Technical Specifications

Yardura Admin

1. INTRODUCTION

1.1 EXECUTIVE SUMMARY

1.1.1 Project Overview

The Yardura Service OS represents a comprehensive end-to-end operations platform designed specifically for dog-waste service companies. This system consolidates fragmented business operations into a unified, modern solution that addresses critical inefficiencies in routing, tracking, billing, and client engagement while introducing innovative wellness insights capabilities.

1.1.2 Core Business Problem

Current dog-waste service operations suffer from fragmented systems where routing is inefficient, shifts and mileage are under-tracked, billing and payroll are error-prone, and client experiences lack transparency and differentiation. Pet owners demand transparency through photos, status updates, and reminders, while seeking useful wellness trends without diagnostic claims.

1.1.3 Key Stakeholders and Users

User Type	Primary Responsibilities	Key Benefits
Business Owners/Managers	Pricing, plans, payroll, reports, cross-sells, franchises	Operational efficiency, revenue optimization
Dispatchers	Schedule board, route optimization, reassignments, weather skips	Streamlined dispatch operations, real-time visibility

User Type	Primary Responsibilities	Key Benefits
Field Technicians	Shift tracking, job completion, photos/notes, navigation	Mobile-first experience, simplified workflows
Accountants	Invoices, payments, refunds, QuickBooks sync	Automated financial processes, accurate reporting
Clients (Residential/Commercial)	Service proofs, wellness insights, billing management	Enhanced transparency, valuable pet health trends
Franchise Owners	Multi-account oversight, brand consistency, royalty management	Scalable business model, centralized control

1.1.4 Expected Business Impact and Value Proposition

The system delivers measurable improvements through reduced drive time and missed appointments, accurate billing and payroll processing, faster cash collection, and higher retention rates. The unique client-facing Wellness Insights feature increases retention and average revenue per user (ARPU), while API/webhooks integration and PWA mobile capabilities enable rapid iteration, lower operational costs, and franchise scalability.

1.2 SYSTEM OVERVIEW

1.2.1 Project Context

Business Context and Market Positioning

The Yardura Service OS positions itself as a premium alternative to existing solutions like Sweep&Go by combining operational depth with modern technology stack advantages. The system leverages Next.js 15, Stripe integration, QuickBooks synchronization, and mapping services to deliver superior performance and user experience compared to legacy systems.

Current System Limitations

Existing solutions in the market suffer from:

- Fragmented operational workflows requiring multiple disconnected systems
- Limited client engagement and transparency features
- Inadequate mobile experiences for field technicians
- Poor integration capabilities with modern business tools
- Lack of innovative features that differentiate service offerings

Integration with Existing Enterprise Landscape

The system integrates with critical business infrastructure through Stripe for payment processing and subscription management, QuickBooks Online for accounting synchronization, utilizing the latest QuickBooks API minor version 75, and various communication platforms including Twilio for SMS/voice and email services for comprehensive business operations.

1.2.2 High-Level Description

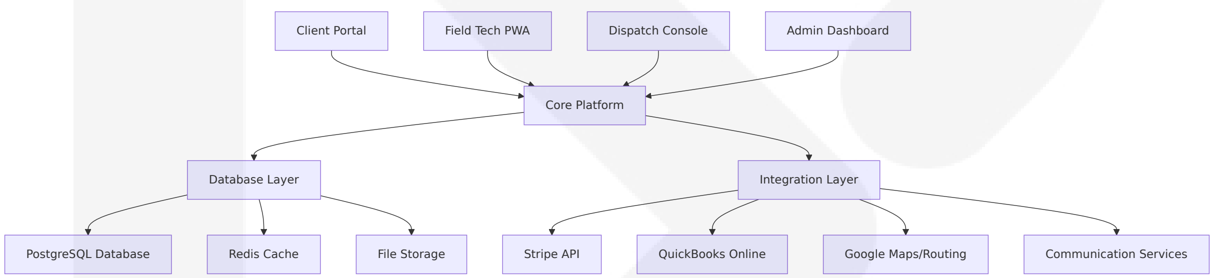
Primary System Capabilities

The Yardura Service OS encompasses the complete business lifecycle from lead generation through service delivery and ongoing client management. Core capabilities include:

- **Lead Management & Quoting**: Preserves existing quote flow and pricing estimator while integrating with CRM systems
- **Client Onboarding**: Streamlined subscription setup with Stripe integration and automated scheduling

- **Dispatch & Routing**: Intelligent route optimization with real-time updates and weather-based adjustments
- **Field Operations**: PWA-based mobile application for technicians with offline capabilities
- **Client Portal**: Comprehensive self-service platform with proof photos and wellness insights
- **Financial Management**: Automated billing, payroll processing, and QuickBooks synchronization
- **Franchise Support**: Multi-tenant architecture supporting brand consistency and royalty management

Major System Components



Core Technical Approach

The system utilizes Next.js 15 with App Router architecture, TypeScript, and modern React patterns. The backend leverages Next.js API routes with Prisma ORM for database operations, while implementing BullMQ for job queuing and Redis for caching and session management. The architecture emphasizes API-first design with comprehensive webhook support for real-time integrations.

1.2.3 Success Criteria

Measurable Objectives

Metric Category	Target Performance	Measurement Method
System Performance	Portal P95 < 500ms cached, Route optimization ≤ 10s for 100 stops	Application monitoring and performance testing
User Experience	Tech app job list render < 200ms from cache, Image upload ≤ 3s on LTE	Mobile performance metrics and user feedback
Business Impact	Reduced drive time, improved billing accuracy, faster cash collection	Operational KPIs and financial reporting
System Reliability	99.9% uptime for staff and client portals	Infrastructure monitoring and SLA tracking

Critical Success Factors

- Seamless preservation of existing quote flow and pricing estimator functionality
- Successful integration with Stripe and QuickBooks Online APIs
- Effective mobile-first experience for field technicians
- Robust wellness insights implementation with appropriate disclaimers
- Scalable multi-tenant architecture supporting franchise operations

Key Performance Indicators (KPIs)

- **Operational Efficiency**: Route completion rates, average service time, missed appointment reduction
- **Financial Performance**: Billing accuracy, collection rates, payroll processing time
- **Client Satisfaction**: Portal usage rates, wellness insights engagement, retention metrics
- **System Adoption**: User login frequency, feature utilization rates, mobile app installation rates

1.3 SCOPE

1.3.1 In-Scope

Core Features and Functionalities

Lead Management & CRM Integration

- Preservation of existing Quote flow & Pricing Estimator with exact implementation
- CRM lead creation with comprehensive tagging and metadata
- Lead-to-customer conversion workflows

Client Onboarding & Subscription Management

- ZIP-based eligibility and zone pricing
- Stripe-powered subscription setup with card-on-file requirements
- Cross-sell upsell opportunities during onboarding
- Terms of service acceptance and portal credential provisioning

Dispatch & Route Management

- Automated job creation from recurring schedules and one-time requests
- Advanced route optimization with real-time ETA calculations
- Mass weather skip functionality with client notifications
- Reschedule, reassign, reclean, and recreate job capabilities

Field Operations (PWA)

- Clock in/out with break tracking and odometer recording
- GPS tracking during active shifts
- Job completion with photo requirements and private notes
- Skip reason documentation with billing implications
- "On the way" client notifications

Client Portal & Wellness Insights

- Service proof photo galleries with timeline view
- Wellness Insights featuring 3Cs (Color/Consistency/Content) analysis
- Moisture/Weight/Frequency trend tracking with non-diagnostic guidance
- Subscription management and billing history access
- Notification preference management

Financial Management

- Automated recurring and one-time invoice generation
- Stripe webhook integration for subscription coordination and payment processing
- QuickBooks Online synchronization using API minor version 75
- Payroll processing with multiple compensation models
- Open balance management and dunning processes

Multi-Tenant & Franchise Support

- Parent account oversight with subaccount switching
- Brand consistency controls and consolidated payment processing
- ACH royalty collection with percentage or minimum-based rules
- Franchise-specific reporting and analytics

Primary User Workflows

1. **Quote to Lead Conversion**: Existing wizard completion → API validation → CRM lead creation → success confirmation
2. **Client Onboarding**: Account creation → Stripe Setup Intent → subscription activation → first visit scheduling
3. **Daily Dispatch**: Job generation → route optimization → device distribution → real-time monitoring
4. **Field Service Delivery**: Shift start → job execution → photo documentation → completion reporting

5. **Wellness Insight Generation**: Photo analysis → 3C classification → trend calculation → client notification

Essential Integrations

- **Stripe API**: Payment processing, subscription management, webhook handling
- **QuickBooks Online API**: Customer synchronization, invoice management, payment recording
- **Google Maps/Distance Matrix API**: Route optimization and navigation services
- **Twilio API**: SMS notifications and voice communication
- **Email Services**: Automated client communications and system notifications

Key Technical Requirements

- Next.js 15 App Router with TypeScript and modern React patterns
- PostgreSQL database with Prisma ORM for data management
- Redis caching for performance optimization and session management
- PWA implementation for offline-capable mobile field operations
- Comprehensive API and webhook architecture for third-party integrations

1.3.2 Implementation Boundaries

System Boundaries

The Yardura Service OS operates as a comprehensive business management platform specifically designed for dog-waste service companies. The system boundary encompasses all operational aspects from initial client contact through ongoing service delivery and financial management.

User Groups Covered

- Business owners and managers across all franchise levels
- Dispatch personnel responsible for daily operations coordination
- Field technicians using mobile devices for service delivery
- Administrative staff handling billing and payroll functions
- Residential and commercial clients accessing self-service portals
- Franchise owners requiring multi-location oversight capabilities

Geographic and Market Coverage

The system supports operations across multiple geographic regions with ZIP-based service area management. Market coverage includes residential and commercial client segments with scalable pricing models and service tier differentiation.

Data Domains Included

- Customer relationship management and lead tracking
- Service scheduling and route optimization data
- Financial transactions and subscription management
- Employee time tracking and payroll information
- Service delivery documentation and photo evidence
- Wellness trend analysis and client health insights
- Franchise operations and royalty management data

1.3.3 Out-of-Scope

Explicitly Excluded Features and Capabilities

Advanced Analytics and Business Intelligence

- Complex predictive analytics beyond basic trend reporting
- Advanced machine learning algorithms for demand forecasting
- Sophisticated business intelligence dashboards with custom report builders

Third-Party Marketplace Integrations

- Integration with pet supply e-commerce platforms
- Veterinary clinic management system connections
- Pet insurance provider API integrations

Advanced Communication Features

- In-app messaging between clients and technicians
- Video calling capabilities for remote consultations
- Social media integration and marketing automation

Inventory Management

- Product inventory tracking and management
- Supply chain optimization features
- Vendor management and procurement systems

Future Phase Considerations

Phase 2 Enhancements (120-180 days)

- React Native mobile application development
- Advanced routing heuristics with machine learning optimization
- Staff performance scorecards and gamification features
- Zapier integration and hardened public API expansion

Phase 3 Expansion (180+ days)

- Franchise theming and white-label customization
- Advanced marketing integrations (Mailchimp, HubSpot)
- International market support with multi-currency capabilities
- Advanced wellness insights with veterinary partner integrations

Integration Points Not Covered

- Direct veterinary clinic system integrations

- Pet microchip registry connections
- Municipal licensing and permit management systems
- Insurance provider claim processing integrations

Unsupported Use Cases

- Multi-service business operations beyond dog-waste management
- Complex multi-location routing across different service types
- Advanced inventory management for retail operations
- Direct veterinary diagnostic capabilities or medical advice provision

2. PRODUCT REQUIREMENTS

2.1 FEATURE CATALOG

2.1.1 Core Business Operations Features

Feature ID	Feature Name	Category	Priority	Status
F-001	Quote & Lead Management System	Lead Generation	Critical	Proposed
F-002	Client Onboarding & Subscription Setup	Customer Management	Critical	Proposed
F-003	Dispatch & Route Management	Operations	Critical	Proposed
F-004	Field Technician PWA	Mobile Operations	Critical	Proposed
F-005	Client Portal & Service Proofs	Customer Experience	High	Proposed
F-006	Wellness Insights Platform	Value-Added Services	High	Proposed

Feature ID	Feature Name	Category	Priority	Status
F-007	Billing & Invoice Management	Financial Operations	Critical	Proposed
F-008	Payroll Processing System	Human Resources	High	Proposed
F-009	Multi-Tenant & Franchise Management	Business Scaling	Medium	Proposed
F-010	Cross-Sell Management	Revenue Optimization	Medium	Proposed

2.1.2 Integration & Technical Features

Feature ID	Feature Name	Category	Priority	Status
F-011	Stripe Payment Integration	Payment Processing	Critical	Proposed
F-012	QuickBooks Online Synchronization	Accounting Integration	Critical	Proposed
F-013	Google Maps Route Optimization	Mapping Services	Critical	Proposed
F-014	Communication Services Integration	Notifications	High	Proposed
F-015	Public API & Webhook System	External Integration	Medium	Proposed
F-016	Reporting & Analytics Dashboard	Business Intelligence	High	Proposed

2.2 FUNCTIONAL REQUIREMENTS

2.2.1 F-001: Quote & Lead Management System

Feature Description

Overview: Preserves existing Quote flow & Pricing Estimator exactly as implemented while integrating with CRM lead creation functionality using QuickBooks Online API minor version 75.

Business Value: Maintains proven conversion funnel while enabling seamless lead-to-customer progression and CRM integration.

User Benefits: Familiar quoting experience with enhanced lead tracking and follow-up capabilities.

Technical Context: Built on Next.js 15 App Router with React 19 support and React Compiler optimizations.

Dependencies

Dependency Type	Description	Requirements
Prerequisite Features	None (foundational feature)	N/A
System Dependencies	Next.js 15, TypeScript, Prisma ORM	Latest stable versions
External Dependencies	CRM API integration	API endpoints and authentication
Integration Requirements	Lead data normalization and validation	Standardized lead schema

Functional Requirements Table

Requirement ID	Description	Acceptance Criteria	Priority	Complexity
F-001-RQ-001	Preserve existing quote wizard flow	All existing wizard steps, validations, and calculations remain identical	Must-Have	Low
F-001-RQ-002	Implement `/api/quote` CRM integration	API returns `leadId` upon successful lead creation	Must-Have	Medium
F-001-RQ-003	Lead data normalization	Standardize frequency, dogs, yard size, areas, monthly total, initial clean data	Must-Have	Medium
F-001-RQ-004	Wellness waitlist flag handling	Capture and store wellness interest during quote process	Should-Have	Low
F-001-RQ-005	Quote success page integration	Display `leadId` and next steps on success page	Must-Have	Low

Technical Specifications

Input Parameters:

- Service frequency selection
- Property details (yard size, areas)
- Dog count and details
- Schedule preferences
- Add-on services selection
- Wellness insights interest flag

Output/Response:

- Validated quote data

- Generated `leadId`
- Success confirmation
- Analytics event triggers

Performance Criteria:

- Quote calculation: < 100ms
- API response time: < 500ms
- Success page load: < 200ms

Data Requirements:

- Lead entity with comprehensive metadata
- Quote history tracking
- Analytics event logging
- CRM integration audit trail

Validation Rules

Business Rules:

- ZIP-based service area validation
- Pricing tier determination (Regular/Premium zones)
- Service frequency constraints
- Add-on service compatibility

Data Validation:

- Required field validation
- Format validation for contact information
- Service area eligibility verification
- Pricing calculation accuracy

Security Requirements:

- Input sanitization and validation
- Rate limiting on quote submissions

- PII data protection
- Audit logging for quote submissions

Compliance Requirements:

- Data retention policy compliance
- Privacy policy acceptance tracking
- Terms of service acknowledgment

2.2.2 F-002: Client Onboarding & Subscription Setup

Feature Description

Overview: Streamlined client onboarding process with Stripe integration using the latest 2025-08-27.basil API version for subscription management and payment processing.

Business Value: Reduces onboarding friction while ensuring secure payment setup and subscription activation.

User Benefits: Simple account creation with transparent pricing and immediate service scheduling.

Technical Context: Stripe Setup Intent integration with card-on-file requirements and subscription lifecycle management.

Dependencies

Dependency Type	Description	Requirements
Prerequisite Features	F-001 (Quote & Lead Management)	Lead data and pricing information
System Dependencies	Stripe API, Next.js authentication	Stripe SDK, NextAuth configuration

Dependency Type	Description	Requirements
External Dependencies	Payment processing, email services	Stripe webhooks, email delivery
Integration Requirements	Subscription management, scheduling	Calendar integration, job creation

Functional Requirements Table

Requirement ID	Description	Acceptance Criteria	Priority	Complexity
F-002-RQ-001	Account creation workflow	User can create account with email/password	Must-Have	Low
F-002-RQ-002	Stripe Setup Intent integration	Secure card storage with PCI compliance	Must-Have	High
F-002-RQ-003	ZIP-based eligibility validation	Service area verification before onboarding	Must-Have	Medium
F-002-RQ-004	Zone pricing implementation	Automatic pricing tier assignment (Regular/Premium)	Must-Have	Medium
F-002-RQ-005	Cross-sell opportunity presentation	Display relevant add-on services during onboarding	Should-Have	Medium
F-002-RQ-006	Terms of service acceptance	Legal agreement acceptance with timestamp	Must-Have	Low
F-002-RQ-007	First visit scheduling	Automatic scheduling of initial service visit	Must-Have	Medium

2.2.3 F-003: Dispatch & Route Management

Feature Description

Overview: Comprehensive dispatch system with automated job creation, intelligent route optimization, and real-time monitoring capabilities.

Business Value: Maximizes operational efficiency through optimized routing and reduces missed appointments.

User Benefits: Streamlined dispatch operations with real-time visibility and flexible reassignment capabilities.

Technical Context: Leverages Next.js 15 performance improvements and build optimizations with Google Maps integration for route calculation.

Dependencies

Dependency Type	Description	Requirements
Prerequisite Features	F-002 (Client Onboarding), F-004 (Field Tech PWA)	Active subscriptions, technician assignments
System Dependencies	Google Maps API, Redis caching	Route optimization algorithms, real-time updates
External Dependencies	Weather services, GPS tracking	Weather API, location services
Integration Requirements	Job scheduling, technician communication	Calendar sync, push notifications

Functional Requirements Table

Requirement ID	Description	Acceptance Criteria	Priority	Complexity
F-003-RQ-001	Automated job creation	Generate jobs from recurring schedules and one-time requests	Must-Have	Medium

Requirement ID	Description	Acceptance Criteria	Priority	Complexity
F-003-RQ-002	Route optimization engine	Optimize routes for up to 100 stops within 10 seconds	Must-Have	High
F-003-RQ-003	Real-time ETA calculations	Dynamic ETA updates based on traffic and progress	Must-Have	High
F-003-RQ-004	Mass weather skip functionality	Bulk skip jobs with automated client notifications	Must-Have	Medium
F-003-RQ-005	Job reassignment capabilities	Reschedule, reassign, reclean, and recreate job functions	Must-Have	Medium
F-003-RQ-006	Dispatch board interface	Visual job management with drag-and-drop functionality	Must-Have	High
F-003-RQ-007	Real-time progress monitoring	Live tracking of technician progress and job completion	Should-Have	High

2.2.4 F-004: Field Technician PWA

Feature Description

Overview: Progressive Web Application for field technicians with offline capabilities, GPS tracking, and comprehensive job management features.

Business Value: Improves field operations efficiency and provides accurate time tracking and service documentation.

User Benefits: Mobile-first experience with offline functionality and streamlined job completion workflows.

Technical Context: PWA implementation with service workers for offline functionality and background synchronization.

Dependencies

Dependency Type	Description	Requirements
Prerequisite Features	F-003 (Dispatch & Route Management)	Job assignments and route information
System Dependencies	PWA capabilities, GPS services	Service workers, geolocation API
External Dependencies	Camera access, navigation services	Device permissions, mapping integration
Integration Requirements	Real-time synchronization, photo storage	WebSocket connections, cloud storage

Functional Requirements Table

Requirement ID	Description	Acceptance Criteria	Priority	Complexity
F-004-RQ-001	Clock in/out functionality	Track shift start/end with break management	Must-Have	Medium
F-004-RQ-002	Odometer recording	Capture vehicle mileage for personal/company vehicles	Must-Have	Low
F-004-RQ-003	GPS tracking during shifts	Continuous location tracking while clocked in	Must-Have	Medium
F-004-RQ-004	Job completion workflow	Photo requirements, private notes, completion timestamps	Must-Have	High
F-004-RQ-005	Skip reason documenta	Categorized skip reasons with billi	Must-Have	Medium

Requirement ID	Description	Acceptance Criteria	Priority	Complexity
	tion	ng implications		
F-004-RQ-006	"On the way" notifications	Automated client notifications when technician en route	Must-Have	Medium
F-004-RQ-007	Offline functionality	Queue actions when offline, sync when connected	Should-Have	High
F-004-RQ-008	Navigation integration	Direct navigation to job locations	Must-Have	Low

2.2.5 F-005: Client Portal & Service Proofs

Feature Description

Overview: Comprehensive client self-service portal with service proof photos, subscription management, and billing access.

Business Value: Enhances client transparency and reduces support overhead through self-service capabilities.

User Benefits: Complete visibility into service history with convenient account management features.

Technical Context: Responsive web interface with secure photo galleries and subscription management integration.

Dependencies

Dependency Type	Description	Requirements
Prerequisite Features	F-004 (Field Tech PWA), F-007 (Billing Management)	Service photos, billing data

Dependency Type	Description	Requirements
System Dependencies	Authentication system, file storage	Secure photo access, subscription data
External Dependencies	Email services, payment processing	Notification delivery, payment updates
Integration Requirements	Stripe integration, photo management	Subscription sync, secure file access

Functional Requirements Table

Requirement ID	Description	Acceptance Criteria	Priority	Complexity
F-005-RQ-001	Service proof photo gallery	Timeline view of service photos with metadata	Must-Have	Medium
F-005-RQ-002	Subscription management interface	View, pause, and cancel subscription requests	Must-Have	Medium
F-005-RQ-003	Billing history access	Complete invoice and payment history	Must-Have	Low
F-005-RQ-004	Payment method management	Add, update, remove payment methods (cannot remove last)	Must-Have	Medium
F-005-RQ-005	Notification preference management	Configure SMS, email, and phone notification preferences	Should-Have	Low
F-005-RQ-006	Service schedule visibility	View upcoming, completed, skipped, and missed services	Must-Have	Low

Requirement ID	Description	Acceptance Criteria	Priority	Complexity
F-005-RQ-007	Dog and yard information management	Update pet and property details with photos	Should-Have	Medium

2.2.6 F-006: Wellness Insights Platform

Feature Description

Overview: Innovative wellness insights feature providing 3Cs analysis (Color/Consistency/Content) with moisture, weight, and frequency trend tracking.

Business Value: Differentiates service offering and increases client retention and ARPU through value-added insights.

User Benefits: Valuable pet health trend information with appropriate non-diagnostic guidance.

Technical Context: Photo analysis system with trend calculation algorithms and client-facing dashboard.

Dependencies

Dependency Type	Description	Requirements
Prerequisite Features	F-004 (Field Tech PWA), F-005 (Client Portal)	Service photos, client access
System Dependencies	Photo analysis algorithms, data storage	Image processing, trend calculations
External Dependencies	None (internal processing)	N/A
Integration Requirements	Photo metadata, client notifications	Image analysis pipeline, alert system

Functional Requirements Table

Requirement ID	Description	Acceptance Criteria	Priority	Complexity
F-006-RQ-001	3Cs classification system	Automated Color, Consistency, Content analysis	Must-Have	High
F-006-RQ-002	Moisture/Weight/Frequency tracking	Trend analysis with timeline visualization	Must-Have	High
F-006-RQ-003	Non-diagnostic guidance	Appropriate disclaimers and "consult veterinarian" messaging	Must-Have	Low
F-006-RQ-004	Photo gallery integration	Wellness insights linked to service proof photos	Must-Have	Medium
F-006-RQ-005	Opt-in anonymized data toggle	Client control over data sharing for research	Should-Have	Low
F-006-RQ-006	Waitlist status management	"90 days free then $59.99/mo" pricing display	Must-Have	Low
F-006-RQ-007	Trend notification system	Alert clients to significant changes in patterns	Should-Have	Medium

2.2.7 F-007: Billing & Invoice Management

Feature Description

Overview: Comprehensive billing system with automated recurring invoices, Stripe webhook integration using the latest API version, and QuickBooks synchronization.

Business Value: Ensures accurate billing, reduces manual processing, and improves cash flow management.

User Benefits: Transparent billing with automated processing and multiple payment options.

Technical Context: Integration with QuickBooks Online API minor version 75 for accounting synchronization.

Dependencies

Dependency Type	Description	Requirements
Prerequisite Features	F-002 (Client Onboarding), F-011 (Stripe Integration)	Active subscriptions, payment methods
System Dependencies	Stripe API, QuickBooks API, job queuing	Payment processing, accounting sync
External Dependencies	Banking systems, tax services	Payment settlement, tax calculations
Integration Requirements	Subscription management, dunning processes	Automated billing cycles, collection workflows

Functional Requirements Table

Requirement ID	Description	Acceptance Criteria	Priority	Complexity
F-007-RQ-001	Automated recurring invoice generation	Generate invoices based on subscription schedules	Must-Have	Medium
F-007-RQ-002	One-time and initial invoice creation	Support for setup fees and ad-hoc charges	Must-Have	Medium

Requirement ID	Description	Acceptance Criteria	Priority	Complexity
F-007-RQ-003	Skip-reason pricing integration	Apply billing rules based on service skip reasons	Must-Have	Medium
F-007-RQ-004	Payment processing and refunds	Handle successful payments and refund requests	Must-Have	High
F-007-RQ-005	Open balance management	Track and manage outstanding balances	Must-Have	Medium
F-007-RQ-006	Dunning process automation	Automated collection workflows for failed payments	Must-Have	High
F-007-RQ-007	QuickBooks synchronization	Sync customers, invoices, and payments with QBO	Must-Have	High

2.2.8 F-008: Payroll Processing System

Feature Description

Overview: Flexible payroll system supporting multiple compensation models including hourly, bonus, and commission structures.

Business Value: Streamlines payroll processing and ensures accurate compensation calculations.

User Benefits: Transparent pay calculations with detailed pay slip exports.

Technical Context: Integration with time tracking data and performance metrics for comprehensive payroll management.

Dependencies

Dependency Type	Description	Requirements
Prerequisite Features	F-004 (Field Tech PWA), F-003 (Dispatch Management)	Time tracking data, job completion records
System Dependencies	Time tracking system, job performance data	Accurate shift records, completion metrics
External Dependencies	Payroll services, tax calculation	Third-party payroll providers, tax compliance
Integration Requirements	HR systems, accounting integration	Employee records, financial reporting

Functional Requirements Table

Requirement ID	Description	Acceptance Criteria	Priority	Complexity
F-008-RQ-001	Multiple compensation models	Support hourly, hourly+bonus, and commission structures	Must-Have	High
F-008-RQ-002	Mileage compensation tracking	Calculate mileage reimbursement for personal/company vehicles	Must-Have	Medium
F-008-RQ-003	Overtime calculation	Automatic overtime calculation based on labor laws	Must-Have	Medium
F-008-RQ-004	Complaints counter integration	Track customer complaints impact on compensation	Should-Have	Low
F-008-RQ-005	Pay slip export functionality	Generate detailed pay slips in CSV/PDF formats	Must-Have	Medium

Requirement ID	Description	Acceptance Criteria	Priority	Complexity
F-008-RQ-006	Approval workflow	Manager approval process for payroll processing	Must-Have	Low
F-008-RQ-007	Performance metrics integration	Link job completion rates to bonus calculations	Should-Have	Medium

2.2.9 F-009: Multi-Tenant & Franchise Management

Feature Description

Overview: Scalable multi-tenant architecture supporting franchise operations with brand consistency and consolidated payment processing.

Business Value: Enables franchise scalability with centralized control and automated royalty management.

User Benefits: Streamlined multi-location oversight with consistent branding and operations.

Technical Context: Multi-tenant database design with role-based access control and franchise-specific configurations.

Dependencies

Dependency Type	Description	Requirements
Prerequisite Features	All core features (F-001 through F-008)	Complete operational system
System Dependencies	Multi-tenant architecture, RBAC system	Tenant isolation, permission management

Dependency Type	Description	Requirements
External Dependencies	ACH processing, banking integration	Automated royalty collection
Integration Requirements	Brand management, consolidated reporting	Theming system, aggregated analytics

Functional Requirements Table

Requirement ID	Description	Acceptance Criteria	Priority	Complexity
F-009-RQ-001	Parent account oversight	View and manage multiple franchise locations	Must-Have	High
F-009-RQ-002	Brand consistency controls	Enforce consistent branding across franchises	Should-Have	Medium
F-009-RQ-003	Consolidated payment processing	Centralized payment handling with franchise allocation	Must-Have	High
F-009-RQ-004	ACH royalty collection	Automated royalty collection by percentage or minimum	Must-Have	High
F-009-RQ-005	Franchise-specific reporting	Individual and consolidated reporting capabilities	Must-Have	Medium
F-009-RQ-006	Subaccount switching interface	Easy navigation between franchise locations	Must-Have	Low
F-009-RQ-007	Royalty rule configuration	Flexible royalty calculation and collection rules	Must-Have	Medium

2.2.10 F-010: Cross-Sell Management

Feature Description

Overview: Comprehensive cross-sell system for additional products and services with commission tracking and completion management.

Business Value: Increases revenue per customer through strategic upselling and cross-selling opportunities.

User Benefits: Access to additional services with transparent pricing and scheduling.

Technical Context: Integration with billing system and technician workflows for seamless cross-sell execution.

Dependencies

Dependency Type	Description	Requirements
Prerequisite Features	F-002 (Client Onboarding), F-007 (Billing Management)	Customer base, billing integration
System Dependencies	Product catalog, pricing engine	Service definitions, tax calculations
External Dependencies	Inventory management (if applicable)	Stock tracking for physical products
Integration Requirements	Technician workflows, commission tracking	Field operations, payroll integration

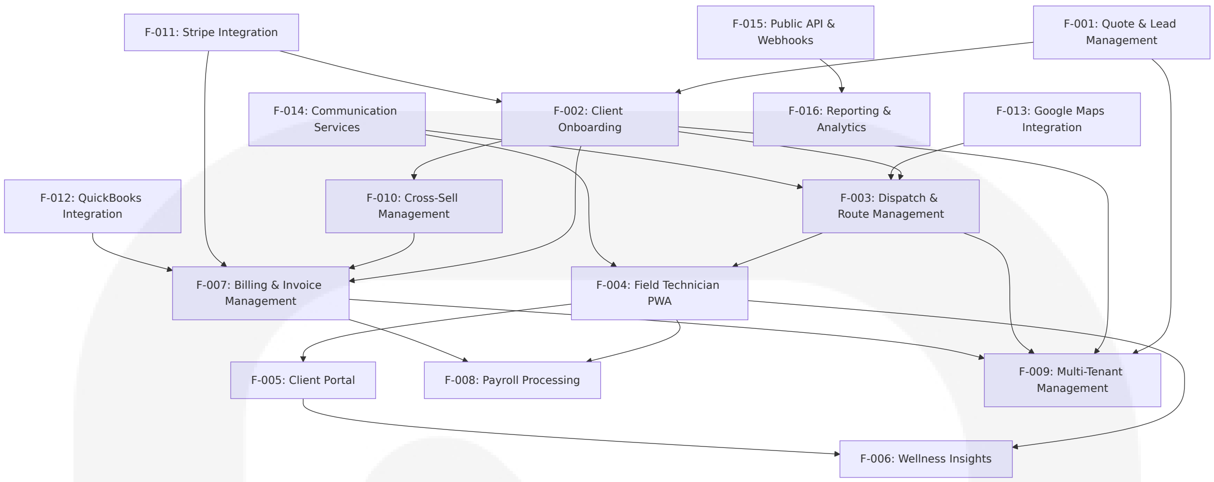
Functional Requirements Table

Requirement ID	Description	Acceptance Criteria	Priority	Complexity
F-010-RQ-001	Product and service catalog	Maintain catalog of cross-sell offerings with pricing	Must-Have	Medium

Requirement ID	Description	Acceptance Criteria	Priority	Complexity
F-010-RQ-002	Taxable flag configuration	Configure tax applicability for different offerings	Must-Have	Low
F-010-RQ-003	Onboarding display integration	Present relevant cross-sells during client onboarding	Should-Have	Medium
F-010-RQ-004	Per-client request management	Track and manage individual client cross-sell requests	Must-Have	Medium
F-010-RQ-005	Frequency and scheduling	Support recurring and one-time cross-sell services	Must-Have	Medium
F-010-RQ-006	Completion and billing integration	Link cross-sell completion to billing and invoicing	Must-Have	High
F-010-RQ-007	Commission tracking and reporting	Track sales performance and commission calculations	Should-Have	Medium

2.3 FEATURE RELATIONSHIPS

2.3.1 Feature Dependencies Map



2.3.2 Integration Points

Integration Point	Connected Features	Shared Components	Common Services
Payment Processing	F-002, F-007, F-009, F-011	Stripe SDK, Payment Forms	Payment validation, Webhook handling
Job Management	F-003, F-004, F-008	Job entities, Status tracking	Scheduling engine, Time tracking
Client Data	F-001, F-002, F-005, F-006	Customer profiles, Preferences	Authentication, Data synchronization
Photo Management	F-004, F-005, F-006	File storage, Image processing	Upload service, Security validation
Reporting Data	F-008, F-009, F-010, F-016	Analytics engine, Data aggregation	Report generation, Export services

2.3.3 Shared Components

Component	Used By Features	Purpose	Technical Requirements
Authentication System	All features	User identity and access control	NextAuth, RBAC, Session manageme

Component	Used By Features	Purpose	Technical Requirements
			nt
Database Layer	All features	Data persistence and retrieval	PostgreSQL, Prisma ORM, Connection pooling
File Storage Service	F-004, F-005, F-006	Photo and document management	S3/R2, Signed URLs, CDN integration
Notification Engine	F-003, F-004, F-006, F-007	Multi-channel communication	Twilio, Email services, Push notifications
Job Queue System	F-003, F-007, F-008, F-012	Background task processing	BullMQ, Redis, Worker processes

2.4 IMPLEMENTATION CONSIDERATIONS

2.4.1 Technical Constraints

Constraint Category	Description	Impact	Mitigation Strategy
Performance	Portal P95 < 500ms cached, Route optimization ≤ 10s for 100 stops	User experience, operational efficiency	Caching strategy, Algorithm optimization
Mobile Performance	Tech app job list render < 200ms, Image upload ≤ 3s on LTE	Field operations efficiency	PWA optimization, Offline capabilities
API Rate Limits	Stripe, QuickBooks, Google Maps API limitations	Integration reliability	Rate limiting, Retry mechanisms, Caching

Constraint Category	Description	Impact	Mitigation Strategy
Data Storage	Photo storage costs and access patterns	Operational costs	Signed URLs, CDN optimization, Lifecycle policies

2.4.2 Performance Requirements

Feature	Performance Metric	Target	Measurement Method
F-001	Quote calculation response time	< 100ms	API monitoring
F-003	Route optimization processing	≤ 10s for 100 stops	Algorithm benchmarking
F-004	Job list rendering	< 200ms from cache	Mobile performance testing
F-005	Portal page load time	P95 < 500ms cached	Web vitals monitoring
F-006	Wellness insight calculation	< 5s per analysis	Processing time tracking

2.4.3 Scalability Considerations

Aspect	Current Requirement	Scaling Strategy	Implementation Notes
Multi-tenancy	Franchise support	Tenant isolation, Shared infrastructure	Database partitioning, Resource allocation
Geographic Distribution	Multiple service areas	Regional deployment, CDN usage	Edge computing, Data locality
User Concurrency	Field technicians, clients, staf	Horizontal scaling, Load balanci	Stateless architecture, Session man

Aspect	Current Requirement	Scaling Strategy	Implementation Notes
	f	ng	agement
Data Volume	Photos, service records, analytics	Storage optimization, Archiving	Data lifecycle management, Compression

2.4.4 Security Implications

Security Domain	Requirements	Implementation Approach	Compliance Considerations
Data Protection	PII encryption at rest	Database encryption, Secure storage	GDPR, CCPA compliance
Payment Security	PCI DSS compliance	Stripe Elements, No raw PAN storage	PCI DSS Level 1 requirements
Access Control	RBAC, Least privilege	Role-based permissions, API security	SOC 2 Type II controls
Audit Logging	Billing, payroll, schedule changes	Comprehensive audit trails	Regulatory compliance, Forensics

2.4.5 Maintenance Requirements

Maintenance Category	Frequency	Scope	Automation Level
Security Updates	Weekly	Dependencies, OS patches	Fully automated
Database Maintenance	Daily	Backups, PITR, Optimization	Automated with monitoring
API Integration Updates	As needed	Stripe, QuickBooks, Maps APIs	Semi-automated with testing

Maintenance Category	Frequency	Scope	Automation Level
Performance Optimization	Monthly	Query optimization, Caching review	Manual analysis, Automated implementation
Feature Flag Management	Continuous	A/B testing, Gradual rollouts	Automated with manual oversight

3. TECHNOLOGY STACK

3.1 PROGRAMMING LANGUAGES

3.1.1 Primary Languages

Language	Platform/Component	Version	Justification
TypeScript	Full-stack development	5.0+	Next.js 15 introduces React 19 support and provides enhanced TypeScript integration. Provides type safety across the entire application stack, essential for complex business logic and API integrations.
JavaScript (ES2022+)	Runtime execution	ES2022+	Modern JavaScript features for optimal performance with Next.js 15 and React 19 compatibility.

3.1.2 Selection Criteria

Type Safety Requirements: The complex business logic involving pricing calculations, route optimization, and financial transactions demands strong typing to prevent runtime errors and ensure data integrity across the system.

Developer Experience: TypeScript's IntelliSense and compile-time error detection are crucial for maintaining code quality across a large codebase with multiple integrations (Stripe, QuickBooks, Google Maps).

Framework Compatibility: Next.js 15 App Router uses React 19 RC with full TypeScript support, ensuring seamless integration with the chosen technology stack.

Team Productivity: Type safety reduces debugging time and improves code maintainability, particularly important for features like wellness insights analysis and multi-tenant franchise management.

3.2 FRAMEWORKS & LIBRARIES

3.2.1 Core Framework

Framework	Version	Purpose	Justification
Next.js	15.x	Full-stack React framework	Next.js 15 is officially stable and ready for production, focused heavily on stability while adding exciting updates. Provides App Router architecture, server-side rendering, and API routes essential for the comprehensive business management platform.
React	19.x	UI library	Next.js 15 aligns with React 19 release, with App Router using React 19 RC and backwards compatibility for React 18 with Pages Router. Required for modern component architecture and state management.

3.2.2 Supporting Libraries

Library	Version	Category	Purpose
Tailwind CSS	3.4+	Styling	Utility-first CSS framework for rapid UI development and consistent design system
shadcn/ui	Latest	UI Components	Pre-built accessible components for consistent user interface across all portals
Framer Motion	11+	Animation	Smooth animations for enhanced user experience in client portal and dispatch interfaces
Lucide React	Latest	Icons	Consistent icon system across all user interfaces

3.2.3 Compatibility Requirements

React 19 Integration: Extensive testing across real-world applications and close work with React team provides confidence in React 19 stability, with core breaking changes well-tested and not affecting existing App Router users.

Performance Optimization: Next.js 15 changes caching semantics with fetch requests, GET Route Handlers, and client navigations no longer cached by default, requiring careful consideration of caching strategies for optimal performance.

Build System: Next.js 15 includes caching improvements and stable Turbopack in development, providing faster development builds and improved developer experience.

3.3 OPEN SOURCE DEPENDENCIES

3.3.1 Database & ORM

Package	Version	Registry	Purpose
@prisma/client	6.10+	npm	Latest version 6.10.1 with PostgreSQL support and performance improvements. Type-safe database client for complex business data models.
prisma	6.10+	npm	Database toolkit and migration system for schema management

3.3.2 Queue Management

Package	Version	Registry	Purpose
bullmq	5.58+	npm	Latest version 5.58.5 published 6 days ago. Lightweight, robust, and fast NodeJS library for creating background jobs and message queues.
ioredis	5.4+	npm	Redis client for BullMQ queue management and caching

3.3.3 Authentication & Security

Package	Version	Registry	Purpose
next-auth	4.24+	npm	Authentication library with OAuth 2 support for secure user management
bcryptjs	2.4+	npm	Password hashing for secure credential storage
jsonwebtoken	9.0+	npm	JWT token generation and validation

3.3.4 Validation & Utilities

Package	Version	Registry	Purpose
zod	3.22+	npm	TypeScript-first schema validation for API endpoints and form data
date-fns	3.6+	npm	Date manipulation utilities for scheduling and reporting features
lodash	4.17+	npm	Utility functions for data manipulation and processing

3.4 THIRD-PARTY SERVICES

3.4.1 Payment Processing

Service	API Version	Purpose	Integration Requirements
Stripe	2025-08-27.basil	Payment processing and subscription management	Current version is 2025-08-27.basil. Basil introduces personalized invoices, ad hoc pricing for Payment Links, and billing improvements with mixed duration subscription phases.

3.4.2 Accounting Integration

Service	API Version	Purpose	Integration Requirements
QuickBooks Online	Minor Version 75	Accounting synchronization	Starting August 1, 2025, all API requests default to minor version 75 with previous minor versions ignored. OAuth2 authentication and webhook support.

3.4.3 Mapping & Location Services

Service	API Version	Purpose	Integration Requirements
Google Maps Platform	v3	Route optimization and navigation	Distance Matrix API, Directions API, and Geocoding API
Google Places API	v1	Address validation and autocomplete	Place details and address validation

3.4.4 Communication Services

Service	API Version	Purpose	Integration Requirements
Twilio	2010-04-01	SMS and voice notifications	Programmable SMS and Voice APIs
Resend	v1	Email delivery	Transactional email API with webhook support

3.4.5 Monitoring & Analytics

Service	Purpose	Integration Requirements
Vercel Analytics	Performance monitoring	Built-in Next.js integration
Sentry	Error tracking and performance monitoring	SDK integration for error reporting

3.5 DATABASES & STORAGE

3.5.1 Primary Database

Database	Version	Purpose	Justification
PostgreSQL	14+	Primary data storage	Prisma ORM supports PostgreSQL versions 9.6 and above, with strong recommendation for currently supported versions. Robust relational database for complex business data with ACID compliance.

3.5.2 Caching Solutions

Technology	Version	Purpose	Configuration
Redis	6.2+	Session management and caching	BullMQ is full Redis™ compliant with version 6.2.0 or newer. Primary cache for route optimization and session storage.
Redis	6.2+	Queue management	BullMQ is backed by Redis, making it easy to scale horizontally and process jobs across multiple servers.

3.5.3 File Storage

Service	Purpose	Configuration
AWS S3 / Cloudflare R2	Photo and document storage	Signed URLs with 15-minute TTL for security
CDN Integration	Global content delivery	Edge caching for photo galleries and static assets

3.5.4 Data Persistence Strategy

Multi-Tenant Architecture: Database design supports franchise operations with tenant isolation and shared infrastructure for scalability.

Backup & Recovery: Automated daily backups with point-in-time recovery capabilities for business continuity.

Performance Optimization: Connection pooling, query optimization, and strategic indexing for sub-500ms response times.

3.6 DEVELOPMENT & DEPLOYMENT

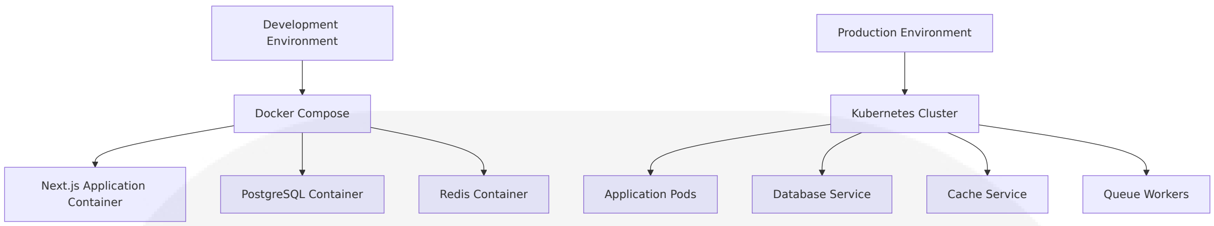
3.6.1 Development Tools

Tool	Version	Purpose
Node.js	20+ LTS	Runtime environment
pnpm	9+	Package management
ESLint	9+	Code linting and quality
Prettier	3+	Code formatting
Husky	9+	Git hooks for quality gates

3.6.2 Build System

Component	Technology	Configuration
Bundler	Next.js 15 with Turbopack	Stable Turbopack in development for improved build performance
TypeScript Compiler	tsc 5.0+	Strict mode with path mapping
CSS Processing	Tailwind CSS + PostCSS	JIT compilation for optimal bundle size

3.6.3 Containerization



3.6.4 CI/CD Pipeline

Stage	Technology	Purpose
Source Control	GitHub	Version control and collaboration
CI/CD	GitHub Actions	Automated testing and deployment
Testing	Jest + Playwright	Unit and end-to-end testing
Deployment	Vercel / AWS	Production hosting and scaling

3.6.5 Quality Assurance

Automated Testing: Comprehensive test suite covering business logic, API endpoints, and user workflows.

Code Quality: ESLint rules enforcing TypeScript best practices and Next.js conventions.

Performance Monitoring: Real-time monitoring of API response times and database query performance.

Security Scanning: Automated dependency vulnerability scanning and security best practices enforcement.

3.6.6 Environment Management

Development: Local Docker environment with hot reloading and debugging capabilities.

Staging: Production-like environment for integration testing and client demonstrations.

Production: Scalable deployment with auto-scaling, monitoring, and disaster recovery capabilities.

