role-based access for different user types including customers, restaurant staff, and administrators. Audit logging provides comprehensive logging of all system activities for compliance and security monitoring.

---

3. Approach: Software Development Models

3.1 Comparison of Development Models

Model	Time	Quality	Cost	Docs	Extens.	Scale	Sec/Comp	Total	Avg
Build-and-Fix	6	3	3	1	3	2	2	20	**2.9**
Waterfall	3	6	4	9	4	5	7	38	**5.4**
Spiral	5	8	4	8	7	7	8	47	**6.7**
Incremental	8	9	7	7	9	8	9	57	**8.1**
Rapid Proto.	9	7	6	5	6	6	6	45	**6.4**

Abbrev: Docs = Documentation, Extens. = Extensibility, Scale = Scalability, Sec/Comp = Security & Compliance; scores 0–10 (higher is better).

The comparison shows that Incremental provides high flexibility with changes between increments, risks addressed per increment, testing after each increment, regular customer feedback, moderate documentation, and fast time-to-market for a working MVP—making it the best fit for RestoSam. Waterfall offers clear, document-driven discipline but low flexibility, testing largely at the end, and longer time to market; it is suitable when requirements are fixed and compliance documentation is paramount. Spiral emphasizes strong risk management and high quality via iterative elaboration, but adds process overhead and cost, so it fits large internal projects with high risk. Rapid Prototyping validates requirements very quickly and aligns to customer needs early, but it can encourage throwaway or shortcut code and lighter documentation, which may hurt maintainability and compliance if not controlled. Build-and-Fix may be fast for trivial tasks, but it is unsuitable for a non-trivial, compliant system due to poor documentation, quality, and security control.

3.2 Selected Model

The Incremental Development Model has been selected for the RestoSam project.

3.3 Alternative Approach

An alternative approach would be the Waterfall Model, which follows a sequential design process where progress flows downward through phases like conception, initiation, analysis, design, construction, testing, implementation, and maintenance.

3.4 Justification for Not Choosing Other Models

The Waterfall Model was not selected because it requires complete requirements upfront, which is challenging for restaurant systems where user needs frequently change. Testing only occurs at the end of development, increasing risk of major issues. No working software is delivered until the final phase, delaying business value realization. Changes are expensive and difficult to implement once a phase is complete.

3.5 Model Benefits

The Incremental Model provides several advantages including early delivery of value where basic functionality like table booking and ordering can be deployed early. Risk reduction occurs as issues are identified and resolved in smaller increments. Flexibility allows changes to be incorporated based on user feedback. Stakeholder satisfaction is maintained through regular deliveries that keep engagement and allow for course corrections.

3.6 Benefits of the Selected Model

The selected Incremental Model provides business value as restaurants can start using core booking functionality while additional features are developed. Quality assurance is ensured as each increment undergoes testing, guaranteeing quality at every stage. The model accommodates the dynamic nature of restaurant operations and customer preferences through adaptability. Cost control is achieved as budget and resources can be allocated more efficiently across increments.

---

4. Methodologies

4.1 Comparison of Software Development Methodologies

4.1.1 Comparison Table of Methodologies

(a) RUP

Parameter	Score (1 to 5)	Justification
Flexibility and Adaptability	3	Well-structured and adaptable through phases, but heavier governance slows changes compared to Agile.
Documentation Management	5	Emphasizes comprehensive artifacts (vision, SRS, architecture docs), ideal for auditability.
Handling Regulatory Requirements	5	Phase gates and required evidence support PCI DSS and security controls.
Team Collaboration	3	Clear roles/processes support collaboration, but ceremony is higher and slower for a 6–9 person team.

(b) Scrum

Parameter	Score (1 to 5)	Justification
Flexibility and Adaptability	5	Time-boxed sprints and backlog refinement handle changing restaurant requirements and MVP evolution.
Documentation Management	3	Just-enough documentation; compliance needs can be covered via Definition of Done templates.
Handling Regulatory Requirements	4	DevSecOps in the sprint (threat modeling, SAST/DAST, evidence in tickets) supports PCI DSS integrations.
Team Collaboration	5	Strong roles (PO, SM, Dev Team), frequent ceremonies, and transparency fit the 6–9 cross-functional team.

(c) XP

Parameter	Score (1 to 5)	Justification
Flexibility and Adaptability	5	Embraces change with continuous planning and small releases.
Documentation Management	2	Prefers code and tests over documents; may be insufficient for compliance trails.
Handling Regulatory Requirements	3	High engineering rigor (TDD, CI) helps quality, but lacks required documentation by default.
Team Collaboration	4	Pair programming and on-site customer improve feedback, but may stress availability and team preference.

(d) Kanban

Parameter	Score (1 to 5)	Justification
Flexibility and Adaptability	4	Continuous flow with WIP limits adapts well, but lacks sprint cadence for fixed milestones.
Documentation Management	3	Process-light; docs discipline must be enforced externally.
Handling Regulatory Requirements	3	Can include compliance checks in the workflow, but provides fewer built-in control points.
Team Collaboration	4	Visual boards and pull-based work enhance collaboration; roles are lightweight.

Overall Methodology Ratings

Methodology	Pros	Cons	Final Score
RUP	- Strong documentation and governance - Excellent for regulated environments and audit trails	- Heavier process for a 6–9 team - Slower to adapt, higher overhead	**4.0**
Scrum	- Highest adaptability and focus on working increments - Excellent team collaboration and transparency	- Requires discipline to maintain compliance documentation within sprints	**4.25**
XP	- Strong engineering practices (TDD, refactoring, CI) improve quality - Rapid feedback	- Minimal documentation and on-site customer expectations may not be feasible	**3.5**

Methodology	Pros	Cons	Final Score
Kanban	- Great visibility and flow; ideal for ops/maintenance and continuous delivery	- Lacks time-boxing; weaker for fixed milestones without additional structure	**3.5**

Taken together, Scrum best matches RestoSam's needs: a small cross-functional team delivering a working MVP quickly, adapting to changing restaurant and payment requirements, and integrating DevSecOps activities each sprint to satisfy PCI-DSS and security evidence. RUP remains a viable choice when auditability must dominate, but its ceremony would slow early iterations. XP provides superb code quality but its documentation lightness and on-site customer expectations are less practical here. Kanban complements Scrum for production support and DevOps flow but, by itself, offers less predictability for milestone-driven delivery.

4.2 Selected Methodology

Scrum has been selected as the primary software development methodology for the RestoSam project.

4.2.1 Benefits

The benefits of Scrum include adaptive planning through time-boxed sprints of 2-4 weeks that allow the team to adapt to changing restaurant requirements and customer feedback. Clear roles and responsibilities are provided through distinct roles including Product Owner, Scrum Master, and Development Team ensuring accountability and efficient decision-making. Incremental delivery is achieved through regular sprint deliveries that ensure working software increments providing immediate business value to restaurant operations. Transparency is maintained through daily stand-ups, sprint reviews, and retrospectives for continuous improvement. The methodology supports cross-functional teams with the diverse skill set required for frontend, backend, mobile, DevOps, and QA development in a multi-platform restaurant system. Risk management is enhanced through short sprints and frequent feedback loops that help identify and mitigate risks early in the development process.

---

5. Constraints

5.1 Technical Constraints

The technical constraints include payment integration that must work with Russian payment systems such as SBP, Mir, Visa, and major banking providers. Platform support requires native iOS and Android apps, responsive web interface, and web-based admin dashboard. Real-time processing is essential for handling order updates, table booking changes, and waiter notifications. The architecture must support 150+ concurrent users per restaurant during peak hours. Third-party API restrictions require all APIs to be accessible from Russia with appropriate localization. Performance requirements demand response time optimization for mobile networks and varying internet speeds. Data synchronization must occur in real-time across mobile, web, and backend systems.

5.2 Security & Compliance Constraints

Security and compliance constraints include data protection compliance with Russian regulations and international standards. Payment security requires PCI DSS compliance for handling payment information. User privacy demands secure handling of personal data, location information, and payment details. Access control implements role-based permissions for customers, restaurant staff, and administrators. Audit requirements necessitate comprehensive logging for regulatory compliance and security monitoring. Encryption standards require end-to-end encryption for sensitive data transmission and storage.

5.3 Organizational & Cost Constraints

Organizational and cost constraints include a team size of 6-9 members with specific roles such as Team Lead, Scrum Master, Product Manager, Web Frontend, 2 Backend developers, Mobile developer, DevOps, SRE, and QA. The timeline requires launching with 1 restaurant and scaling to 10 restaurants within several months. Budget limitations require a cost-effective technology stack while maintaining quality and scalability. The business culture demands an innovative environment with low bureaucracy requiring agile decision-making. Resource availability is limited for specialized talent in restaurant technology domain. Market competition requires differentiation from existing restaurant booking and ordering systems.

---

6. Quality Attributes

6.1 Key Attributes

Extensibility

RestoSam uses modular microservices, allowing independent development and deployment of features. The API-first design enables easy integration of external services such as payment providers or loyalty programs without affecting existing functionality.

Maintainability

Code quality is ensured through clean architecture, documentation, and automated testing. CI pipelines verify every commit with unit and integration tests. Modular design isolates changes to individual services, simplifying debugging and updates.

Security and Compliance

Security follows DevSecOps principles. The system enforces multi-factor authentication (MFA), secure sessions, and role-based access control. Data is encrypted with TLS 1.3 in transit and AES-256 at rest. Regular scans, penetration tests, and audit logs ensure PCI DSS and Russian data-protection compliance.

Scalability

RestoSam supports horizontal and vertical scaling using containerized microservices orchestrated by Kubernetes. Microservices architecture supporting independent scaling of components, database sharding and caching strategies for performance, and load balancing and auto-scaling capabilities.

Reliability and Performance

The system guarantees 99.9 % uptime with fault-tolerant infrastructure and automated failover Health checks and retry logic prevent cascading failures. Average response time stays under 1 second, with real-time

monitoring to detect bottlenecks. Backup and disaster-recovery plans ensure rapid restoration of services and zero data loss in critical scenarios.

6.2 DevOps

RestoSam applies a unified DevOps pipeline for CI/CD, and monitoring.