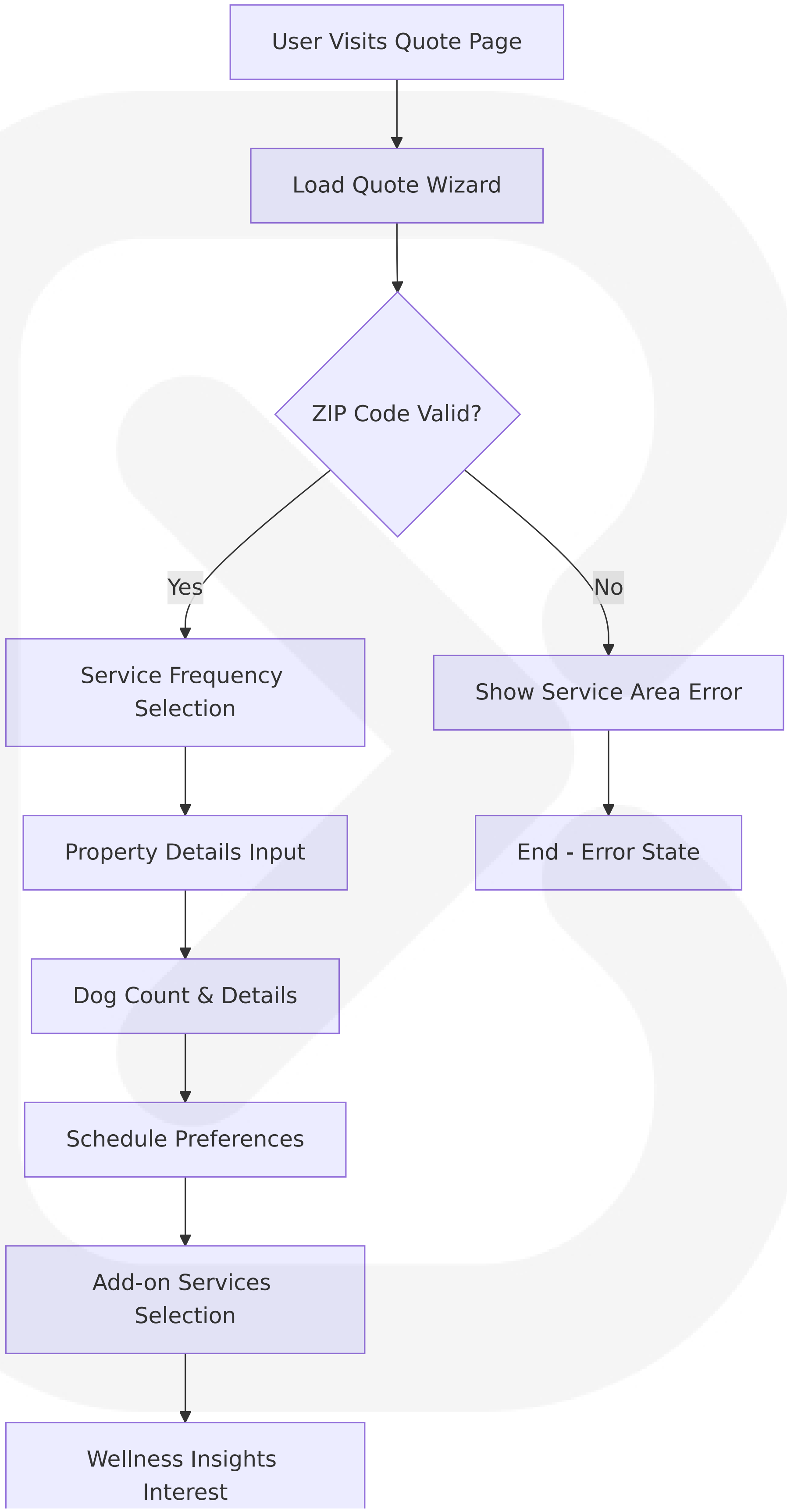
4. PROCESS FLOWCHART

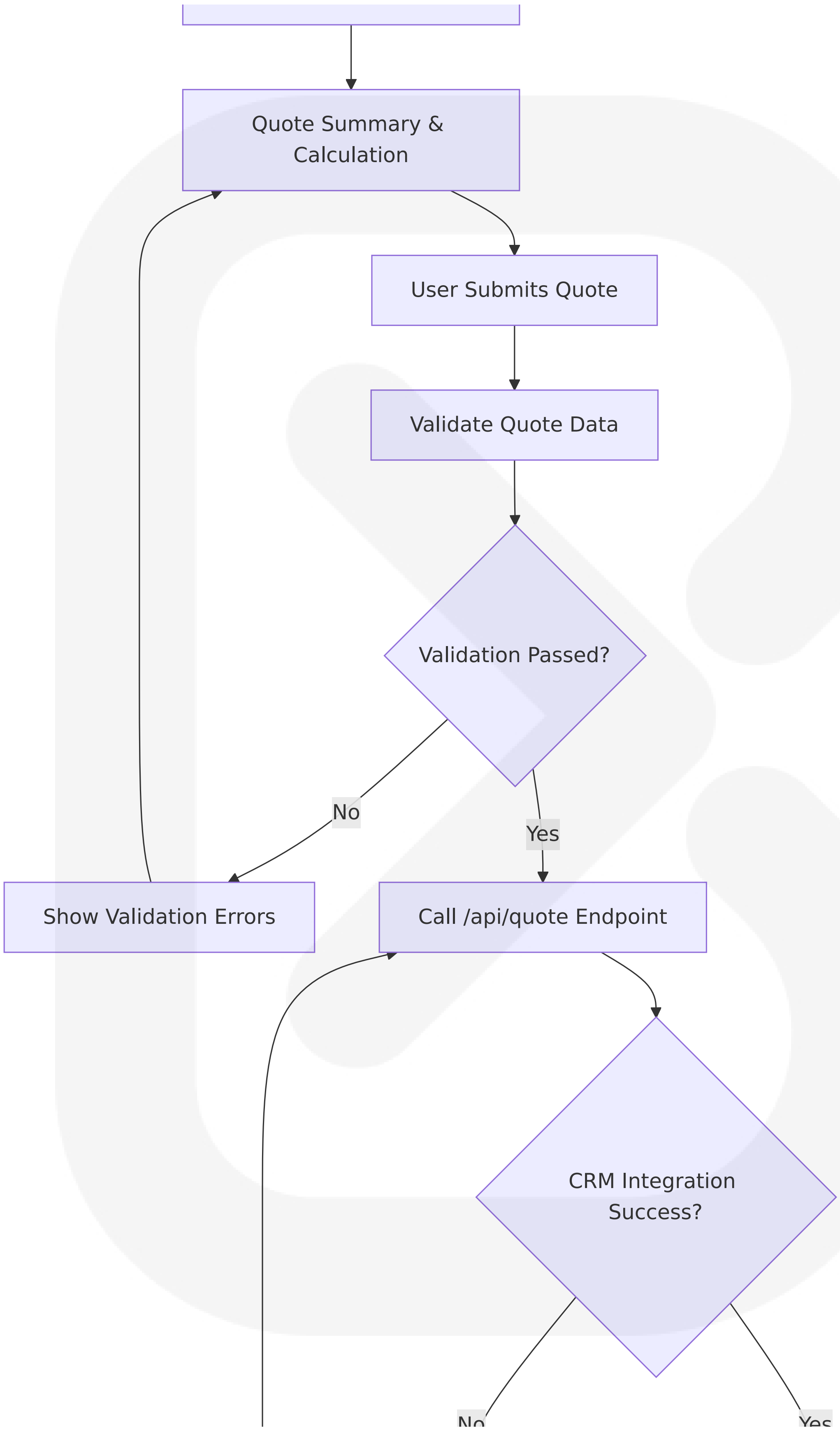
4.1 SYSTEM WORKFLOWS

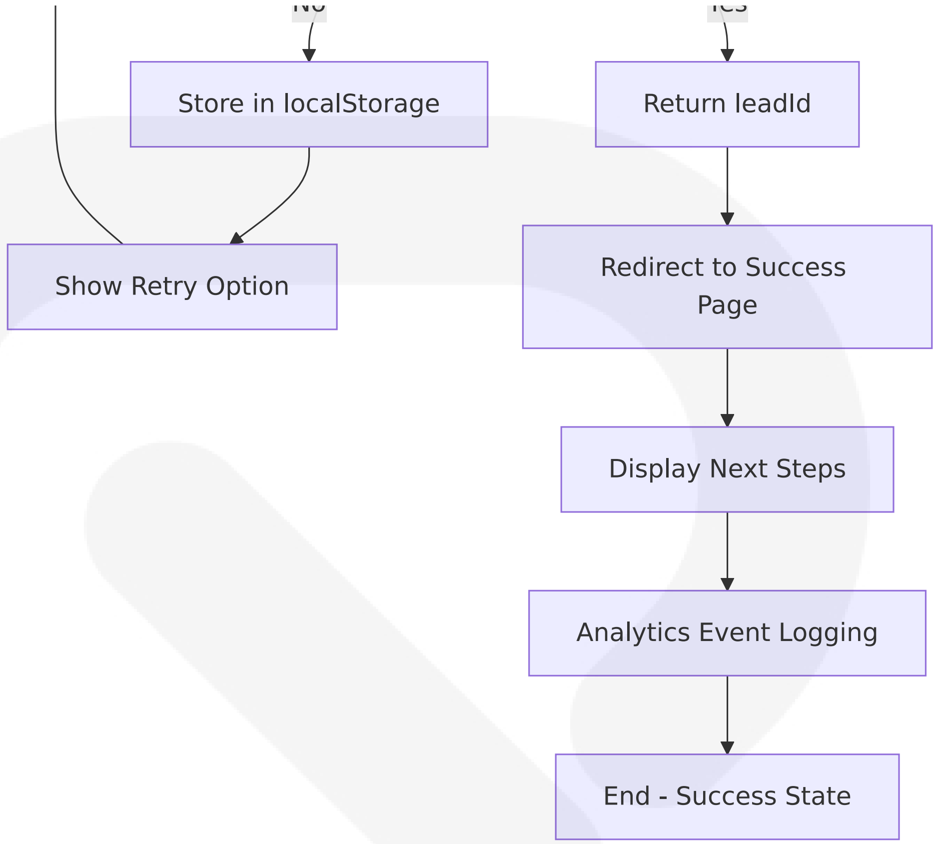
4.1.1 Core Business Processes

Quote to Lead Conversion Workflow

The system preserves the existing Quote flow & Pricing Estimator exactly as implemented while integrating with CRM lead creation functionality. Next.js 15 App Router uses React 19 RC with extensive testing across real-world applications providing confidence in stability.

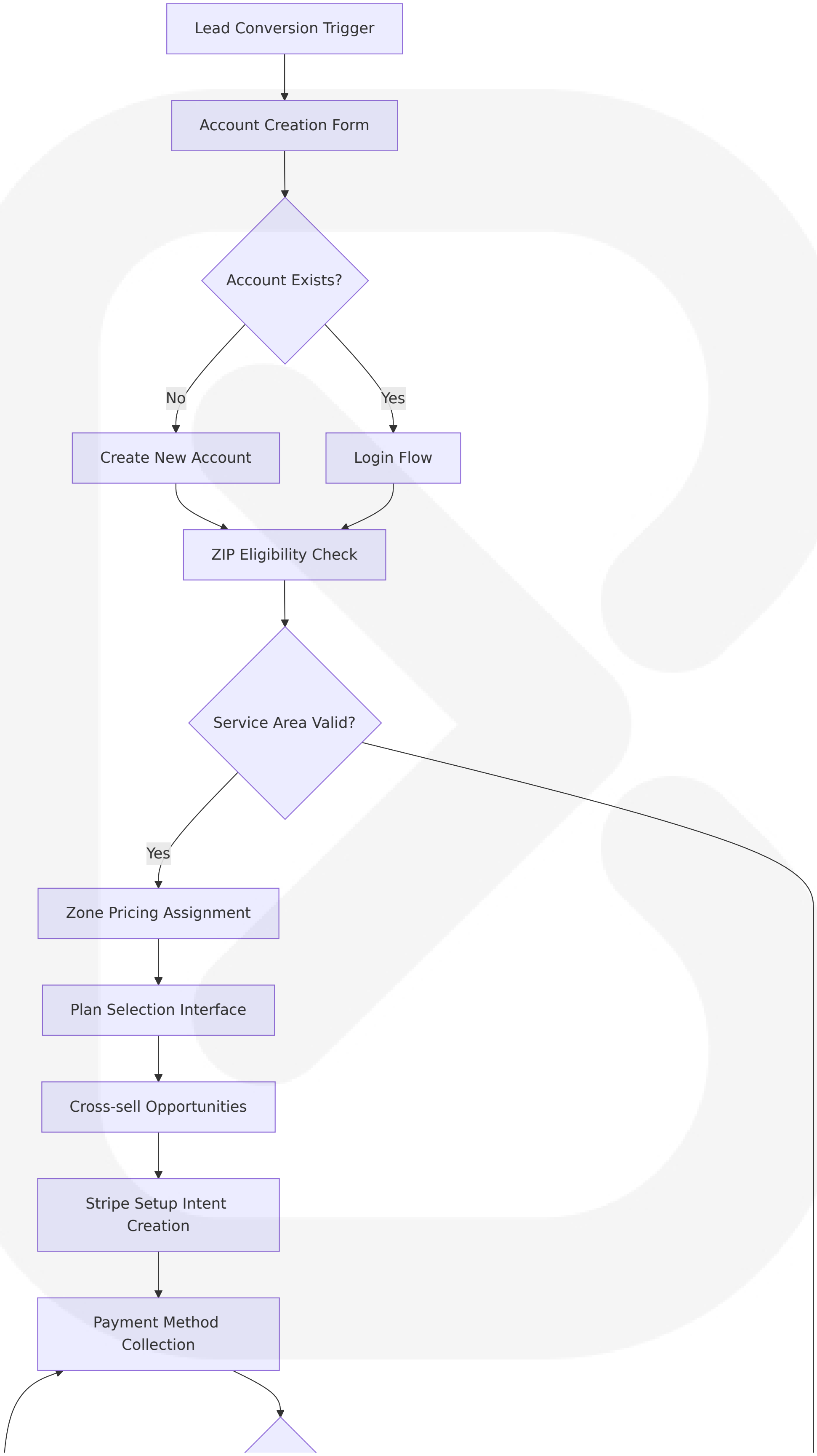

User Visits Quote Page
Load Quote Wizard
ZIP Code Valid?
Yes
No
Service Frequency Selection
Show Service Area Error
Property Details Input
End - Error State
Dog Count & Details
Schedule Preferences
Add-on Services Selection
Wellness Insights Interest

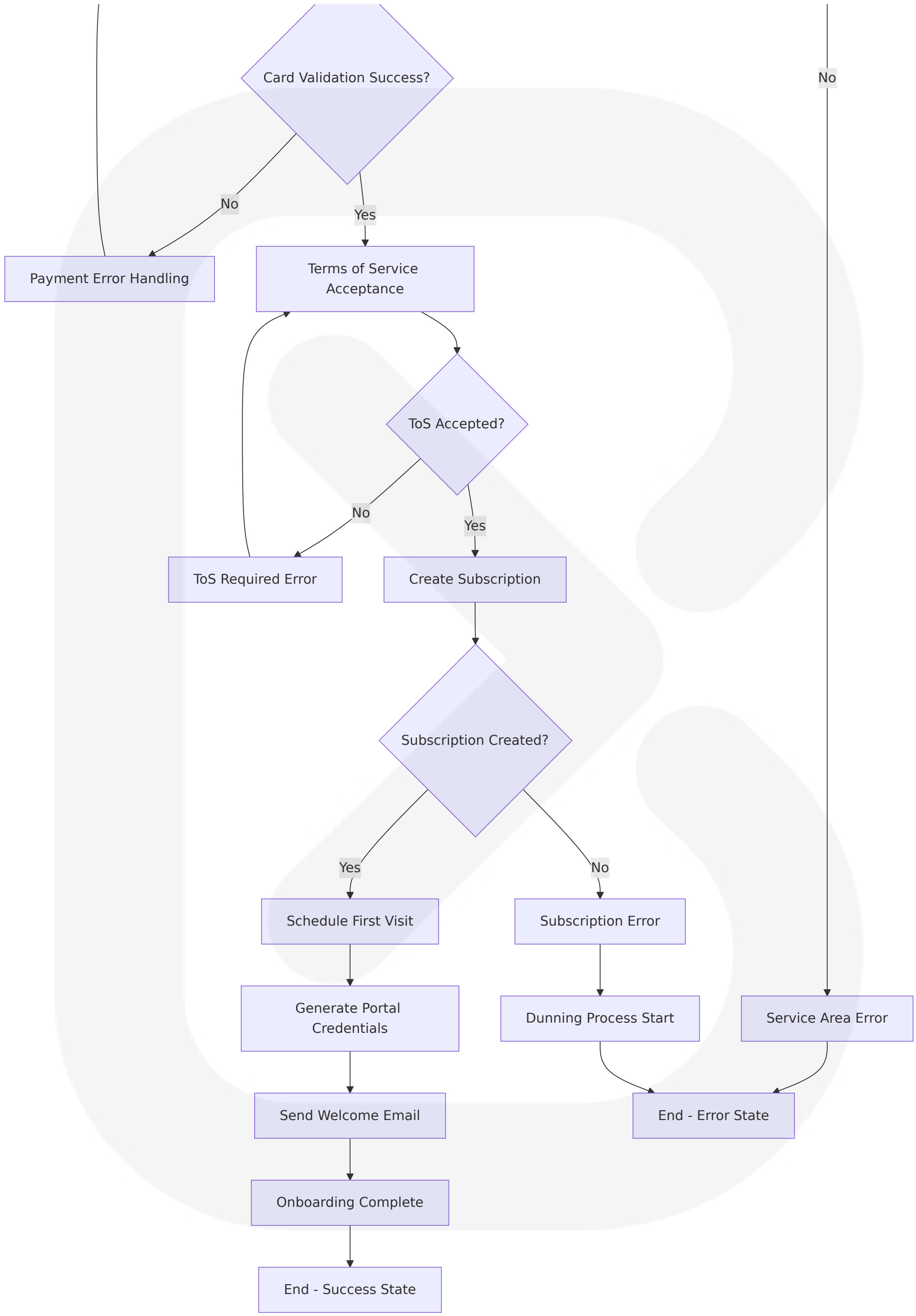

Quote Summary & Calculation
User Submits Quote
Validate Quote Data
Validation Passed?
No
Yes
Show Validation Errors
Call /api/quote Endpoint
CRM Integration Success?
No
Yes



Client Onboarding & Subscription Setup Workflow

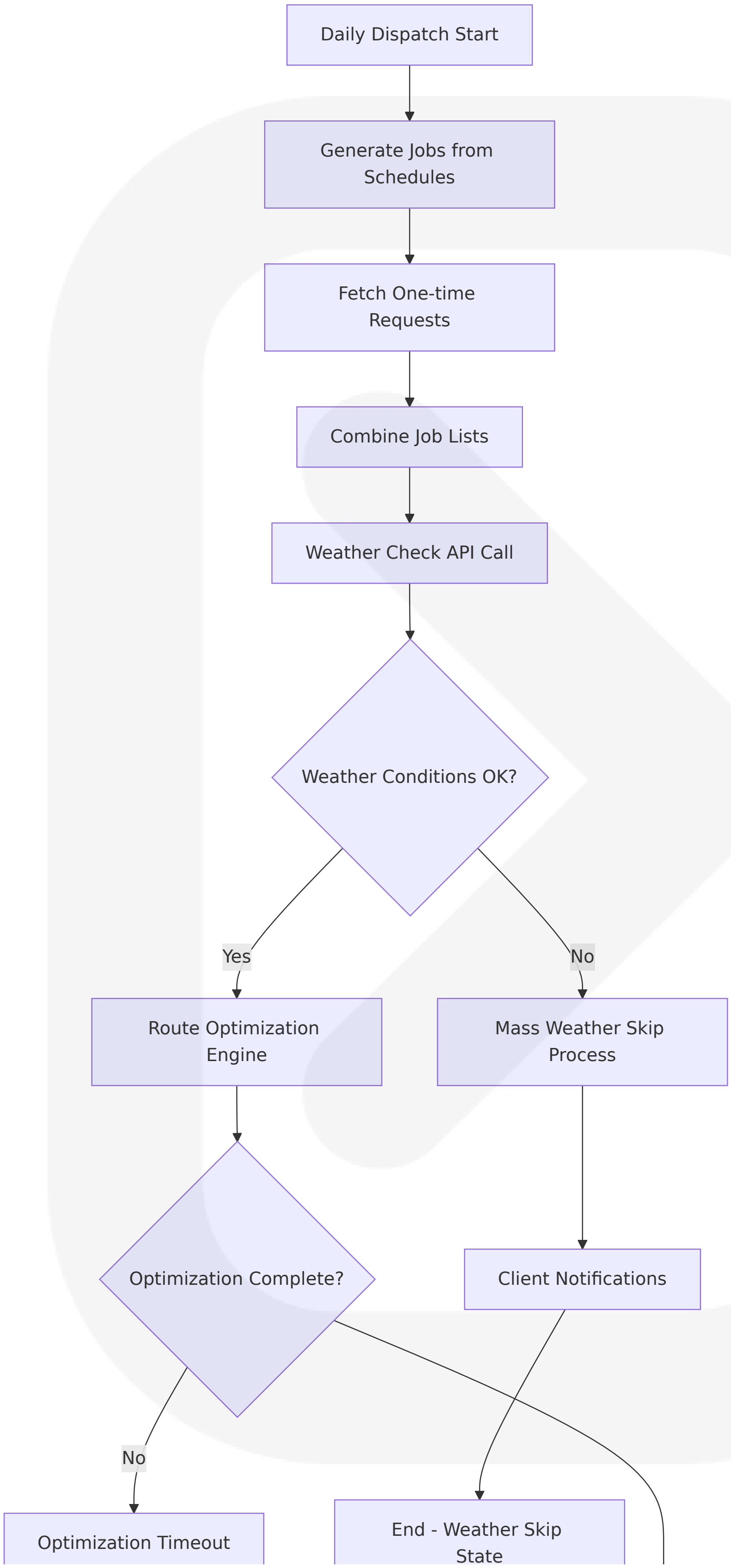
The system integrates with Stripe API version 2025-08-27.basil, which introduces personalized invoices and ad hoc pricing for Payment Links, providing enhanced subscription management capabilities.

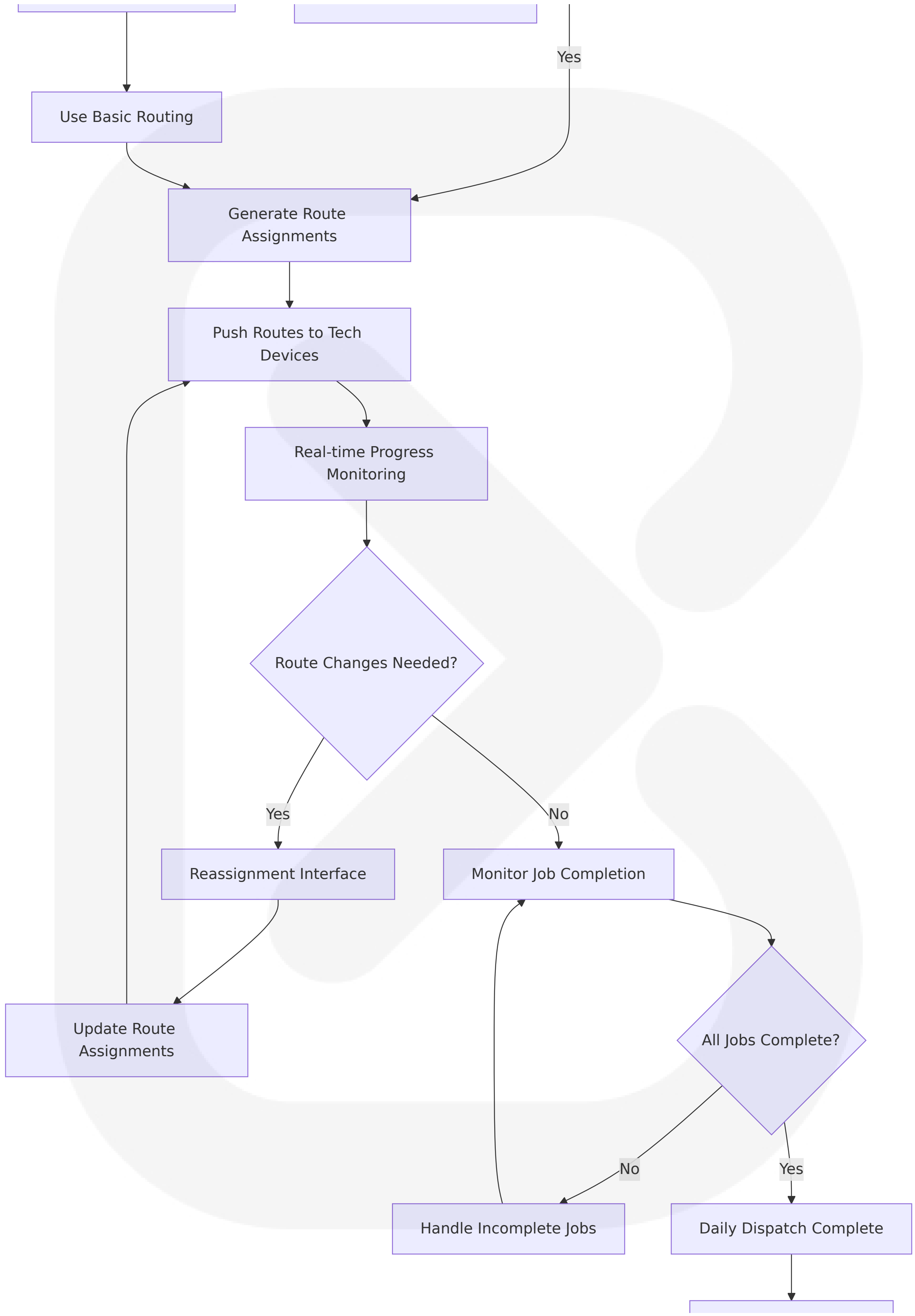

Lead Conversion Trigger
Account Creation Form
Account Exists?
No
Yes
Create New Account
Login Flow
ZIP Eligibility Check
Service Area Valid?
Yes
Zone Pricing Assignment
Plan Selection Interface
Cross-sell Opportunities
Stripe Setup Intent Creation
Payment Method Collection


Card Validation Success?
No
No
Yes
Payment Error Handling
Terms of Service Acceptance
ToS Accepted?
No
Yes
ToS Required Error
Create Subscription
Subscription Created?
Yes
No
Schedule First Visit
Subscription Error
Generate Portal Credentials
Dunning Process Start
Service Area Error
Send Welcome Email
End - Error State
Onboarding Complete
End - Success State

Daily Dispatch & Route Optimization Workflow

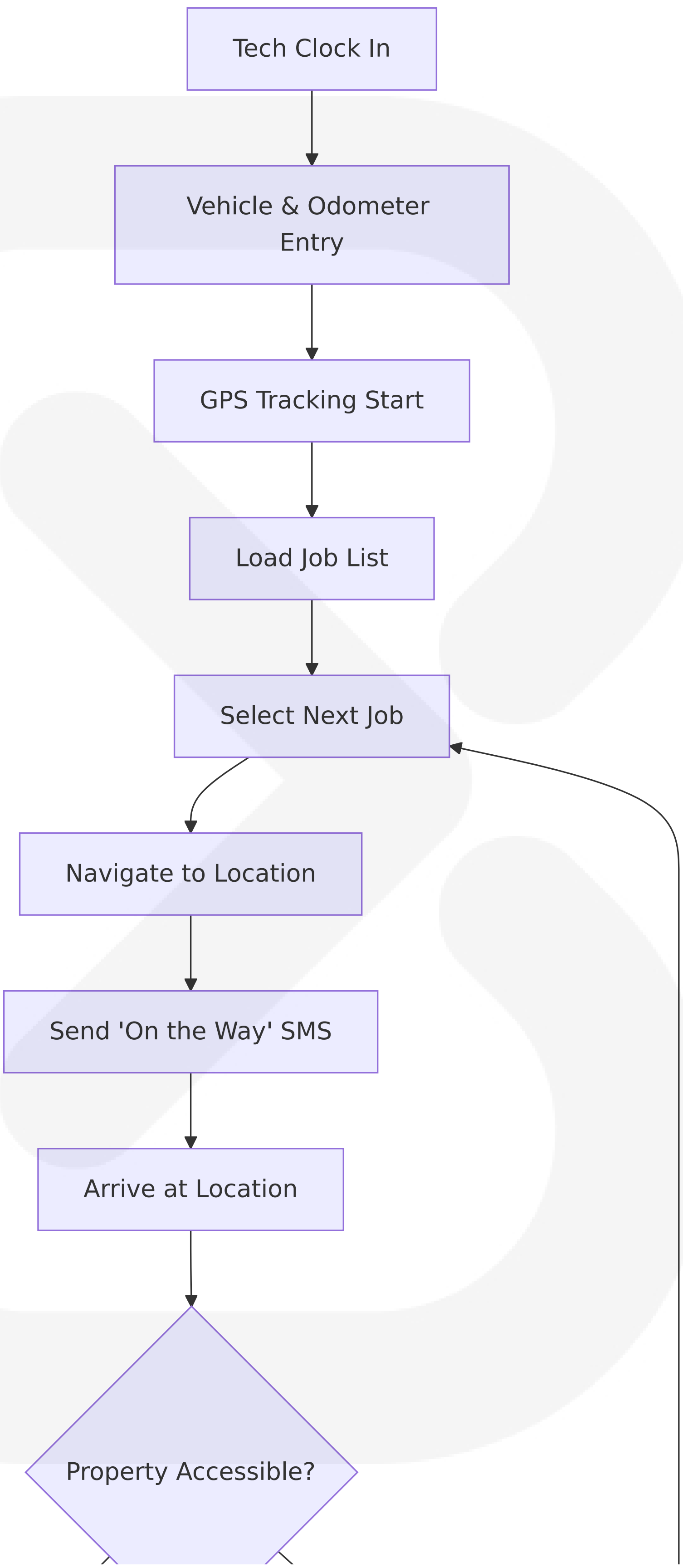
BullMQ version 5.58.5 provides the fastest, most reliable Redis-based distributed queue for Node.js with rock solid stability and atomicity for background job processing.

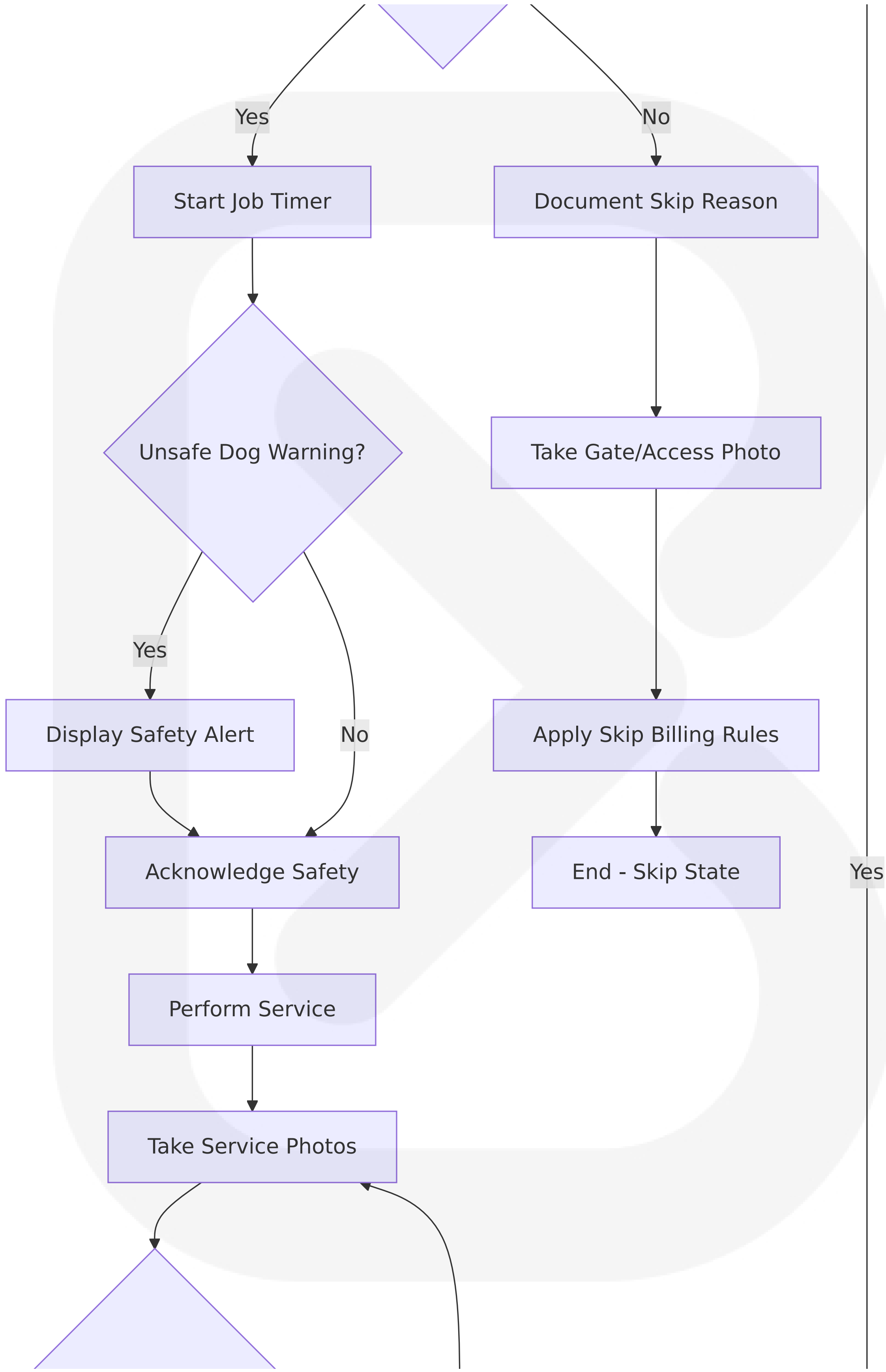

Daily Dispatch Start
Generate Jobs from Schedules
Fetch One-time Requests
Combine Job Lists
Weather Check API Call
Weather Conditions OK?
Yes
No
Route Optimization Engine
Mass Weather Skip Process
Optimization Complete?
Client Notifications
No
Optimization Timeout
End - Weather Skip State


Yes
Use Basic Routing
Generate Route Assignments
Push Routes to Tech Devices
Real-time Progress Monitoring
Route Changes Needed?
Yes
No
Reassignment Interface
Monitor Job Completion
Update Route Assignments
All Jobs Complete?
No
Yes
Handle Incomplete Jobs
Daily Dispatch Complete

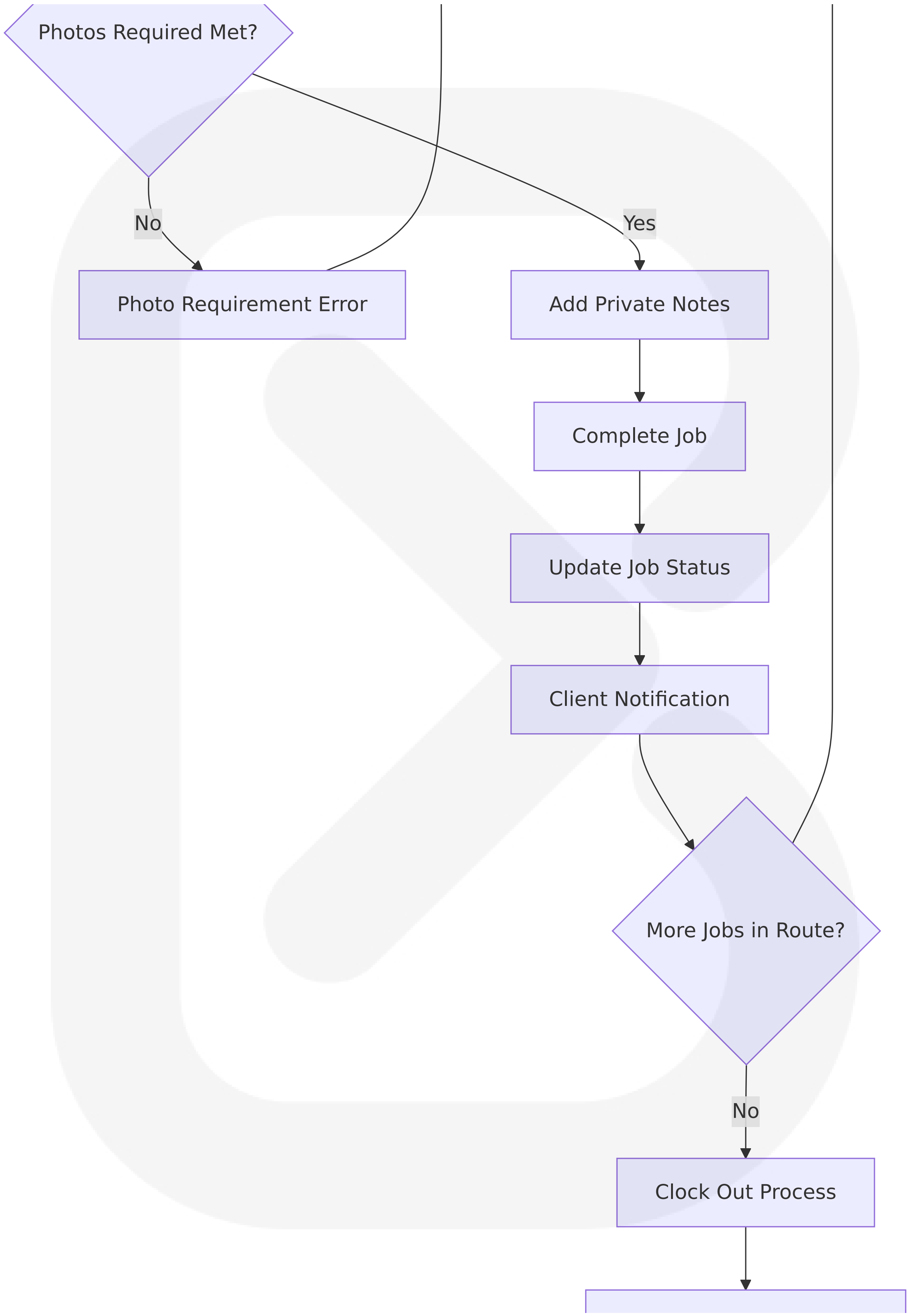
Field Technician Service Delivery Workflow

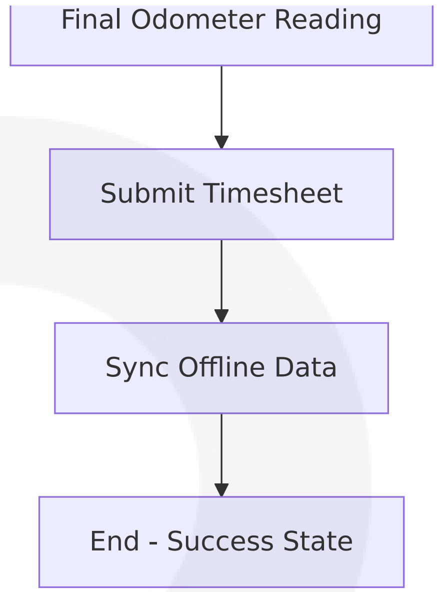
Next.js 15 introduces React 19 support, caching improvements, and stable Turbopack in development with improved build times and faster Fast Refresh for enhanced PWA performance.


Tech Clock In
Vehicle & Odometer Entry
GPS Tracking Start
Load Job List
Select Next Job
Navigate to Location
Send 'On the Way' SMS
Arrive at Location
Property Accessible?



Yes
No
Start Job Timer
Document Skip Reason
Unsafe Dog Warning?
Take Gate/Access Photo
Yes
No
Display Safety Alert
Apply Skip Billing Rules
Acknowledge Safety
End - Skip State
Yes
Perform Service
Take Service Photos

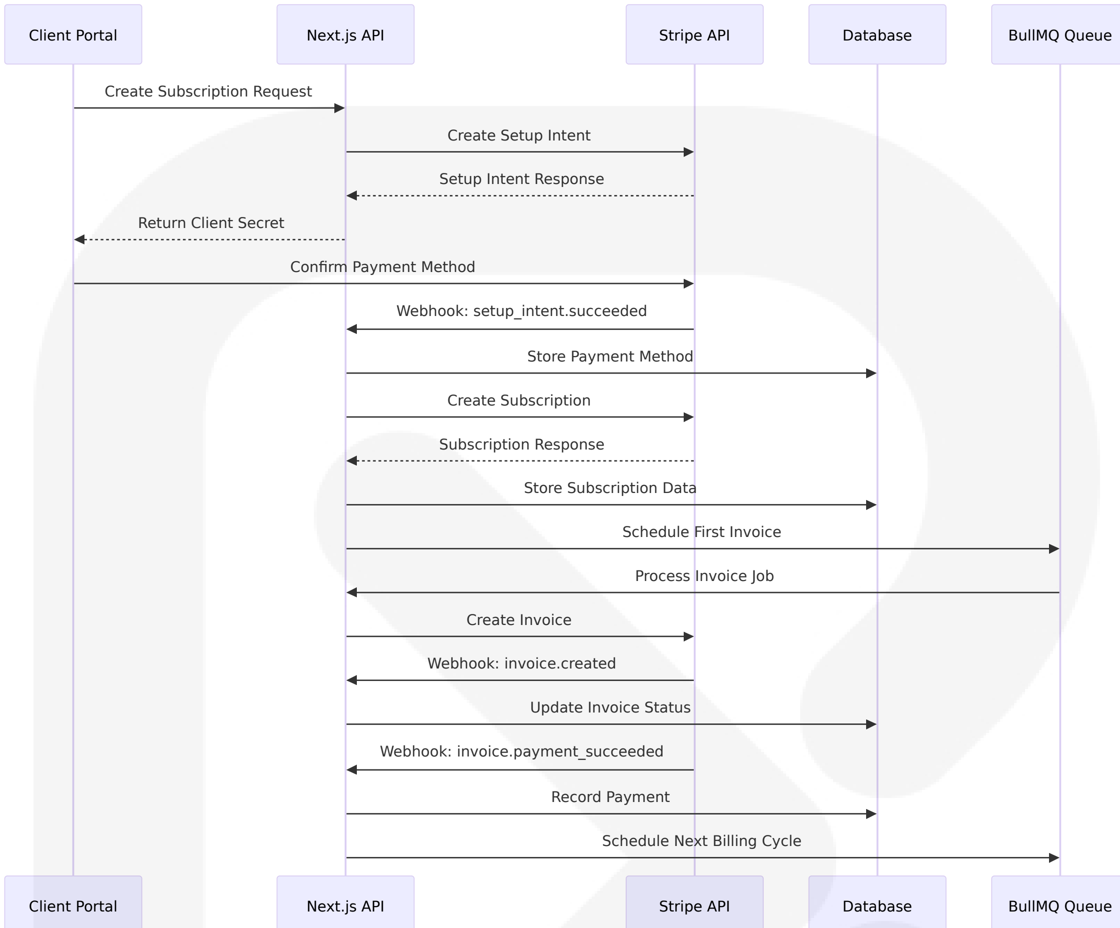

Photos Required Met?
No
Yes
Photo Requirement Error
Add Private Notes
Complete Job
Update Job Status
Client Notification
More Jobs in Route?
No
Clock Out Process



4.1.2 Integration Workflows

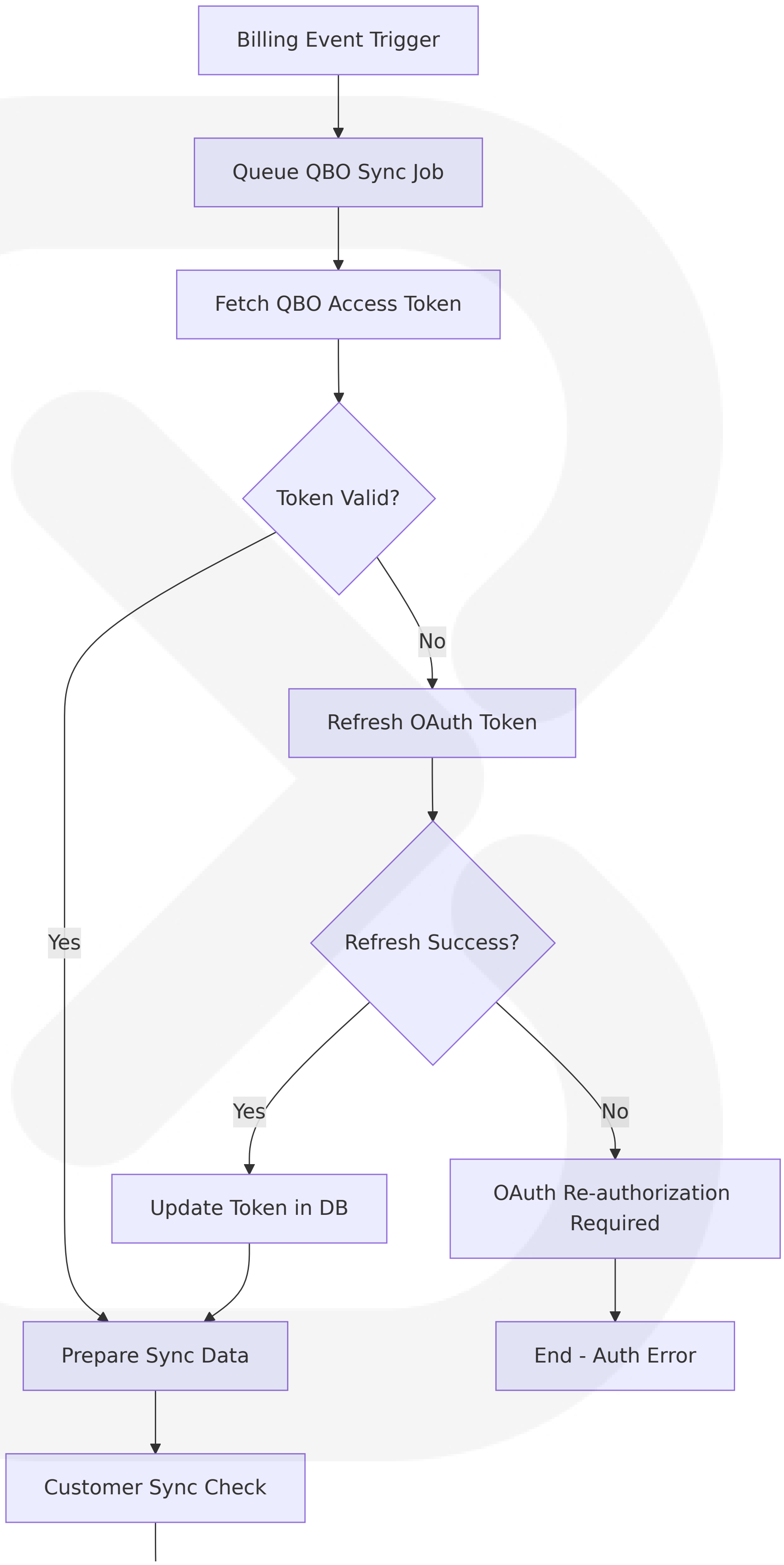
Stripe Payment Processing Integration Flow

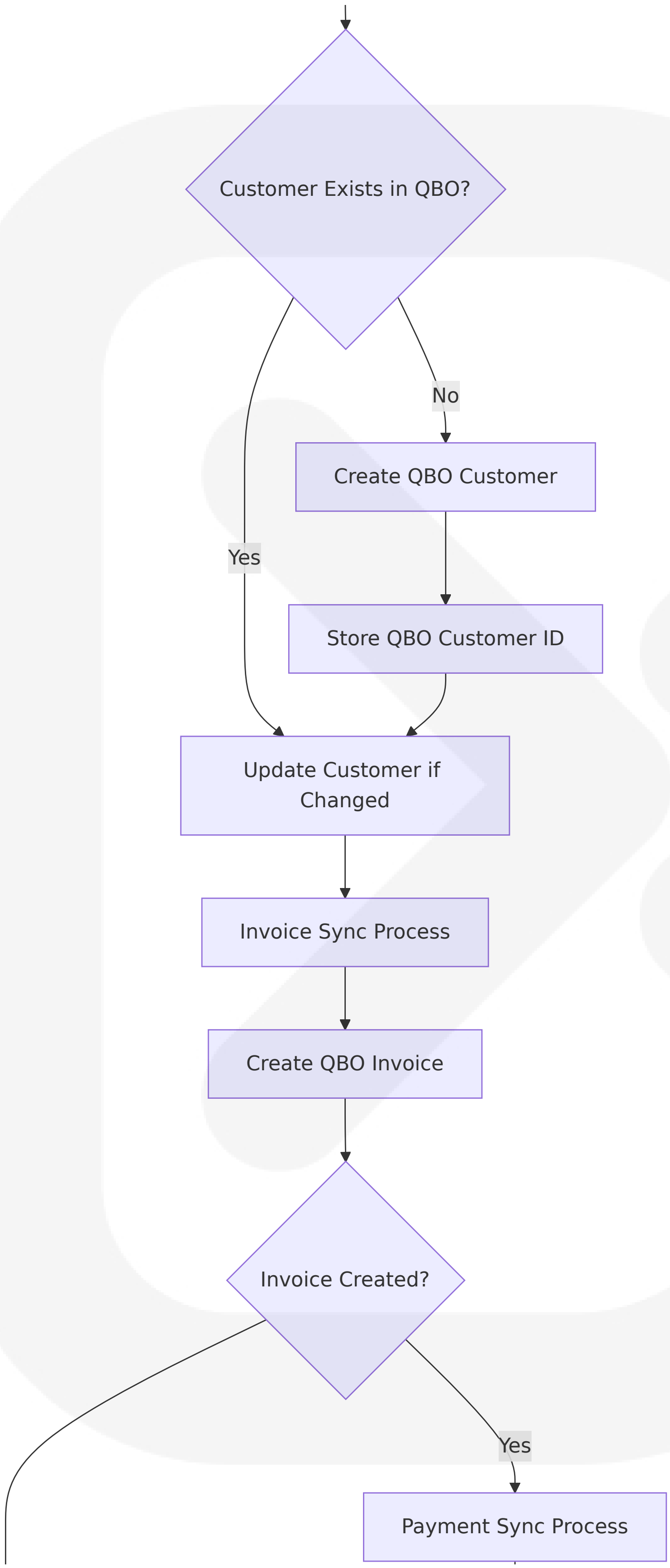
Stripe Basil introduces billing improvements with subscription schedules supporting phases with mixed durations and flexible billing mode with thresholds for usage-based billing.

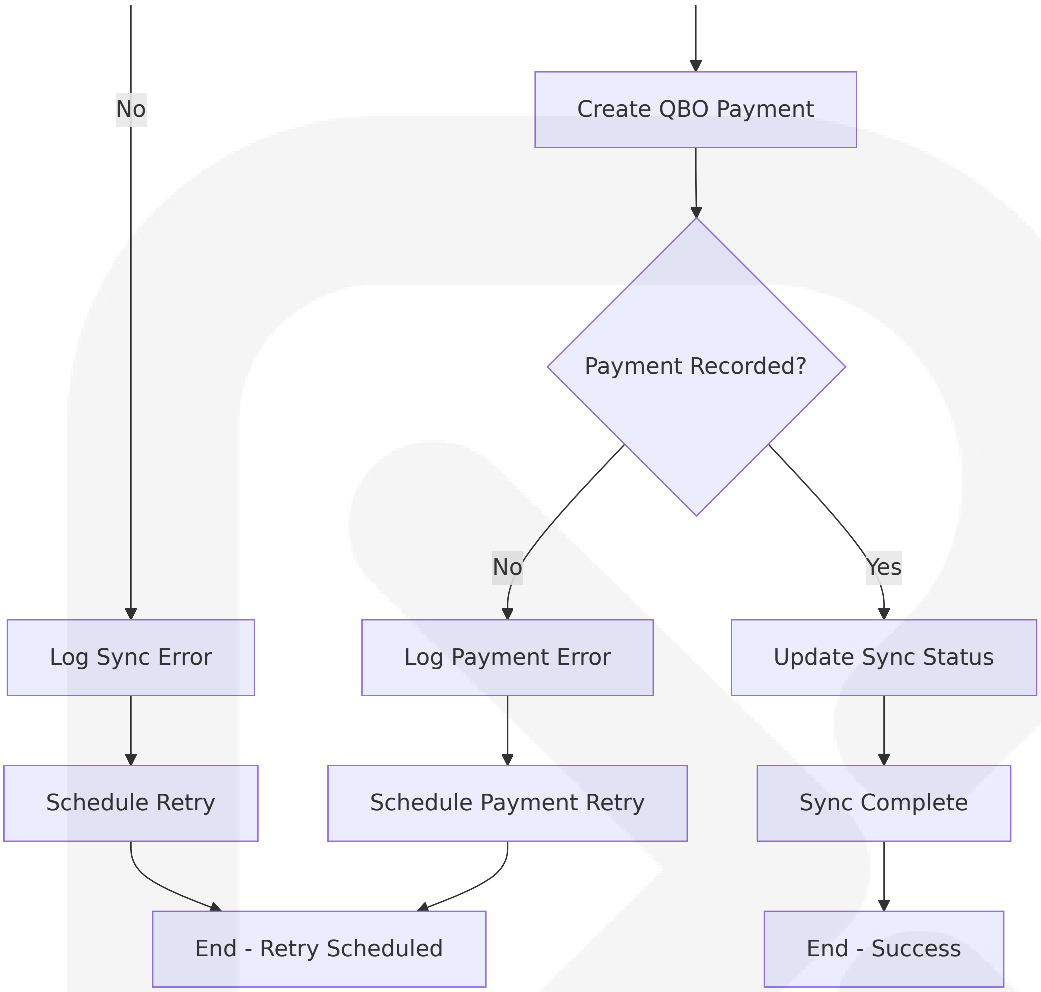


QuickBooks Online Synchronization Flow

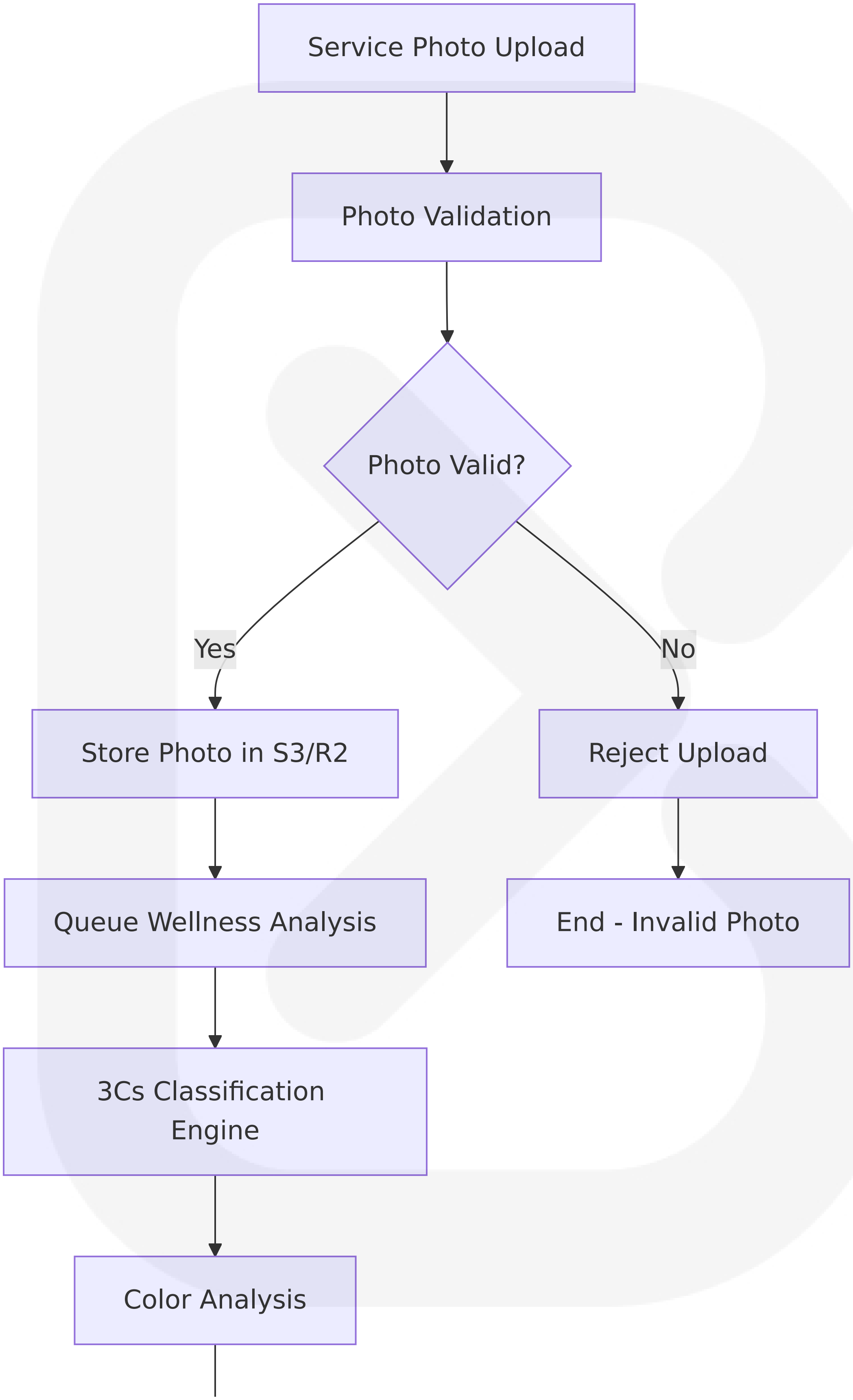
Starting August 1, 2025, all API requests to the QuickBooks Online Accounting API will default to minor version 75, with previous minor versions being ignored.

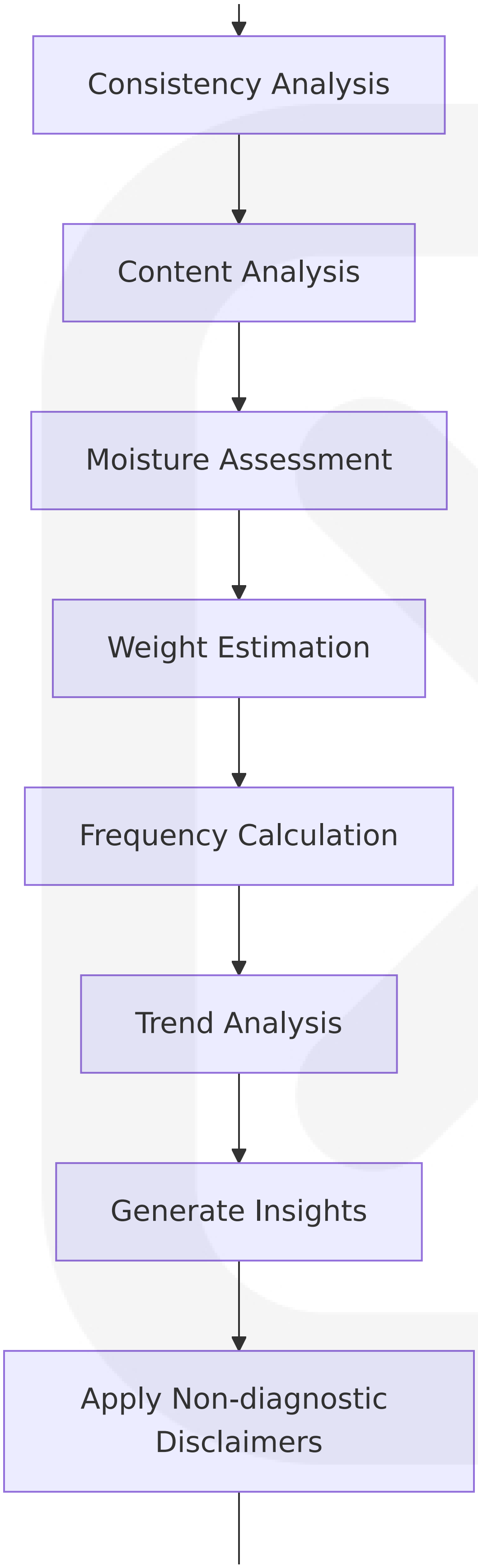

Billing Event Trigger
Queue QBO Sync Job
Fetch QBO Access Token
Token Valid?
No
Refresh OAuth Token
Refresh Success?
Yes
Yes
No
Update Token in DB
OAuth Re-authorization Required
Prepare Sync Data
End - Auth Error
Customer Sync Check

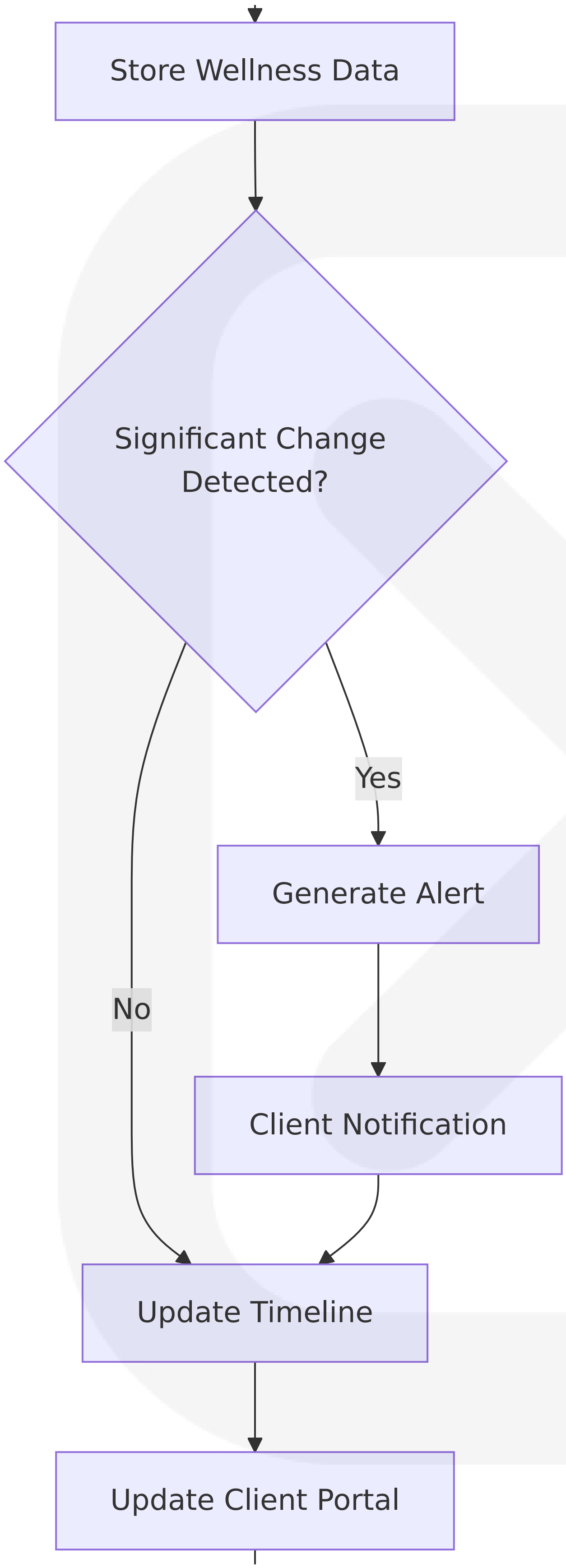

Customer Exists in QBO?
No
Yes
Create QBO Customer
Store QBO Customer ID
Update Customer if Changed
Invoice Sync Process
Create QBO Invoice
Invoice Created?
Yes
Payment Sync Process

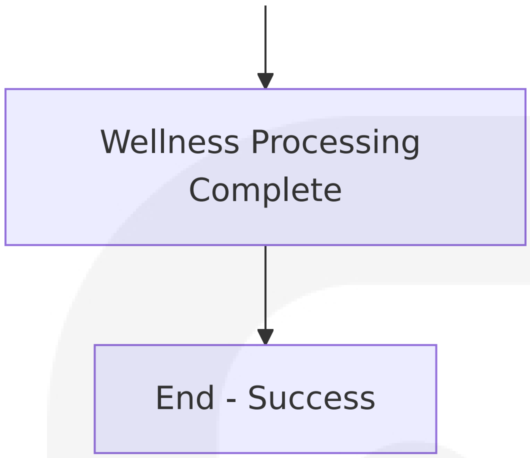


Wellness Insights Processing Flow


Service Photo Upload
Photo Validation
Photo Valid?
Yes
No
Store Photo in S3/R2
Reject Upload
Queue Wellness Analysis
End - Invalid Photo
3Cs Classification Engine
Color Analysis


Consistency Analysis
Content Analysis
Moisture Assessment
Weight Estimation
Frequency Calculation
Trend Analysis
Generate Insights
Apply Non-diagnostic Disclaimers

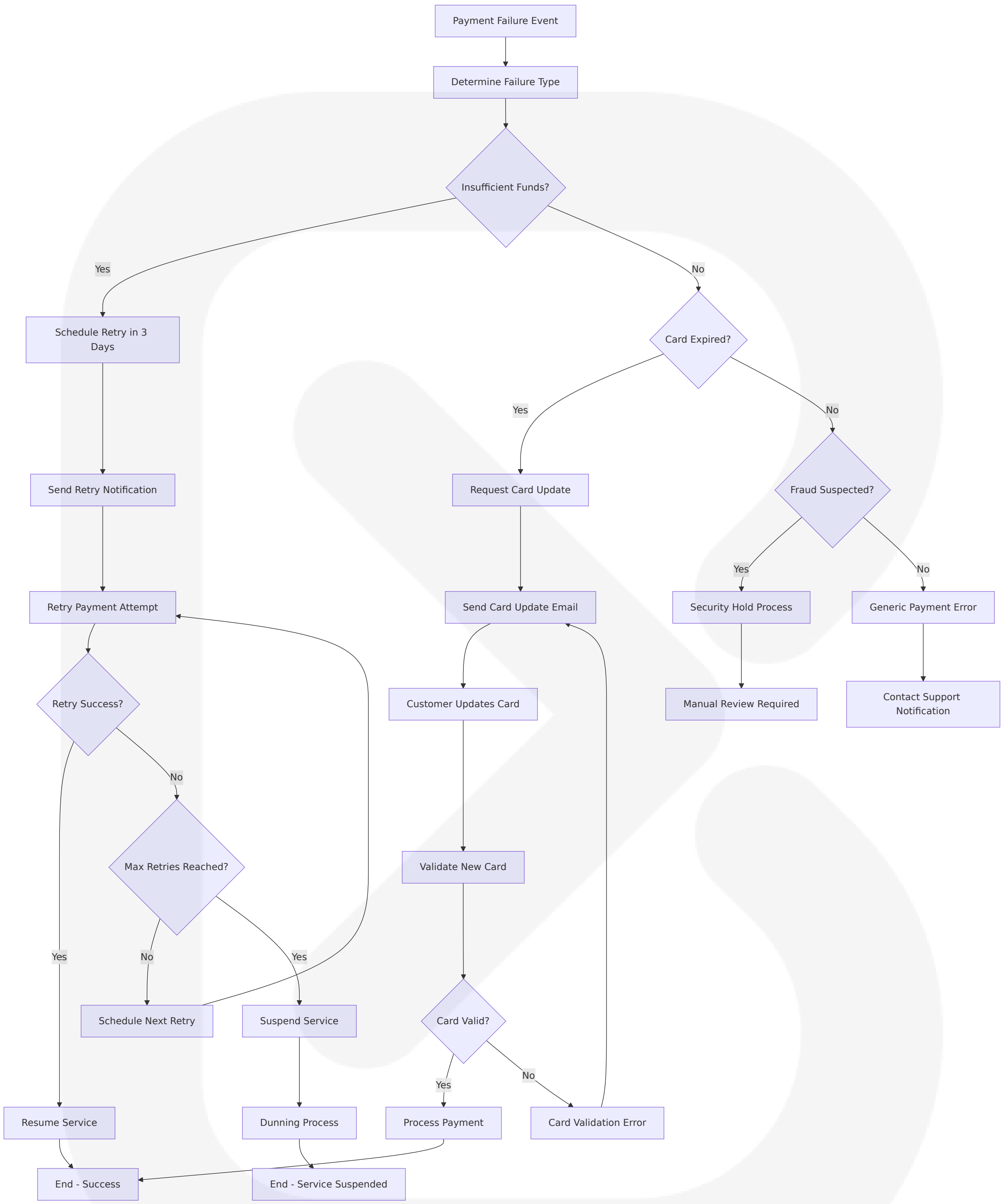

Store Wellness Data
Significant Change Detected?
Yes
No
Generate Alert
Client Notification
Update Timeline
Update Client Portal



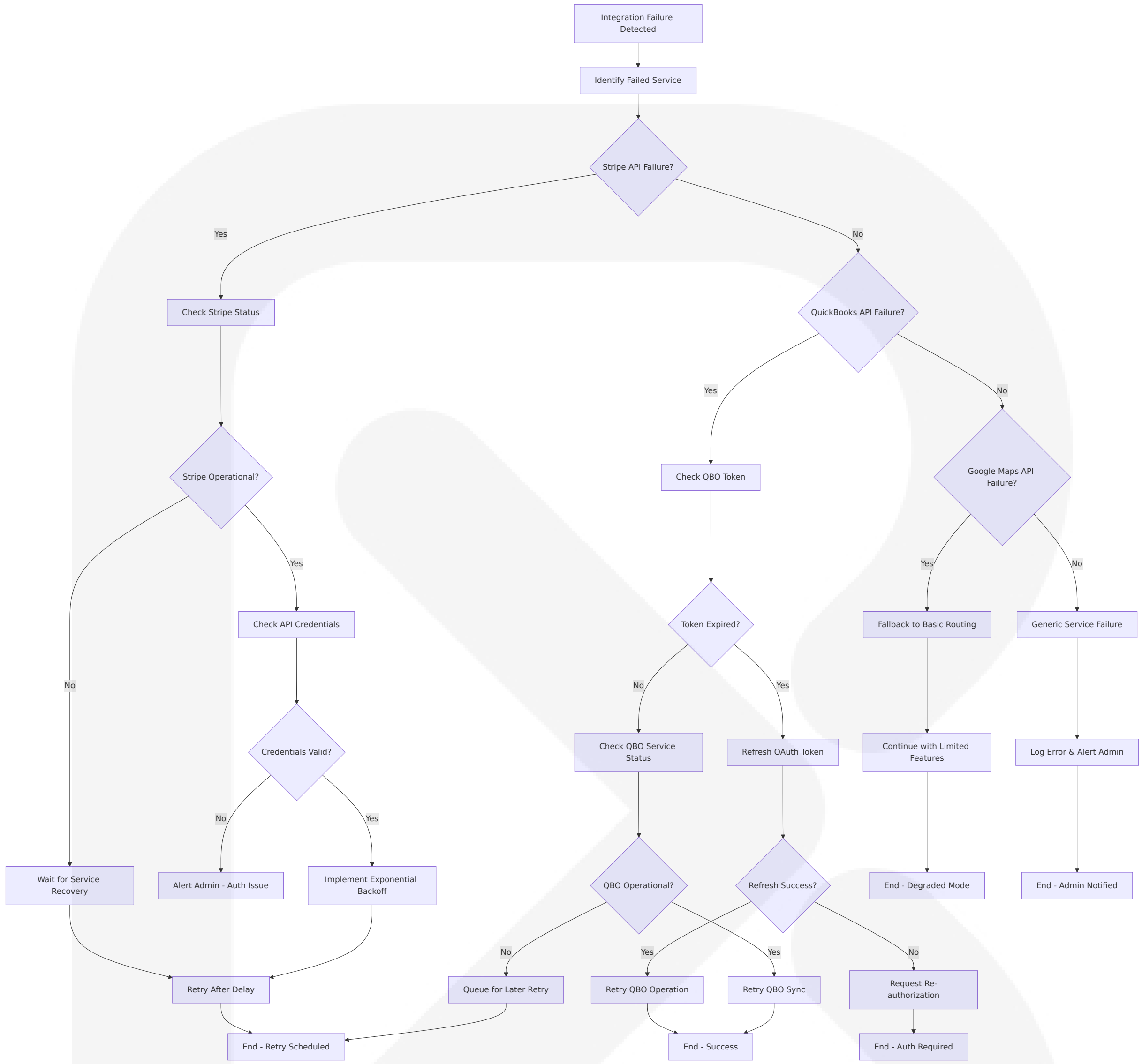
4.2 FLOWCHART REQUIREMENTS

4.2.1 Error Handling and Recovery Workflows

Payment Failure Recovery Process

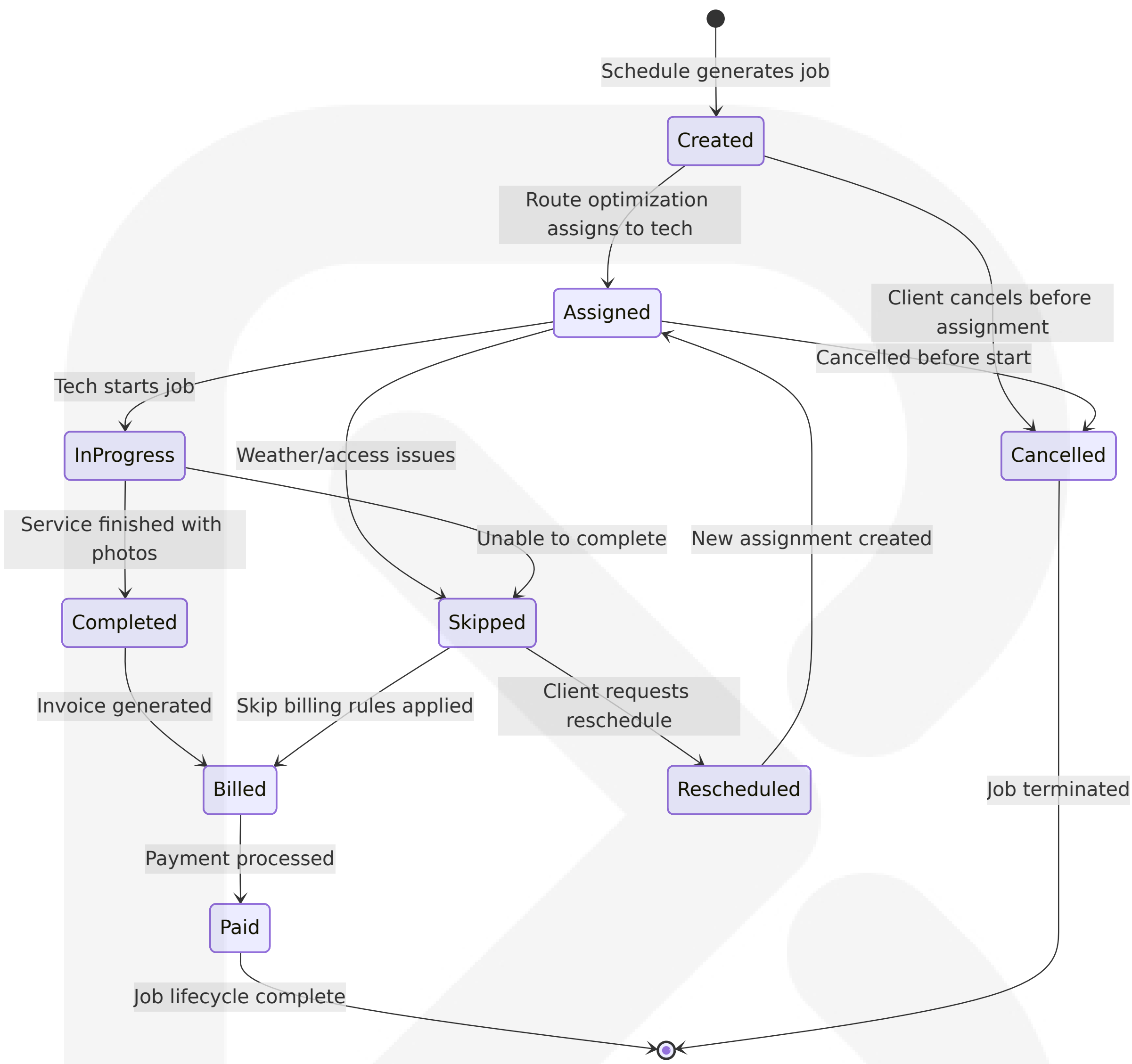


System Integration Failure Handling

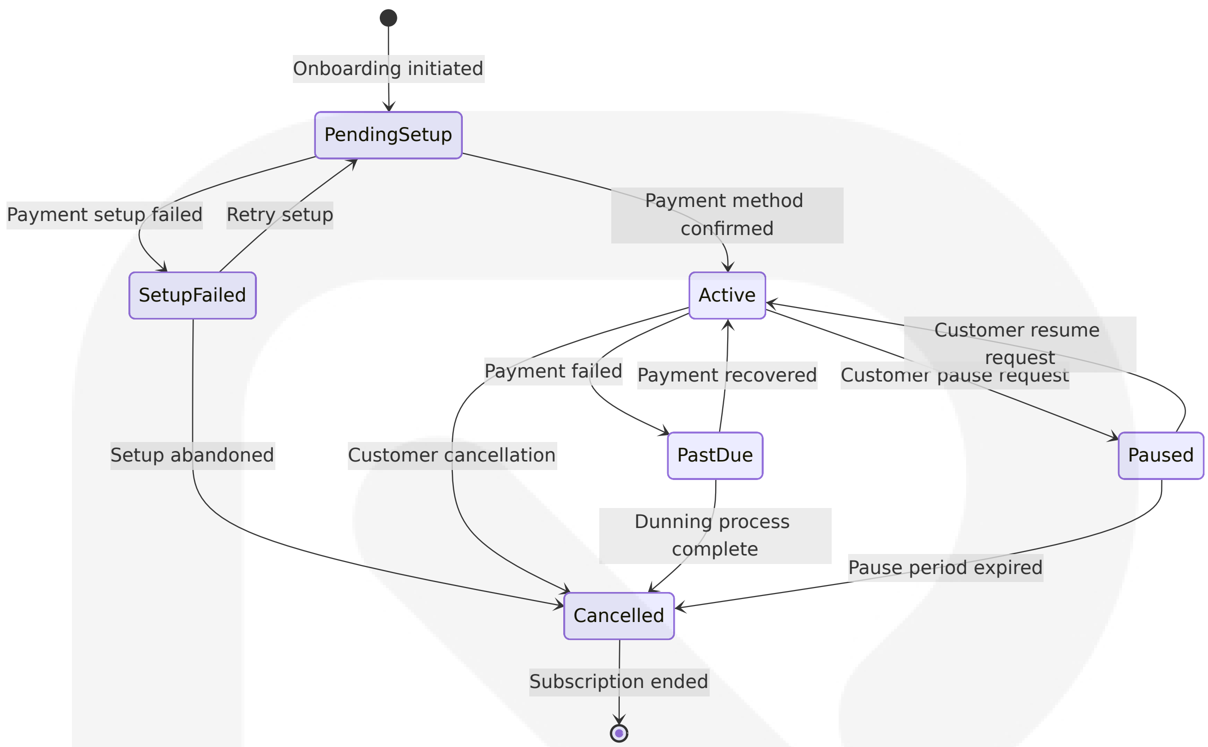


4.2.2 State Management and Data Flow

Job State Transition Diagram



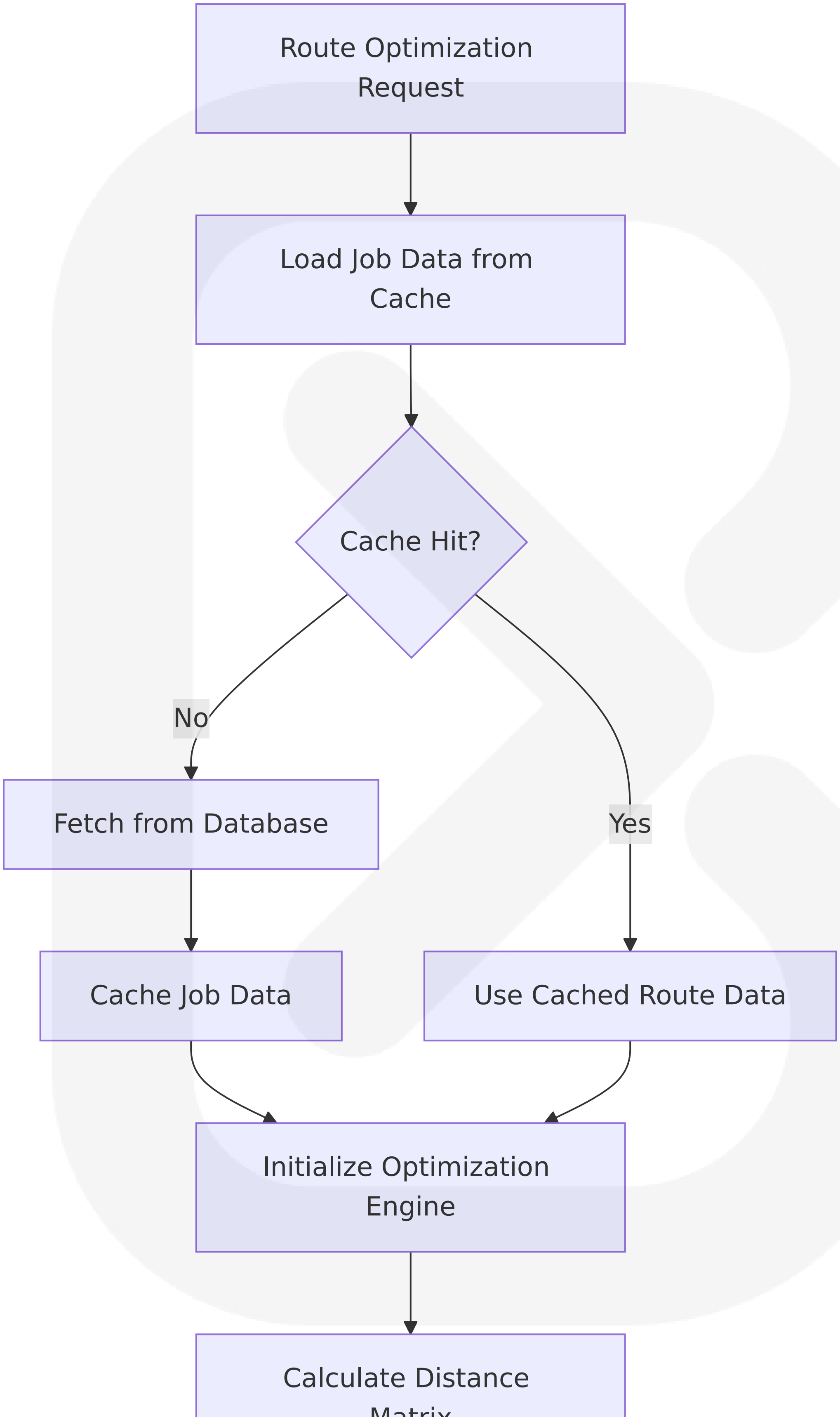
Subscription Lifecycle State Management

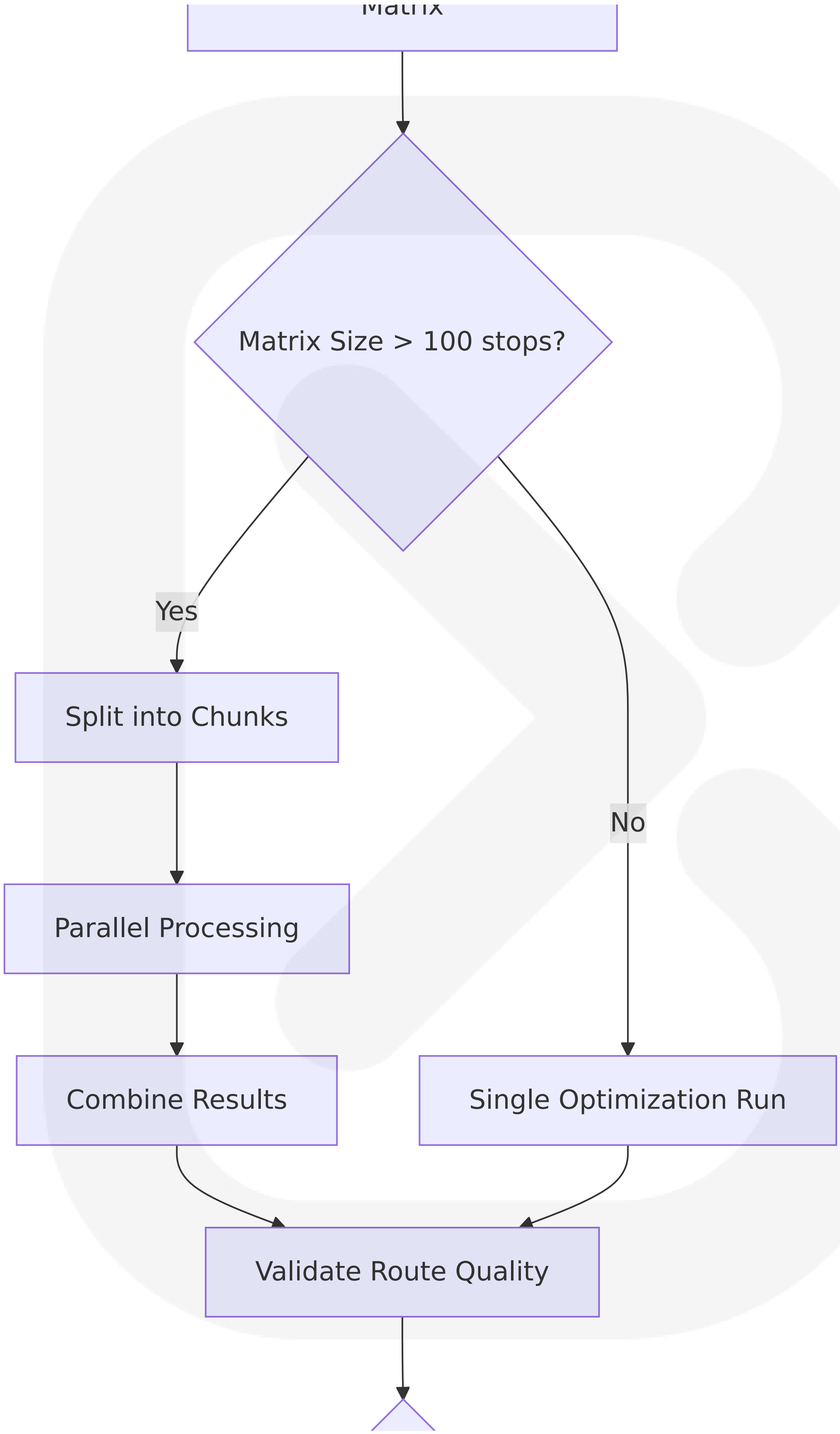


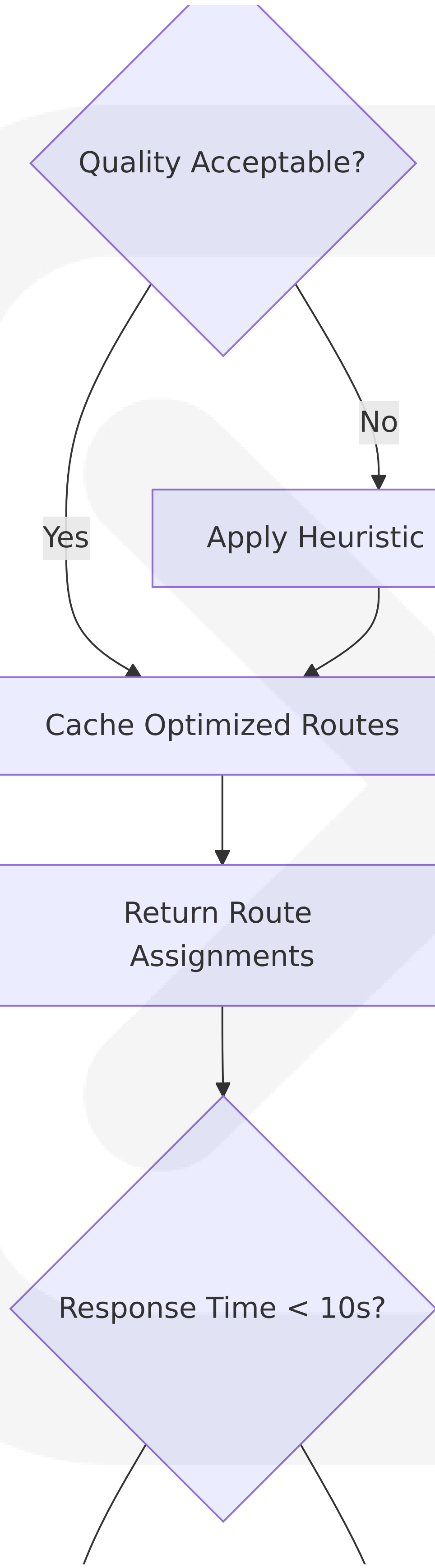
4.2.3 Performance and Scalability Considerations

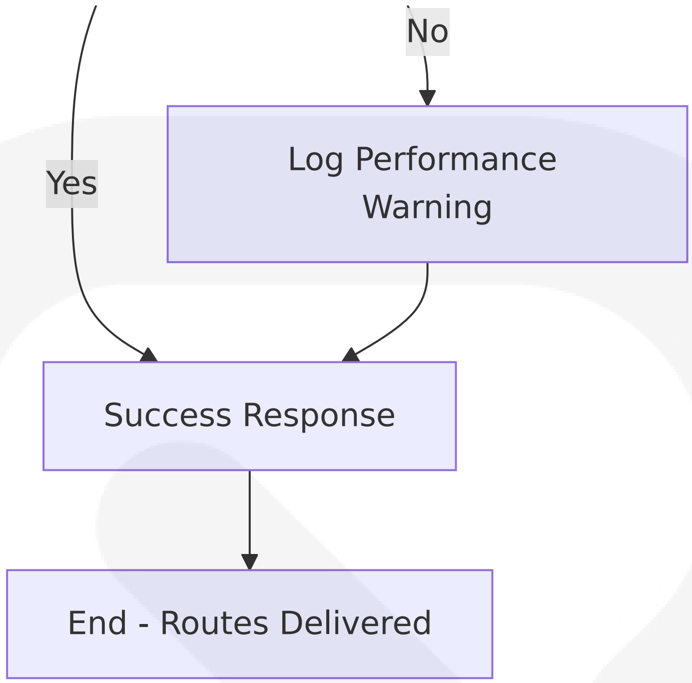
Route Optimization Performance Flow

BullMQ is designed for high performance, trying to get the highest possible throughput from Redis by combining efficient Lua scripts and pipelining.


Route Optimization Request
Load Job Data from Cache
Cache Hit?
No
Yes
Fetch from Database
Cache Job Data
Use Cached Route Data
Initialize Optimization Engine
Calculate Distance Matrix


Matrix
Matrix Size > 100 stops?
Yes
No
Split into Chunks
Parallel Processing
Combine Results
Single Optimization Run
Validate Route Quality

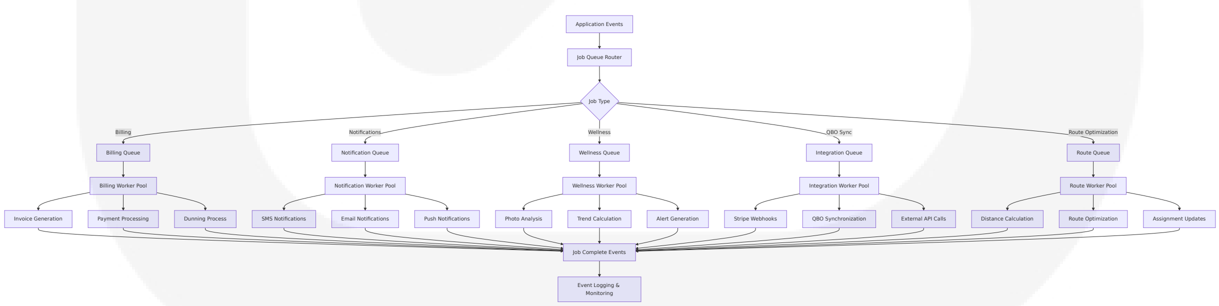

Quality Acceptable?
Yes
No
Apply Heuristic Fallback
Cache Optimized Routes
Return Route Assignments
Response Time < 10s?



4.3 TECHNICAL IMPLEMENTATION

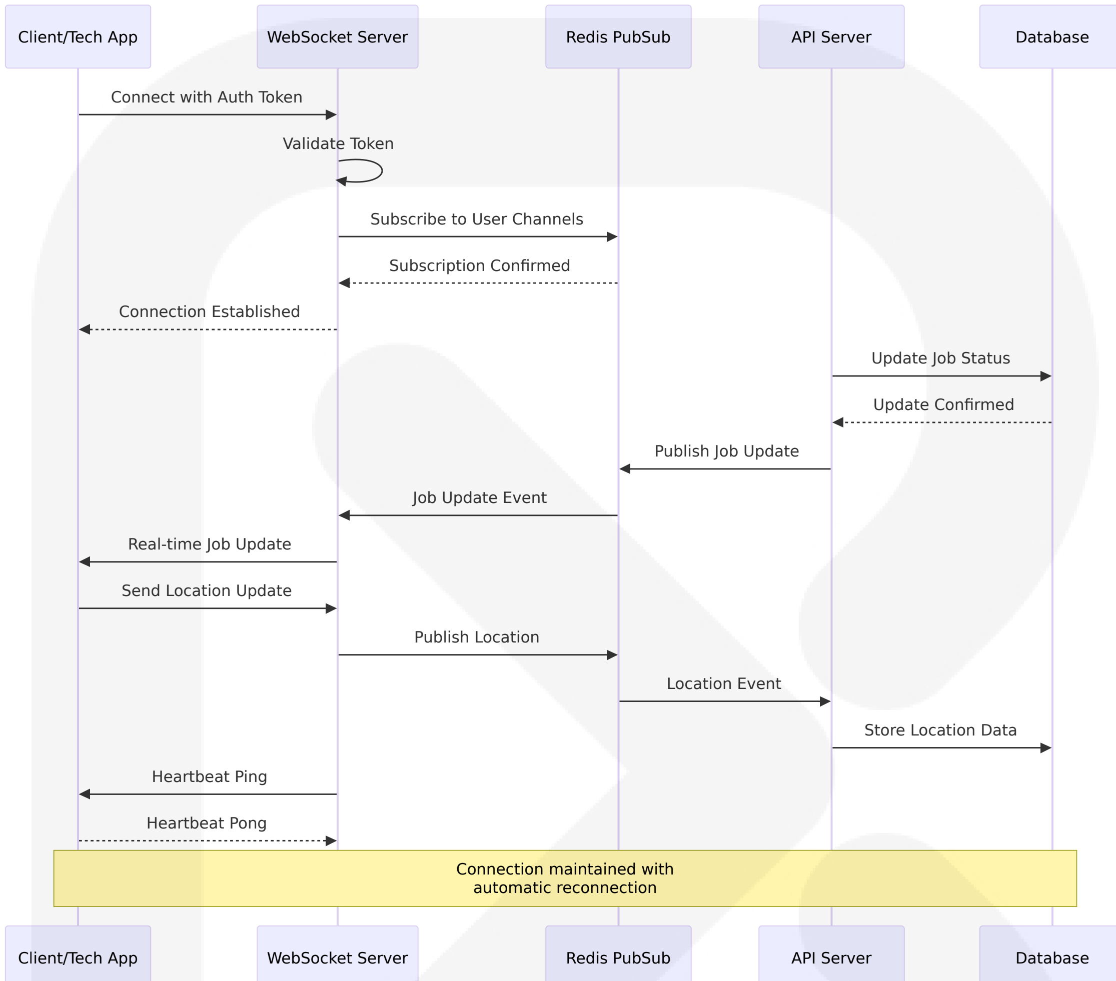
4.3.1 Background Job Processing Architecture

BullMQ is a lightweight, robust, and fast NodeJS library for creating background jobs and sending messages using queues, designed to be easy to use but also powerful and highly configurable, backed by Redis for easy horizontal scaling.