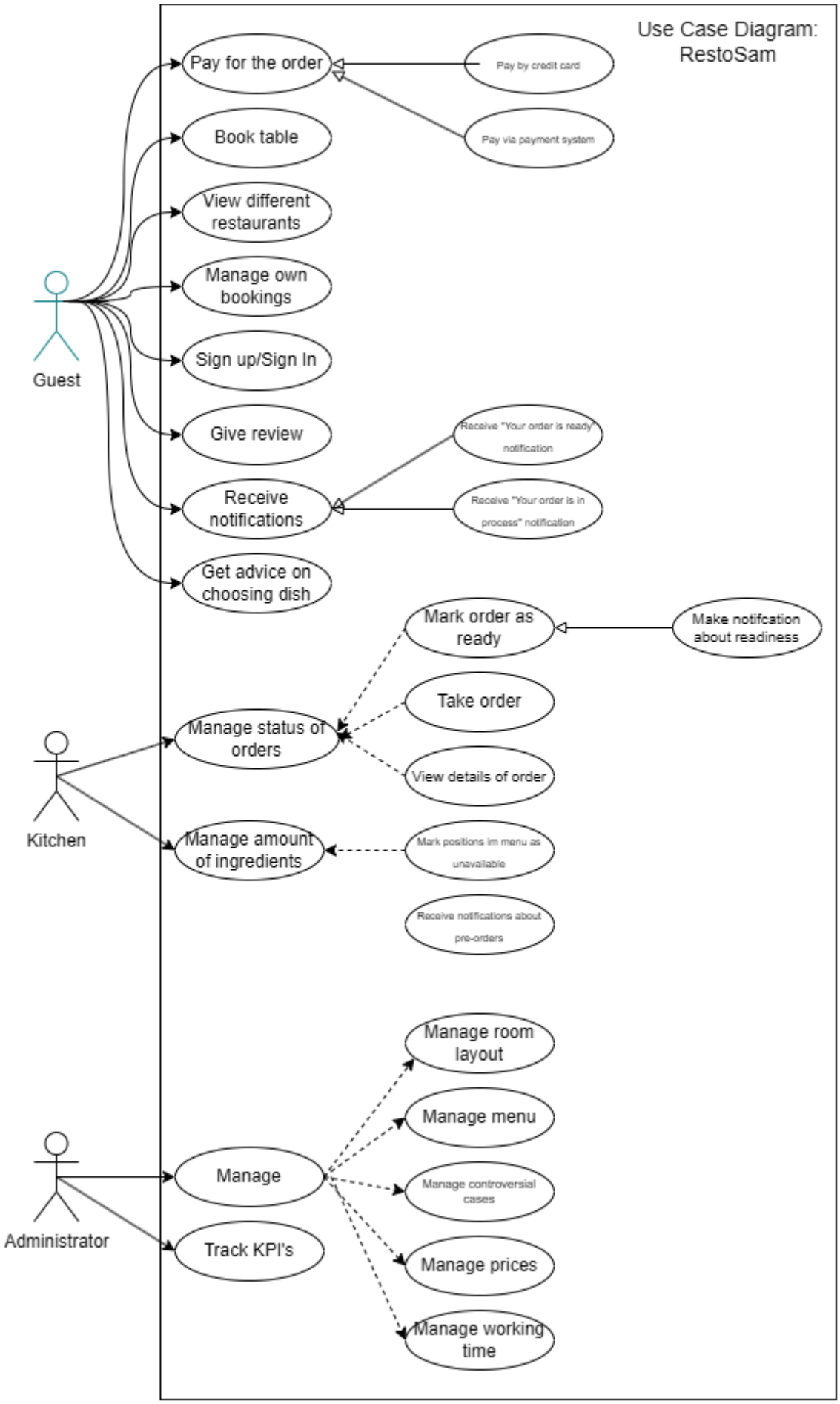
- CI/CD: automated build, test, and deployment with rollback.
- Monitoring: Prometheus, Grafana.
- Security: SAST/DAST scans in CI/CD detect vulnerabilities early.
- Collaboration: shared responsibility of dev, QA, and ops teams.

This approach ensures quick, reliable releases and stable operation.