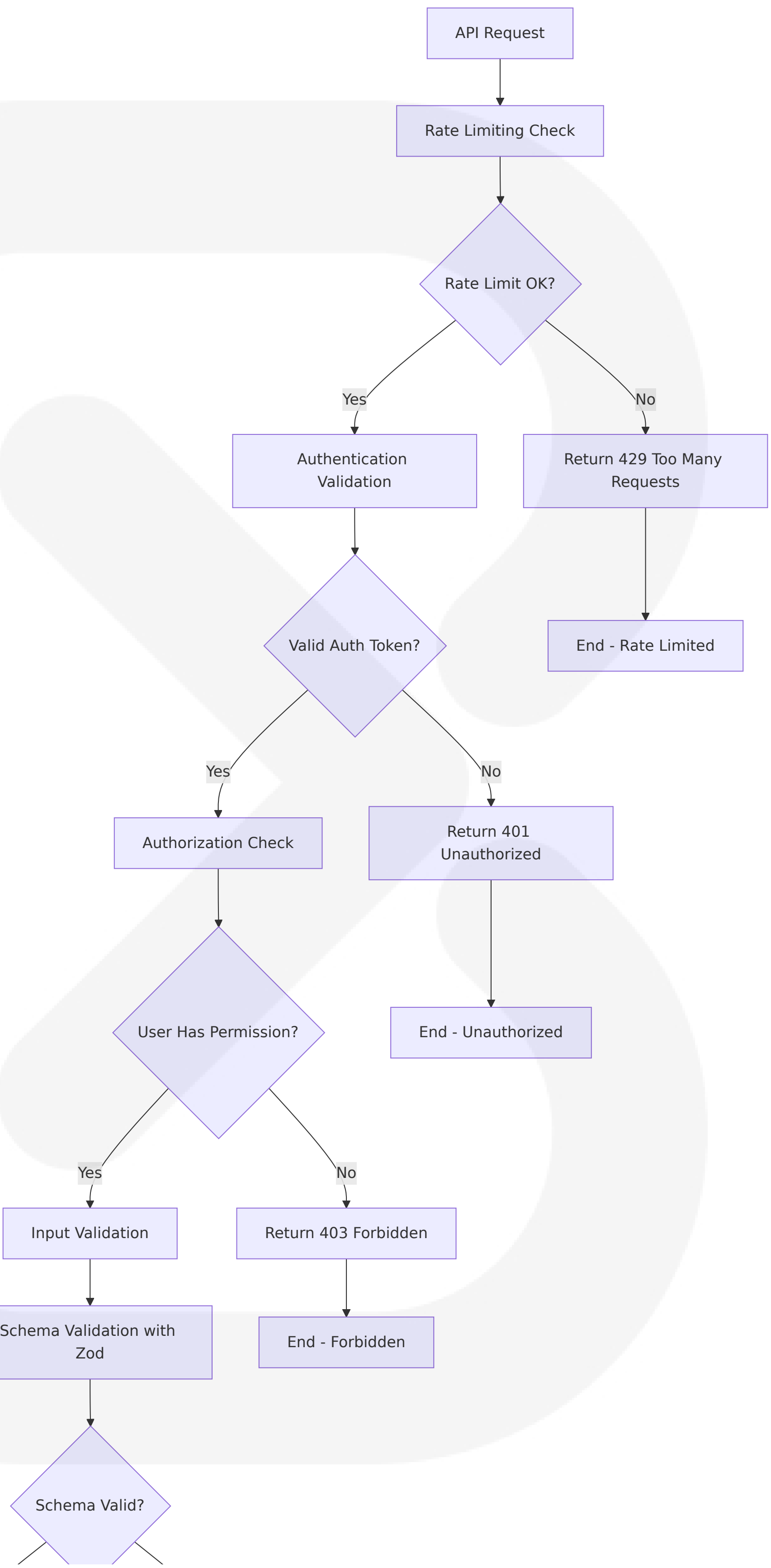
4.3.2 Real-time Data Synchronization

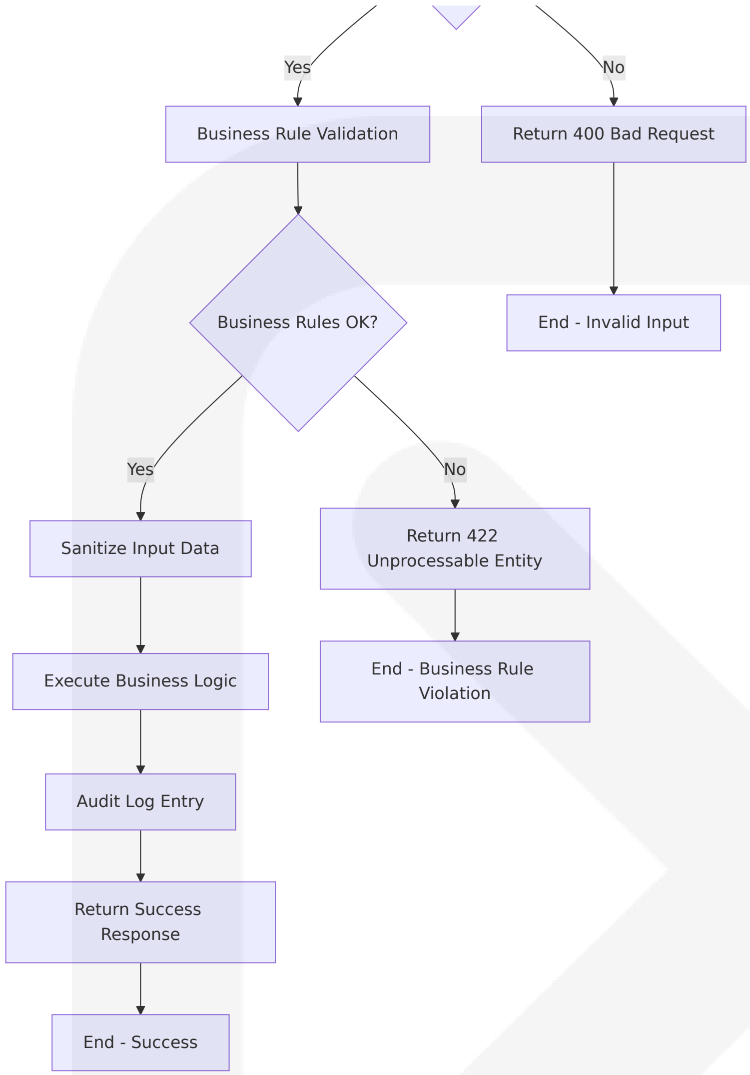
WebSocket Connection Management



4.3.3 Data Validation and Security Workflows

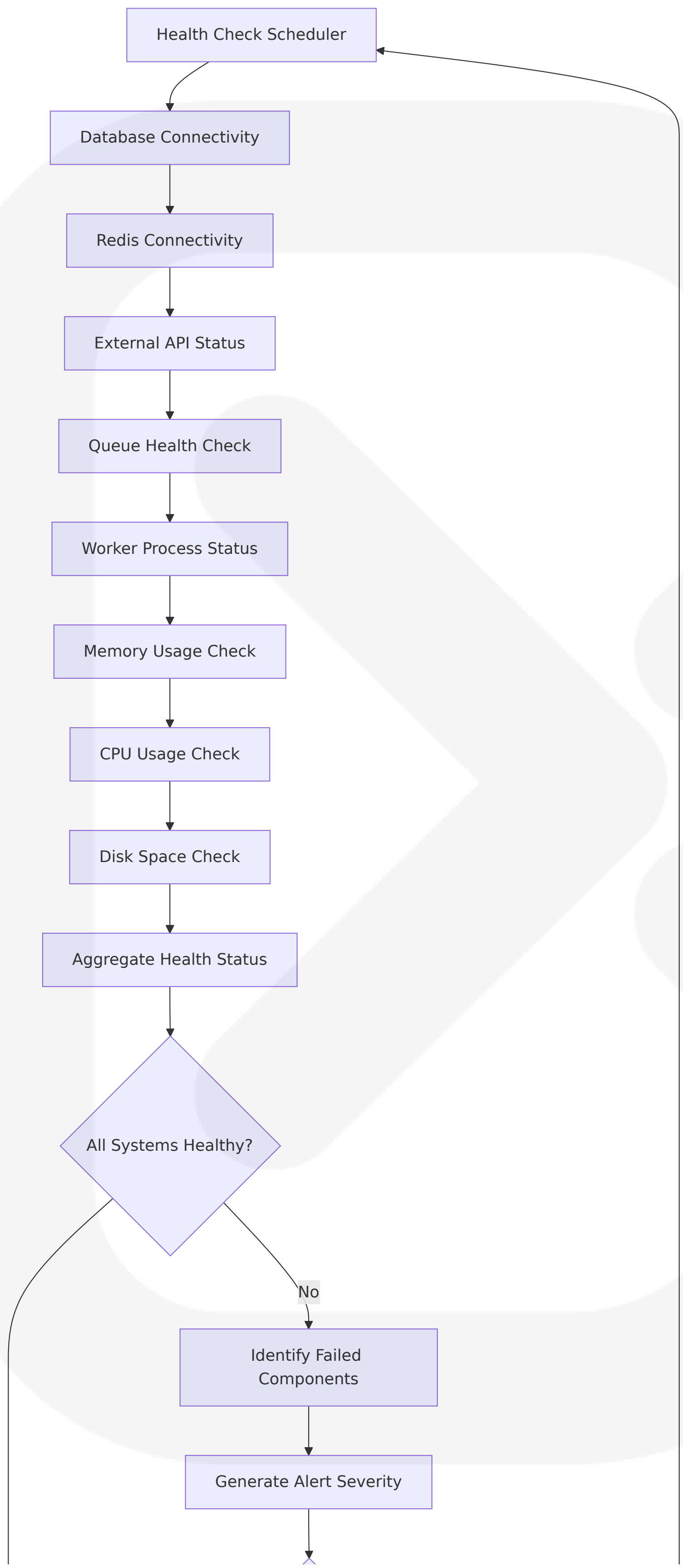
API Request Validation Pipeline

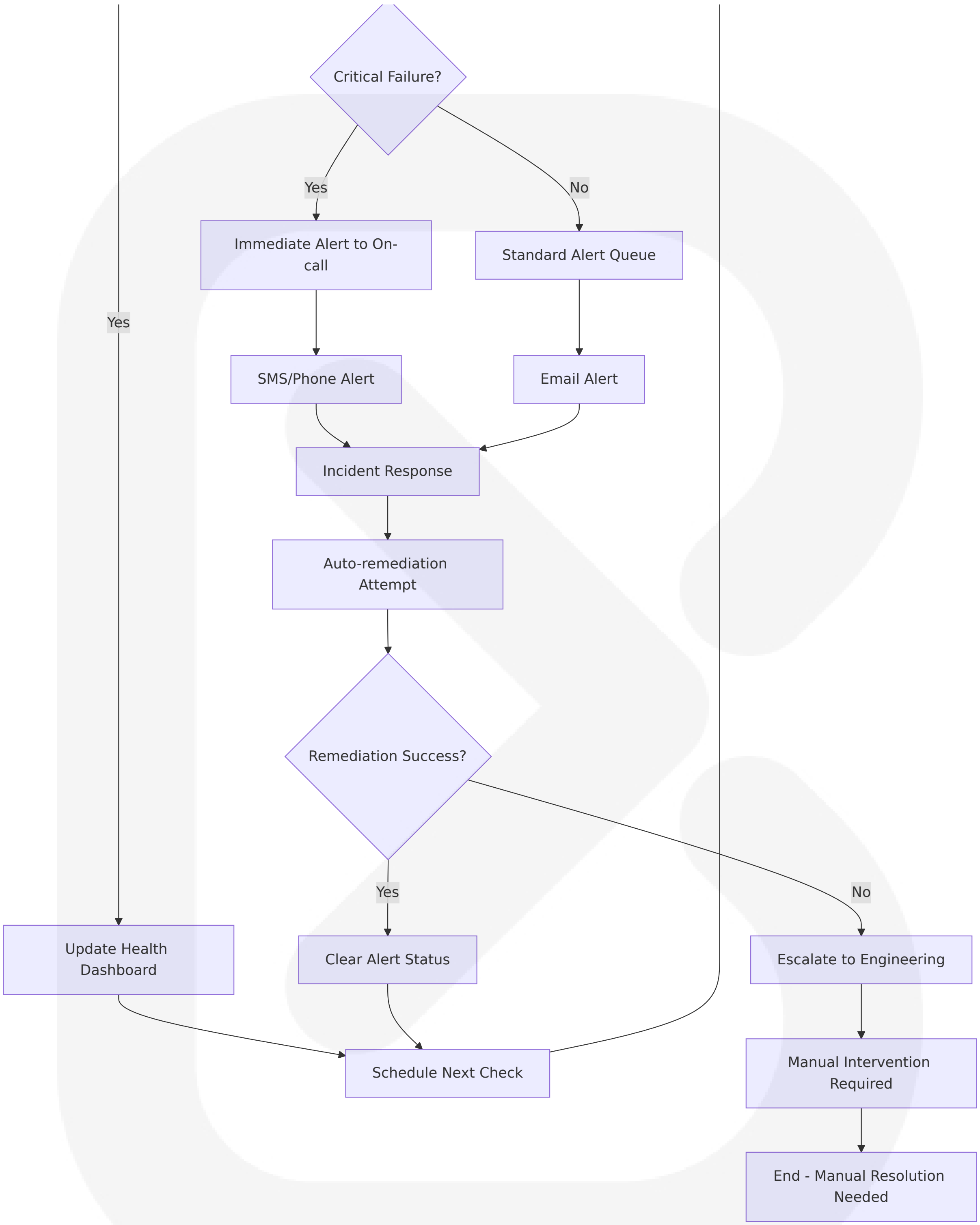

API Request
Rate Limiting Check
Rate Limit OK?
Yes
No
Authentication Validation
Return 429 Too Many Requests
Valid Auth Token?
End - Rate Limited
Yes
No
Authorization Check
Return 401 Unauthorized
User Has Permission?
End - Unauthorized
Yes
No
Input Validation
Return 403 Forbidden
Schema Validation with Zod
End - Forbidden
Schema Valid?



4.3.4 Monitoring and Alerting Workflows

System Health Monitoring Flow


Health Check Scheduler
Database Connectivity
Redis Connectivity
External API Status
Queue Health Check
Worker Process Status
Memory Usage Check
CPU Usage Check
Disk Space Check
Aggregate Health Status
All Systems Healthy?
No
Identify Failed Components
Generate Alert Severity



This comprehensive process flowchart section provides detailed workflows for all major system operations, integration patterns, error handling procedures, and technical implementation considerations. The diagrams use proper Mermaid.js syntax and include clear decision points, timing constraints, and system boundaries as required for the Yardura Service OS technical specification.

5. SYSTEM ARCHITECTURE

5.1 HIGH-LEVEL ARCHITECTURE

5.1.1 System Overview

The Yardura Service OS employs a **modern monolithic architecture** with microservice-ready patterns, built on Next.js 15 with React 19 support and App Router architecture. This architectural approach provides the operational simplicity of a monolith while maintaining clear service boundaries and integration patterns that support future decomposition as the system scales.

The system follows **Domain-Driven Design (DDD) principles** with clear bounded contexts for Lead Management, Client Operations, Field Services, Financial Management, and Wellness Analytics. Each domain maintains its own data models, business logic, and integration interfaces while sharing common infrastructure services.

Key Architectural Principles:

- **API-First Design**: All business operations expose REST APIs with comprehensive webhook support for external integrations
- **Event-Driven Architecture**: BullMQ provides fast and robust queue system with exactly-once delivery semantics for asynchronous processing
- **Progressive Web Application**: Field technician interface built as PWA with offline-first capabilities and background synchronization

- **Multi-Tenant by Design**: Database and application architecture supports franchise operations with tenant isolation and shared infrastructure

System Boundaries and Major Interfaces:

The system integrates with three critical external services: Stripe API version 2025-08-27.basil with personalized invoices and subscription improvements, QuickBooks Online API defaulting to minor version 75 as of August 1, 2025, and Google Maps Platform for route optimization. Internal boundaries separate the client-facing portal, staff operations interface, field technician PWA, and administrative dashboards.

5.1.2 Core Components Table

Component Name	Primary Respons ibility	Key Depend encies	Integration P oints
Next.js Appli cation Serve r	Request routing, se rver-side rendering, API endpoints	React 19, Type Script, Prisma ORM	All external AP Is, database, c ache
BullMQ Job P rocessing	Background task e xecution, queue m anagement	Redis, job wor kers	Billing, notifica tions, integrati ons
PostgreSQL Database	Primary data persis tence, transactiona l integrity	Prisma ORM, c onnection pool ing	All application components
Redis Cache & Sessions	Performance optimi zation, session ma nagement	BullMQ, applic ation cache	Web server, jo b queues

5.1.3 Data Flow Description

Primary Data Flows:

The system processes data through several key pipelines. **Quote-to-Lead conversion** flows from the preserved pricing estimator through API validation to CRM lead creation. **Client onboarding** integrates Stripe Setup Intent creation with subscription activation and job scheduling. **Daily dispatch operations** combine recurring schedules with one-time requests, process through route optimization algorithms, and distribute to field technician devices.

Integration Patterns:

External integrations follow **webhook-first patterns** with BullMQ handling retry logic and failure recovery. Stripe webhooks trigger subscription lifecycle events, payment processing, and billing synchronization. QuickBooks integration uses OAuth2 authentication with eventual consistency patterns for customer, invoice, and payment synchronization.

Data Transformation Points:

Critical transformation occurs at the quote normalization layer where pricing calculations convert to standardized lead data. Photo uploads trigger wellness analysis pipelines that classify 3Cs (Color/Consistency/Content) and generate trend data. Route optimization transforms job lists into optimized technician assignments with ETA calculations.

Key Data Stores and Caches:

PostgreSQL serves as the primary transactional store with Prisma ORM providing type-safe database access. Redis provides multi-layered caching including route optimization results, session data, and job queue persistence. File storage uses S3/R2 with signed URLs for secure photo access and CDN integration for global content delivery.

5.1.4 External Integration Points

System Name	Integration Type	Data Exchange Pattern	Protocol/Format
Stripe API	Payment Processing	Webhook + REST API	HTTPS/JSON, API version 2025-08-27.basil
QuickBooks Online	Accounting Sync	OAuth2 + REST API	HTTPS/JSON, Minor version 75
Google Maps Platform	Route Optimization	REST API	HTTPS/JSON, Distance Matrix API
Communication Services	Notifications	REST API + Webhooks	HTTPS/JSON, SMS/Email delivery

5.2 COMPONENT DETAILS

5.2.1 Next.js Application Server

Purpose and Responsibilities:

The Next.js 15 application server serves as the primary application runtime, handling HTTP request routing, server-side rendering, and API endpoint management. The App Router provides opinionated caching defaults with GET Route Handlers and Client Router Cache now uncached by default, requiring careful caching strategy implementation for optimal performance.

Technologies and Frameworks:

Built on Node.js 20+ runtime with TypeScript 5.0+ for type safety across the full stack. Leverages React Server Components (RSC) by default with all page and layout files as Server Components rendering on the server. Client Components marked with "use client" directive handle interactivity and browser APIs.

Key Interfaces and APIs:

Exposes REST API endpoints through App Router route handlers for all business operations. Server Actions provide form handling and mutations with automatic POST endpoint generation. WebSocket connections support real-time updates for dispatch operations and field technician tracking.

Data Persistence Requirements:

Integrates with PostgreSQL through Prisma ORM with connection pooling for scalability. Implements database migrations and schema versioning for production deployments. Supports read replicas for reporting queries and write/read separation patterns.

Scaling Considerations:

Stateless architecture enables horizontal scaling with load balancing. Session data stored in Redis for multi-instance compatibility. Background job processing separated into dedicated worker processes for independent scaling.

5.2.2 BullMQ Job Processing System

Purpose and Responsibilities:

BullMQ implements a fast and robust queue system with high performance, trying to get the highest possible throughput from Redis by combining efficient Lua scripts and pipelining. Handles background processing for billing cycles, route optimization, wellness analysis, and external API synchronization.

Technologies and Frameworks:

BullMQ is a fast, robust and opinionated message queue library for Node.js based on Redis, implementing job queues that handle background jobs, scheduling recurring tasks, and retrying failed jobs. Requires Redis 6.2+ for full compatibility and optimal performance.

Key Interfaces and APIs:

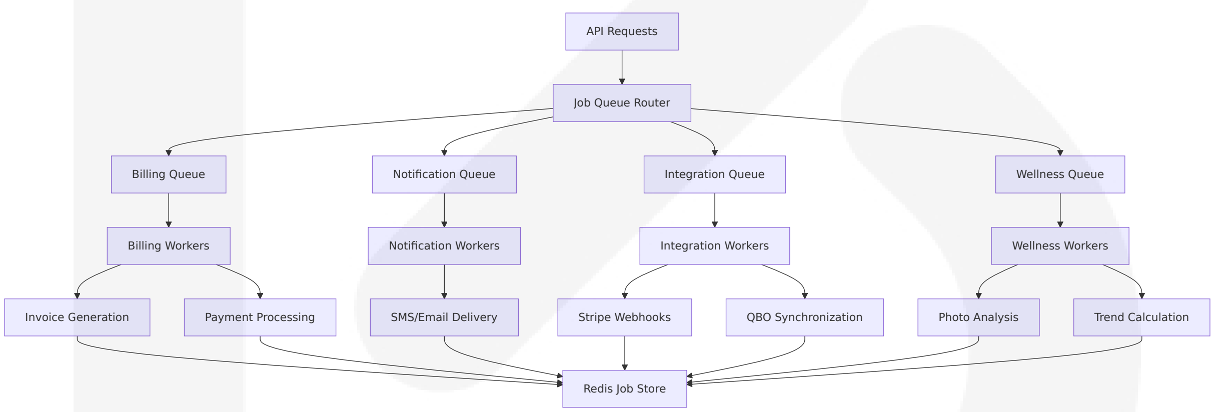
Provides job queue interfaces for different processing categories: billing operations, notification delivery, wellness analysis, external API synchronization, and route optimization. Supports job prioritization, delayed execution, and retry mechanisms with exponential backoff.

Data Persistence Requirements:

Uses Redis data types like lists, sets, sorted sets, and hashes to store and manage jobs, with each job stored as a JSON object inside a Redis hash, indexed by job ID. Implements job state persistence for recovery and monitoring.

Scaling Considerations:

Easy to scale horizontally by adding more workers for processing jobs in parallel. Worker processes can be distributed across multiple servers with Redis as the central coordination point.



5.2.3 PostgreSQL Database Layer

Purpose and Responsibilities:

Serves as the primary transactional data store with ACID compliance for all business-critical operations. Manages complex relational data including multi-tenant franchise structures, subscription billing, job scheduling, and wellness analytics with referential integrity.

Technologies and Frameworks:

PostgreSQL 14+ with Prisma ORM providing type-safe database access and migration management. Implements connection pooling for performance optimization and supports read replica configurations for reporting workloads.

Key Interfaces and APIs:

Prisma Client provides the primary database interface with generated TypeScript types. Raw SQL support for complex queries and reporting. Database triggers and stored procedures for data consistency enforcement.

Data Persistence Requirements:

Multi-tenant database design with tenant isolation through row-level security policies. Automated backup strategies with point-in-time recovery capabilities. Data retention policies for compliance with privacy regulations.

Scaling Considerations:

Vertical scaling for primary database with read replica support for reporting queries. Database partitioning strategies for high-volume tables like job records and photo metadata. Connection pooling and query optimization for performance at scale.

5.2.4 Redis Cache and Session Management

Purpose and Responsibilities:

Provides high-performance caching layer for route optimization results, session management, and BullMQ job queue persistence. Implements distributed locking for concurrent operations and rate limiting for API endpoints.

Technologies and Frameworks:

Redis 6.2+ with ioredis client library for Node.js integration. Supports Redis Cluster for horizontal scaling and Redis Sentinel for high availability configurations.

Key Interfaces and APIs:

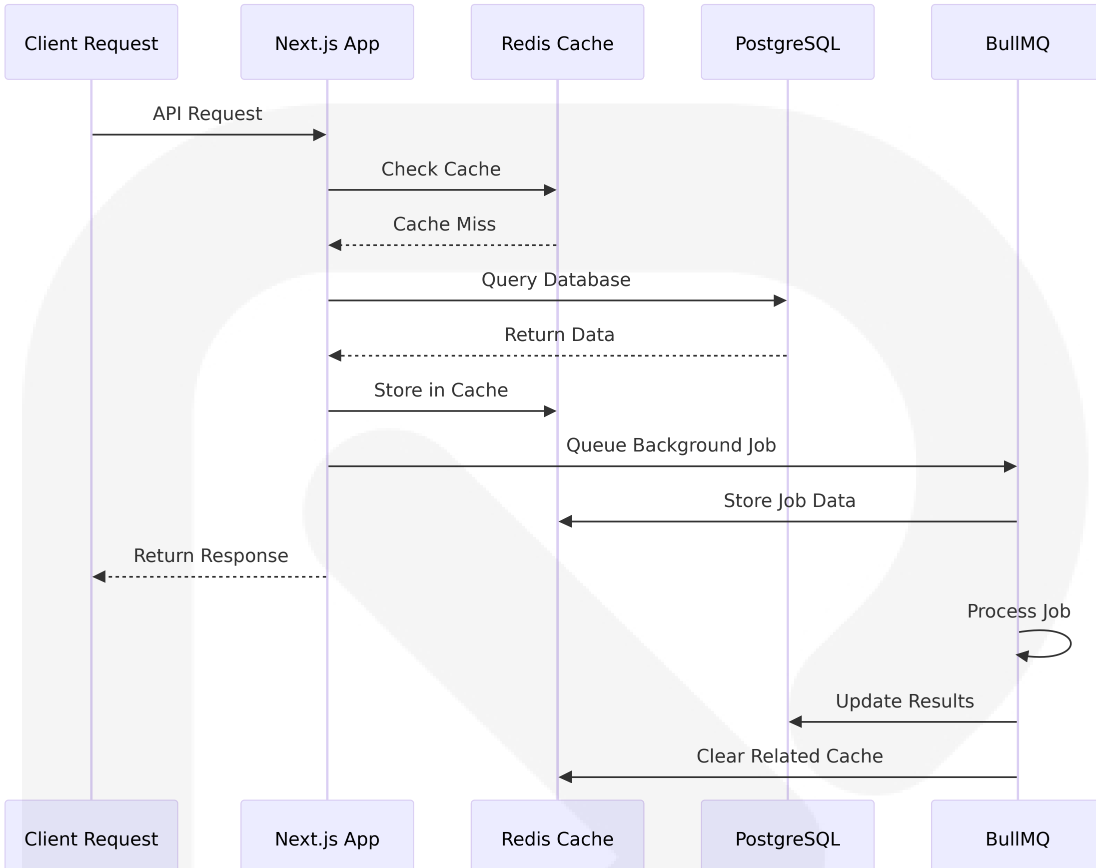
Standard Redis commands through ioredis client with connection pooling. Pub/Sub capabilities for real-time notifications. Lua scripting for atomic operations and complex data manipulations.

Data Persistence Requirements:

Configurable persistence options from in-memory only to full disk persistence. Backup and replication strategies for production deployments. Memory optimization for large datasets and cache eviction policies.

Scaling Considerations:

Redis connections have quite low overhead, so multiple connections can be used unless service provider imposes hard limitations. Supports horizontal scaling through Redis Cluster and vertical scaling through memory optimization.



5.3 TECHNICAL DECISIONS

5.3.1 Architecture Style Decisions and Tradeoffs

Monolithic Architecture with Microservice Patterns

The decision to implement a monolithic architecture with clear service boundaries provides operational simplicity while maintaining future flexibility. This approach reduces deployment complexity, simplifies debugging, and enables rapid development cycles essential for the initial product launch.

Tradeoffs:

- **Benefits**: Simplified deployment, easier debugging, reduced network latency, transactional consistency
- **Drawbacks**: Potential scaling bottlenecks, technology lock-in, larger deployment units
- **Mitigation**: Clear domain boundaries, API-first design, event-driven patterns for future decomposition

Next.js Full-Stack Framework Selection

Next.js 15 aligns with React 19 release with extensive testing across real-world applications providing confidence in stability. The framework provides server-side rendering, API routes, and build optimizations essential for the comprehensive business management platform.

Tradeoffs:

- **Benefits**: Unified development experience, optimized performance, strong TypeScript integration
- **Drawbacks**: Framework lock-in, learning curve for team, potential over-engineering for simple features
- **Mitigation**: Gradual adoption, clear separation of business logic, comprehensive testing strategy

5.3.2 Communication Pattern Choices

Event-Driven Architecture with BullMQ

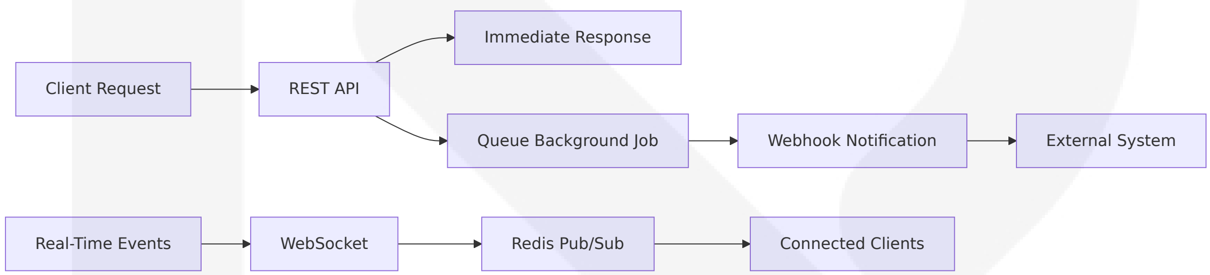
BullMQ implements producer-consumer pattern where producers add jobs to queues while separate consumers remove and process jobs, decoupling production and consumption into separate concerns. This pattern enables asynchronous processing essential for billing cycles, notifications, and external API integrations.

REST API with Webhook Integration

Primary communication uses REST APIs for synchronous operations with webhook patterns for asynchronous event notification. This hybrid approach provides immediate response for user interactions while enabling reliable background processing.

WebSocket for Real-Time Updates

Dispatch operations and field technician tracking require real-time updates implemented through WebSocket connections with Redis Pub/Sub for multi-instance coordination.



5.3.3 Data Storage Solution Rationale

PostgreSQL as Primary Database

PostgreSQL provides ACID compliance, complex query capabilities, and JSON support essential for multi-tenant operations and flexible data models. The relational model supports complex business relationships while JSON columns enable schema flexibility for evolving requirements.

Redis for Caching and Queues

Redis provides the speed, consistency, and scalability needed for modern backend tasks. The in-memory architecture delivers sub-millisecond response times for route optimization caching and session management.

S3/R2 for File Storage

Object storage provides scalable, cost-effective solution for photo storage with CDN integration for global content delivery. Signed URLs ensure security while maintaining performance for client access.

5.3.4 Caching Strategy Justification

Multi-Layer Caching Architecture

Implements caching at multiple levels: CDN for static assets, Redis for application data, and database query result caching. This approach optimizes performance across different access patterns and data types.

Cache Invalidation Strategy

Event-driven cache invalidation ensures data consistency while maintaining performance. BullMQ jobs handle complex invalidation patterns that span multiple cache keys and data relationships.

Performance Requirements Alignment

Caching strategy designed to meet specific performance targets: portal P95 < 500ms cached, route optimization ≤ 10s for 100 stops, and tech app job list render < 200ms from cache.

5.3.5 Security Mechanism Selection

Role-Based Access Control (RBAC)

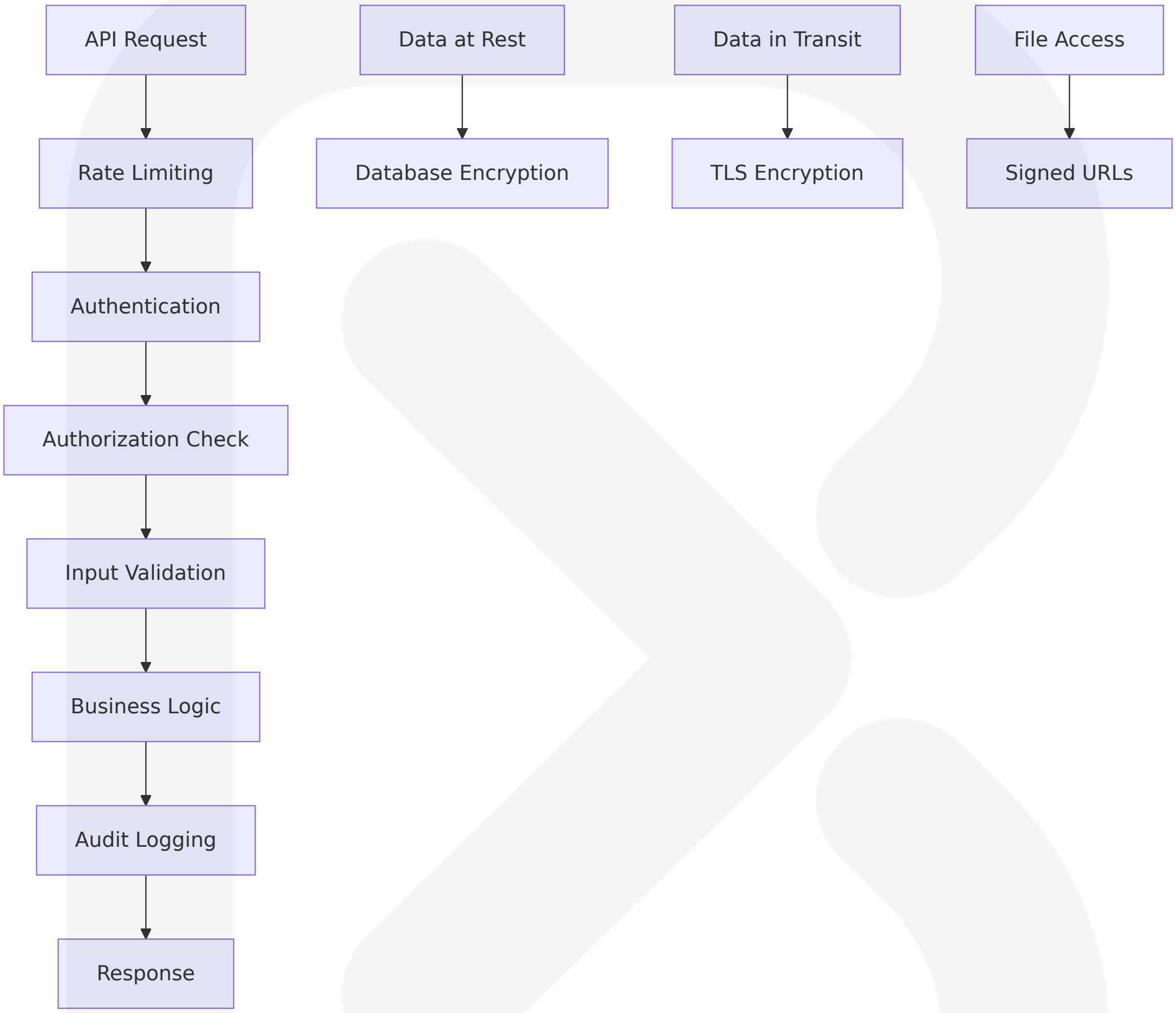
Implements comprehensive RBAC system with role hierarchy supporting franchise operations. Permissions cascade from parent accounts to subaccounts while maintaining tenant isolation.

API Security Patterns

All API endpoints implement authentication, authorization, input validation, and rate limiting. Audit logging captures all business-critical operations for compliance and forensic analysis.

Data Protection Strategy

PII encryption at rest, TLS for data in transit, and signed URLs with time-based expiration for file access. Payment data handled through Stripe Elements to maintain PCI DSS compliance.



5.4 CROSS-CUTTING CONCERNS

5.4.1 Monitoring and Observability Approach

Application Performance Monitoring

Implements comprehensive monitoring using Vercel Analytics for Next.js-specific metrics and Sentry for error tracking and performance monitoring. Custom metrics track business KPIs including route optimization performance, billing success rates, and client portal usage patterns.

Distributed Tracing

Request tracing spans across API endpoints, database queries, external API calls, and background job processing. Correlation IDs enable end-to-end request tracking through complex workflows involving multiple system components.

Business Metrics Dashboard

Real-time dashboards display operational metrics including active technician count, job completion rates, billing processing status, and system health indicators. Alerts trigger on threshold violations for critical business operations.

5.4.2 Logging and Tracing Strategy

Structured Logging Architecture

All application components emit structured JSON logs with consistent schema including timestamp, correlation ID, user context, and business operation metadata. Log aggregation enables complex queries and analysis across distributed system components.

Audit Trail Implementation

Comprehensive audit logging captures all business-critical operations including billing changes, payroll processing, schedule modifications, and data exports. Audit records include user identity, timestamp, operation details, and data change tracking.

Log Retention and Compliance

Implements tiered log retention with immediate access for operational logs, medium-term storage for audit trails, and long-term archival for compliance requirements. Log data anonymization supports privacy regulation compliance.

5.4.3 Error Handling Patterns

Graceful Degradation Strategy

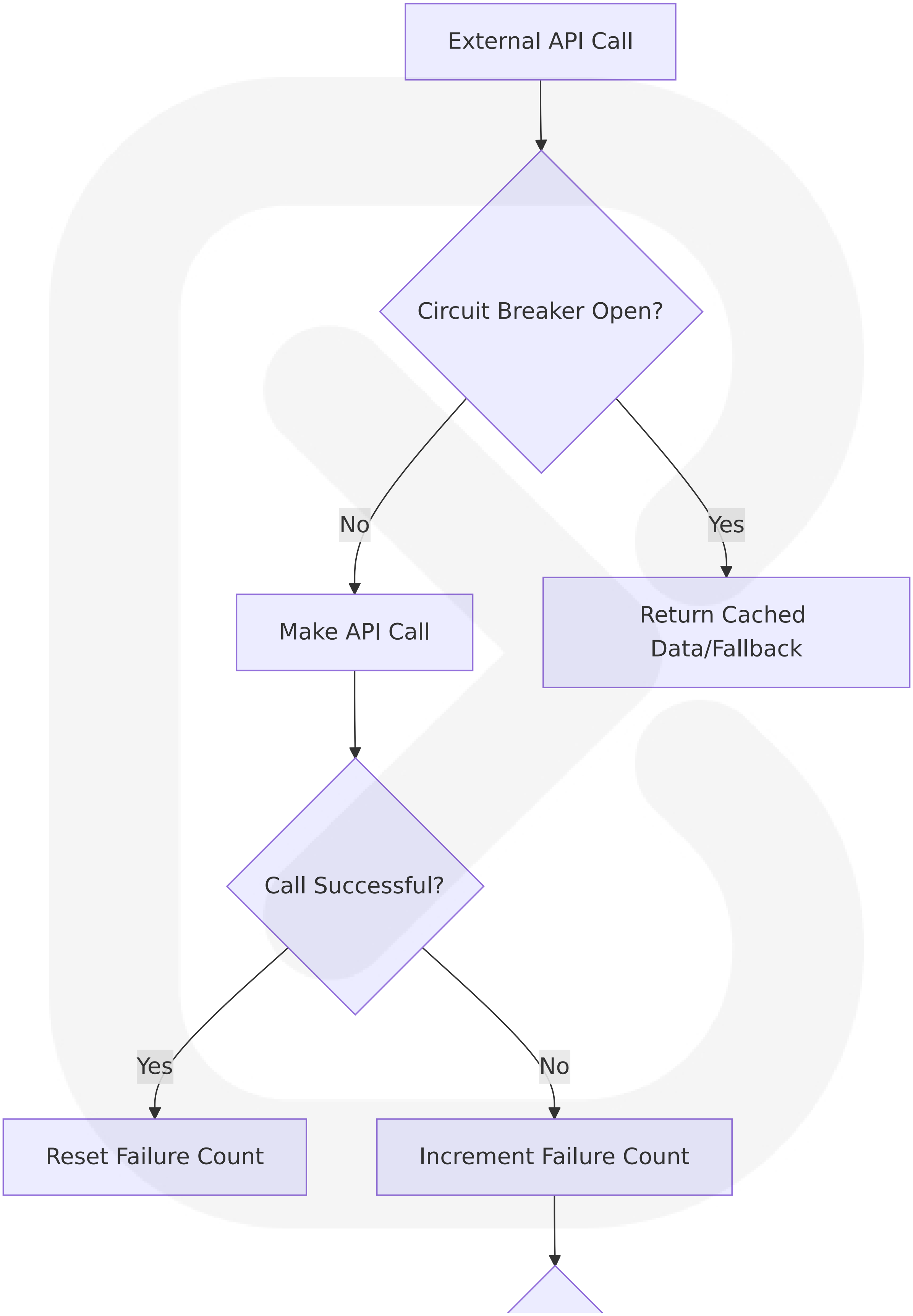
System components implement graceful degradation when external services become unavailable. Route optimization falls back to basic algorithms when Google Maps API fails, and billing operations queue for retry when Stripe API is unavailable.

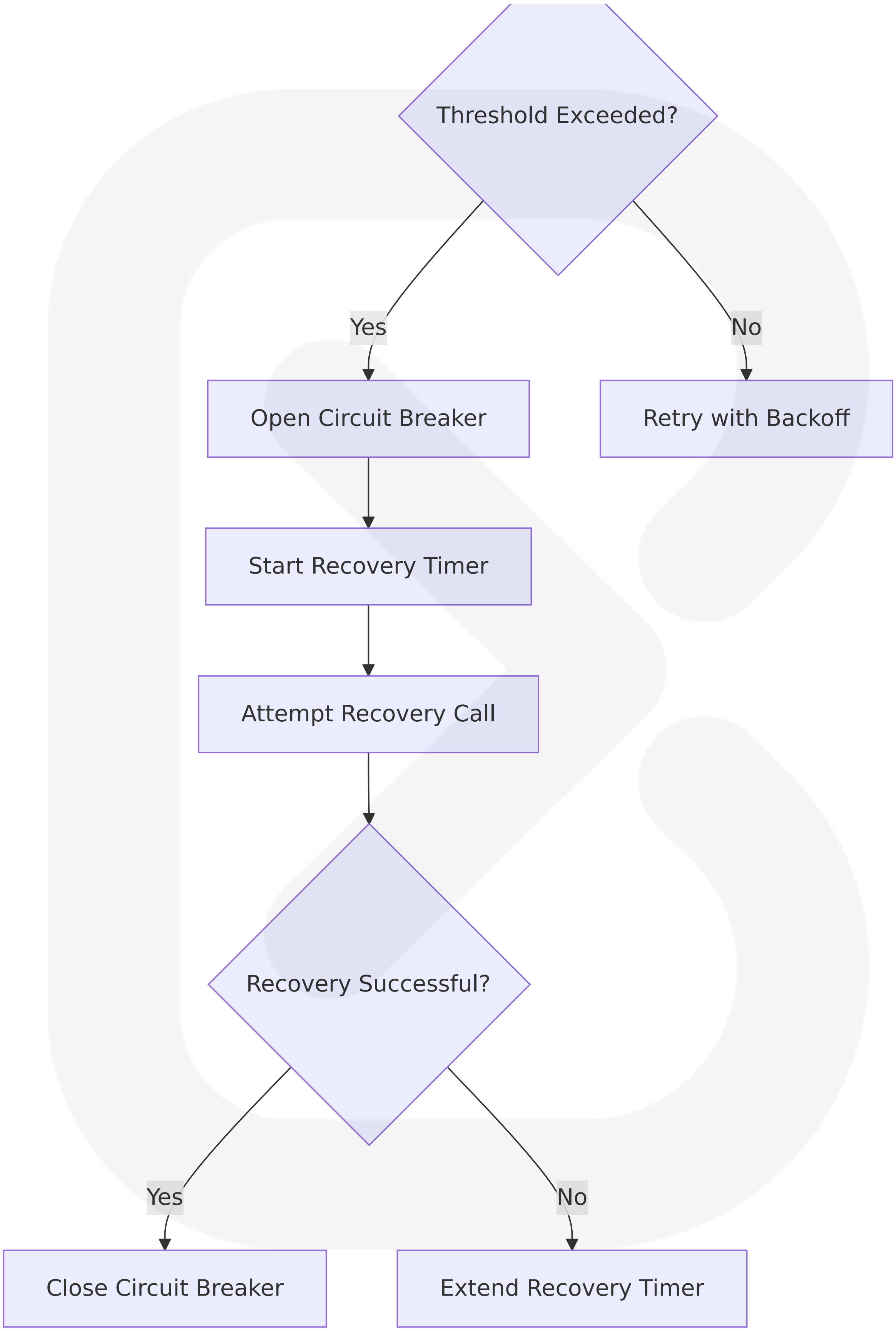
Circuit Breaker Pattern

External API integrations implement circuit breaker patterns to prevent cascade failures. Failed requests trigger circuit opening with exponential backoff retry strategies and automatic recovery detection.

User Experience Preservation

Error handling prioritizes user experience preservation through meaningful error messages, alternative workflows, and offline capabilities where applicable. Field technician PWA maintains full functionality during network interruptions.


External API Call
Circuit Breaker Open?
No
Yes
Make API Call
Return Cached Data/Fallback
Call Successful?
Yes
No
Reset Failure Count
Increment Failure Count


Threshold Exceeded?
Yes
No
Open Circuit Breaker
Retry with Backoff
Start Recovery Timer
Attempt Recovery Call
Recovery Successful?
Yes
No
Close Circuit Breaker
Extend Recovery Timer

5.4.4 Authentication and Authorization Framework

Multi-Tenant Authentication

NextAuth provides OAuth2 and credential-based authentication with session management stored in Redis for multi-instance compatibility. Tenant context embedded in session data enables automatic data isolation and access control.

Hierarchical Authorization Model

RBAC implementation supports franchise hierarchies with permission inheritance and override capabilities. Parent account users can access subaccount data while maintaining appropriate restrictions and audit trails.

API Security Implementation

All API endpoints require authentication with JWT tokens for external integrations and session-based authentication for web interfaces. Rate limiting prevents abuse while audit logging captures all access attempts.

5.4.5 Performance Requirements and SLAs

Response Time Targets

System architecture designed to meet specific performance requirements: portal P95 < 500ms cached, route optimization ≤ 10s for 100 stops, tech app job list render < 200ms from cache, and image upload ≤ 3s on LTE networks.

Availability Requirements

Target 99.9% uptime for staff and client portals with planned maintenance windows during low-usage periods. Database backup and recovery procedures support RTO of 4 hours and RPO of 15 minutes for business continuity.

Scalability Planning

Architecture supports horizontal scaling of web servers and background workers with database scaling through read replicas and connection pooling. Auto-scaling policies respond to traffic patterns and resource utilization metrics.

5.4.6 Disaster Recovery Procedures

Backup and Recovery Strategy

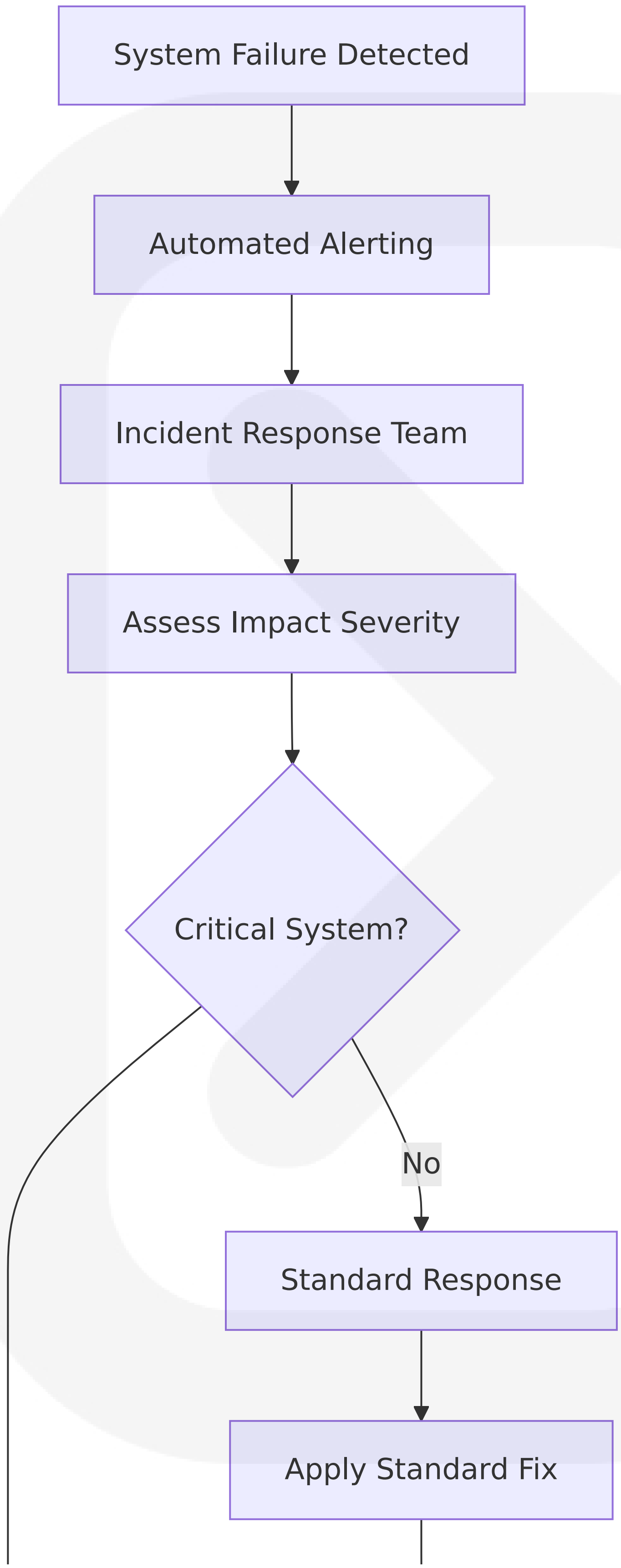
Automated daily database backups with point-in-time recovery capabilities support business continuity requirements. File storage implements cross-region replication for photo data protection and CDN failover capabilities.

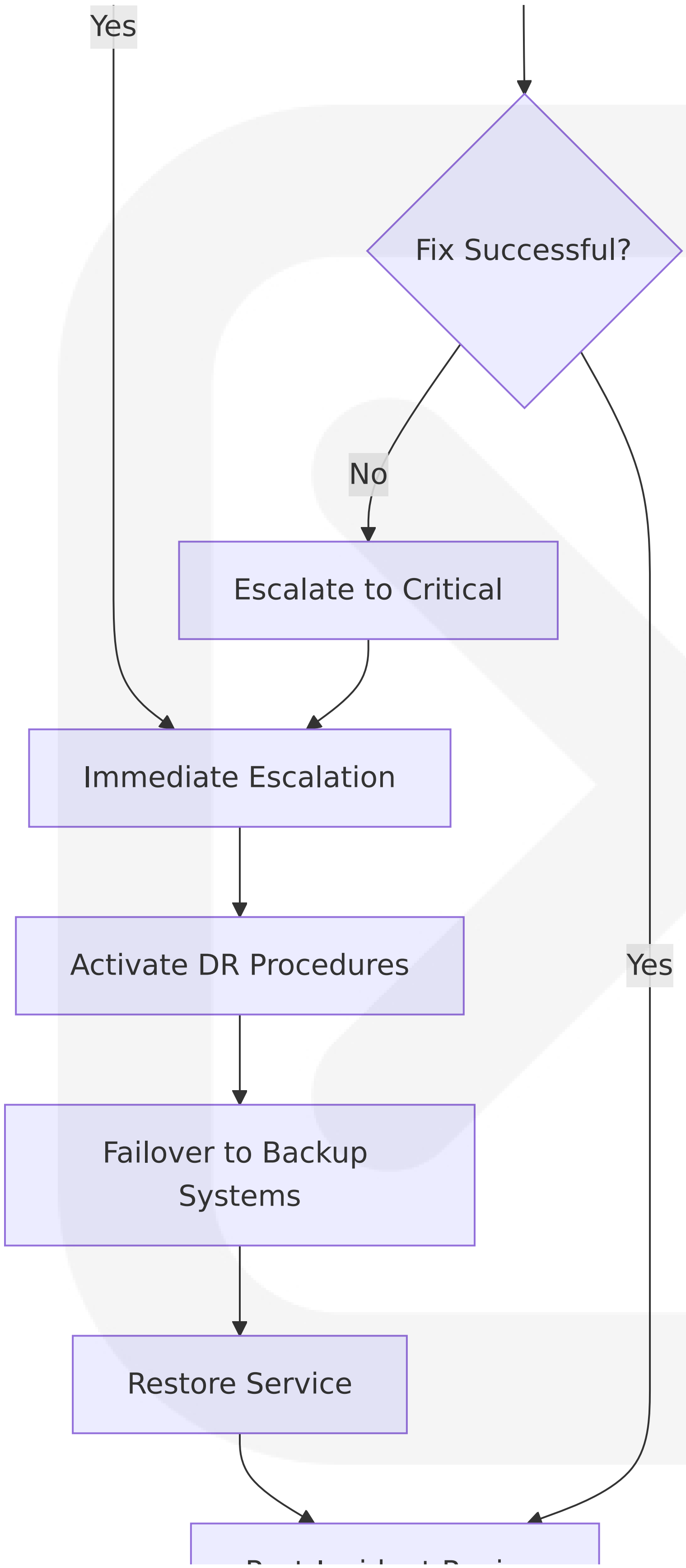
Incident Response Procedures

Defined incident response procedures include automated alerting, escalation paths, and communication protocols. Runbooks provide step-by-step recovery procedures for common failure scenarios including database corruption, external API outages, and infrastructure failures.

Business Continuity Planning

Critical business operations maintain functionality during partial system failures through graceful degradation and offline capabilities. Field technician PWA enables continued service delivery during network interruptions with automatic synchronization upon reconnection.


System Failure Detected
Automated Alerting
Incident Response Team
Assess Impact Severity
Critical System?
No
Standard Response
Apply Standard Fix


Yes
Fix Successful?
No
Escalate to Critical
Immediate Escalation
Activate DR Procedures
Yes
Failover to Backup Systems
Restore Service

6. SYSTEM COMPONENTS DESIGN

6.1 CORE APPLICATION COMPONENTS

6.1.1 Next.js Application Server Architecture

Component Overview

The Next.js 15 application server utilizes App Router architecture with TypeScript, Server Actions, Server and Client Components, providing the primary runtime environment for the Yardura Service OS. Next.js by Vercel is the full-stack React framework for the web, built on the latest React features with React 19 support and comprehensive server-side rendering capabilities.

Technical Architecture

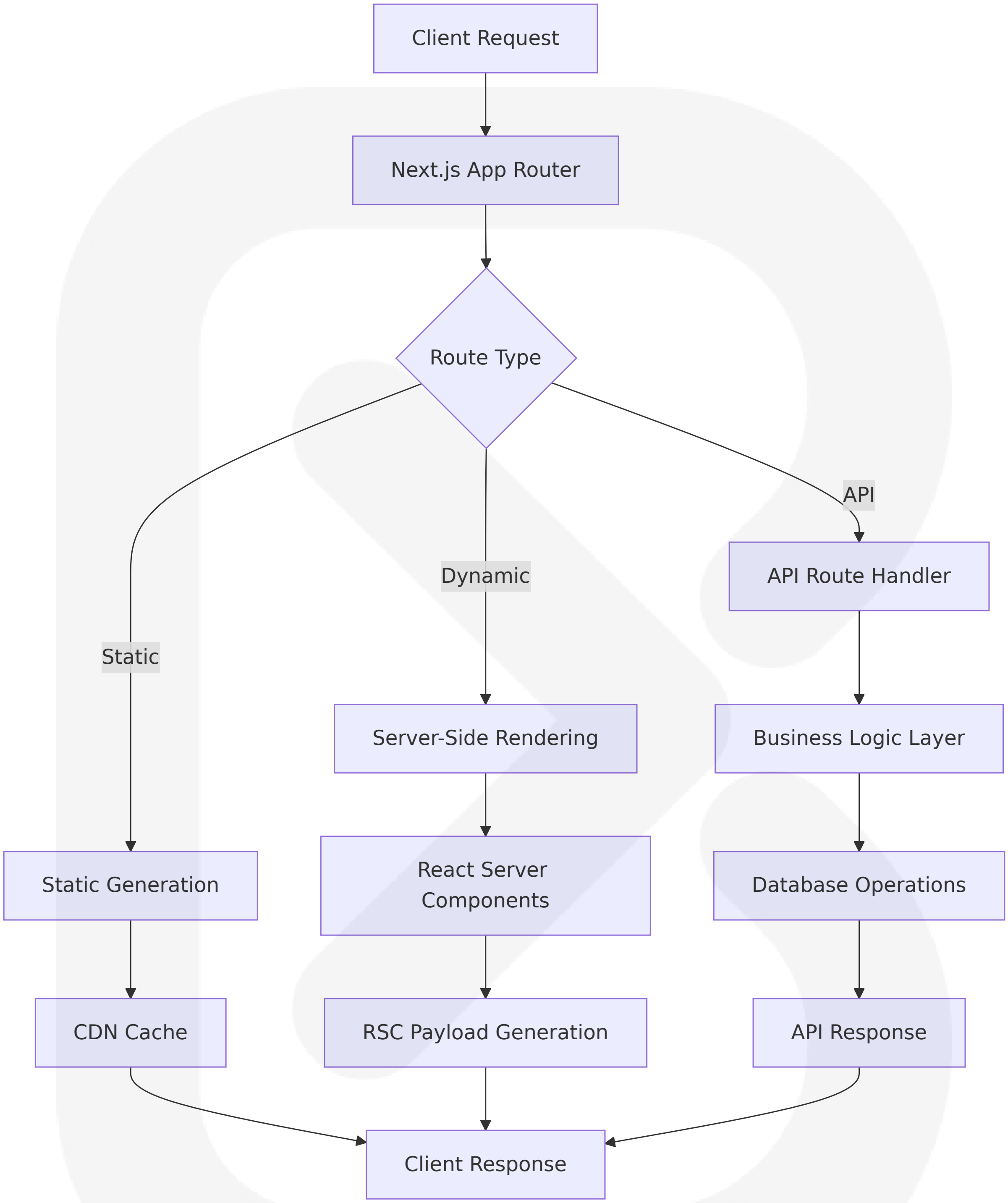
Layer	Technology	Purpose	Implementation Details
Presentation Layer	React 19 Server/Client Components	UI rendering and interactivity	Server Components rendered into RSC Payload, Client Components for prerendered HTML
API Layer	Next.js API Routes	RESTful endpoints and webhooks	API endpoints to securely connect with third-party services for handling auth or listening for webhooks
Server Actions	React Server Actions	Form handling and mutations	Run server code by calling a function, skip the API

Layer	Technology	Purpose	Implementation Details
Routing Layer	App Router	File-system based routing	Create routes using the file system, including support for more advanced routing patterns and UI layouts

Component Responsibilities

- **Request Processing**: Handles HTTP requests through App Router with automatic route generation based on file system structure
- **Server-Side Rendering**: Next.js uses React's APIs to orchestrate rendering, split into chunks by individual route segments
- **API Endpoint Management**: Provides RESTful API endpoints for all business operations including quote processing, client onboarding, and webhook handling
- **Authentication Integration**: Integrates with NextAuth for session management and role-based access control
- **Static Asset Optimization**: Flexible rendering and caching options, including Incremental Static Regeneration (ISR), on a per-page level

Performance Optimizations



Scaling Considerations

- **Horizontal Scaling**: Stateless architecture enables multiple application instances behind load balancers

- **Edge Computing**: Take control of the incoming request, use code to define routing and access rules through middleware for global distribution
- **Resource Optimization**: Reduce the amount of JavaScript sent to the browser, improve the First Contentful Paint (FCP)

6.1.2 BullMQ Job Processing Engine

Component Overview

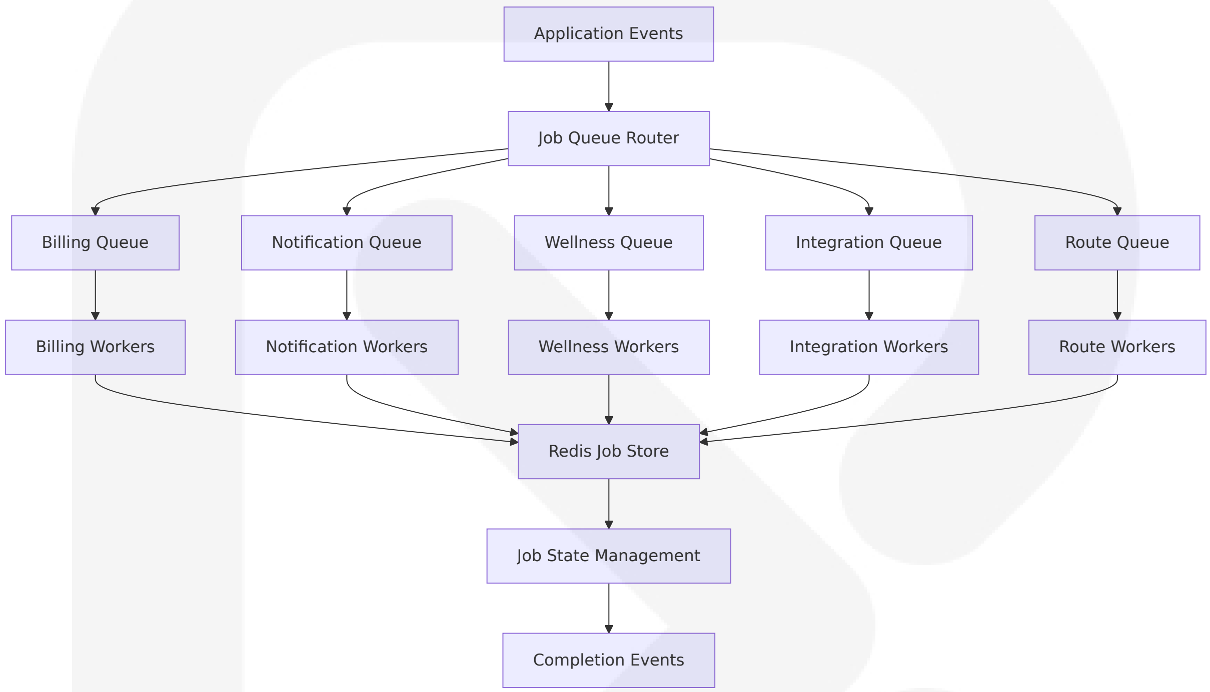
BullMQ is a Node.js library that implements a fast and robust queue system built on top of Redis with exactly once queue semantics, serving as the backbone for all asynchronous operations in the Yardura Service OS. BullMQ is a lightweight, robust, and fast NodeJS library for creating background jobs and sending messages using queues, designed to be easy to use but also powerful and highly configurable.

Queue Architecture Design

Queue Name	Purpose	Worker Pool Size	Processing Pattern
billing-queue	Invoice generation, payment processing, dunning	5 workers	Sequential with retry
notification-queue	SMS, email, push notifications	10 workers	Parallel processing
wellness-queue	Photo analysis, trend calculation	3 workers	CPU-intensive batch
integration-queue	Stripe webhooks, QuickBooks sync	5 workers	External API calls
route-queue	Route optimization, job assignment	2 workers	Memory-intensive

Technical Implementation

A message queue works by having a producer component add a job or message to the queue, while a separate consumer component removes jobs from the queue and processes them, decoupling the production and consumption of jobs into separate concerns.



Job Lifecycle Management

Robust job lifecycle handling with states like waiting, active, delayed, completed, failed, repeated etc. ensures reliable processing across all business operations.

Performance Characteristics

- **Throughput Optimization**: High performant, try to get the highest possible throughput from Redis by combining efficient .lua scripts and pipelining
- **Horizontal Scaling**: Easy to scale horizontally, add more workers for processing jobs in parallel
- **Reliability**: The fastest, most reliable, Redis-based distributed queue for Node, carefully written for rock solid stability and atomicity

Connection Management

Redis connections have quite low overhead, so you should not need to care about reusing connections unless your service provider imposes hard limitations. The system implements connection pooling strategies optimized for different queue types and processing patterns.

6.1.3 PostgreSQL Database Layer

Component Overview

The PostgreSQL data source connector connects Prisma ORM to a PostgreSQL database server, with the PostgreSQL connector containing a database driver responsible for connecting to your database. The database layer serves as the primary transactional data store with ACID compliance for all business-critical operations.

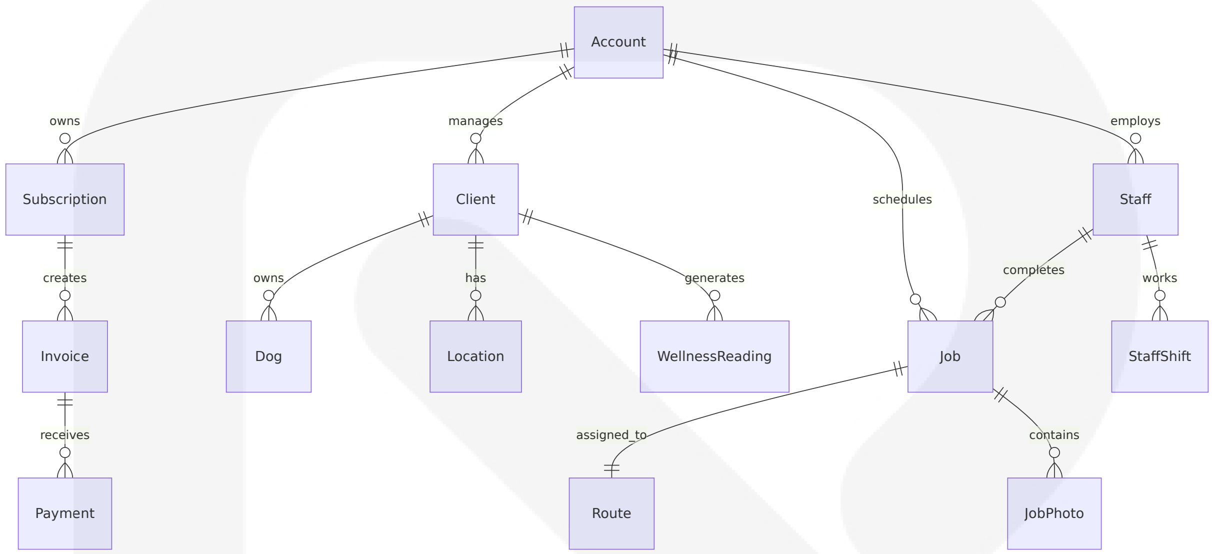
Database Architecture

Schema	Purpose	Key Tables	Scaling Strategy
core	User management, authentication	users, roles, sessions	Read replicas
business	Clients, subscriptions, billing	clients, subscriptions, invoices	Partitioning by tenant
operations	Jobs, routes, schedules	jobs, routes, schedules	Time-based partitioning
analytics	Wellness data, reporting	wellness_readings, metrics	Separate analytics DB
audit	Compliance, change tracking	audit_logs, data_changes	Long-term archival

Prisma ORM Integration

The ORM you know and love: fully type-safe queries, easy schema management, migrations and auto-completion. Productivity becomes higher because it gets combined with end-to-end type-safety using TypeScript.

Multi-Tenant Data Architecture



Performance Optimization Strategies

- **Connection Pooling**: Connection pooling, query caching, and automated backups are all included in the database configuration
- **Query Optimization**: A prepared statement is a feature that can be used to optimize performance, parsed, compiled, and optimized only once
- **Type Safety**: Prisma ORM strikes an excellent balance by providing type-safe database access with an intuitive API while maintaining transparency and control over database operations

Data Persistence Requirements

- **ACID Compliance**: Full transactional integrity for billing, payroll, and schedule operations
- **Multi-Tenant Isolation**: Row-level security policies ensure tenant data separation
- **Backup Strategy**: Automated backups are all included with point-in-time recovery capabilities

- **Migration Management**: The migrations are a breeze, and I love how easy it is to get a full view of your database from the Prisma schema file

6.1.4 Redis Cache and Session Management

Component Overview

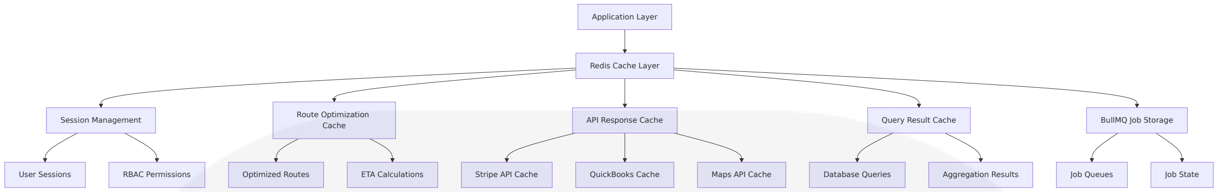
At the core of BullMQ's architecture is Redis, an in-memory data structure store that enables the queue system to be fast, reliable, and persistent. Redis provides the speed, consistency, and scalability needed for modern backend tasks.

Cache Architecture Design

Cache Layer	Purpose	TTL Strategy	Eviction Policy
Session Cache	User authentication, RBAC	24 hours	LRU
Route Cache	Optimized routes, ETAs	1 hour	TTL-based
API Cache	External API responses	15 minutes	TTL-based
Query Cache	Database query results	5 minutes	LRU with TTL
File Cache	Signed URLs, metadata	15 minutes	TTL-based

Data Structure Utilization

BullMQ uses Redis data types like lists, sets, sorted sets, and hashes to store and manage jobs, optimizing for different access patterns and performance requirements.



Performance Characteristics

- **Sub-millisecond Response**: In-memory architecture delivers optimal performance for frequently accessed data
- **Pub/Sub Capabilities**: BullMQ uses Redis' Pub/Sub system to broadcast real-time events (such as job completion or failure) to workers and dashboards
- **Horizontal Scaling**: Multiple workers can interact with the same Redis instance, enabling BullMQ Redis setups to scale horizontally across machines and environments

Connection Management Strategy

BullMQ uses the node module ioredis, and the options you pass to BullMQ are just passed to the constructor of ioredis. If you do not provide any options, it will default to port 6379 and localhost.

6.2 INTEGRATION COMPONENTS

6.2.1 Stripe Payment Processing Integration

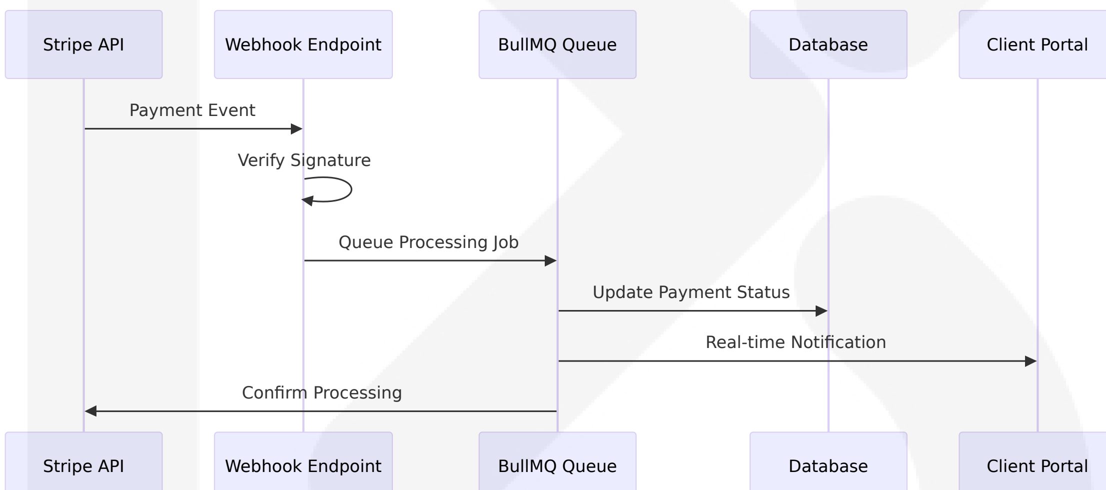
Component Architecture

The Stripe integration component handles all payment processing operations using API version 2025-08-27.basil, which introduces personalized invoices, ad hoc pricing for Payment Links, and billing improvements with mixed duration subscription phases.

Integration Patterns

Operation Type	Stripe API Endpoint	Webhook Events	Error Handling
Setup Intent	`/v1/setup_intents`	`setup_intent.succeeded`	Retry with exponential backoff
Subscription Management	`/v1/subscriptions`	`customer.subscription.*`	Dunning process automation
Invoice Processing	`/v1/invoices`	`invoice.payment_succeeded`	Failed payment recovery
Payment Methods	`/v1/payment_methods`	`payment_method.attached`	Card validation errors
Refund Processing	`/v1/refunds`	`charge.dispute.created`	Dispute management

Webhook Processing Architecture



Data Synchronization Patterns

- **Idempotent Operations**: All Stripe operations use idempotency keys to prevent duplicate processing
- **Eventual Consistency**: Webhook processing ensures eventual consistency between Stripe and local database
- **Retry Mechanisms**: Failed webhook processing triggers exponential backoff retry strategies

6.2.2 QuickBooks Online Synchronization

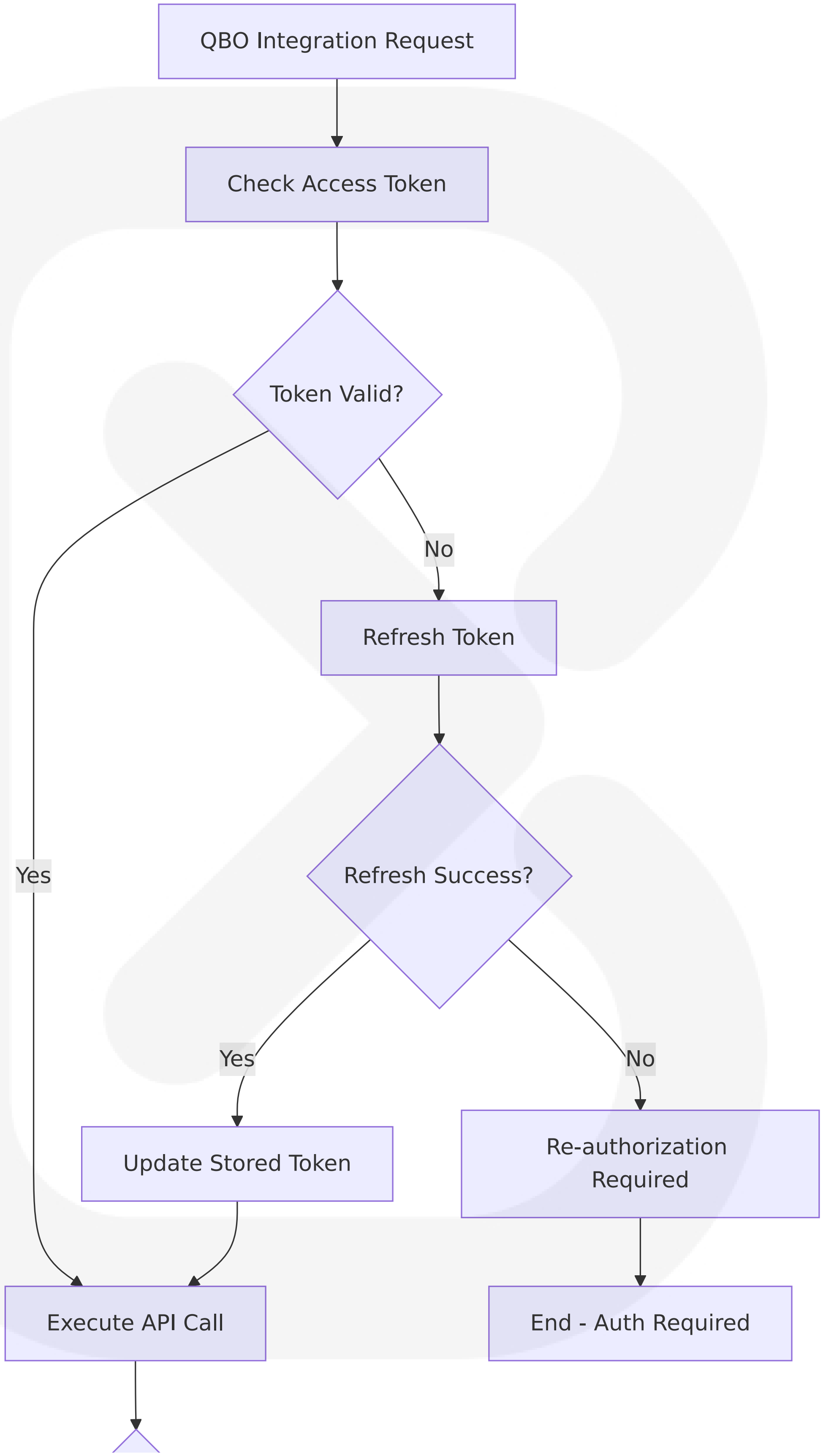
Component Architecture

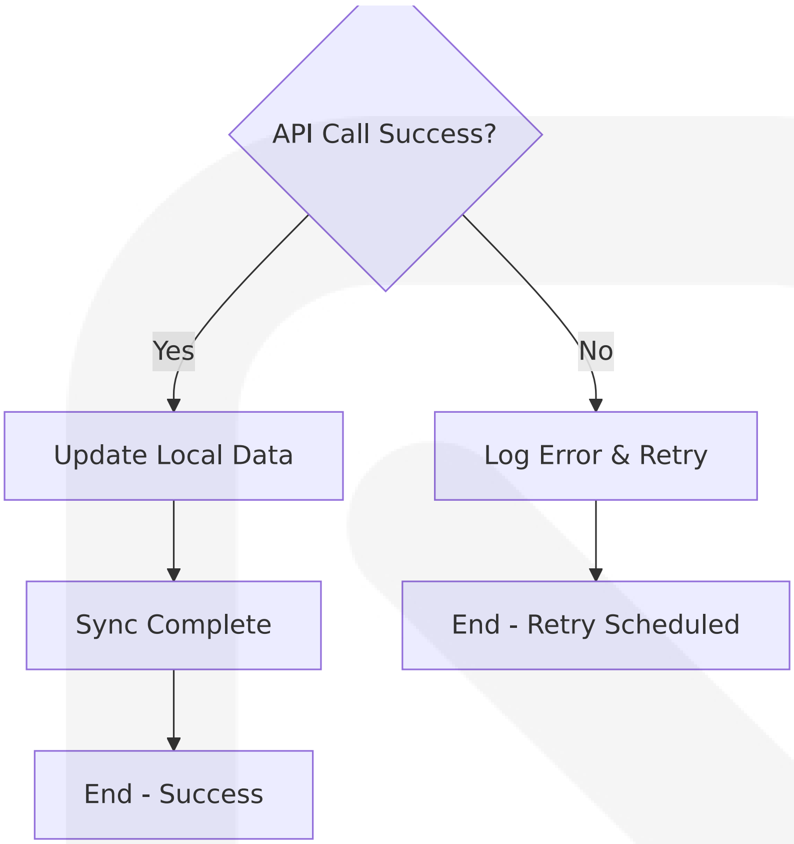
Starting August 1, 2025, all API requests to the QuickBooks Online Accounting API will default to minor version 75, requiring updated integration patterns for customer, invoice, and payment synchronization.

Synchronization Components

Entity Type	QBO Endpoint	Sync Frequency	Conflict Resolution
Customers	`/v3/companyid/customers`	Real-time	Last-write-wins
Invoices	`/v3/companyid/invoices`	Batch hourly	Manual review
Payments	`/v3/companyid/payments`	Real-time	Automatic reconciliation
Items	`/v3/companyid/items`	Daily	Version tracking

OAuth2 Authentication Flow


QBO Integration Request
Check Access Token
Token Valid?
Yes
No
Refresh Token
Refresh Success?
Yes
No
Update Stored Token
Re-authorization Required
Execute API Call
End - Auth Required



Data Mapping and Transformation

- **Customer Mapping**: Yardura clients map to QBO customers with custom field extensions
- **Invoice Synchronization**: Subscription invoices sync with QBO invoice structure
- **Payment Reconciliation**: Stripe payments reconcile with QBO payment records
- **Tax Handling**: Cross-sell items sync with appropriate tax classifications

6.2.3 Google Maps Route Optimization

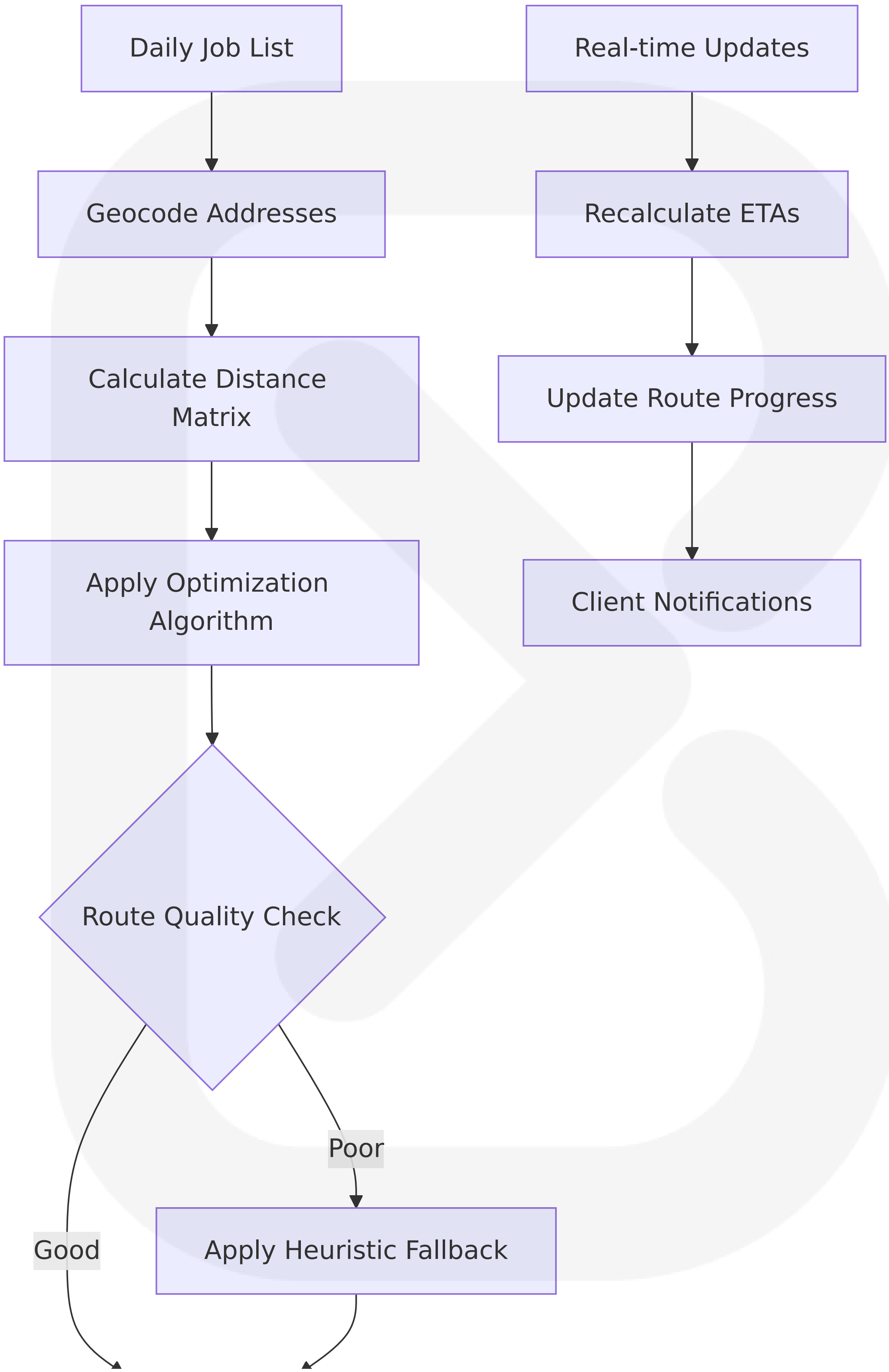
Component Architecture

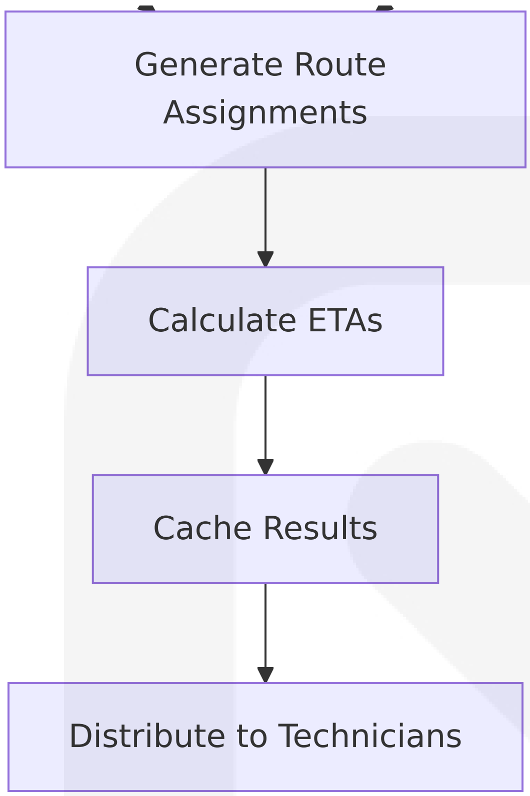
The Google Maps integration provides route optimization capabilities using Distance Matrix API and Directions API for efficient technician routing and ETA calculations.

API Integration Components

Service	API Endpoint	Usage Pattern	Rate Limiting
Distance Matrix	`/maps/api/distancematrix/json`	Batch optimization	100 elements/request
Directions	`/maps/api/directions/json`	Individual routes	50 requests/second
Geocoding	`/maps/api/geocode/json`	Address validation	50 requests/second
Places	`/maps/api/place/details/json`	Address autocomplete	100 requests/second

Route Optimization Algorithm


Daily Job List
Geocode Addresses
Calculate Distance Matrix
Apply Optimization Algorithm
Route Quality Check
Good
Poor
Apply Heuristic Fallback
Real-time Updates
Recalculate ETAs
Update Route Progress
Client Notifications



Performance Optimization Strategies

- **Batch Processing**: Distance matrix calculations processed in batches to minimize API calls
- **Intelligent Caching**: Route results cached with geographic and temporal keys
- **Fallback Mechanisms**: Basic routing algorithms when API limits exceeded
- **Real-time Adaptation**: Dynamic route adjustments based on traffic and completion status

6.2.4 Communication Services Integration

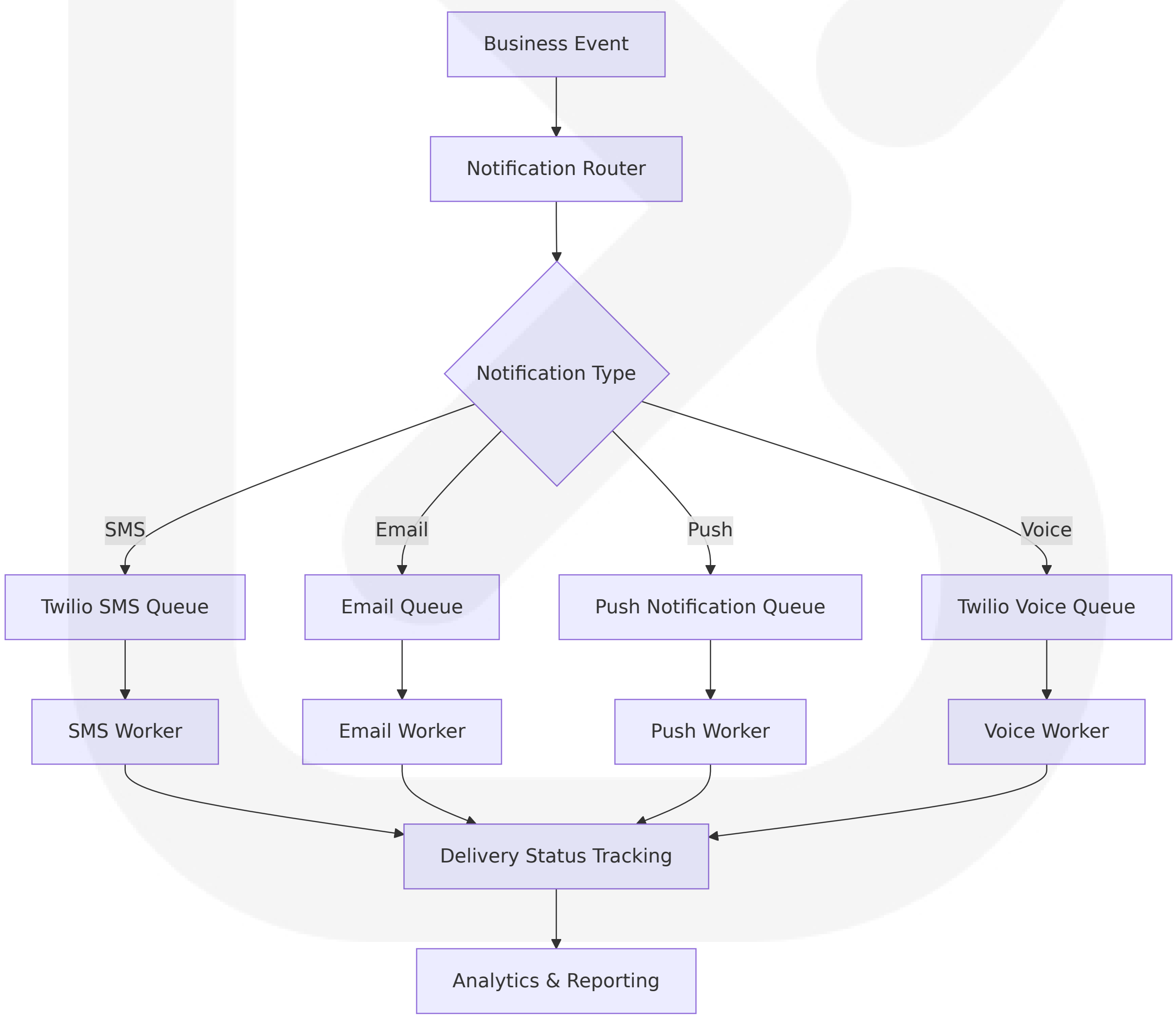
Component Architecture

Multi-channel communication system integrating Twilio for SMS/voice and email services for comprehensive client and staff notifications.

Communication Channels

Channel	Service Provider	Use Cases	Delivery Patterns
SMS	Twilio	On-the-way notifications, alerts	Real-time
Voice	Twilio	Emergency notifications	Immediate
Email	Resend/SendGrid	Invoices, reports, marketing	Batch/scheduled
Push	Web Push API	PWA notifications	Real-time
In-app	WebSocket	Dashboard updates	Real-time

Notification Processing Pipeline



Delivery Optimization

- **Preference Management**: Client notification preferences control delivery channels
- **Rate Limiting**: Intelligent rate limiting prevents spam and reduces costs
- **Retry Logic**: Failed deliveries trigger retry mechanisms with exponential backoff
- **Analytics Integration**: Delivery metrics feed into business intelligence dashboards

6.3 USER INTERFACE COMPONENTS

6.3.1 Progressive Web Application (PWA) Architecture

Component Overview

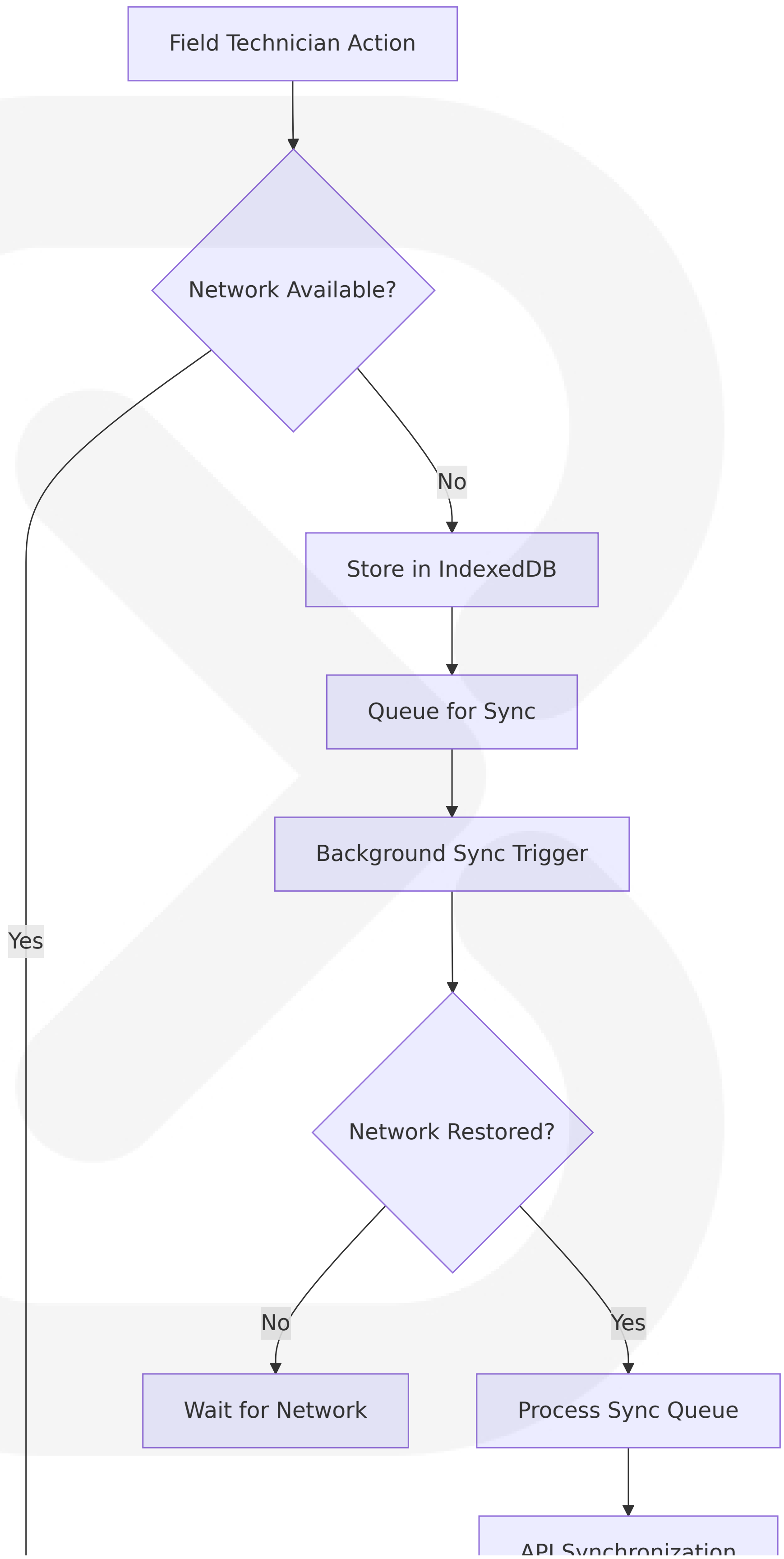
The field technician PWA provides offline-first functionality with background synchronization, built using Next.js 15 with service worker integration for reliable field operations.

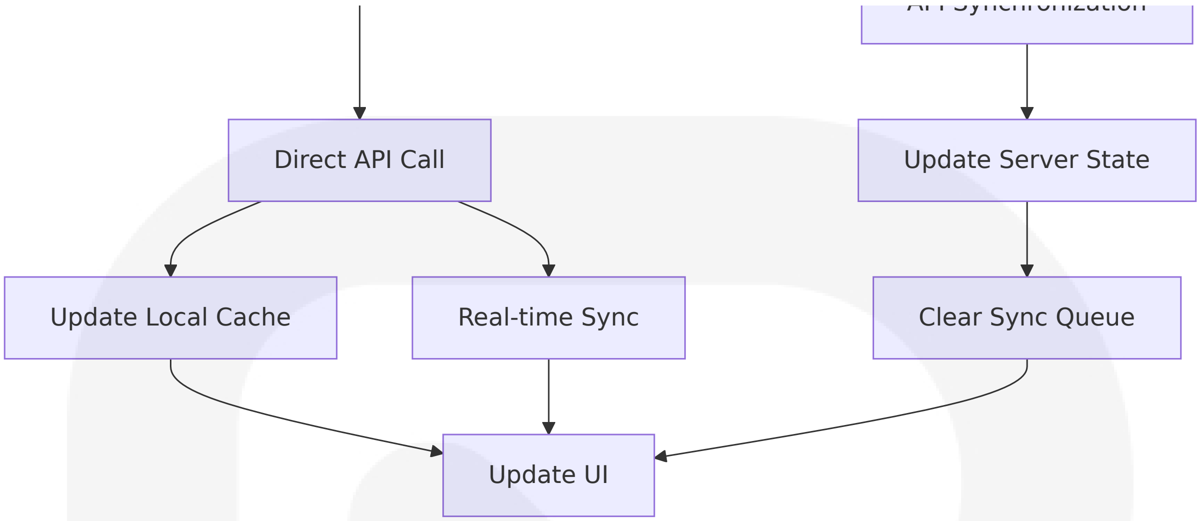
PWA Architecture Components

Component	Technology	Purpose	Offline Capability
Service Worker	Workbox	Offline functionality, caching	Full offline support
Background Sync	Background Sync API	Data synchronization	Queue operations
Push Notifications	Web Push API	Real-time updates	Notification delivery
Local Storage	IndexedDB	Offline data storage	Complete job data

Component	Technology	Purpose	Offline Capability
Camera Integration	MediaDevices API	Photo capture	Local storage

Offline-First Data Flow


Field Technician Action
Network Available?
No
Store in IndexedDB
Queue for Sync
Background Sync Trigger
Yes
Network Restored?
No
Yes
Wait for Network
Process Sync Queue



Performance Optimization

- **Lazy Loading**: Components and routes loaded on-demand to minimize initial bundle size
- **Image Optimization**: Photo compression and progressive loading for mobile networks
- **Caching Strategy**: Intelligent caching of job data, routes, and client information
- **Battery Optimization**: GPS tracking optimized for battery conservation

6.3.2 Client Portal Interface Components

Component Architecture

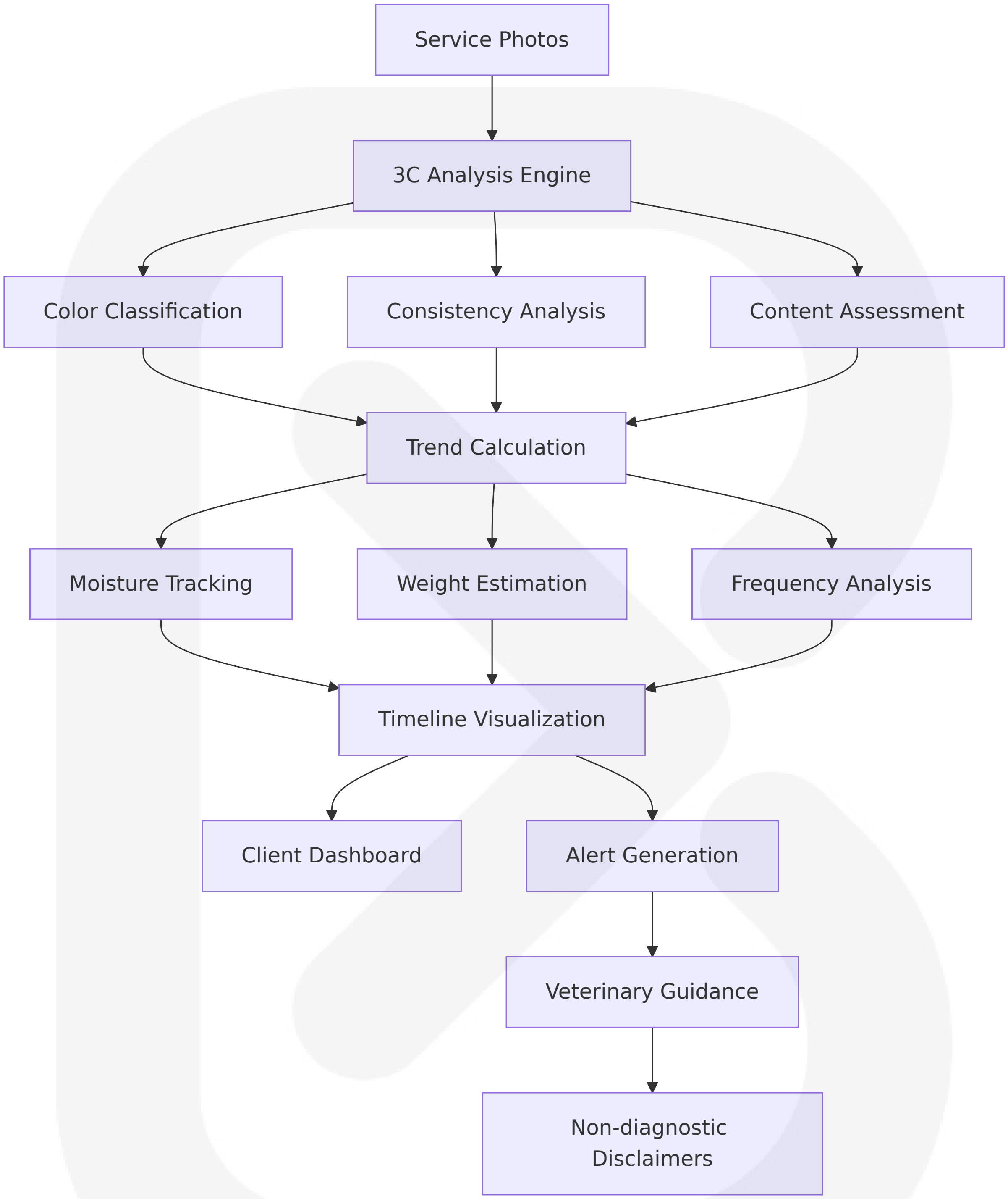
The client portal provides comprehensive self-service capabilities with wellness insights, billing management, and service history access.