---

7. Diagrams

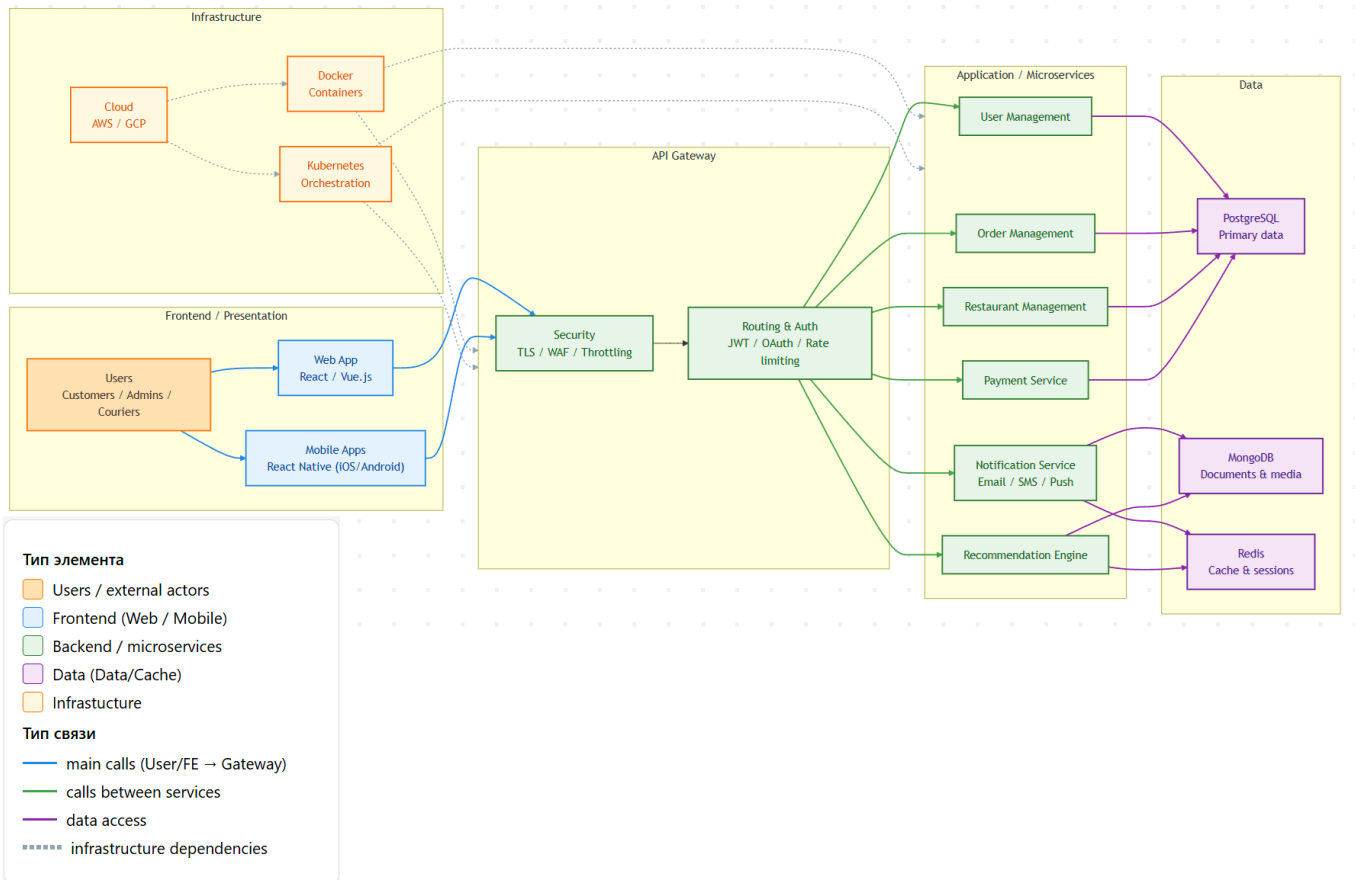
7.1 Use Case Diagram



This use case diagram shows three actors (Guest, Kitchen, Administrator) and their interactions with the system. Guests browse restaurants, sign up/sign in, book tables and manage bookings, pay for orders (with credit card and payment-system options as extensions of the core payment use case), leave reviews, get

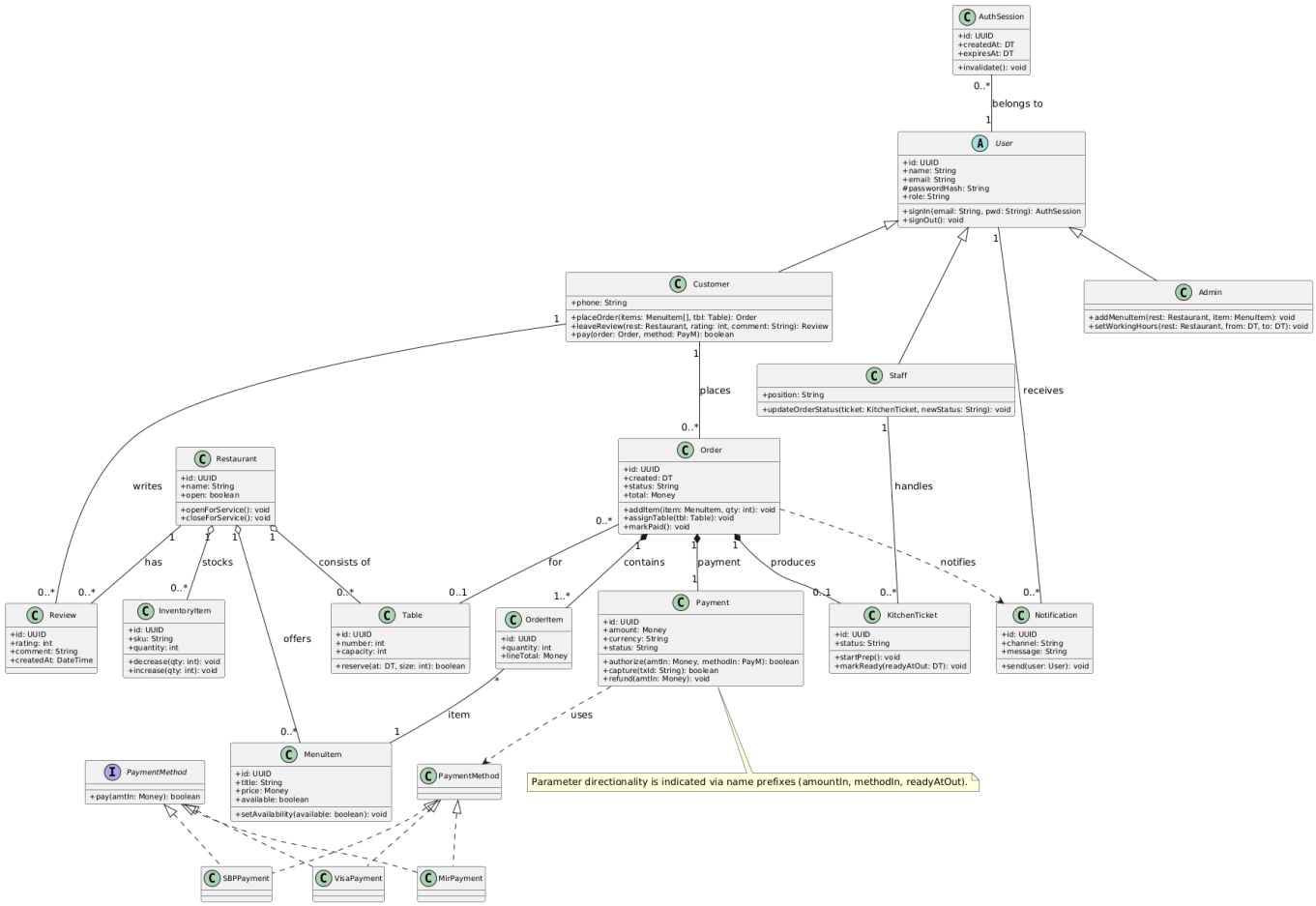
advice on choosing dishes, and receive status notifications such as “order in process” and “order is ready.” Kitchen staff take and view orders, update statuses and mark orders ready (triggering readiness notifications), and manage ingredient availability by marking menu items unavailable and planning stock based on pre-order notifications. Administrators maintain the room layout, menu, prices and working time, resolve controversial cases, and track KPIs. Associations indicate who performs which actions, while include/extend relationships show payment options and the notification flow attached to order management.