Portal Component Structure

Component Category	Key Components	Data Sources	Update Frequency
Dashboard	Service overview, next visit	Jobs, schedules	Real-time

Component Category	Key Components	Data Sources	Update Frequency
Wellness Insights	3C analysis, trends	Photo analysis, readings	Daily batch
Service History	Photo gallery, timeline	Job photos, completion data	Real-time
Billing Management	Invoices, payments	Stripe, local billing	Real-time
Account Settings	Profile, preferences	User data, notifications	On-demand

Wellness Insights Visualization



Responsive Design Patterns

- **Mobile-First**: Optimized for mobile devices with progressive enhancement
- **Accessibility**: WCAG 2.1 AA compliance with screen reader support

- **Performance**: Sub-500ms P95 response times with intelligent caching
- **Progressive Enhancement**: Core functionality available without JavaScript

6.3.3 Administrative Dashboard Components

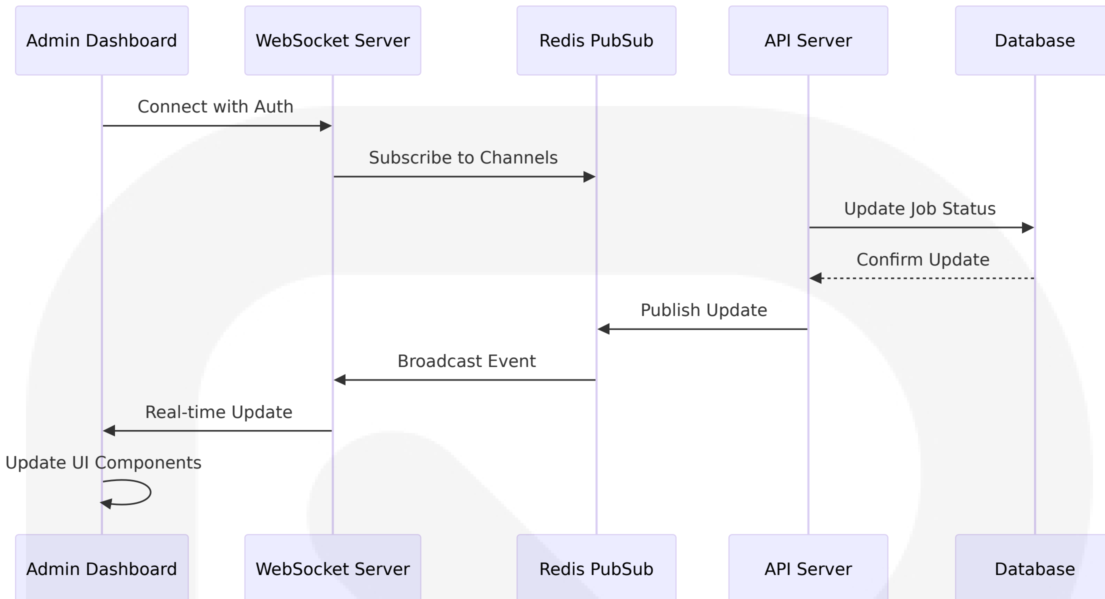
Component Architecture

Comprehensive administrative interface supporting dispatch operations, billing management, payroll processing, and franchise oversight.

Dashboard Component Hierarchy

Dashboard Type	Primary Components	User Roles	Data Refresh
Dispatch Board	Route visualization, job management	Dispatchers	Real-time
Billing Console	Invoice management, payment processing	Accountants	Hourly
Payroll Dashboard	Time tracking, compensation calculation	Managers	Daily
Analytics Portal	KPI tracking, reporting	Owners	Real-time
Franchise Management	Multi-tenant oversight, royalties	Franchise owners	Daily

Real-time Data Synchronization



Component State Management

- **Global State**: Redux Toolkit for complex state management across dashboard components
- **Local State**: React hooks for component-specific state management
- **Server State**: React Query for server state synchronization and caching
- **Real-time Updates**: WebSocket integration for live data updates

6.4 SECURITY COMPONENTS

6.4.1 Authentication and Authorization Framework

Component Architecture

Multi-layered security architecture implementing NextAuth for authentication with comprehensive role-based access control (RBAC) supporting franchise hierarchies.

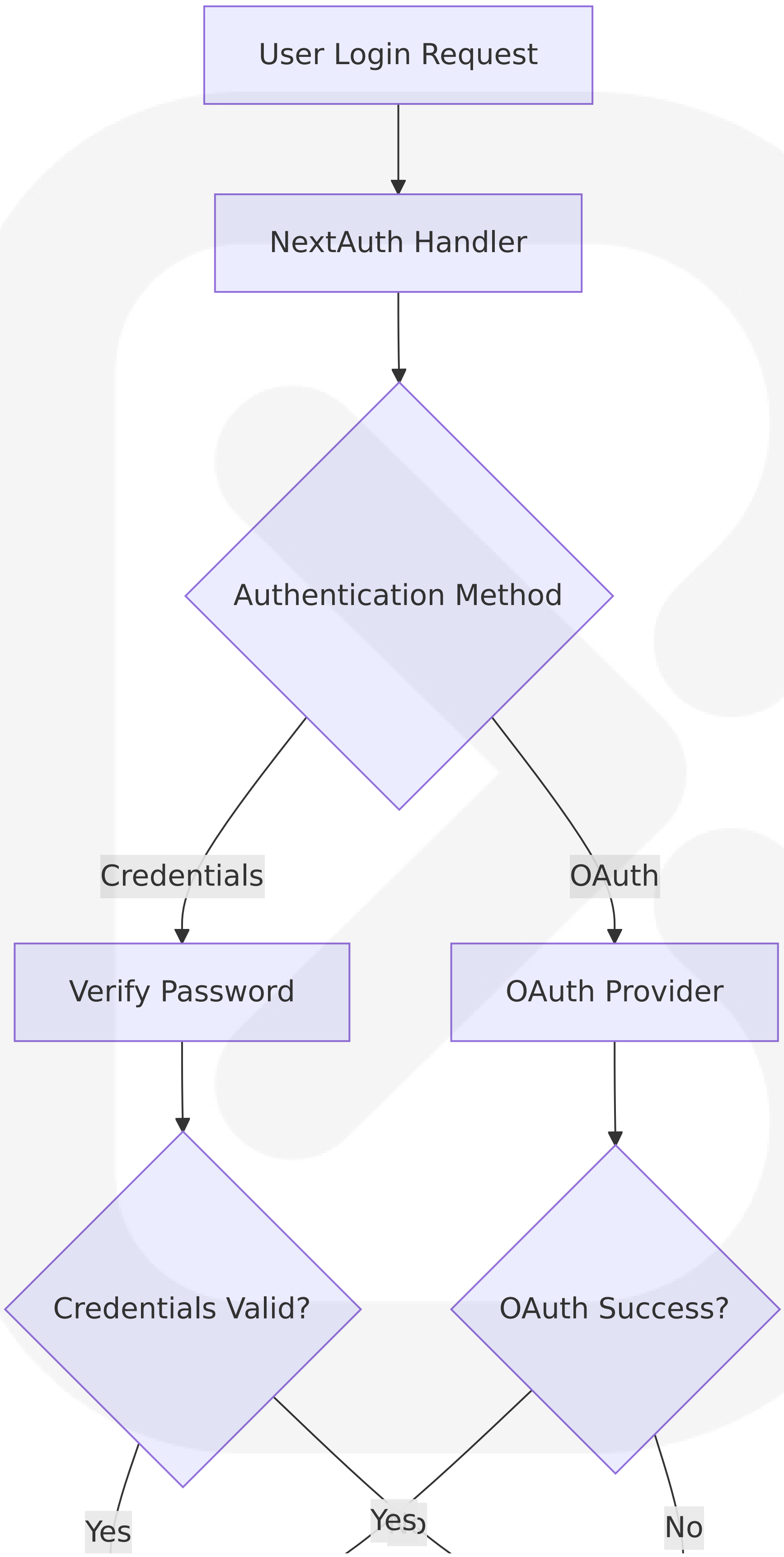
Security Component Stack

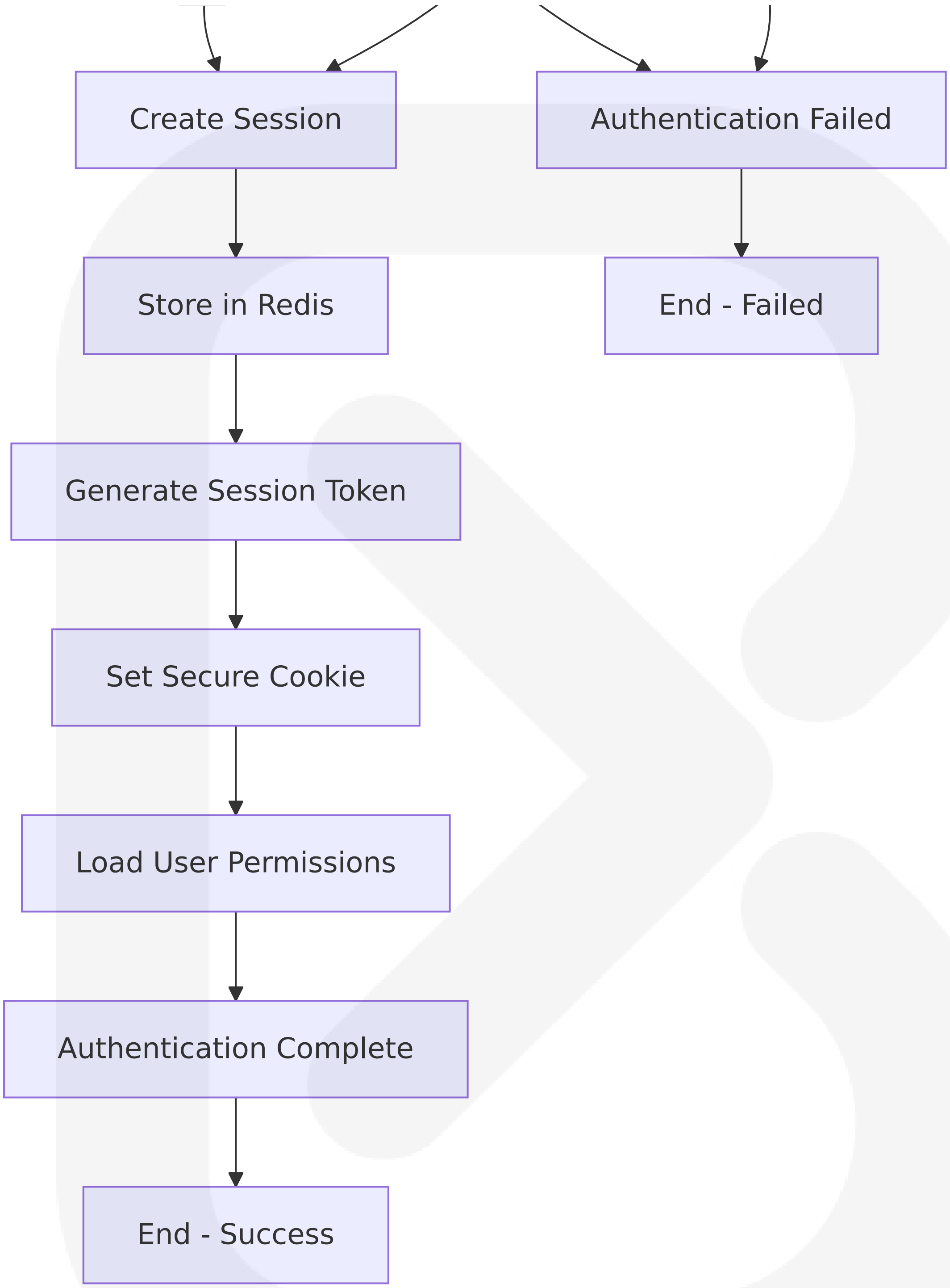
Layer	Technology	Purpose	Implementation
Authentication	NextAuth	Identity verification	OAuth2, credentials
Authorization	Custom RBAC	Permission management	Role hierarchy
Session Management	Redis	Session storage	Distributed sessions
API Security	JWT/Session	API protection	Token validation
Audit Logging	Custom	Compliance tracking	Comprehensive logs

Role-Based Access Control Matrix

Role	Client Portal	Field Operations	Dispatch	Billing	Admin
Client	Full access	None	None	View only	None
Field Tech	None	Full access	View assigned	None	None
Dispatcher	None	View all	Full access	None	None
Accountant	None	None	View only	Full access	None
Manager	View all	Full access	Full access	Full access	Limited
Owner	Full access	Full access	Full access	Full access	Full access

Authentication Flow Architecture


User Login Request
NextAuth Handler
Authentication Method
Credentials
OAuth
Verify Password
OAuth Provider
Credentials Valid?
OAuth Success?
Yes
Yes
No



6.4.2 Data Protection and Encryption

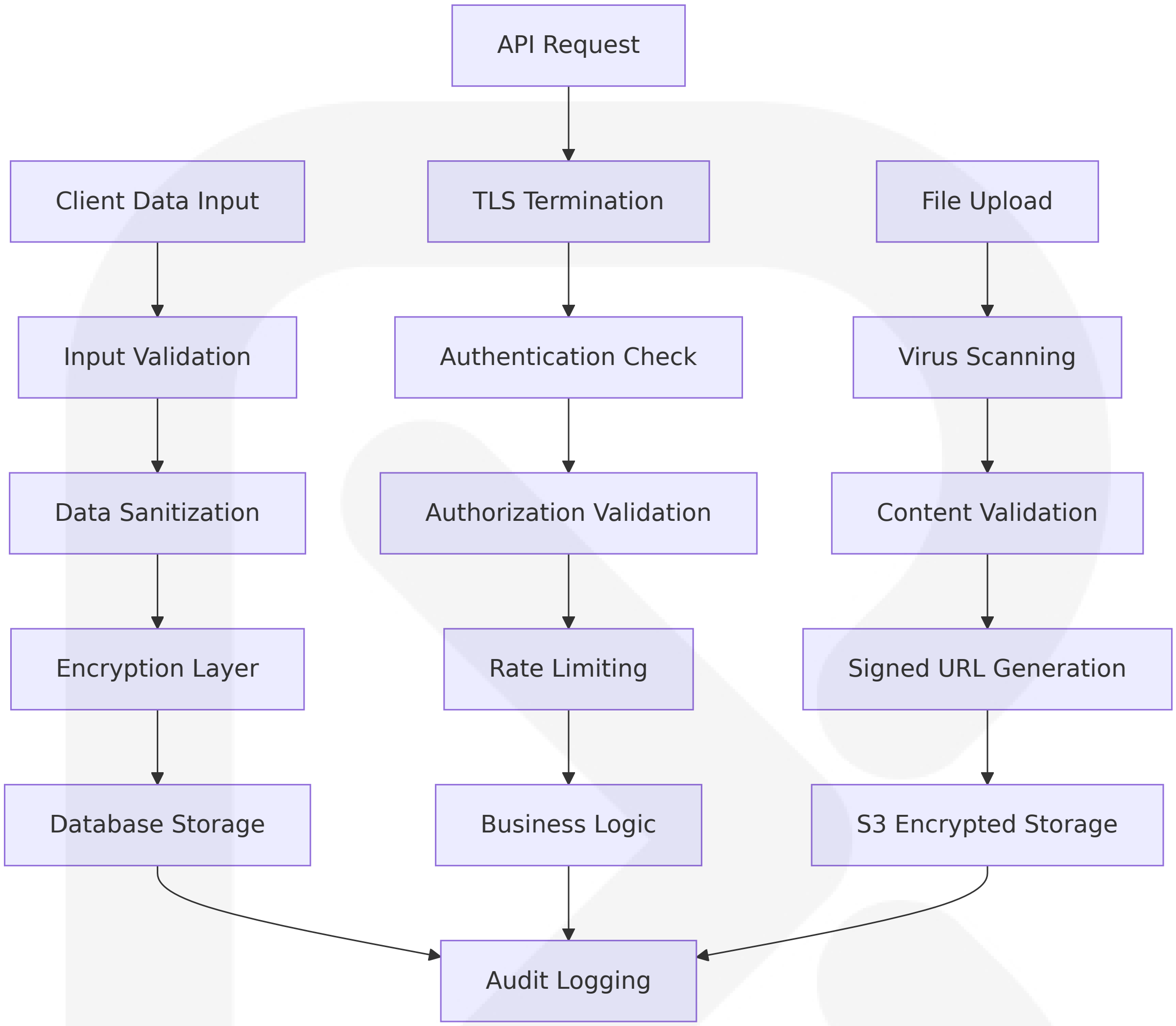
Component Architecture

Comprehensive data protection strategy implementing encryption at rest, in transit, and in processing with PCI DSS compliance for payment data.

Encryption Implementation

Data Type	Encryption Method	Key Management	Compliance
PII Data	AES-256 at rest	AWS KMS/HashiCorp Vault	GDPR, CCPA
Payment Data	Stripe Elements	Stripe managed	PCI DSS Level 1
Session Data	Redis encryption	Application keys	SOC 2
File Storage	S3 server-side	AWS managed	Industry standard
Database	PostgreSQL TDE	Database encryption	ACID compliance

Data Flow Security



6.4.3 API Security and Rate Limiting

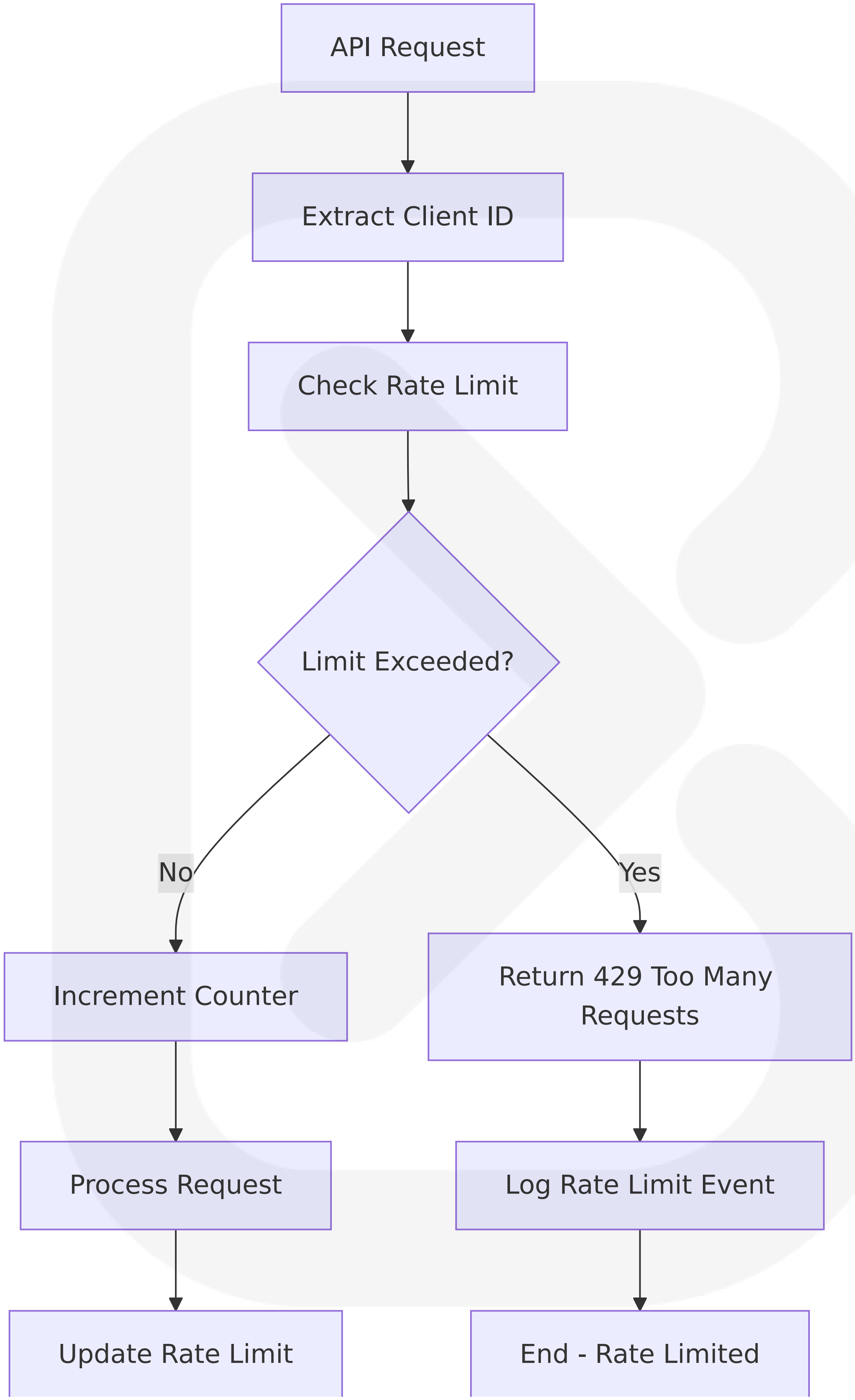
Component Architecture

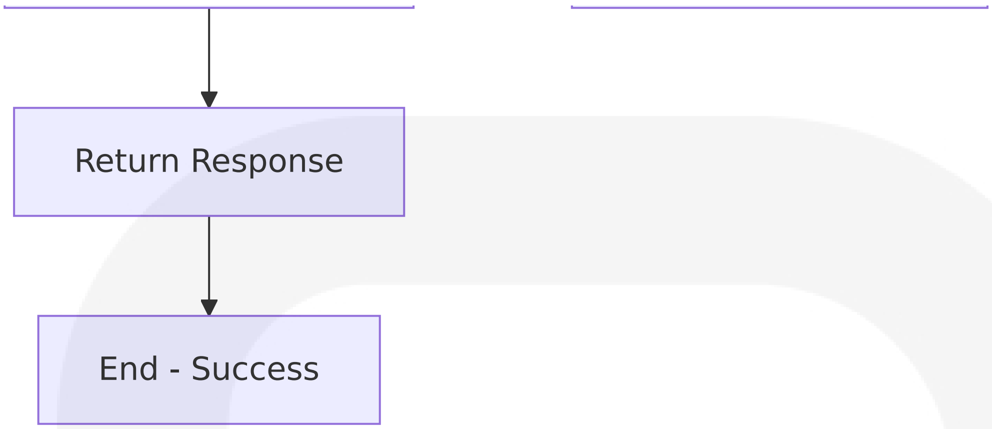
Comprehensive API security framework implementing authentication, authorization, input validation, and intelligent rate limiting across all endpoints.

Security Middleware Stack

Middleware	Purpose	Implementation	Configuration
Rate Limiting	Abuse prevention	Redis-based counters	Per-user, per-endpoint
Authentication	Identity verification	JWT/Session validation	Required for all APIs
Authorization	Permission checking	RBAC enforcement	Role-based rules
Input Validation	Data integrity	Zod schema validation	Strict typing
Audit Logging	Compliance tracking	Structured logging	All operations

Rate Limiting Strategy


API Request
Extract Client ID
Check Rate Limit
Limit Exceeded?
No
Yes
Increment Counter
Return 429 Too Many Requests
Process Request
Log Rate Limit Event
Update Rate Limit
End - Rate Limited



API Security Patterns

- **Defense in Depth**: Multiple security layers prevent single points of failure
- **Zero Trust Architecture**: Every request validated regardless of source
- **Principle of Least Privilege**: Minimal permissions granted for each operation
- **Comprehensive Monitoring**: All security events logged and monitored

This comprehensive system components design provides detailed architecture for all major components of the Yardura Service OS, ensuring scalable, secure, and maintainable implementation across the entire platform.

6.1 CORE SERVICES ARCHITECTURE

6.1.1 Architecture Decision

Core Services Architecture is not applicable for this system as a traditional microservices approach. The Yardura Service OS implements a **modern monolithic architecture with microservice-ready patterns** built on Next.js 15 with clear service boundaries and integration patterns.

Rationale for Monolithic Approach:

The monolithic architecture is a classic approach where the entire application is built as a single, cohesive unit. Simplified development: Developers can focus on building features without worrying about integrating different services or managing multiple repositories. Easier deployment: A single build and deploy process reduce complexity, making it easier to manage and scale your application.

Next.js, on the other hand, enables the development of full-stack applications. It allows the developer to focus on both the frontend and the backend in one framework.

The system's operational requirements, team structure, and business complexity align better with a monolithic approach that provides:

- **Operational Simplicity**: Single deployment unit reduces complexity for initial market entry
- **Development Velocity**: Unified codebase enables rapid feature development and iteration
- **Transactional Consistency**: Complex business operations (billing, payroll, scheduling) benefit from ACID transactions
- **Cost Efficiency**: Lower operational overhead compared to distributed systems management

6.1.2 Service-Oriented Internal Architecture

While maintaining a monolithic deployment model, the system implements **clear service boundaries** using Domain-Driven Design principles with distinct bounded contexts:

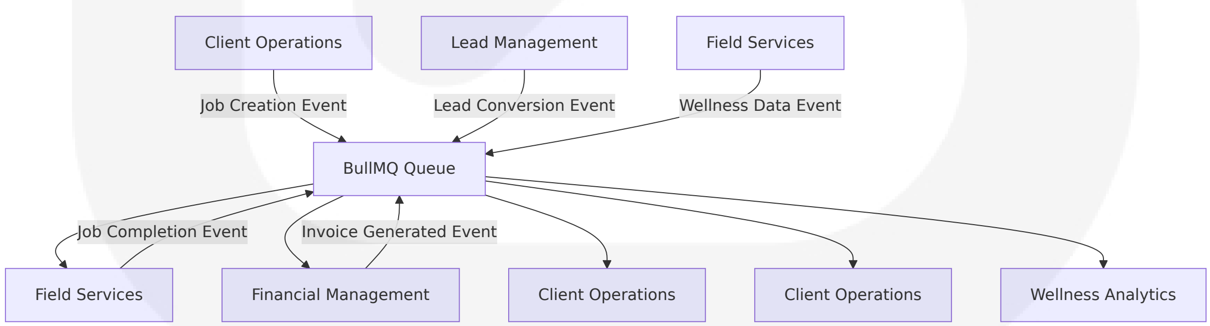
6.1.2.1 Internal Service Boundaries

Service Domain	Responsibilities	Key Components	Data Ownership
Lead Management	Quote processing, CRM integration	Quote wizard, lead creation, analytics	Leads, quotes, pricing
Client Operations	Onboarding, portal, subscriptions	Account creation, Stripe integration	Clients, subscriptions, billing
Field Services	Dispatch, routing, job completion	PWA, GPS tracking, photo management	Jobs, routes, technician data
Financial Management	Billing, payroll, QuickBooks sync	Invoice generation, payment processing	Invoices, payments, accounting

6.1.2.2 Inter-Service Communication Patterns

Internal API Boundaries: Each domain exposes internal APIs through Next.js API routes with clear contracts and validation schemas using Zod for type safety.

Event-Driven Communication: Queues can solve many different problems in an elegant way, from smoothing out processing peaks to creating robust communication channels between micro-services or offloading heavy work from one server to many smaller workers



6.1.3 Background Processing Architecture

6.1.3.1 Queue-Based Service Coordination

BullMQ is a lightweight, robust, and fast NodeJS library for creating background jobs and sending messages using queues. It is backed by Redis, which makes it easy to scale horizontally and process jobs across multiple servers.

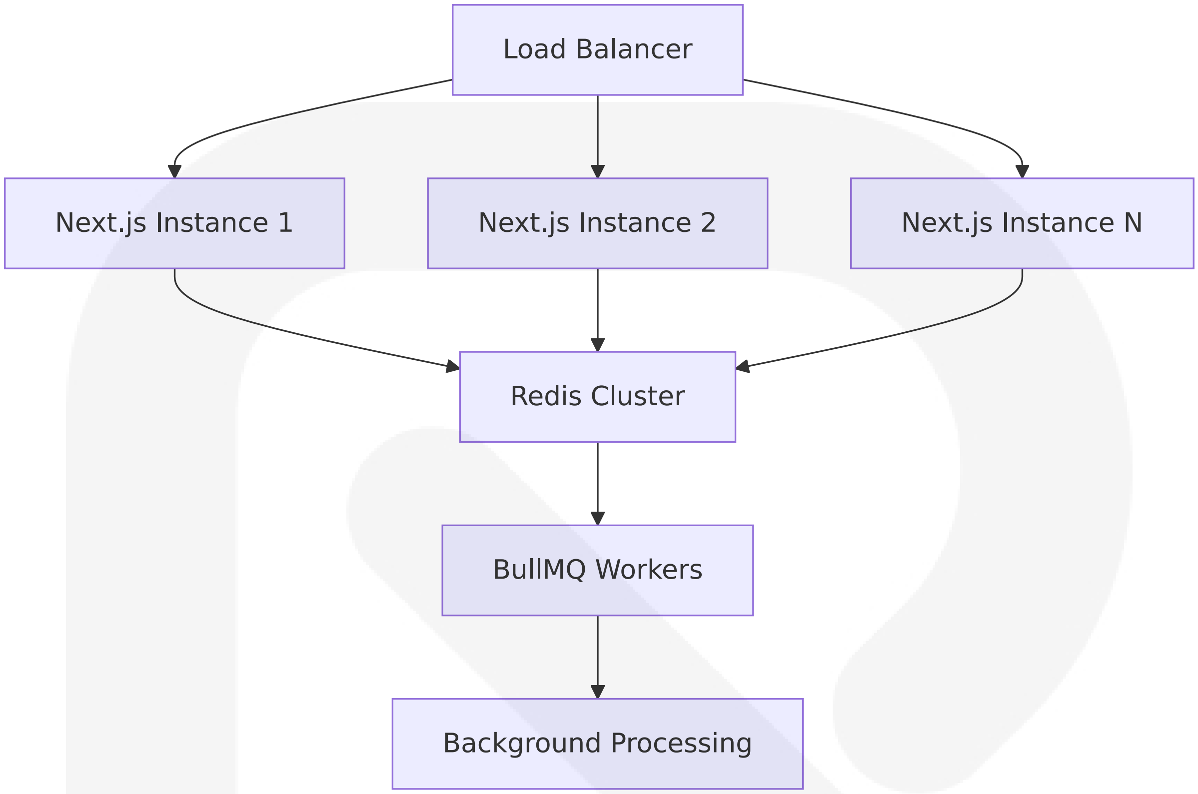
Queue Architecture Design:

Queue Name	Service Domain	Processing Pattern	Scaling Strategy
billing-queue	Financial Management	Sequential with retry	5 workers
notification-queue	Client Operations	Parallel processing	10 workers
wellness-queue	Wellness Analytics	CPU-intensive batch	3 workers
integration-queue	External APIs	API rate limiting	5 workers

6.1.3.2 Service Discovery and Load Balancing

Internal Service Discovery: Services communicate through well-defined internal APIs with automatic service registration through Next.js API routes.

Load Balancing Strategy: If you can afford many connections, by all means just use them. Redis connections have quite low overhead, so you should not need to care about reusing connections unless your service provider imposes hard limitations.



6.1.4 Scalability Design

6.1.4.1 Horizontal Scaling Approach

Application Tier Scaling: Scalability is one of the key features of BullMQ. Since it leverages Dragonfly/Redis, you can easily scale your applications by increasing the number of workers that process jobs. You have complete control over how many workers you want to use based on the workload.

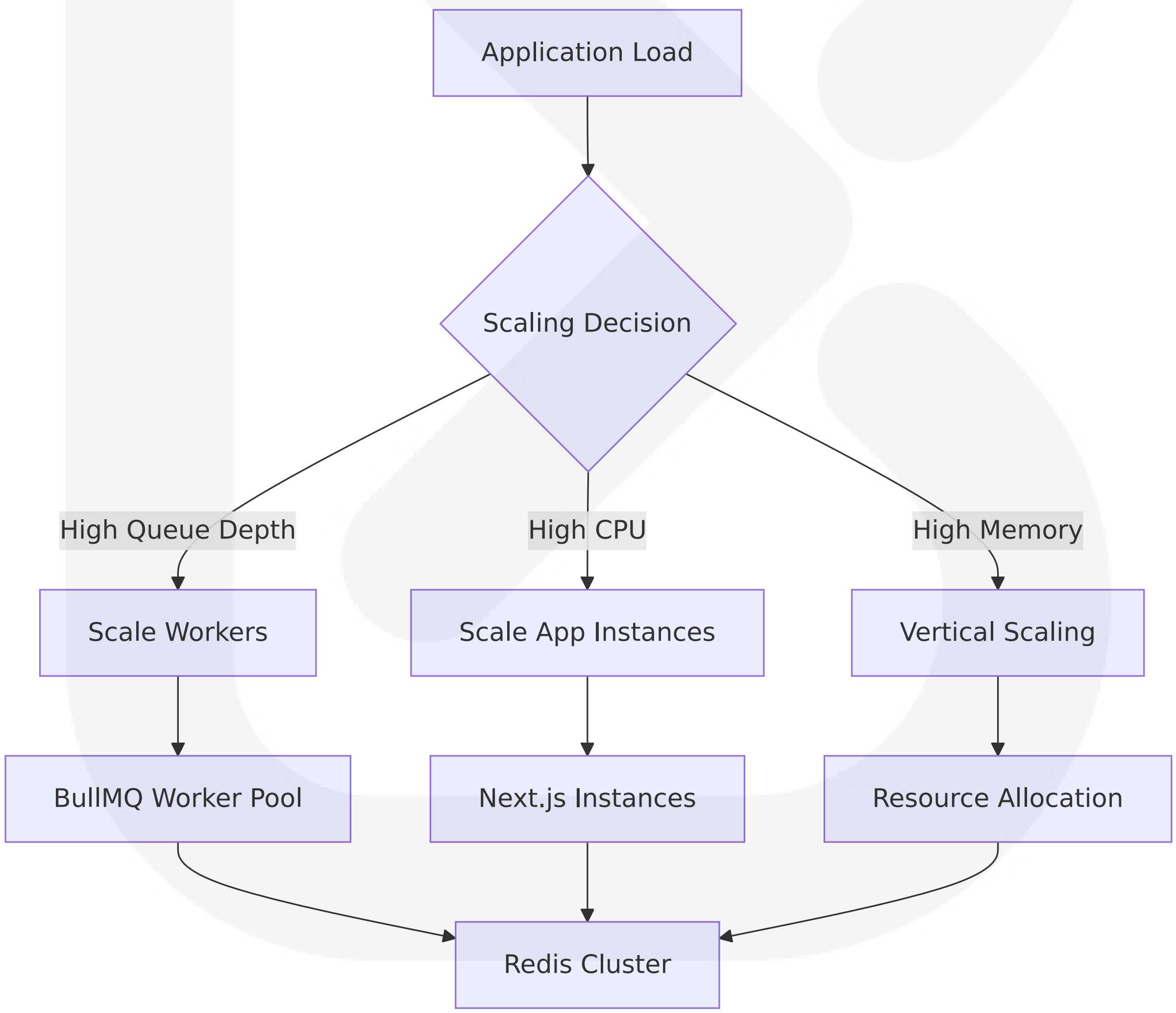
Scaling Triggers and Rules:

Metric	Threshold	Scaling Action	Implementation
CPU Utilization	> 70% for 5 minutes	Add application instance	Horizontal pod autoscaling
Queue Depth	> 1000 jobs	Add worker processes	BullMQ worker scaling

Metric	Threshold	Scaling Action	Implementation
Response Time	P95 > 1000ms	Scale application tier	Load balancer adjustment
Memory Usage	> 80%	Vertical scaling trigger	Resource limit increase

6.1.4.2 Resource Allocation Strategy

Worker Pool Management: Scalability: BullMQ scales well with options like local concurrency and multiple workers. We configured local concurrency to handle large job volumes efficiently, ensuring smooth performance under heavy load.



6.1.4.3 Performance Optimization Techniques

Caching Strategy: Multi-layer caching with Redis for route optimization, session management, and API response caching to meet performance targets of portal P95 < 500ms cached.

Database Optimization: Connection pooling, read replicas for reporting queries, and strategic indexing for complex business queries.

Queue Optimization: High performant. Try to get the highest possible throughput from Redis by combining efficient .lua scripts and pipelining.

6.1.5 Resilience Patterns

6.1.5.1 Fault Tolerance Mechanisms

Circuit Breaker Implementation: External API integrations implement circuit breaker patterns to prevent cascade failures with exponential backoff retry strategies.

Graceful Degradation: System components implement graceful degradation when external services become unavailable:

- Route optimization falls back to basic algorithms when Google Maps API fails
- Billing operations queue for retry when Stripe API is unavailable
- Field technician PWA maintains functionality during network interruptions

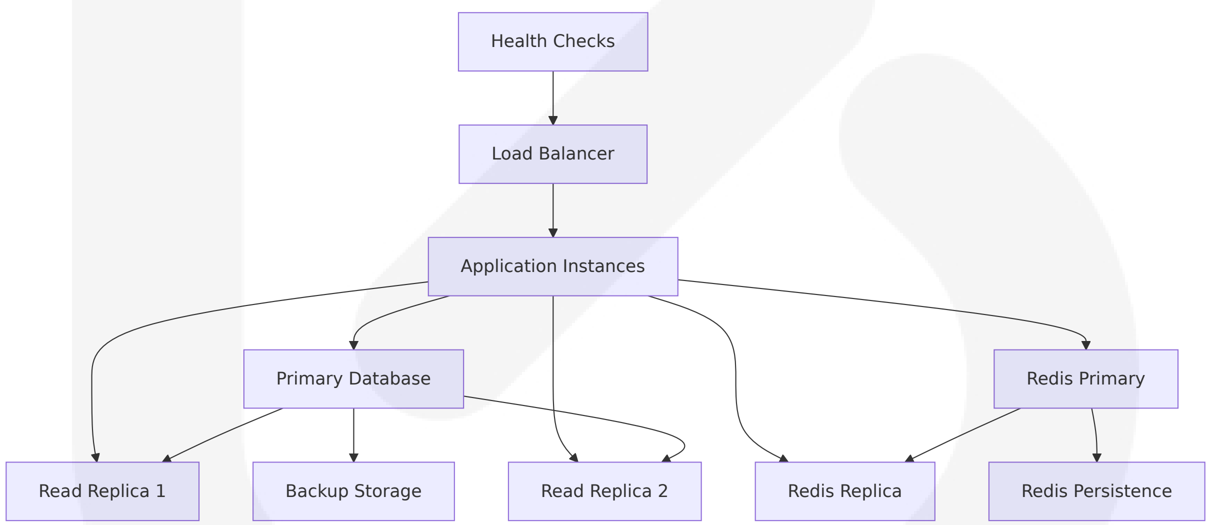
6.1.5.2 Disaster Recovery Procedures

Backup and Recovery Strategy:

Component	Backup Frequency	Recovery Time Objective	Recovery Point Objective
PostgreSQL Database	Daily automated	4 hours	15 minutes
Redis Cache	Continuous replication	1 hour	5 minutes
File Storage (S3/R2)	Cross-region replication	2 hours	Real-time
Application State	Stateless architecture	Immediate	N/A

6.1.5.3 Data Redundancy and Failover

Database Resilience: Redis ensures that job states are persisted and accessible across multiple instances of your application, enabling horizontal scaling. Persistence Options: Configure Redis to persist data to disk, ensuring no job data is lost during restarts.



6.1.5.4 Service Degradation Policies

Progressive Degradation Strategy:

1. **Level 1 - Full Functionality**: All services operational with real-time features

2. **Level 2 - Core Operations**: Essential business functions with delayed processing
3. **Level 3 - Read-Only Mode**: Data access only, no modifications
4. **Level 4 - Emergency Mode**: Critical alerts and basic status information

Failure Recovery Patterns: Implement robust error handling to manage failed jobs. Use BullMQ's retry mechanisms and dead-letter queues to ensure no job is lost.

6.1.6 Future Microservices Migration Path

6.1.6.1 Service Extraction Strategy

The current monolithic architecture with clear service boundaries enables future migration to microservices when business requirements justify the operational complexity:

Phase 1 Candidates (120-180 days):

- Wellness Analytics Service (CPU-intensive, independent processing)
- Notification Service (high-volume, stateless operations)

Phase 2 Candidates (180+ days):

- External Integration Service (API rate limiting, independent scaling)
- Reporting and Analytics Service (read-heavy, different scaling patterns)

6.1.6.2 Migration Readiness Indicators

Technical Indicators:

- Service boundaries well-defined with minimal cross-cutting concerns
- Event-driven communication patterns established
- Independent data models with clear ownership
- Comprehensive API contracts and testing

Business Indicators:

- Team size exceeds 8-10 developers
- Different services require independent deployment cycles
- Scaling requirements vary significantly between domains
- Regulatory or compliance requirements demand service isolation

This architecture approach provides the operational simplicity needed for initial market success while maintaining the flexibility to evolve toward distributed systems as business requirements and team capabilities mature.

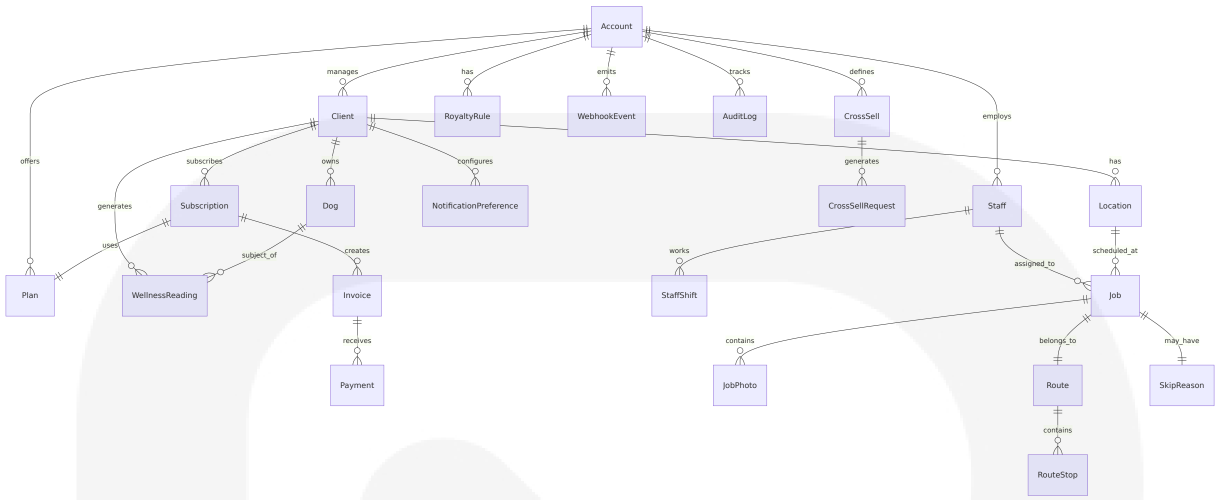
6.2 DATABASE DESIGN

6.2.1 Schema Design

Entity Relationships

The Yardura Service OS database design implements a **shared database, shared schema multi-tenant architecture** with tenant isolation through row-level security policies. This approach uses a tenant or customer id on tables, with the customer id denormalized on every table to make future scaling and safety of data easier, despite database purists considering this a bad idea.

Prisma ORM is a powerful and type-safe database toolkit for Node.js and TypeScript that simplifies database access and management, making it a great choice for Next.js applications. The system utilizes PostgreSQL as the datasource with Prisma Client generation, creating models with one-to-many relationships.



Data Models and Structures

Core Business Entities

Entity	Primary Purpose	Key Relationships	Tenant Isolation
Account	Multi-tenant root entity	Parent to all business data	Root tenant identifier
Client	Customer management	Dogs, Locations, Subscriptions	account_id foreign key
Subscription	Billing and service plans	Invoices, Payments	Via client relationship
Job	Service delivery tracking	Routes, Photos, Staff	Via client relationship

Multi-Tenant Architecture Implementation

The shared table data approach uses a single database and schema for all tenants, with tenant data stored in shared tables with a tenant identifier column such as tenant_id to ensure data isolation. Each table includes an `account_id` column that serves as the tenant identifier with appropriate foreign key constraints.

Wellness Analytics Schema

Table	Structure	Purpose	Data Types
WellnessReading	id, dog_id, job_photo_id, color_score, consistency_score, content_score	3Cs analysis storage	JSONB for flexible scoring
WellnessTrend	id, dog_id, metric_type, value, recorded_at	Trend calculations	Time-series data
WellnessAlert	id, client_id, alert_type, message, created_at	Client notifications	Structured alert data

Indexing Strategy

Performance-Critical Indexes

Table	Index Type	Columns	Purpose
jobs	Composite	(account_id, scheduled_date, status)	Daily dispatch queries
job_photos	Composite	(job_id, created_at)	Photo timeline access
wellness_readings	Composite	(dog_id, recorded_at)	Trend analysis
invoices	Composite	(client_id, status, due_date)	Billing operations

Multi-Tenant Query Optimization

Once you have your customer id on every table you want to ensure you're joining on that key, with libraries helping with enforcement of joining on your customer id. All queries include `account_id` filters with dedicated indexes to ensure optimal performance across tenant boundaries.

Specialized Indexes for Business Operations

- **Route Optimization**: Geospatial indexes on location coordinates for efficient route calculation

- **Wellness Analytics**: Time-series indexes on wellness readings for trend analysis
- **Audit Compliance**: Composite indexes on audit logs for regulatory reporting

Partitioning Approach

Time-Based Partitioning Strategy

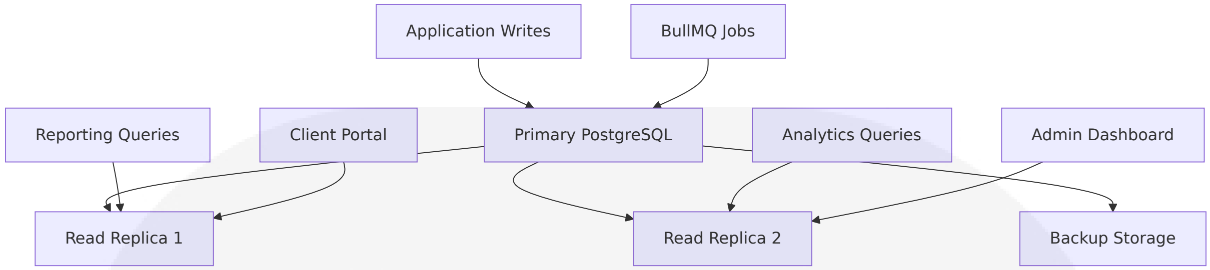
Table	Partition Key	Retention Policy	Archive Strategy
audit_logs	created_at (monthly)	7 years active	Cold storage migration
job_photos	created_at (quarterly)	3 years active	S3 Glacier transition
wellness_readings	recorded_at (monthly)	5 years active	Compressed storage
webhook_events	created_at (weekly)	90 days active	Automated cleanup

Tenant-Based Considerations

While implementing shared schema architecture, the system maintains partition alignment with tenant boundaries where possible to optimize query performance and enable future scaling to tenant-specific partitions if required.

Replication Configuration

Primary-Replica Architecture



Replication Strategy

- **Synchronous Replication**: Critical billing and payment operations
- **Asynchronous Replication**: Reporting and analytics workloads
- **Cross-Region Backup**: Disaster recovery and compliance requirements

Backup Architecture

Multi-Tier Backup Strategy

Backup Type	Frequency	Retention	Recovery Objective
Point-in-Time Recovery	Continuous	30 days	RPO: 15 minutes
Full Database Backup	Daily	90 days	RTO: 4 hours
Archive Backup	Weekly	7 years	Compliance restore
Cross-Region Backup	Daily	30 days	Disaster recovery

6.2.2 Data Management

Migration Procedures

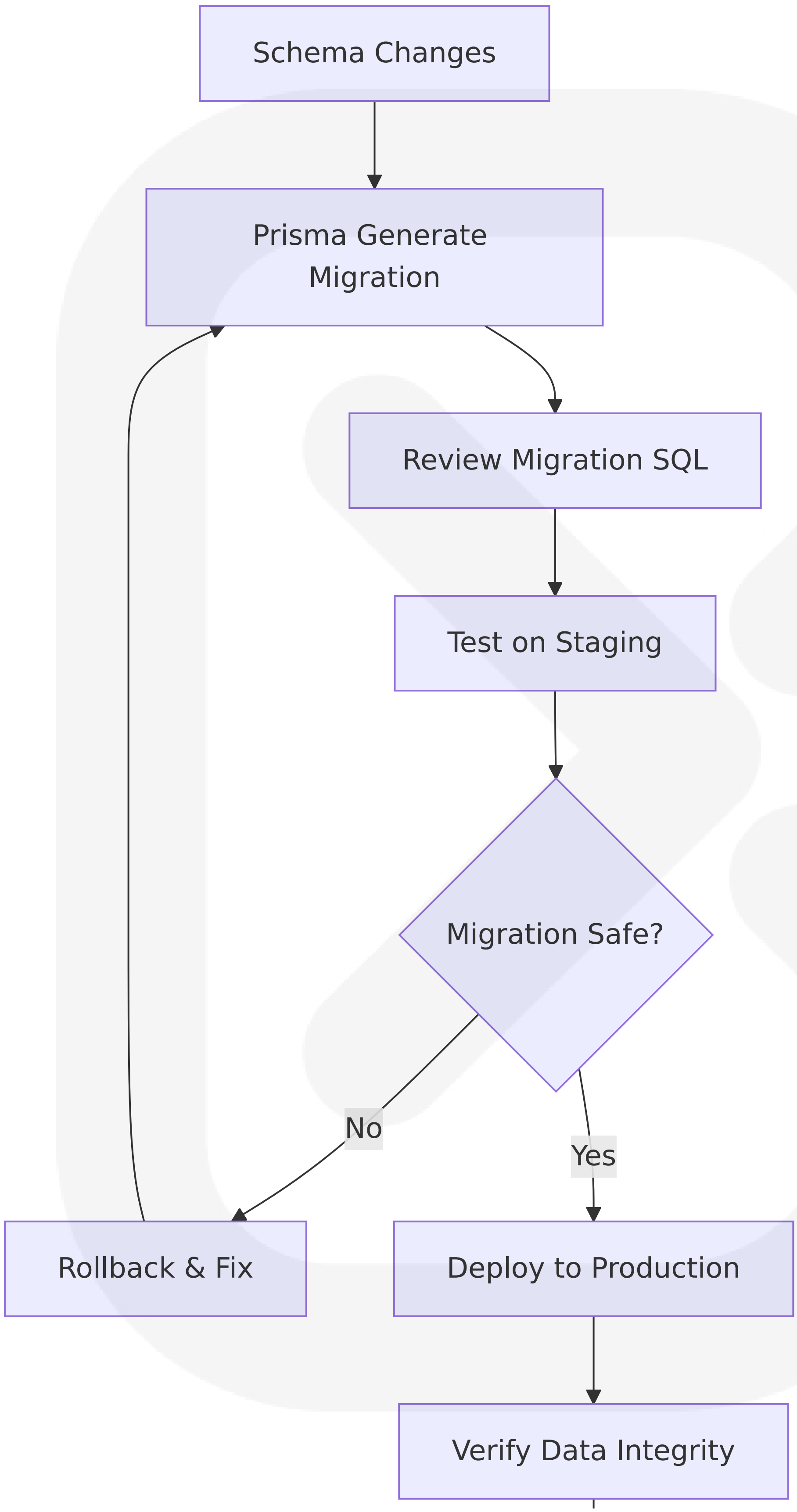
Prisma Migration Strategy

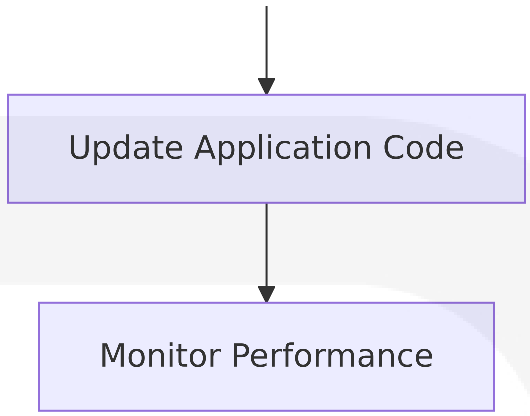
Prisma Migrate auto-generates SQL migrations from your Prisma schema, with migration files being fully customizable, giving full control and ultimate flexibility from local development to production environments.