7.2 Architecture Diagram



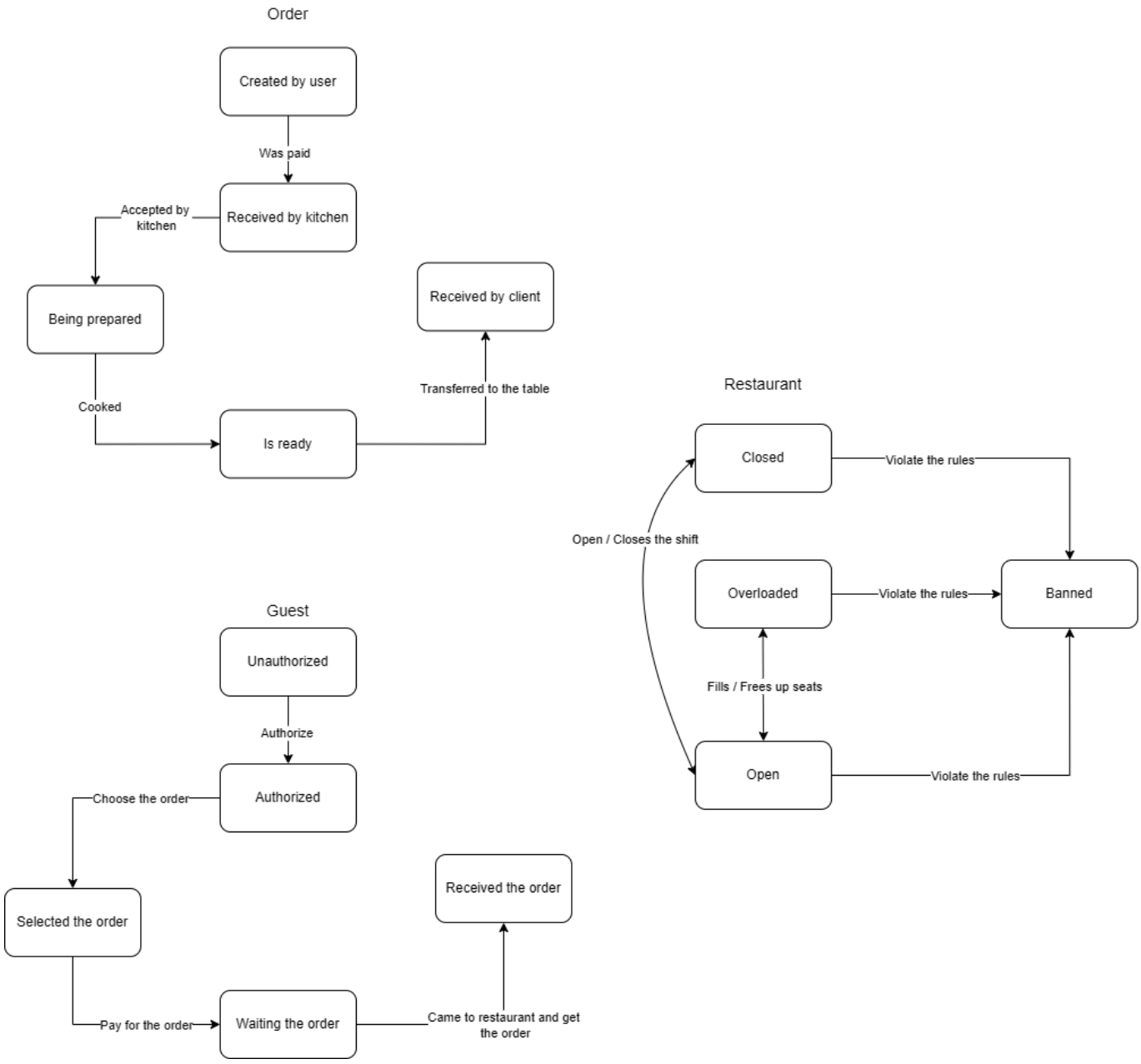
The system follows a microservices architecture with a frontend layer consisting of React or Vue.js web application and React Native mobile applications for iOS and Android. The API Gateway handles request routing and authentication with rate limiting and security. Microservices include User Management Service, Order Management Service, Restaurant Management Service, Payment Service, Notification Service, and Recommendation Engine Service. The data layer uses PostgreSQL for primary data, MongoDB for documents and media, and Redis for caching and sessions. Infrastructure includes Docker containers, Kubernetes orchestration, and cloud deployment on AWS or GCP.