Multi-Tenant Migration Challenges

Schema migrations are inherently challenging, with complexity including change history tracking across tenant databases, coordinated deployment ensuring consistent changes, rollback management, tenant-specific customizations, and testing validation across representative databases.

Migration Workflow


Schema Changes
Prisma Generate Migration
Review Migration SQL
Test on Staging
Migration Safe?
No
Yes
Rollback & Fix
Deploy to Production
Verify Data Integrity



Versioning Strategy

Database Schema Versioning

Version Component	Strategy	Implementation	Rollback Support
Schema Structure	Semantic versioning	Prisma migrations	Automated rollback
Data Transformations	Sequential numbering	Custom scripts	Manual verification
Index Changes	Performance-based	Background creation	Online rebuilding
Constraint Updates	Compatibility-first	Phased deployment	Constraint dropping

Archival Policies

Data Lifecycle Management

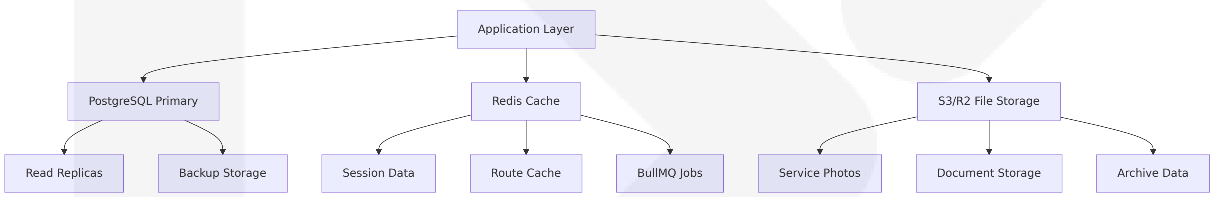
Backup and restoration are more complex in multi-tenant environments, so not all providers offer reliable restoration services. The system implements comprehensive archival policies to manage data growth and compliance requirements.

Archival Timeline

Data Category	Active Period	Archive Period	Deletion Policy
Service Photos	3 years	7 years (compressed)	Client consent required
Wellness Data	5 years	Permanent (anonymized)	Research opt-out honored
Financial Records	7 years	Permanent	Regulatory compliance
Audit Logs	7 years	Permanent	Legal requirements

Data Storage and Retrieval Mechanisms

Hybrid Storage Architecture



Storage Optimization

- **Hot Data**: PostgreSQL with SSD storage for active operations
- **Warm Data**: Read replicas for reporting and analytics
- **Cold Data**: S3 Glacier for archived photos and compliance records

Caching Policies

Multi-Layer Caching Strategy

BullMQ uses Redis data types like lists, sets, sorted sets, and hashes to store and manage jobs, with each job stored as a JSON object inside a Redis hash, indexed by job ID.

Cache Configuration

Cache Layer	TTL Strategy	Eviction Policy	Use Cases
Application Cache	5-15 minutes	LRU	Database query results
Session Cache	24 hours	TTL-based	User authentication
Route Cache	1 hour	TTL-based	Optimization results
API Response Cache	15 minutes	TTL-based	External API calls

6.2.3 Compliance Considerations

Data Retention Rules

Regulatory Compliance Framework

Regulation	Data Types	Retention Period	Deletion Requirements
GDPR	Personal data	User-controlled	Right to erasure
CCPA	Consumer information	24 months minimum	Opt-out compliance
SOX	Financial records	7 years	Audit trail preservation
State Regulations	Service records	Varies by jurisdiction	Local compliance

Automated Compliance Enforcement

The system implements automated data retention policies through scheduled BullMQ jobs that identify and process data according to regulatory requirements while maintaining audit trails of all compliance actions.

Backup and Fault Tolerance Policies

High Availability Architecture

PostgreSQL can handle large datasets and concurrent users important for growing multi-tenant applications, with ACID compliance ensuring data integrity and consistency across transactions.

Fault Tolerance Measures

Component	Redundancy	Failover Time	Data Loss Tolerance
Primary Database	Synchronous replica	< 30 seconds	Zero data loss
Cache Layer	Redis Cluster	< 5 seconds	Acceptable loss
File Storage	Cross-region replication	< 60 seconds	Zero data loss
Application Tier	Load balancer	< 10 seconds	Stateless recovery

Privacy Controls

Data Protection Implementation

The system utilizes PostgreSQL's Row-Level Security feature to ensure data isolation between tenants, with trigger functions reducing the cognitive burden of developers in managing different tenants.

Privacy Control Mechanisms

- **Encryption at Rest**: AES-256 encryption for all PII data
- **Access Controls**: Role-based permissions with least privilege
- **Data Masking**: Automated PII masking in non-production environments
- **Consent Management**: Granular consent tracking for data usage

Audit Mechanisms

Comprehensive Audit Framework

Audit Category	Scope	Retention	Access Controls
Data Access	All PII queries	7 years	Security team only
Financial Operations	Billing, payments, refunds	7 years	Finance and audit
System Changes	Schema, configuration	7 years	Engineering leads
User Actions	Portal, admin operations	3 years	Account managers

Access Controls

Multi-Tenant Security Model

Adding tenant_id fields to resources enables where clause restrictions, but creates issues including code clutter, maintenance difficulty, forgotten clauses by newcomers, and lack of true data isolation between tenants.

Row-Level Security Implementation

```
-- Example RLS policy for client data isolation
CREATE POLICY client_isolation ON clients
    FOR ALL TO application_role
    USING (account_id = current_setting('app.current_account_id')::intege

-- Automatic tenant context setting
CREATE OR REPLACE FUNCTION set_tenant_context(tenant_id integer)
RETURNS void AS $$
BEGIN
    PERFORM set_config('app.current_account_id', tenant_id::text, true);
END;
$$ LANGUAGE plpgsql;
```

6.2.4 Performance Optimization

Query Optimization Patterns

Multi-Tenant Query Strategies

Within Citus when you shard your data on a tenant or customer id all the data gets co-located on the same instance, meaning when you join you're not doing cross shard joins, with the join pushed down to the node where all the data is located.

Optimization Techniques

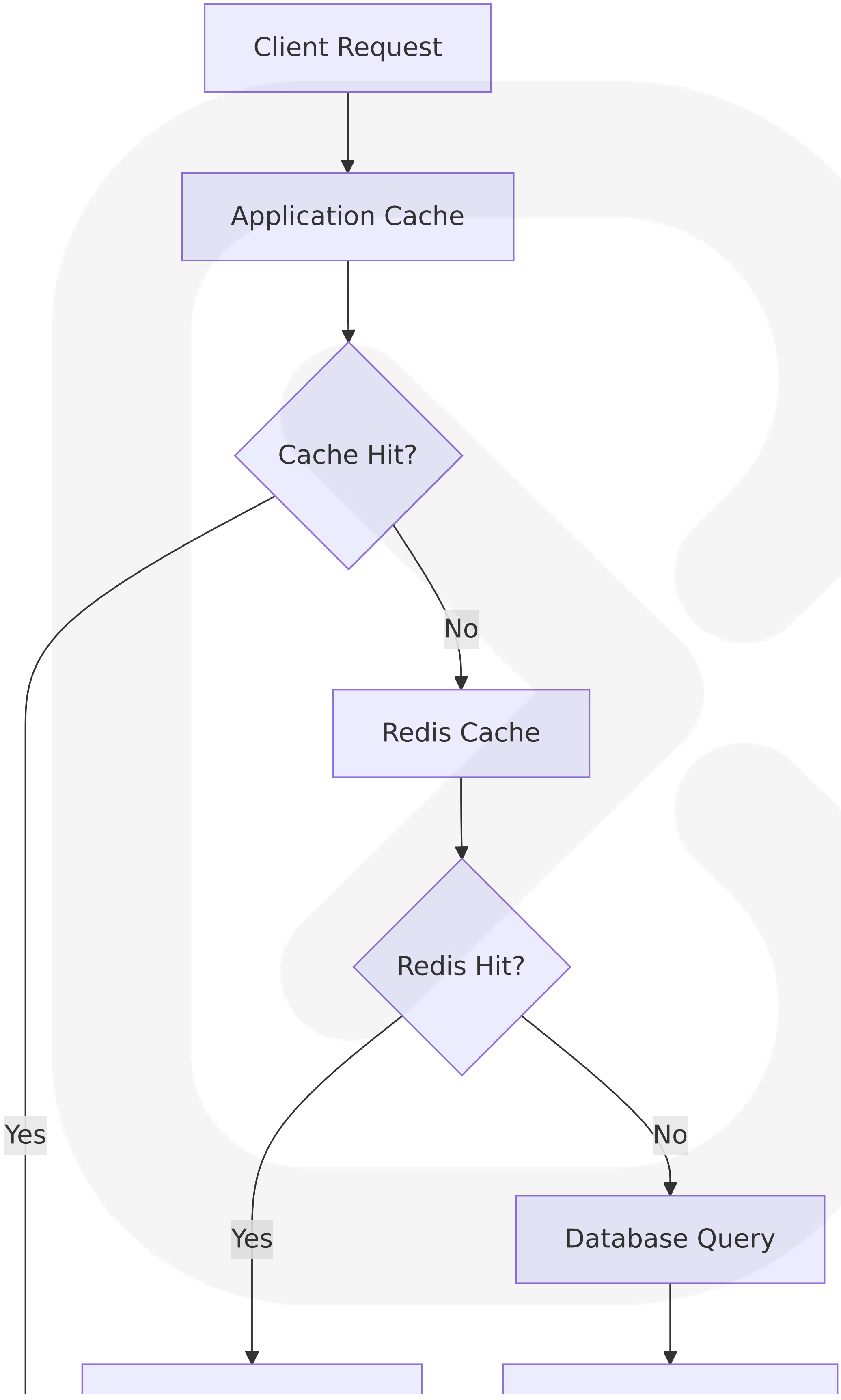
Pattern	Implementation	Performance Gain	Use Cases
Tenant-Aware Indexing	Composite indexes with account_id	80% query improvement	All multi-tenant queries
Query Plan Caching	Prepared statements	60% execution improvement	Repeated operations
Connection Pooling	PgBouncer integration	90% connection efficiency	High concurrency
Read Replica Routing	Query classification	70% primary load reduction	Reporting queries

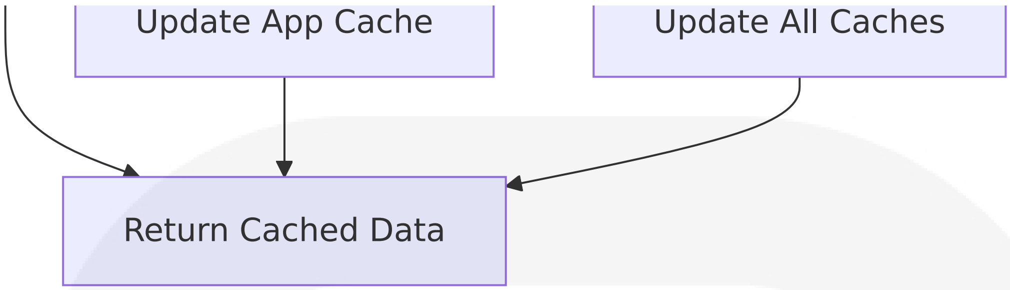
Caching Strategy

Redis-Based Performance Enhancement

Redis operates entirely in memory making it extremely fast, with BullMQ Redis integration providing the speed, consistency, and scalability needed for modern backend tasks.

Cache Hierarchy


Client Request
Application Cache
Cache Hit?
No
Redis Cache
Redis Hit?
Yes
No
Yes
Database Query



Connection Pooling

Database Connection Management

To manage database connections effectively, create a dedicated module that ensures Prisma Client is used as a singleton across your application, crucial for maintaining efficient and reliable database connections.

Connection Pool Configuration

Pool Parameter	Development	Production	Reasoning
Max Connections	10	100	Resource optimization
Min Connections	2	20	Connection warmup
Idle Timeout	30 seconds	300 seconds	Resource cleanup
Connection Lifetime	1 hour	4 hours	Connection refresh

Read/Write Splitting

Query Routing Strategy

Operation Type	Target	Consistency	Performance Benefit
Transactional Writes	Primary	Strong	ACID compliance

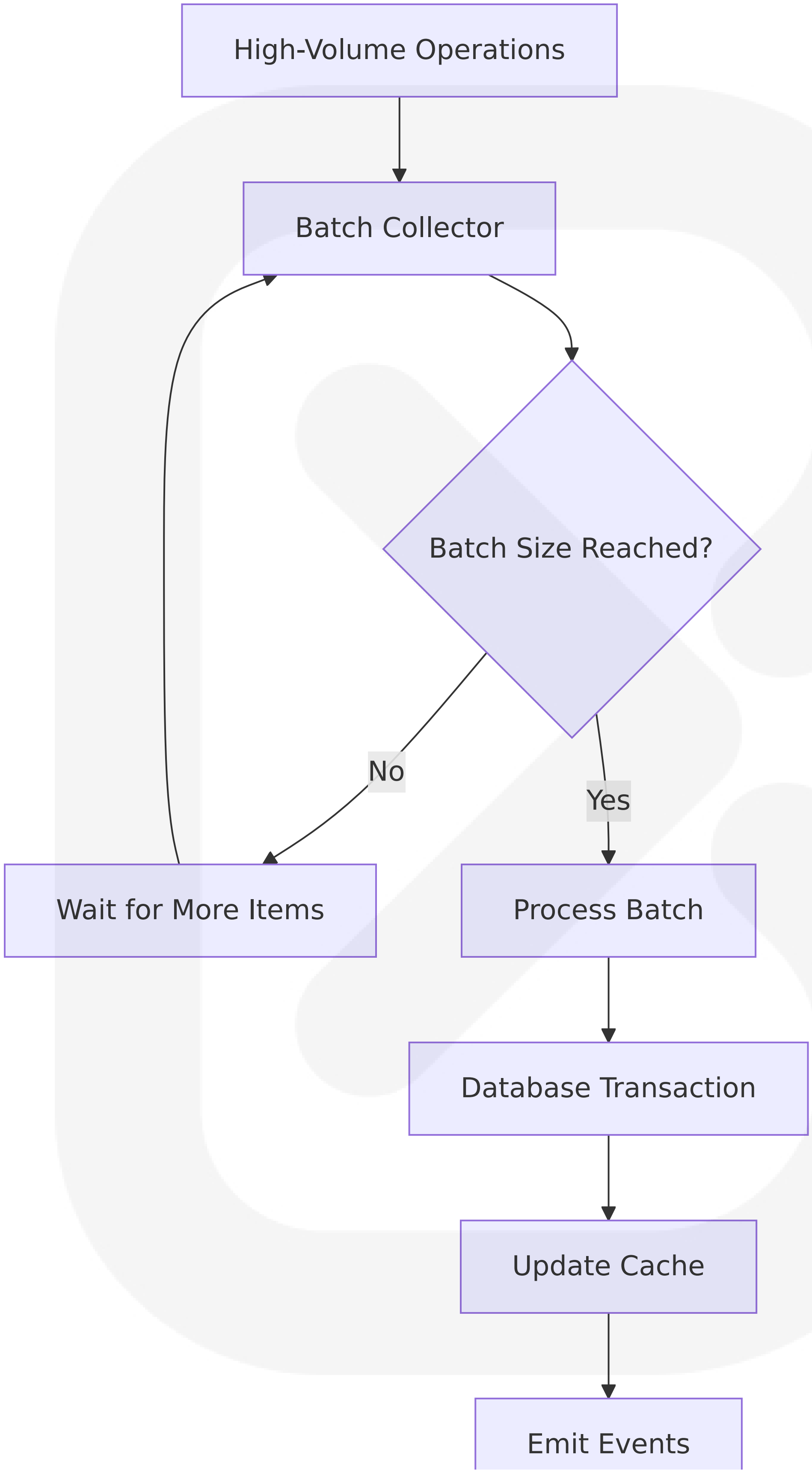
Operation Type	Target	Consistency	Performance Benefit
Real-time Reads	Primary	Strong	Immediate consistency
Reporting Queries	Read Replica	Eventual	70% load reduction
Analytics	Dedicated Replica	Eventual	Isolated workload

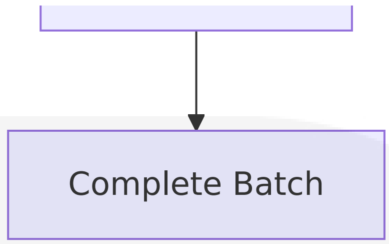
Batch Processing Approach

BullMQ Job Processing Optimization

BullMQ is really fast, with modern hardware easily achieving 50k jobs/sec, with bottlenecks likely on network IO and job processing rather than BullMQ itself, and batching can increase these numbers by an order of magnitude.

Batch Processing Patterns


High-Volume Operations
Batch Collector
Batch Size Reached?
No
Yes
Wait for More Items
Process Batch
Database Transaction
Update Cache
Emit Events



Batch Configuration

Operation	Batch Size	Frequency	Performance Gain
Invoice Generation	100 records	Every 5 minutes	10x throughput
Wellness Analysis	50 photos	Every 2 minutes	5x processing speed
Notification Delivery	200 messages	Every 1 minute	15x delivery rate
Audit Log Writing	500 entries	Every 30 seconds	20x write efficiency

This comprehensive database design provides a robust foundation for the Yardura Service OS, implementing modern multi-tenant patterns with PostgreSQL and Prisma ORM while ensuring scalability, security, and compliance requirements are met through careful architectural decisions and performance optimizations.

6.3 INTEGRATION ARCHITECTURE

6.3.1 API DESIGN

6.3.1.1 Protocol Specifications

The Yardura Service OS implements a comprehensive RESTful API architecture built on Next.js 15 App Router with TypeScript, providing standardized endpoints for all business operations and external integrations.

Core API Standards

Protocol	Version	Content Type	Response Format
HTTP/HTTPS	1.1/2.0	application/json	JSON with consistent schema
WebSocket	RFC 6455	text/binary	Real-time event streaming
Webhook	HTTP POST	application/json	Event-driven notifications

API Endpoint Structure

```
https://api.yardura.com/v1/{resource}
├── /leads              # Quote and lead management
├── /clients            # Client onboarding and management
├── /jobs               # Service scheduling and completion
├── /billing            # Invoice and payment processing
├── /wellness           # Health insights and analytics
├── /integrations       # External service connections
└── /webhooks           # Event notification endpoints
```

Request/Response Standards

All API endpoints follow consistent patterns with standardized error handling, pagination, and metadata inclusion:

```
{
  "data": {},
  "meta": {
    "timestamp": "2025-09-10T12:00:00Z",
    "version": "v1",
    "request_id": "req_123456789"
  },
  "pagination": {
    "page": 1,
    "limit": 50,
    "total": 1250,
```

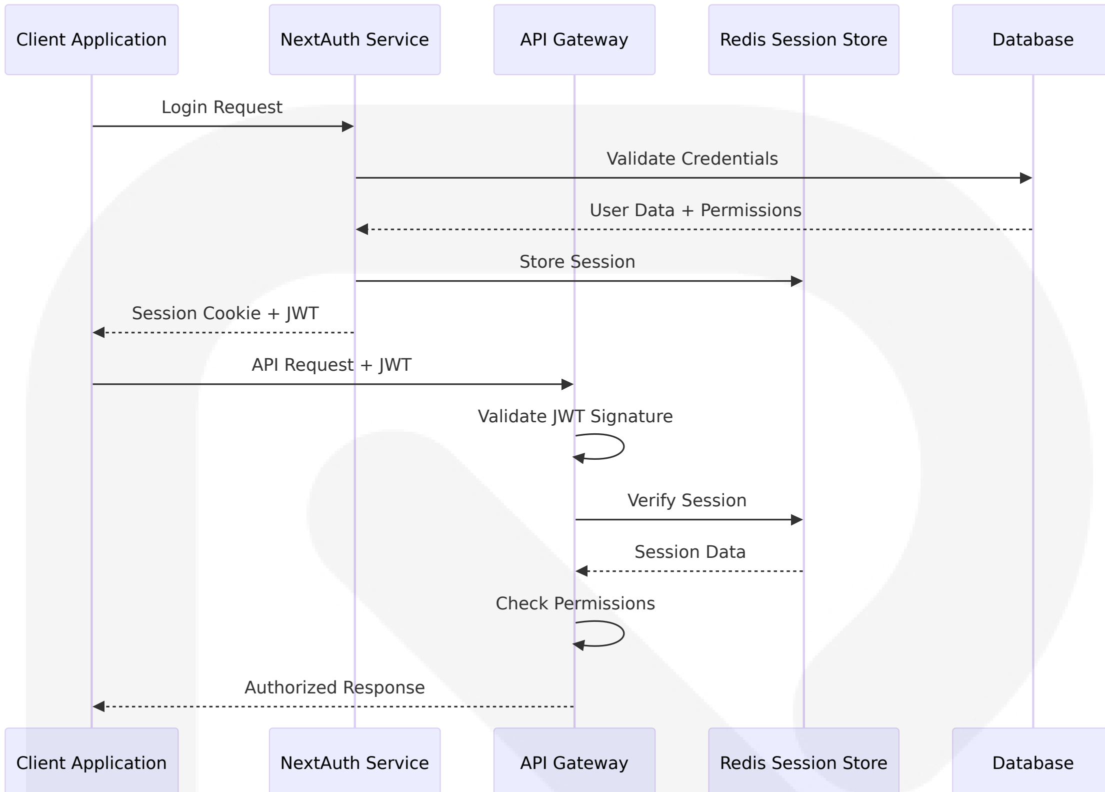
```
    "has_more": true
  }
}
```

6.3.1.2 Authentication Methods

Multi-Layered Authentication Strategy

Authentication Type	Use Case	Implementation	Token Lifetime
NextAuth Session	Web portal access	Session cookies with Redis	24 hours
JWT Bearer Tokens	API access	RS256 signed tokens	1 hour
API Keys	External integrations	HMAC-SHA256 signed requests	Permanent
Webhook Signatures	Event verification	HMAC-SHA256 payload signing	Per-request

Authentication Flow Architecture



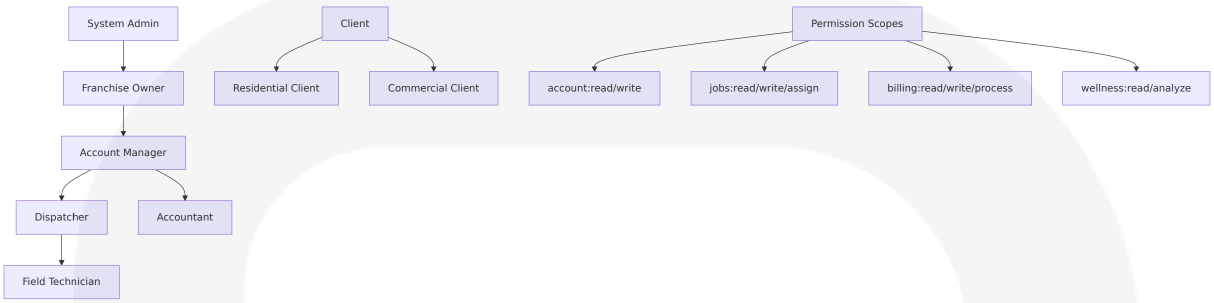
Role-Based Access Control Matrix

Role	API Scope	Rate Limit	Special Permissions
Client	/clients, /billing, /wellness	100 req/min	Own data only
Field Tech	/jobs, /routes	200 req/min	Assigned jobs only
Dispatcher	/jobs, /routes, /schedules	500 req/min	All operational data
Manager	All endpoints	1000 req/min	Account-wide access
System	All endpoints	10000 req/min	Cross-tenant access

6.3.1.3 Authorization Framework

Hierarchical Permission Model

The system implements a comprehensive RBAC framework supporting franchise hierarchies with permission inheritance and tenant isolation:



Permission Enforcement Patterns

- **Resource-Level**: Access control at entity level (clients, jobs, invoices)
- **Field-Level**: Granular permissions for sensitive data (PII, financial)
- **Operation-Level**: Action-specific permissions (create, read, update, delete)
- **Tenant-Level**: Multi-tenant isolation with row-level security

6.3.1.4 Rate Limiting Strategy

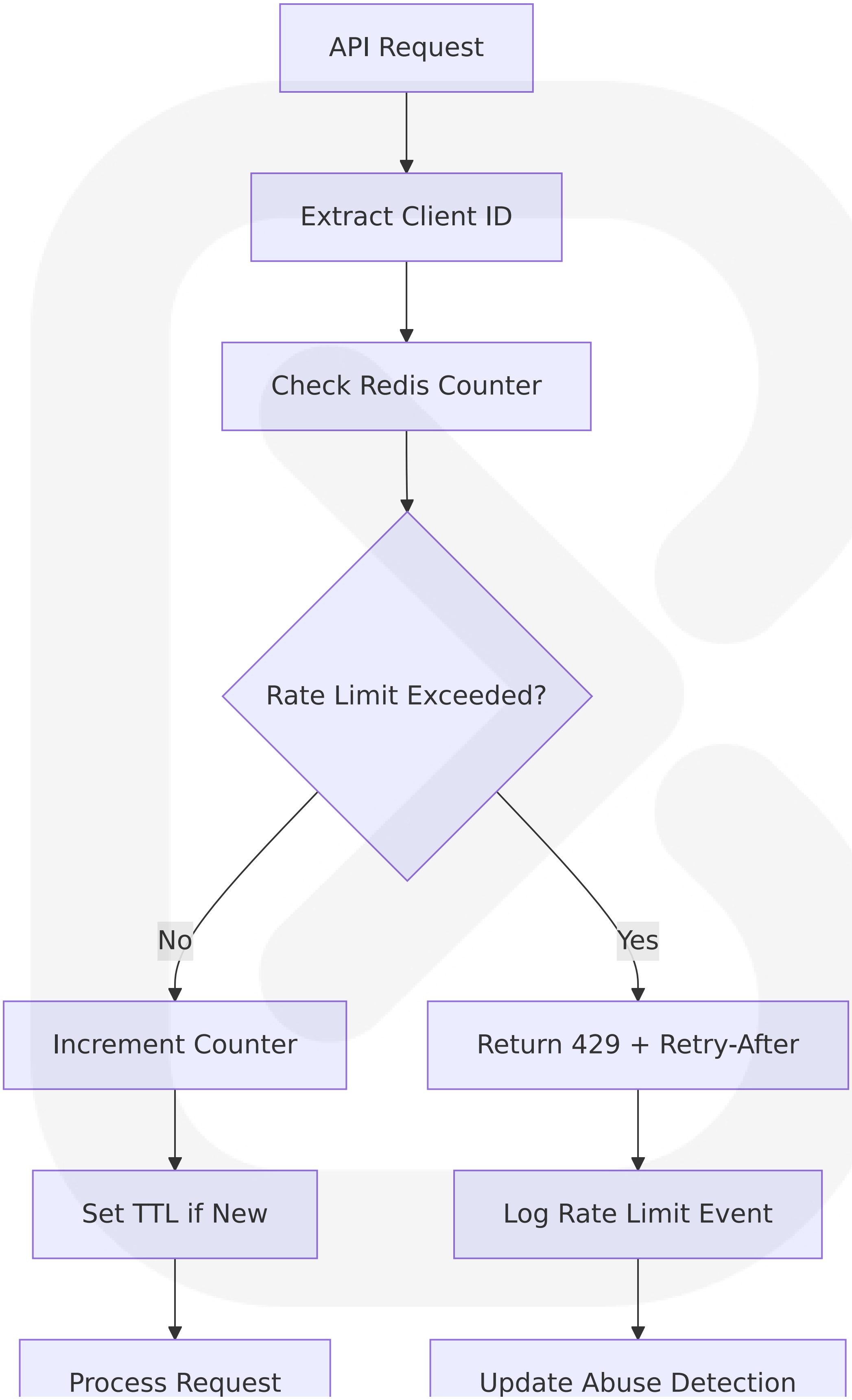
Intelligent Rate Limiting Implementation

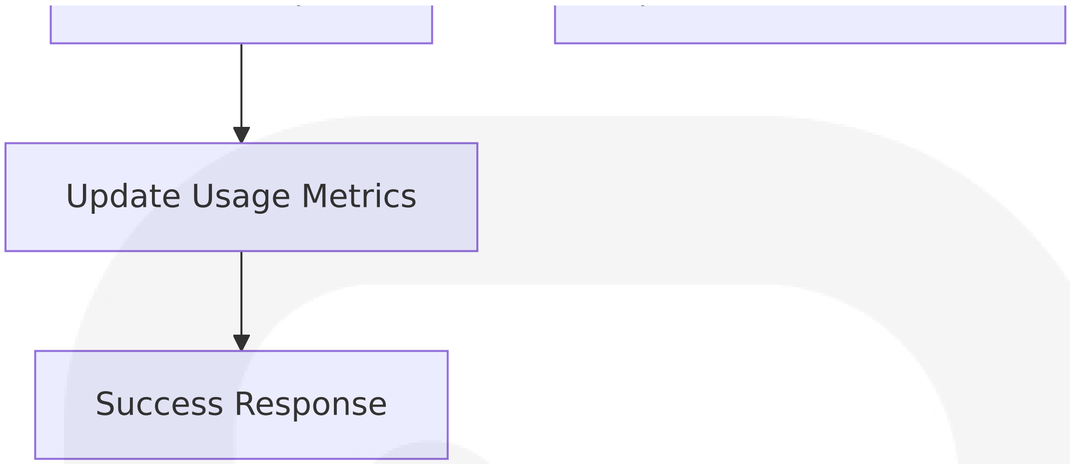
BullMQ version 5.58.5 provides the fastest, most reliable, Redis-based distributed queue for Node, carefully written for rock solid stability and atomicity, enabling sophisticated rate limiting patterns across the platform.

Rate Limiting Tiers

Tier	Requests/Minute	Burst Allowance	Sliding Window
Basic Client	100	150	1 minute
Premium Client	200	300	1 minute
Staff User	500	750	1 minute
System Integration	2000	3000	1 minute

Dynamic Rate Limiting Logic


API Request
Extract Client ID
Check Redis Counter
Rate Limit Exceeded?
No
Yes
Increment Counter
Return 429 + Retry-After
Set TTL if New
Log Rate Limit Event
Process Request
Update Abuse Detection



Adaptive Rate Limiting Features

- **Burst Handling**: Temporary allowance above base limits for legitimate traffic spikes
- **Sliding Windows**: Smooth rate limiting without hard reset boundaries
- **Priority Queuing**: Critical operations bypass standard rate limits
- **Abuse Detection**: Automatic escalation for suspicious traffic patterns

6.3.1.5 Versioning Approach

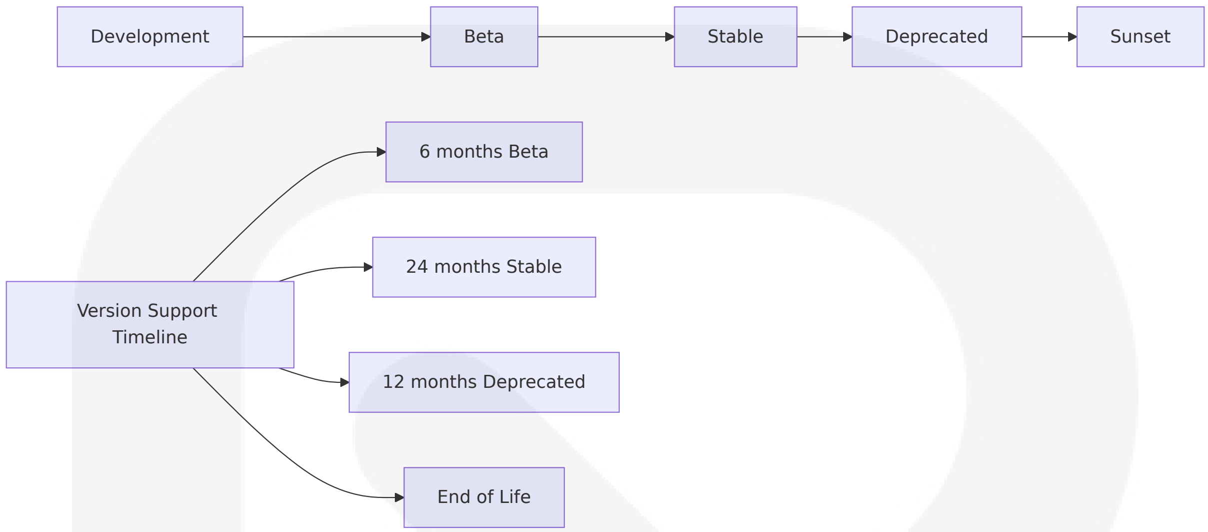
API Versioning Strategy

The system implements semantic versioning with backward compatibility guarantees and graceful deprecation cycles:

Versioning Methods

Method	Implementation	Use Case	Example
URL Path	/v1/, /v2/	Major versions	/v1/clients, /v2/clients
Header	API-Version: 2025-09-10	Minor versions	API-Version: 2025-09-10
Query Parameter	?version=1.2	Development/testing	/clients?version=1.2

Version Lifecycle Management



6.3.1.6 Documentation Standards

Comprehensive API Documentation Framework

Documentation Type	Tool/Format	Update Frequency	Audience
API Reference	OpenAPI 3.0	Automated from code	Developers
Integration Guides	Markdown + Diagrams	Monthly	Partners
SDK Documentation	Auto-generated	Per release	Client developers
Webhook Specifications	JSON Schema	Per event change	Integration teams

Documentation Architecture

- **Interactive Documentation**: Swagger UI with live API testing capabilities
- **Code Examples**: Multi-language examples for all endpoints
- **Postman Collections**: Pre-configured API collections for testing

- **SDK Generation**: Automated client library generation for popular languages

6.3.2 MESSAGE PROCESSING

6.3.2.1 Event Processing Patterns

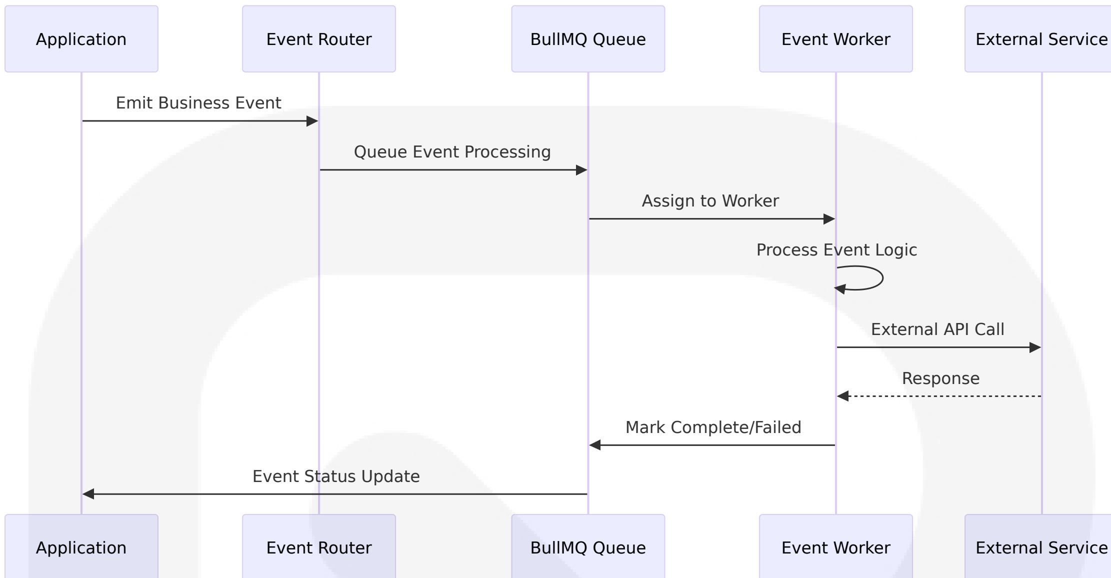
Event-Driven Architecture Implementation

Message queues are a great way to decouple your application components and scale your application by distributing the load across multiple workers, while increasing reliability by adding retries and delays to your jobs.

Core Event Categories

Event Category	Processing Pattern	Reliability Level	Example Events
Business Events	Async with retry	Exactly-once	lead.created, job.completed
System Events	Fire-and-forget	At-least-once	user.login, api.request
Integration Events	Sync + Async	Exactly-once	stripe.payment, qbo.sync
Notification Events	Async batch	At-least-once	sms.send, email.deliver

Event Processing Flow



6.3.2.2 Message Queue Architecture

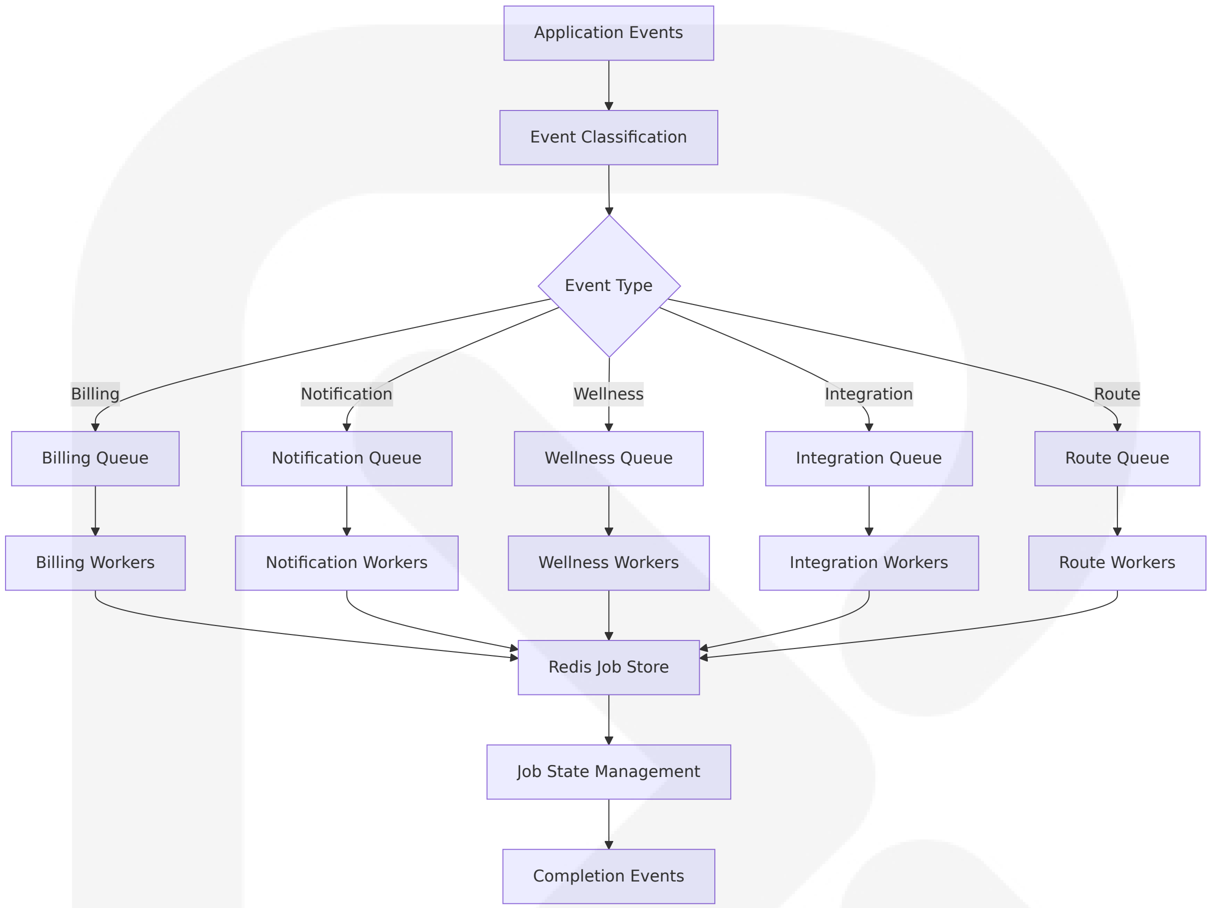
BullMQ-Based Queue System

BullMQ is the fastest, most reliable, Redis-based distributed queue for Node, carefully written for rock solid stability and atomicity, providing the foundation for all asynchronous processing in the Yardura Service OS.

Queue Architecture Design

Queue Name	Purpose	Concurrency	Retry Strategy
billing-queue	Invoice generation, payments	5 workers	Exponential backoff, 5 retries
notification-queue	SMS, email, push notifications	10 workers	Linear backoff, 3 retries
wellness-queue	Photo analysis, trend calculation	3 workers	Fixed delay, 2 retries
integration-queue	External API synchronization	5 workers	Exponential backoff, 10 retries
route-queue	Route optimization processing	2 workers	No retry (time-sensitive)

Message Processing Patterns



6.3.2.3 Stream Processing Design

Real-Time Data Streaming Architecture

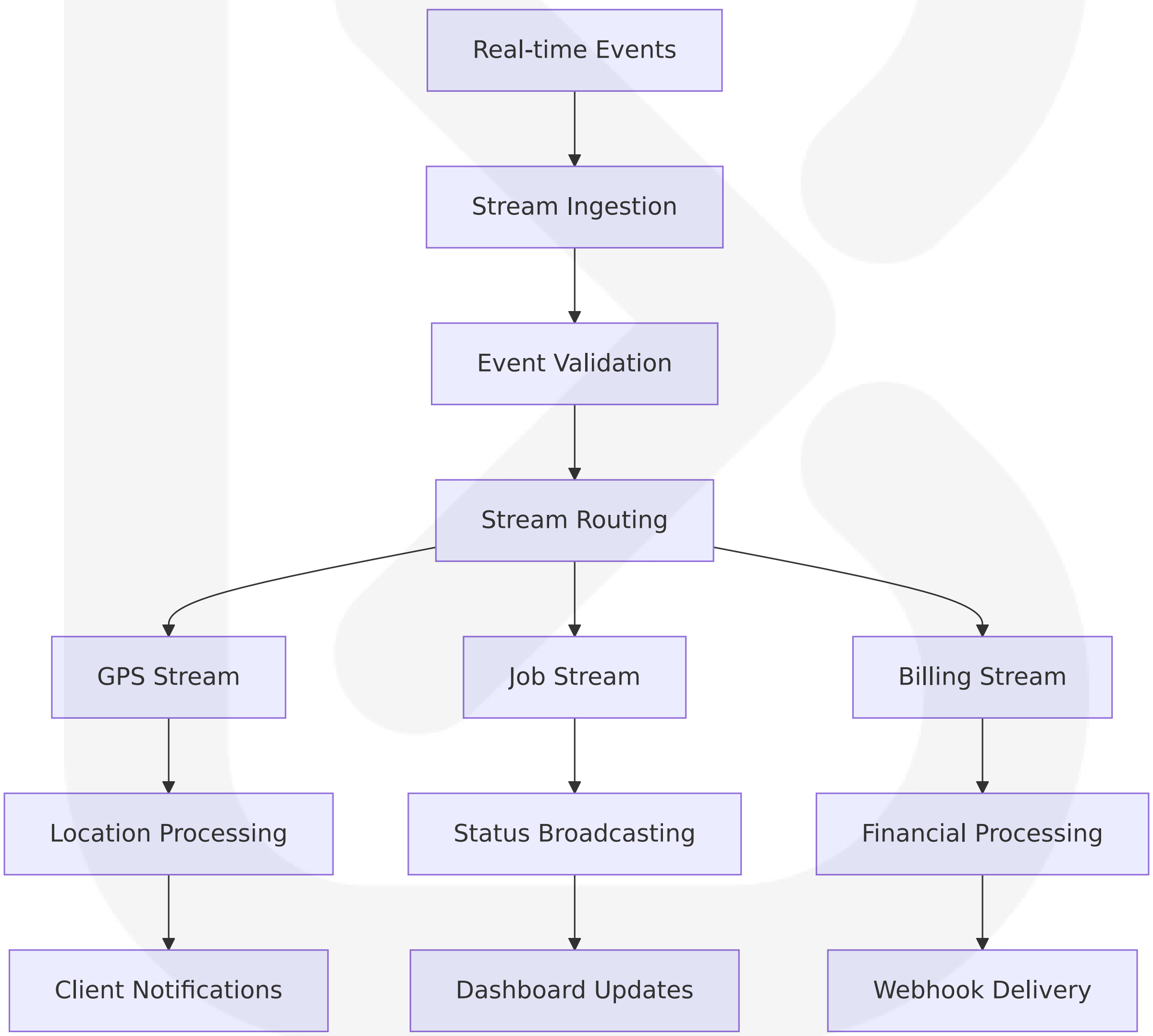
The system implements real-time streaming for critical business operations including GPS tracking, job status updates, and billing events.