7.3 Class Diagram



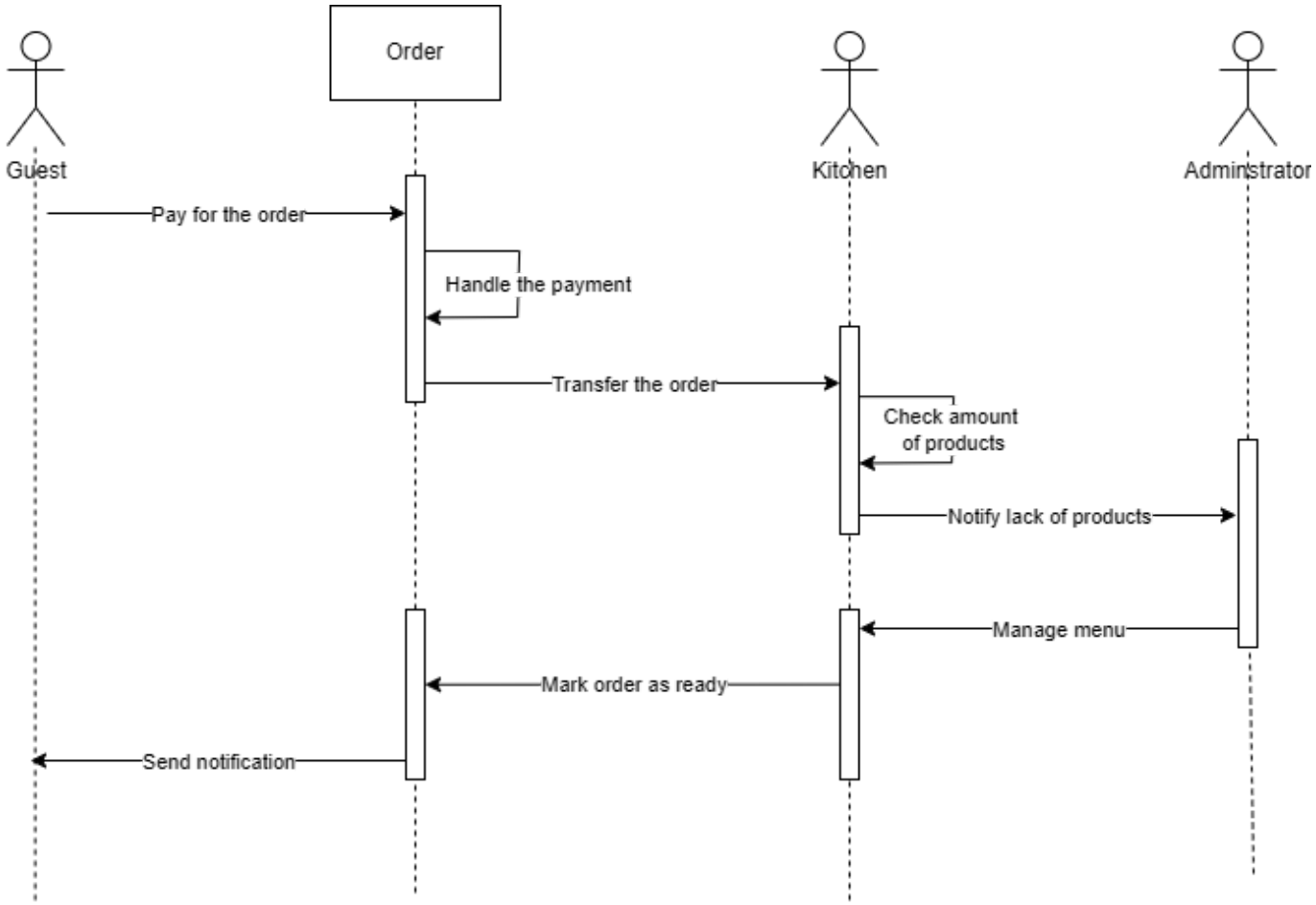
This diagram shows the core domain of RestoSam. Users are abstract and specialized as Customer, Staff, and Admin. Customers place Orders composed of OrderItems, optionally tied to a Table, and each Order has a Payment. Payments follow a strategy pattern via the PaymentMethod interface with concrete methods for SBP, Mir, and Visa. Restaurants aggregate Tables, MenuItems, and InventoryItems; Staff operate on KitchenTickets produced from Orders to drive preparation and readiness. Users receive Notifications, and Customers write Reviews that belong to Restaurants. AuthSession represents user sessions. Methods capture typical actions such as sign-in, reserving tables, adding items, marking orders paid/ready, and authorizing or refunding payments; associations include cardinalities and a mix of generalization, realization, composition/aggregation, and dependencies.

7.4 State Transition Diagram



This diagram depicts three parallel lifecycles. For orders: an order is created by the user, then paid, then received and accepted by the kitchen, then being prepared, then marked ready (cooked), then received by the client, then transferred to the table, and finally closed. The guest journey mirrors this with selecting an order, paying, waiting, arriving to pick it up, and confirming receipt. For restaurants: a restaurant starts open, can become overloaded as occupancy grows beyond planned capacity, shifts can set it to closed, and rule violations from the open or overloaded states move the restaurant to a banned state. For users and authorization: an unauthorized user becomes authorized after a successful sign in. Transitions are triggered by guest actions, kitchen updates, occupancy changes, and policy violations, and the system sends notifications at key steps such as order in process and order ready.

7.5 Sequence Diagram



This sequence diagram illustrates the end-to-end processing of an order across four participants: the guest using the client app, the order service, the kitchen subsystem, and the administrator console. The guest pays for the order and the order service handles the payment and records the outcome; once payment succeeds, the order service transfers the order for execution to the kitchen. The kitchen checks the amount of required products and either notifies a lack of products for resolution or proceeds to preparation. The administrator maintains the menu and product availability in parallel, which influences kitchen checks and prevents unavailable items from being ordered again. When preparation completes, the kitchen marks the order as ready and the order service sends a notification to the guest about the updated status. The interaction emphasizes payment confirmation before fulfillment, ingredient verification to avoid stock issues, and timely notifications that keep the guest informed.

---

8. Implementation, Testing & Maintenance

8.1 Implementation Plan

Technologies & Tools

Languages: Go, TypeScript, JavaScript, SQL

Frontend: React, React Native

Backend: Go (REST/gRPC), PostgreSQL, Redis, MongoDB

Version control: Git, GitHub

CI/CD: GitHub Actions

Infrastructure: Docker, Kubernetes, Nginx, GitHub Actions

Monitoring: Prometheus, Grafana

IDEs: GoLand, VS Code, Android Studio

Other: gRPC, JWT, OpenAPI

Phase 1 (Months 1-3) — Core Platform

- Set up cloud infrastructure (Kubernetes, CI/CD, monitoring)
- Implement API gateway, authentication (JWT)
- Build core services: User, Restaurant, Table, Menu
- Deliverable: API spec, dev environment

Phase 2 (Months 4-5) — Booking

- Hall plan editor
- Booking flow with table selection
- Real-time availability (Redis)
- Deliverable: Booking UI

Phase 3 (Months 6-7) — Pre-order & Payments

- Payment integration (Mir/SBP/Visa)
- Prepayment flows and refunds
- Multimedia menu system
- Deliverable: Secure payment flows, PCI DSS compliance

Phase 4 (Months 8-9) — Kitchen & Waiter Features

- Kitchen ticket system, order tracking
- Waiter call, bill split, in-place payment
- Offline resilience
- Deliverable: Kitchen dashboard, waiter notifications

Phase 5 (Months 10-11) — Recommendations & Analytics

- AI recommendation engine
- Restaurant search
- Admin dashboards
- Deliverable: Analytics

Phase 6 (Month 12) — Production

- Pilot with 1-3 restaurants
- Security audit
- Deliverable: Production release

Resources: Product Owner, Scrum Master, Backend (2), Frontend (2), Mobile (1), DevOps/SRE (1), QA (1-2)

Tools: Git, Docker, Kubernetes, GitHub Actions, Prometheus, Grafana

8.2 Testing Plan

Unit Testing

- Tools: Go testing, Testify, Jest
- Coverage: 80%+
- Run: Every PR in CI

API Testing

- Tools: Postman, httptest
- Run: After builds

Integration Testing

- Tools: TestContainers
- Run: Each phase

E2E Testing

- Tools: Selenium, Playwright
- Test: booking → payment → kitchen → delivery
- Run: Before release

Release Ready: All tests pass, 80%+ coverage, no critical bugs

8.3 Maintenance Plan

RestoSam requires three types of maintenance: Corrective, Adaptive, and Perfective, applied sequentially after each release.

- **Corrective Maintenance** focuses on resolving bugs affecting booking, payments, or kitchen workflows. Activities include issue reporting via Help Desk, prioritization, fixing, and regression testing.
- **Adaptive Maintenance** ensures compatibility with changing external systems such as Mir/SBP/Visa APIs, restaurant business rules, OS/browser updates, and regulatory changes. Activities include monitoring changes, requirements analysis, architecture adjustments, implementation, and testing.
- **Perfective Maintenance** improves performance, UX, and functionality, including enhancements to recommendation accuracy, booking flow, and dashboards. Activities include backlog grooming, planning, implementation, and validation.
- **Resources:** Backend (1–2), Frontend (1), Mobile (1), QA (1–2), DevOps/SRE (1), Product Owner, Support.
- **Methodology & SDLC:** Maintenance follows the same Incremental SDLC and Scrum methodology as development, with fixes and improvements integrated into 2–4-week sprints and hotfixes delivered as needed.

Appendix 1 - ADRs

ADR-001: Microservices Architecture Selection

Status: accepted

Context:

- **Technical constraints:** Real-time data synchronization across mobile, web, and backend components; support for multiple platforms (iOS, Android, Web); need for independent deployments
- **Non-functional requirements (NFRs):** High availability (99.5% uptime); sub-3-second response times for critical operations; PCI DSS compliance for payment processing
- **Business requirements:** Scale from supporting 1 restaurant to 10+ restaurants within several months; handle 150+ concurrent users per restaurant during peak hours
- **Risks and challenges:** Technology stack complexity; inter-service communication overhead; distributed system debugging challenges
- **Scale considerations:** Horizontal scaling from 1 to 10+ restaurants with independent service scaling capabilities
- **Timeline and resource constraints:** 6-9 person development team; timeline for scaling within several months

Decision: We will adopt a microservices architecture using Spring Boot for Java services, with API Gateway for request routing, service mesh for inter-service communication, and container orchestration with Kubernetes. Services will be independently deployable with domain-driven design boundaries.

Alternatives:

1. **Monolithic Architecture**: Considered for simpler initial development but rejected due to scalability limitations and inability to scale individual components independently.
2. **Serverless Architecture**: Evaluated for automatic scaling but rejected due to cold start latency issues unacceptable for real-time restaurant operations and higher operational complexity for the team.
3. **Modular Monolith**: Considered as a middle ground but rejected because it doesn't provide the independent deployment capabilities needed for scaling from 1 to 10+ restaurants.

Consequences: (+) Independent scaling and deployment of services (+) Technology diversity (can use different stacks per service) (+) Fault isolation and resilience (+) Team autonomy and faster development cycles (-) Increased operational complexity and monitoring overhead (-) Distributed system debugging challenges (-) Eventual consistency issues across services (-) Higher infrastructure and DevOps requirements

Links:

- Architecture Diagram: `image/documentation/1761672498075.png`
- Class Diagram: `image/documentation/1761672507726.png`
- Implementation: `/backend/services/`

ADR-002: Database Technology Selection

Status: accepted

Context:

- **Technical constraints:** Handling structured transactional data (orders, users, payments); unstructured content (menu items, customer preferences, multimedia); high-performance caching (table availability,

frequently accessed data); support for real-time queries and complex analytics
- **Non-functional requirements (NFRs):** Sub-3-second response times for critical operations; PCI DSS compliance for secure data handling; data consistency for financial transactions
- **Business requirements:** Scale from 1 to 10+ restaurants; evolving menu structures and user preferences requiring flexible schema
- **Risks and challenges:** Balancing data consistency with flexibility; managing multiple database technologies; data synchronization across different storage types
- **Scale considerations:** Horizontal scaling across multiple restaurants while maintaining performance and consistency
- **Timeline and resource constraints:** Need to support rapid scaling from single restaurant to multi-restaurant operations

Decision: We will implement a hybrid data storage architecture with PostgreSQL as the primary relational database for transactional data, MongoDB for flexible document storage, and Redis for high-performance caching and session management.

Alternatives:

1. **Single PostgreSQL Database**: Considered for ACID compliance but rejected due to poor performance with unstructured data and lack of built-in caching capabilities.
2. **Single MongoDB Database**: Evaluated for flexibility but rejected due to weaker transactional guarantees needed for payment processing and complex relational queries.
3. **Traditional RDBMS Only**: Considered for maturity but rejected due to scaling limitations and inability to handle multimedia content efficiently.

Consequences: (+) Optimal data storage for each data type (+) Strong consistency for financial transactions (+) Flexible schema for evolving requirements (+) High-performance caching capabilities (-) Increased operational complexity (-) Data synchronization challenges (-) Multiple technology stacks to maintain (-) Higher infrastructure costs

Links:

- Data Architecture: Section 2.2 of `documentation.md`
- Implementation: `/backend/infrastructure/database/`

ADR-003: Frontend Technology Stack

Status: accepted

Context:

- **Technical constraints:** Native mobile apps for iOS and Android; responsive web interface; admin dashboard; real-time updates; complex user interactions; offline capabilities
- **Non-functional requirements (NFRs):** Consistent user experience across platforms; rapid feature development support
- **Business requirements:** Cross-platform mobile and web presence; admin dashboard functionality
- **Risks and challenges:** Limited mobile development experience; learning curve for cross-platform development; platform-specific limitations
- **Scale considerations:** Support for multiple platforms with shared codebase and consistent user experience

- **Timeline and resource constraints:** 6-9 person team with web expertise but limited mobile experience; budget constraints limiting hiring of specialized mobile developers

Decision: We will use React Native for cross-platform mobile development (iOS and Android) and React for web interfaces, with a shared component library and state management layer.

Alternatives:

1. **Native iOS/Android Development**: Considered for optimal performance but rejected due to higher development costs, longer time-to-market, and maintenance complexity with separate codebases.
2. **Progressive Web Apps (PWA)**: Evaluated for unified codebase but rejected due to limited native device access and poorer user experience for complex restaurant interactions.
3. **Flutter**: Considered as alternative cross-platform solution but rejected due to team's existing React expertise and larger ecosystem of React libraries.

Consequences: (+) Single codebase for mobile platforms (+) Faster development and maintenance (+) Consistent UI/UX across platforms (+) Access to native device features (-) Performance overhead compared to native apps (-) Potential platform-specific limitations (-) Dependency on React Native ecosystem stability (-) Steeper learning curve for complex native integrations

Links:

- Mobile App Requirements: Section 5.1 of `documentation.md`
- Implementation: `/frontend/mobile/`, `/frontend/web/`

ADR-004: Development Methodology Selection

Status: accepted

Context:

- **Technical constraints:** Cross-functional team with diverse skills (frontend, backend, mobile, DevOps, QA); agile culture with low bureaucracy
- **Non-functional requirements (NFRs):** Regulatory compliance (PCI DSS) requiring proper documentation and audit trails; quality and security standards maintenance
- **Business requirements:** Evolving restaurant requirements based on user feedback and market conditions; incremental delivery of working software every 2-4 weeks
- **Risks and challenges:** Managing changing requirements; balancing speed with quality and compliance; team coordination across diverse skill sets
- **Scale considerations:** Growing from MVP to full system while maintaining delivery cadence
- **Timeline and resource constraints:** 6-9 person team; MVP delivery within months; 2-4 week delivery cycles

Decision: We will use Scrum as the primary development methodology with 2-4 week sprints, supplemented by DevSecOps practices for security and compliance integration.

Alternatives:

1. **Waterfall**: Considered for comprehensive documentation but rejected due to inability to handle changing restaurant requirements and delayed delivery of working software.

2. **Kanban**: Evaluated for continuous flow but rejected due to need for time-boxed deliveries and sprint-based planning for stakeholder management.
3. **XP (Extreme Programming)**: Considered for code quality practices but rejected due to team size constraints and lack of on-site customer availability.

Consequences: (+) Adaptability to changing requirements (+) Regular delivery of working software (+) Clear roles and accountability (+) Early identification and mitigation of risks (-) Requires discipline for documentation (-) Overhead of ceremonies for small team (-) Need for Product Owner availability (-) Sprint planning overhead

Links:

- Methodology Comparison: Section 4.1 of `documentation.md`
- Implementation Plan: Section 8.1 of `documentation.md`