Stream Processing Components

Stream Type	Technology	Throughput	Latency Target
GPS Tracking	WebSocket + Redis Streams	1000 events/sec	< 100ms

Stream Type	Technology	Throughput	Latency Target
Job Updates	Server-Sent Events	500 events/sec	< 200ms
Billing Events	BullMQ + Webhooks	100 events/sec	< 500ms
Wellness Analytics	Batch processing	50 events/sec	< 5 seconds

Stream Processing Flow



6.3.2.4 Batch Processing Flows

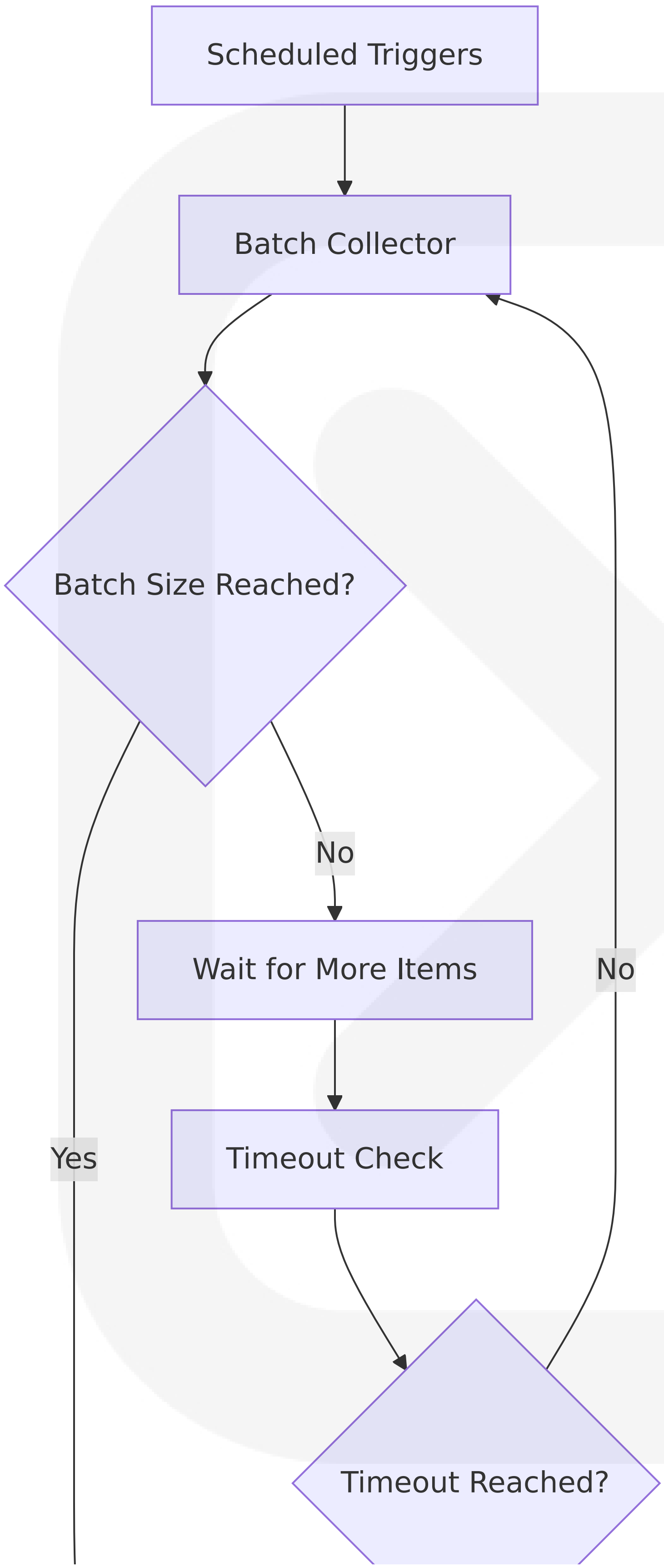
Efficient Batch Processing Implementation

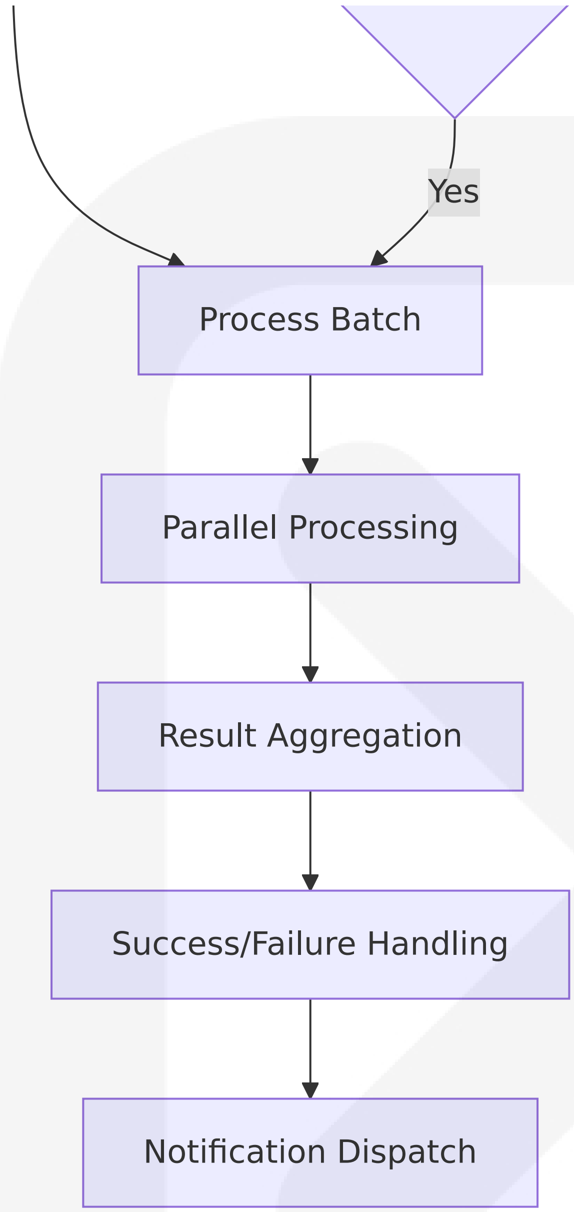
BullMQ increases efficiency by consuming jobs in batches, a strategy that minimizes overhead and can boost throughput, enabling high-performance processing for bulk operations.

Batch Processing Categories

Batch Type	Schedule	Batch Size	Processing Time
Daily Billing	2:00 AM daily	1000 invoices	15 minutes
Wellness Analysis	Every 2 hours	100 photos	10 minutes
Route Optimization	6:00 AM daily	500 jobs	5 minutes
QBO Synchronization	Every 4 hours	200 records	8 minutes

Batch Processing Architecture


Scheduled Triggers
Batch Collector
Batch Size Reached?
No
Wait for More Items
No
Yes
Timeout Check
Timeout Reached?



6.3.2.5 Error Handling Strategy

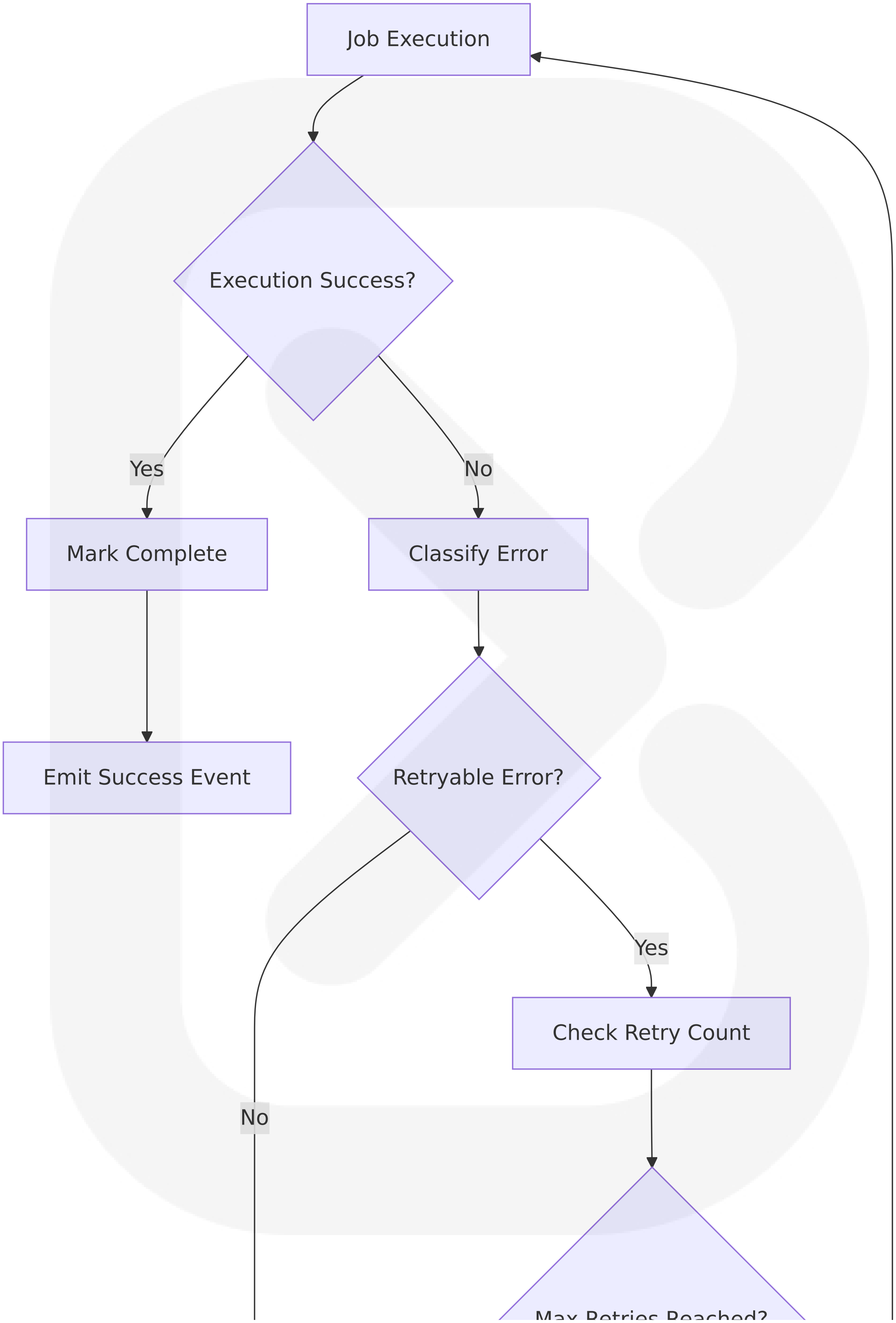
Comprehensive Error Recovery Framework

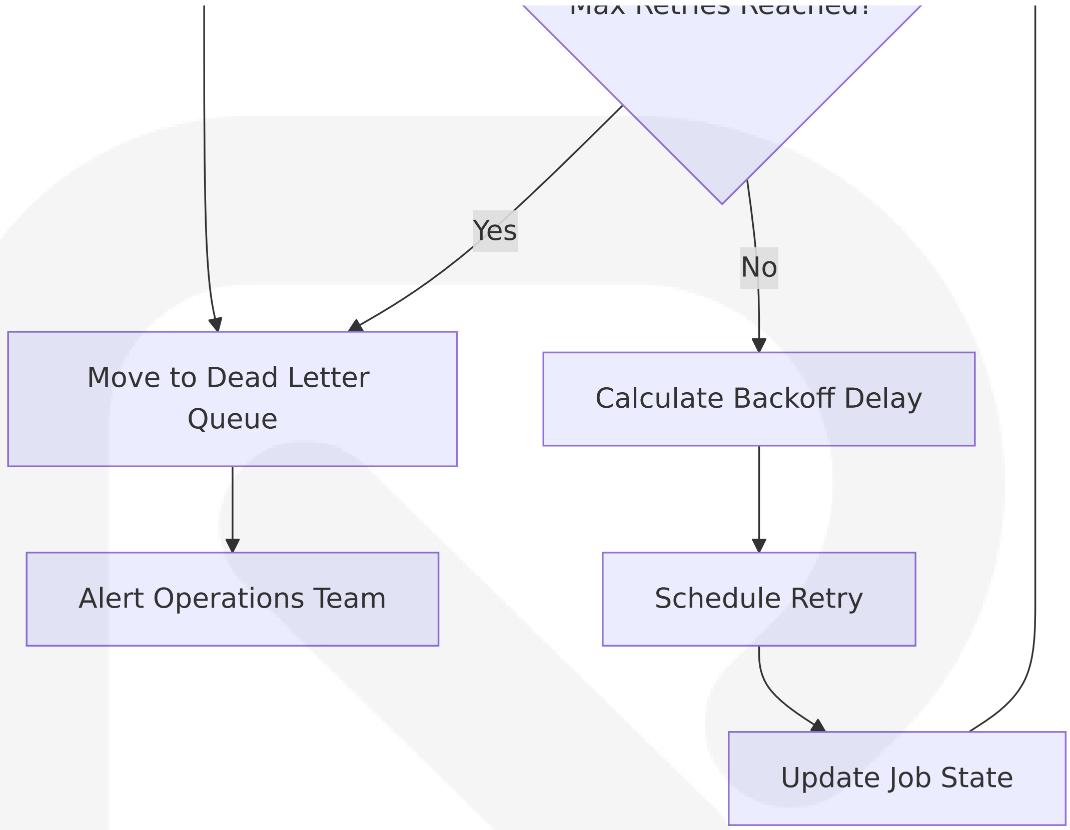
The system implements sophisticated error handling with automatic recovery, dead letter queues, and manual intervention capabilities.

Error Classification and Handling

Error Type	Recovery Strategy	Escalation Path	Manual Intervention
Transient Network	Exponential backoff retry	After 5 failures	Support notification
Authentication	Token refresh + retry	After 3 failures	Admin notification
Rate Limiting	Delayed retry	Queue backlog alert	Capacity planning
Data Validation	Immediate failure	Error logging	Data correction

Error Recovery Flow


Job Execution
Execution Success?
Yes
No
Mark Complete
Classify Error
Emit Success Event
Retryable Error?
Yes
No
Check Retry Count



6.3.3 EXTERNAL SYSTEMS

6.3.3.1 Third-Party Integration Patterns

Comprehensive External Integration Framework

The Yardura Service OS integrates with multiple external systems using standardized patterns for reliability, security, and maintainability.

Primary External Integrations

Service	Integration Type	Data Flow	Update Frequency
Stripe API	REST + Webhooks	Bidirectional	Real-time
QuickBooks Online	REST + OAuth2	Bidirectional	Hourly batch

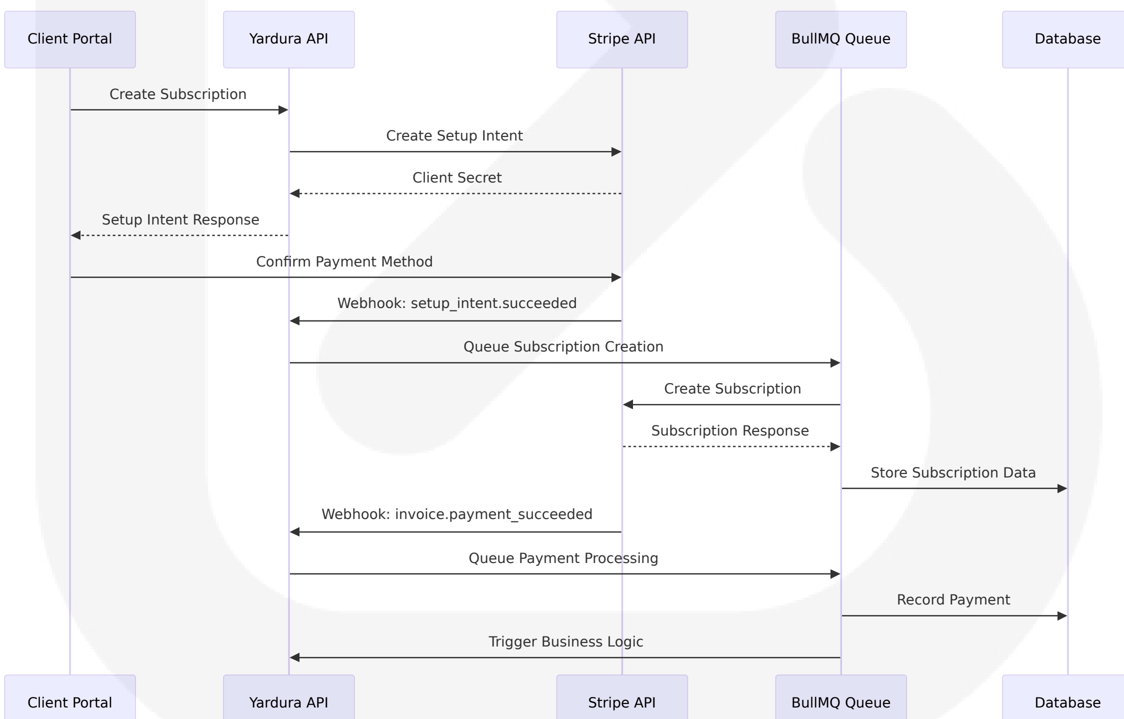
Service	Integration Type	Data Flow	Update Frequency
Google Maps	REST API	Unidirectional	On-demand
Twilio	REST API	Unidirectional	Real-time

6.3.3.2 Stripe Payment Integration

Advanced Payment Processing Architecture

Stripe API version 2025-08-27.basil introduces personalized invoices, ad hoc pricing for Payment Links, billing improvements with subscription schedules supporting phases with mixed durations, and flexible billing mode with thresholds for usage-based billing.

Stripe Integration Components



Stripe Webhook Processing

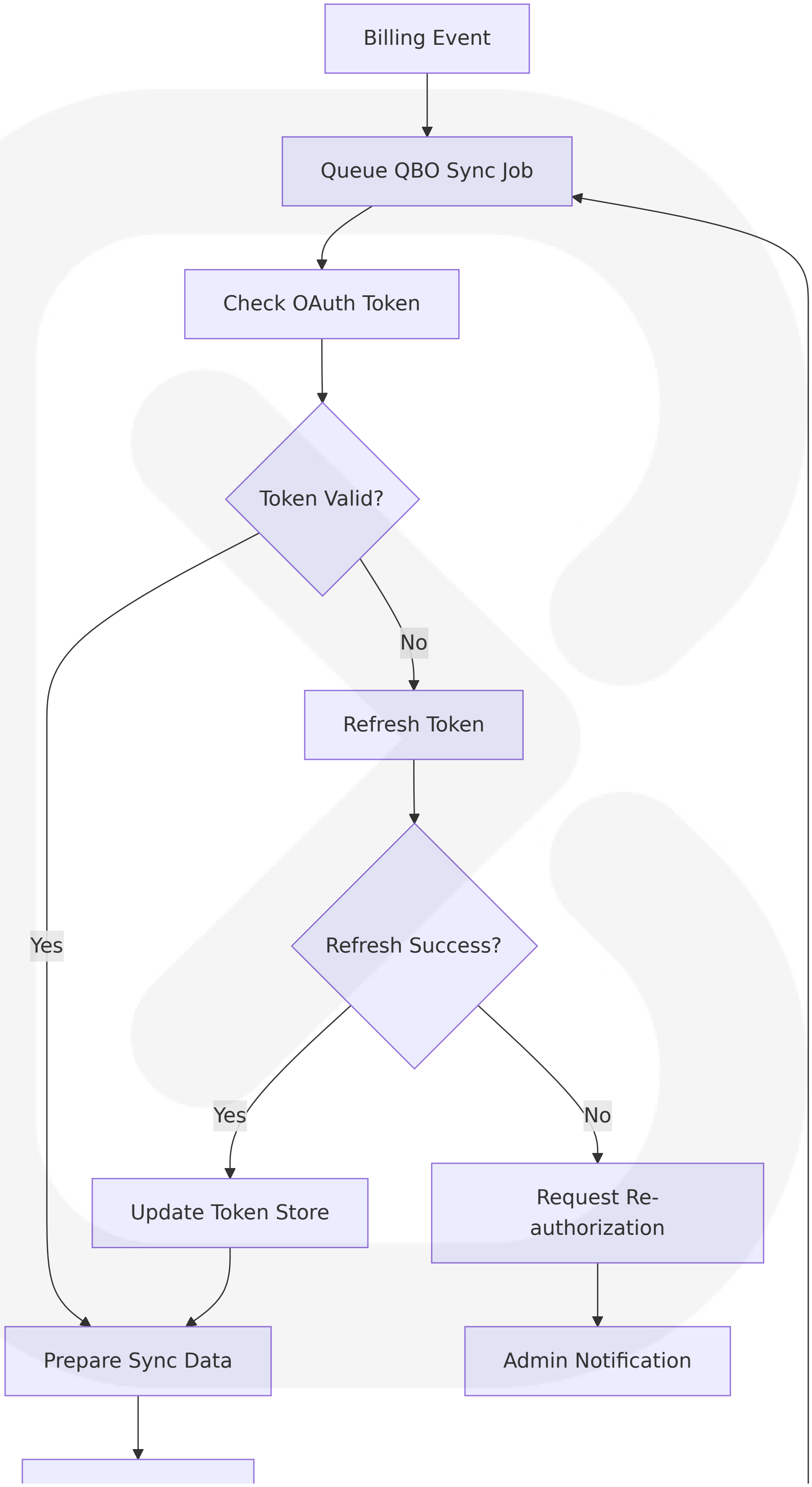
Webhook Event	Processing Priority	Retry Strategy	Business Impact
payment_intent.succeeded	High	5 retries, exponential backoff	Service activation
invoice.payment_failed	Critical	10 retries, immediate + delayed	Dunning process
customer.subscription.updated	Medium	3 retries, linear backoff	Service modification
setup_intent.succeeded	High	5 retries, exponential backoff	Payment method setup

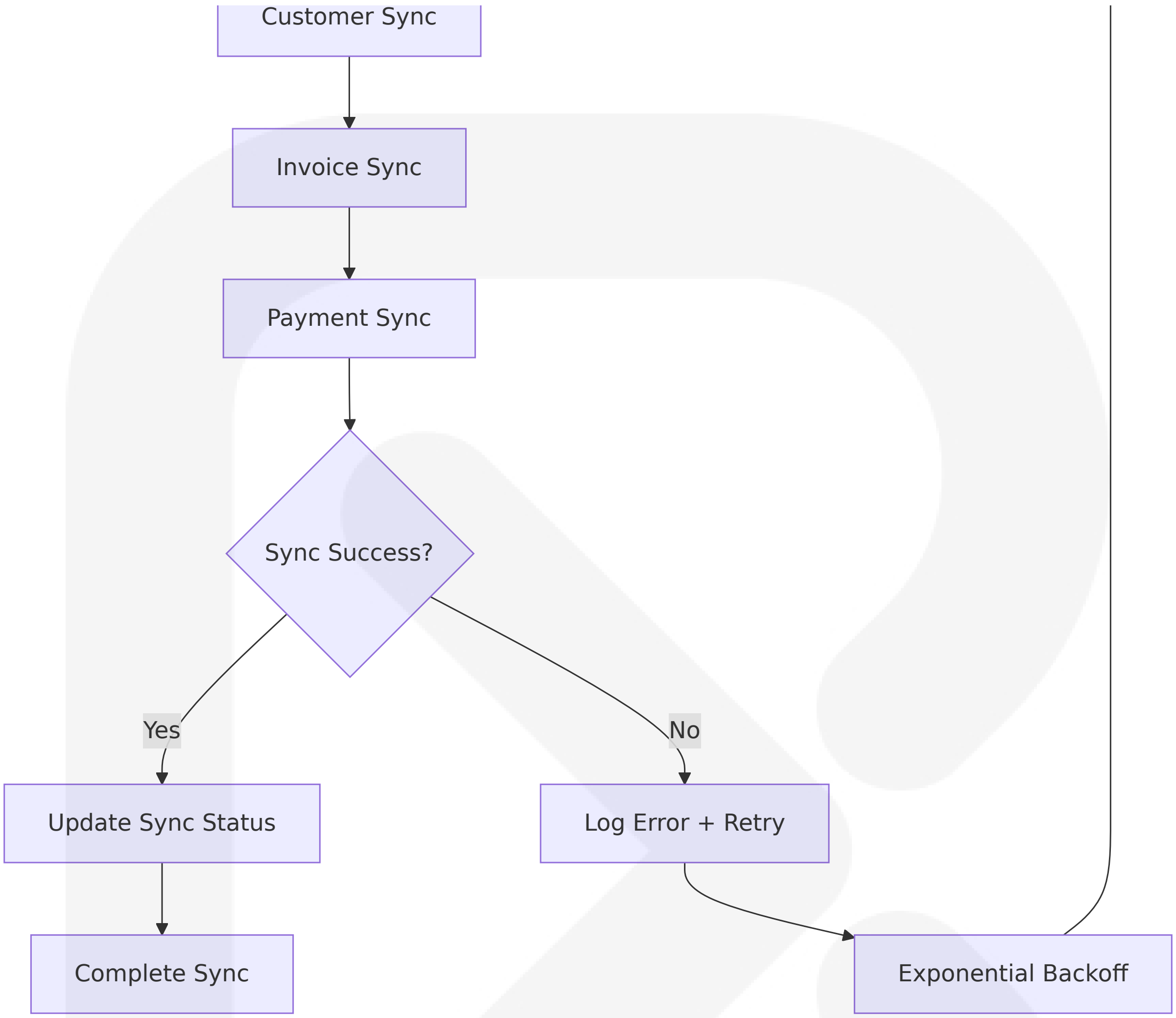
6.3.3.3 QuickBooks Online Integration

Accounting Synchronization Architecture

Starting August 1, 2025, all API requests to the QuickBooks Online Accounting API will default to minor version 75, with previous minor versions being ignored.

QBO Integration Flow


Billing Event
Queue QBO Sync Job
Check OAuth Token
Token Valid?
No
Yes
Refresh Token
Refresh Success?
Yes
No
Update Token Store
Request Re-authorization
Prepare Sync Data
Admin Notification



QBO Data Mapping Strategy

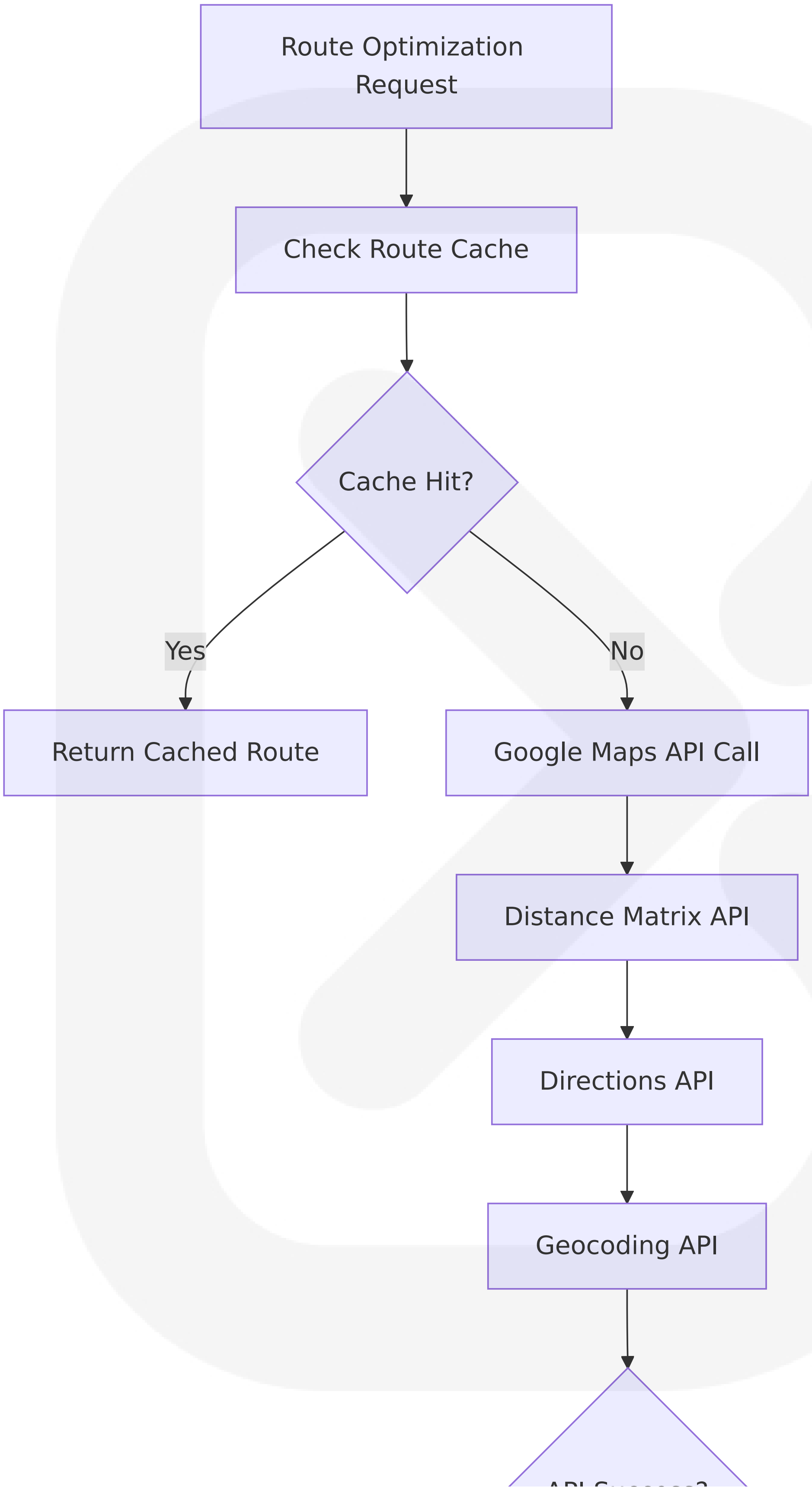
Yardura Entity	QBO Entity	Sync Direction	Conflict Resolution
Client	Customer	Bidirectional	Last-write-wins
Subscription	Recurring Template	Yardura → QBO	Yardura authoritative
Invoice	Invoice	Bidirectional	Manual review required
Payment	Payment	Bidirectional	Automatic reconciliation

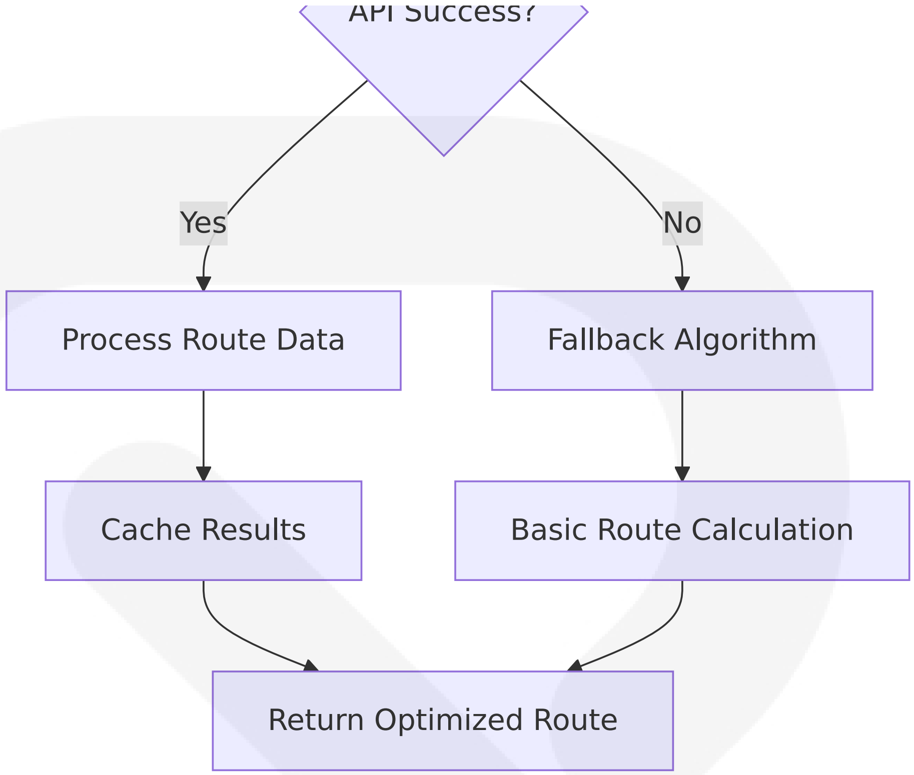
6.3.3.4 Google Maps Integration

Route Optimization Service Integration

The system leverages Google Maps Platform APIs for comprehensive routing and location services with intelligent caching and fallback mechanisms.

Maps API Integration Architecture


Route Optimization Request
Check Route Cache
Cache Hit?
Yes
No
Return Cached Route
Google Maps API Call
Distance Matrix API
Directions API
Geocoding API



Maps API Usage Optimization

API Endpoint	Usage Pattern	Caching Strategy	Rate Limiting
Distance Matrix	Batch optimization	1 hour TTL	100 elements/request
Directions	Individual routes	30 minutes TTL	50 requests/second
Geocoding	Address validation	24 hours TTL	50 requests/second
Places	Address autocomplete	1 hour TTL	100 requests/second

6.3.3.5 Communication Services Integration

Multi-Channel Communication Architecture

The system integrates multiple communication channels for comprehensive client and staff notifications.

Communication Integration Matrix

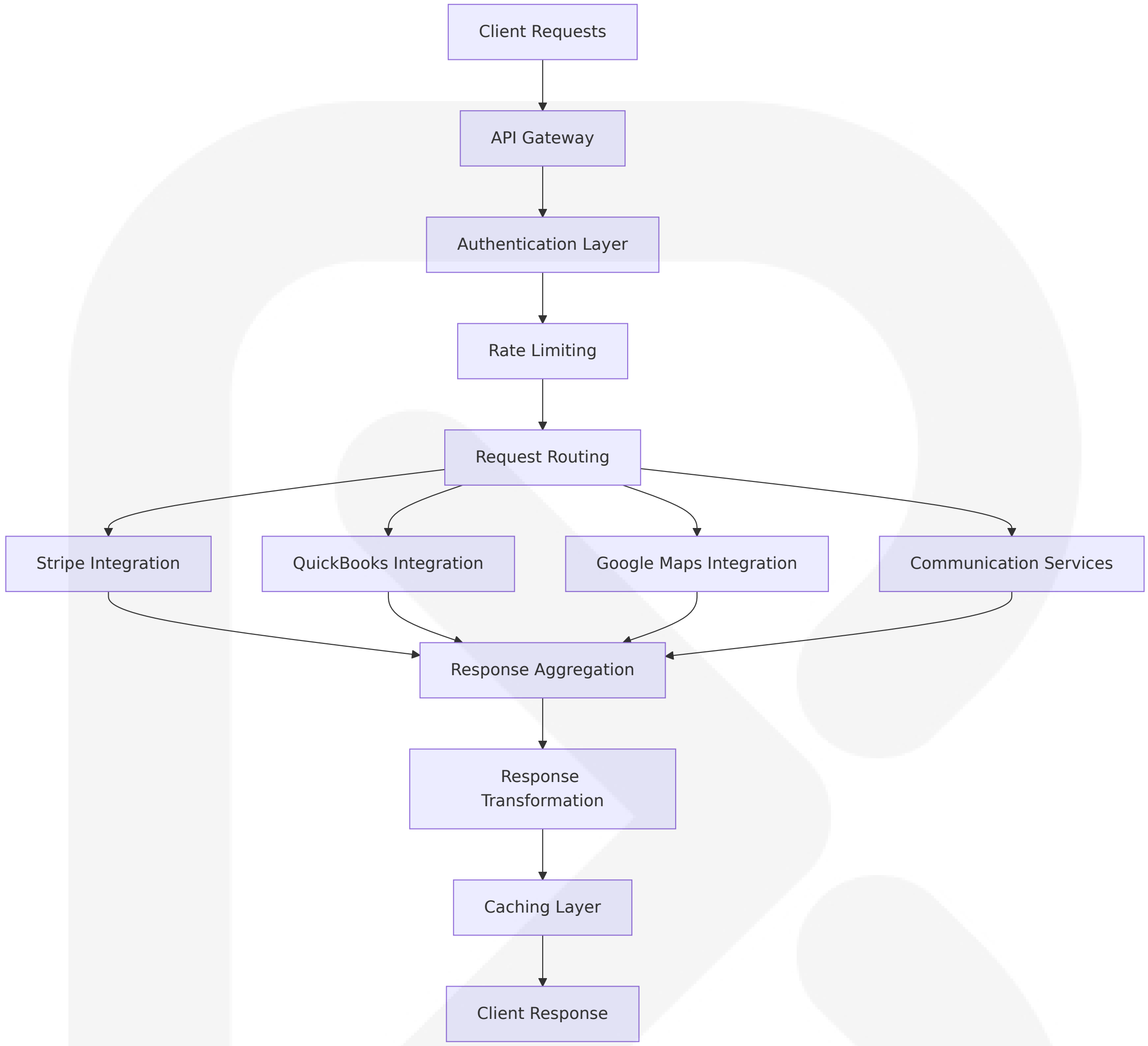
Channel	Service Provider	Use Cases	Delivery SLA
SMS	Twilio	On-the-way notifications, alerts	< 30 seconds
Voice	Twilio	Emergency notifications	< 10 seconds
Email	Resend/SendGrid	Invoices, reports, marketing	< 2 minutes
Push	Web Push API	PWA notifications	< 5 seconds

6.3.3.6 API Gateway Configuration

Centralized API Management

The system implements a comprehensive API gateway pattern for external service management, security, and monitoring.

Gateway Architecture Components



Gateway Configuration Standards

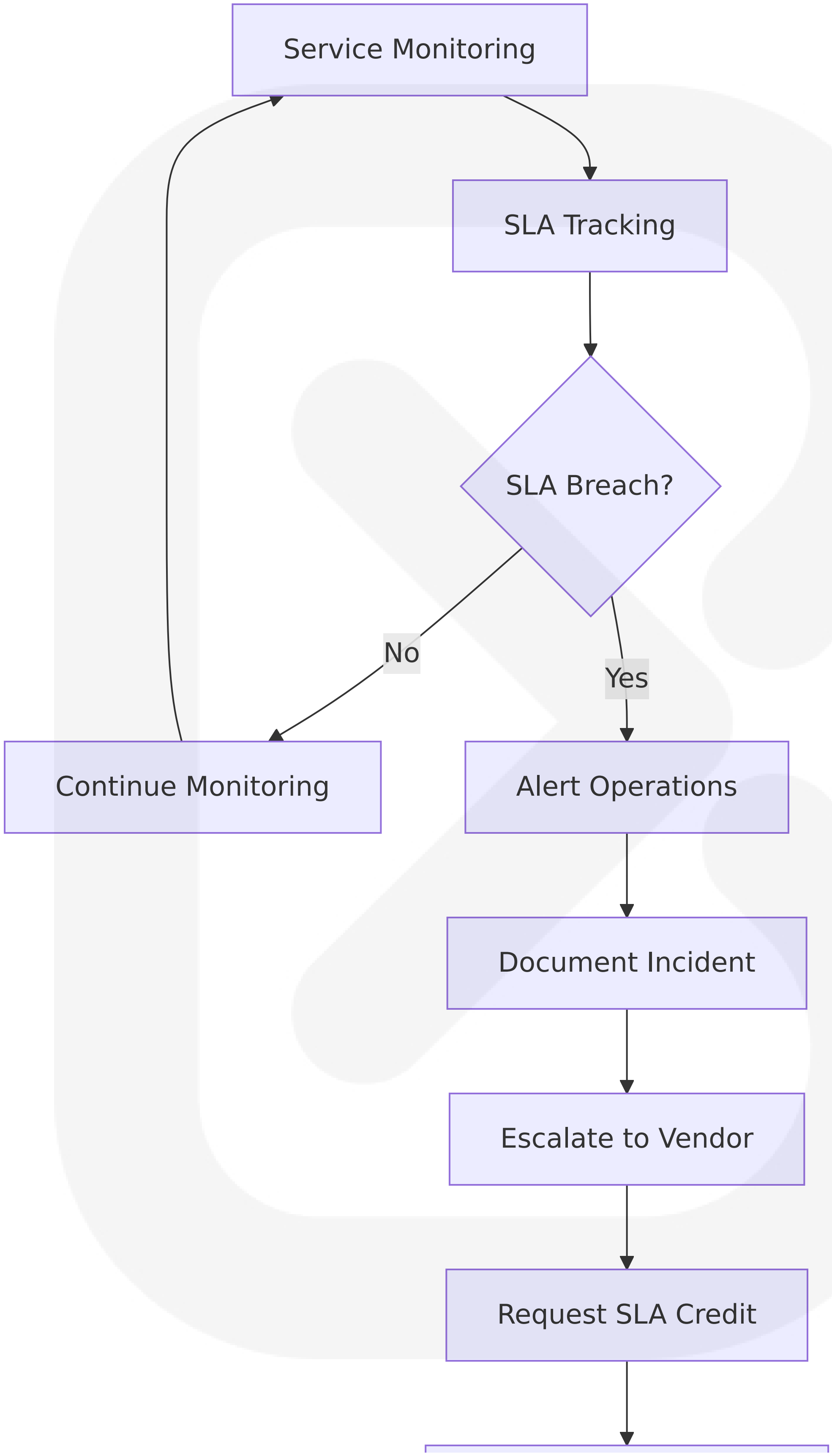
Configuration	Value	Purpose	Monitoring
Timeout	30 seconds	Prevent hanging requests	Response time metrics
Retry Policy	3 attempts, exponential backoff	Handle transient failures	Retry rate tracking
Circuit Breaker	50% failure rate, 60s timeout	Prevent cascade failures	Circuit state monitoring
Load Balancing	Round-robin with health checks	Distribute load evenly	Health check status

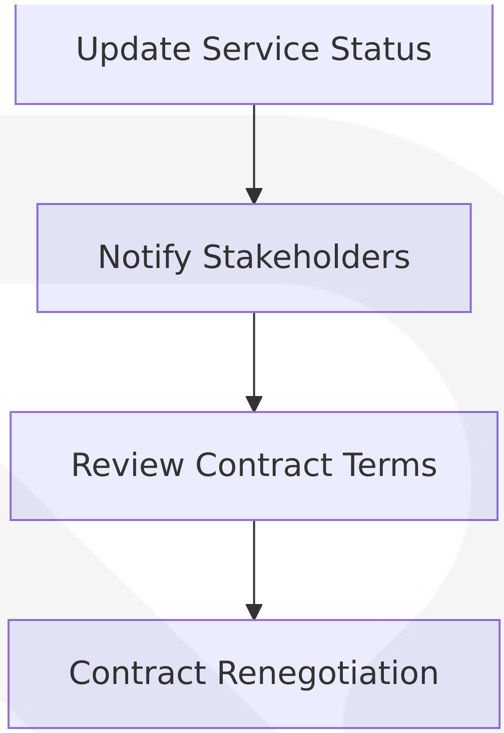
6.3.3.7 External Service Contracts

Service Level Agreements and Contracts

Service	Availability SLA	Response Time SLA	Support Level
Stripe	99.99%	< 500ms P95	24/7 Enterprise
QuickBooks Online	99.5%	< 2000ms P95	Business hours
Google Maps	99.9%	< 1000ms P95	Standard support
Twilio	99.95%	< 200ms P95	24/7 Premium

Contract Monitoring and Compliance


Service Monitoring
SLA Tracking
SLA Breach?
No
Yes
Continue Monitoring
Alert Operations
Document Incident
Escalate to Vendor
Request SLA Credit



This comprehensive integration architecture provides robust, scalable, and maintainable connections to all external services while ensuring reliability, security, and performance requirements are met across the entire Yardura Service OS platform.

6.4 SECURITY ARCHITECTURE

6.4.1 AUTHENTICATION FRAMEWORK

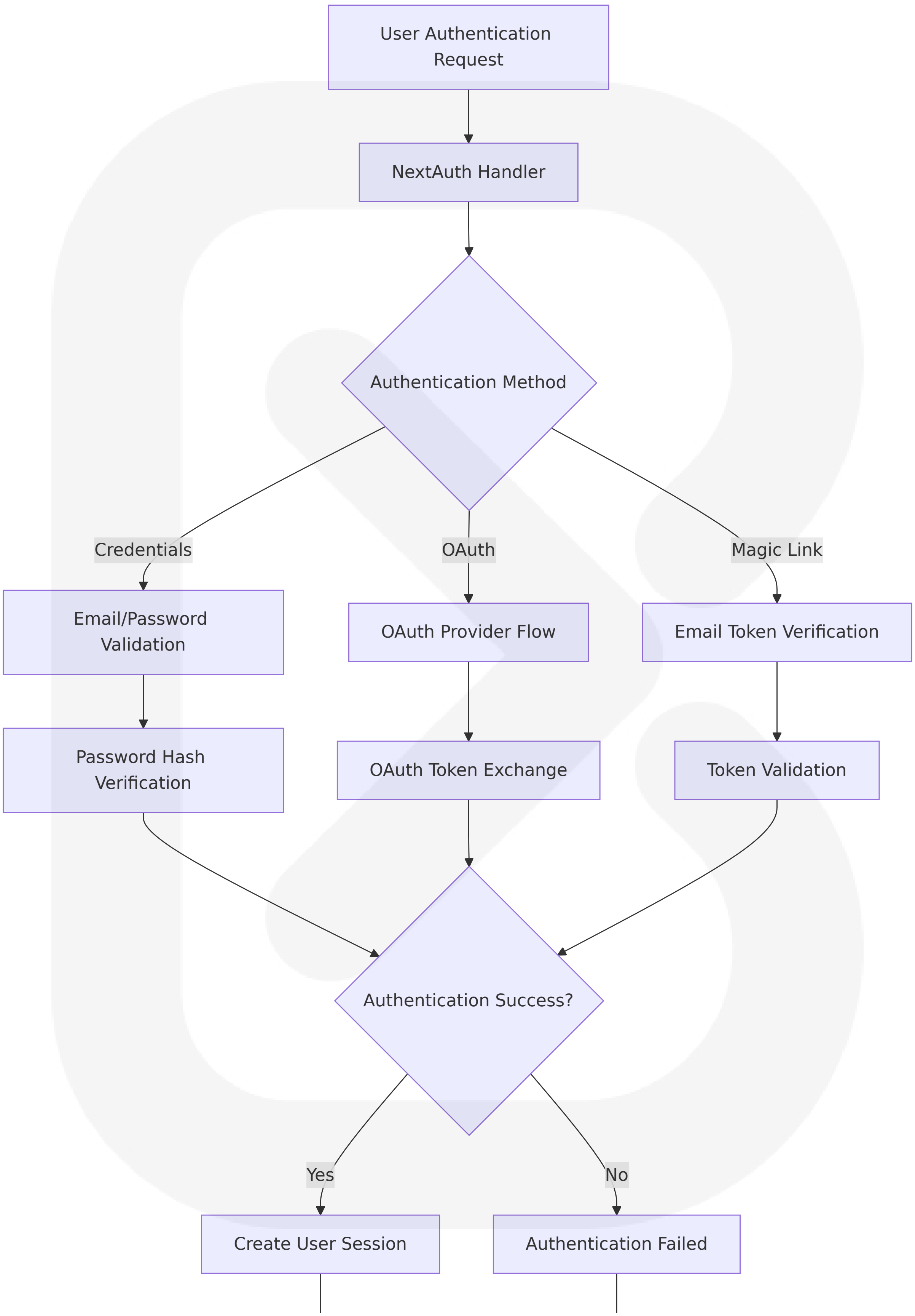
6.4.1.1 Identity Management System

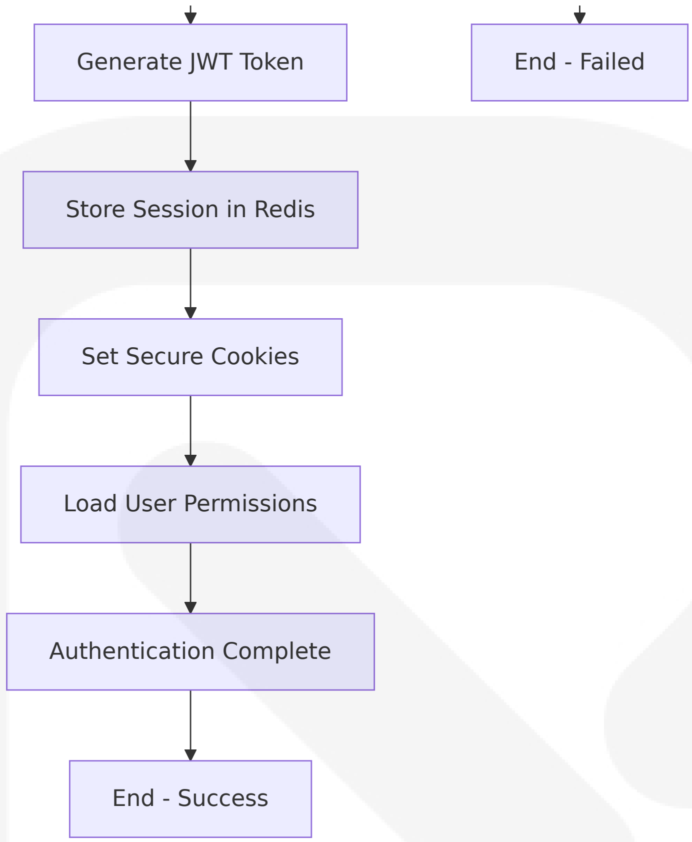
The Yardura Service OS implements a comprehensive identity management system built on NextAuth for increased security and simplicity, offering built-in solutions for authentication, session management, and authorization, as well as additional features such as social logins, multi-factor authentication, and role-based access control.

Identity Provider Integration

Provider Type	Implementation	Use Cases	Security Features
Email/Password	NextAuth Credentials	Staff and client accounts	bcrypt hashing, rate limiting
OAuth2 Providers	NextAuth OAuth	Google, Microsoft SSO	PKCE, state validation
Magic Links	NextAuth Email	Passwordless authentication	Time-limited tokens
API Keys	Custom HMAC	External integrations	SHA-256 signatures

User Identity Architecture


User Authentication Request
NextAuth Handler
Authentication Method
Credentials
OAuth
Magic Link
Email/Password Validation
OAuth Provider Flow
Email Token Verification
Password Hash Verification
OAuth Token Exchange
Token Validation
Authentication Success?
Yes
No
Create User Session
Authentication Failed



6.4.1.2 Multi-Factor Authentication

MFA Implementation Strategy

Implement short-lived access tokens and refresh tokens to ensure sessions remain secure. You can use next-auth's session management features for this, combined with comprehensive MFA options for enhanced security.

MFA Methods and Configuration

MFA Method	Implementation	Target Users	Security Level
SMS OTP	Twilio integration	All staff users	Medium
TOTP Apps	Google Authenticator, Authy	Admin users	High
Hardware Keys	WebAuthn/FIDO2	Executive users	Very High

MFA Method	Implementation	Target Users	Security Level
Backup Codes	Encrypted storage	All MFA users	Recovery only

MFA Enforcement Policies

- **Mandatory MFA**: All staff accounts with billing, payroll, or administrative access
- **Optional MFA**: Client accounts with enhanced security preferences
- **Risk-Based MFA**: Triggered by unusual login patterns or high-risk operations
- **Grace Period**: 7-day enrollment period for new staff accounts

6.4.1.3 Session Management

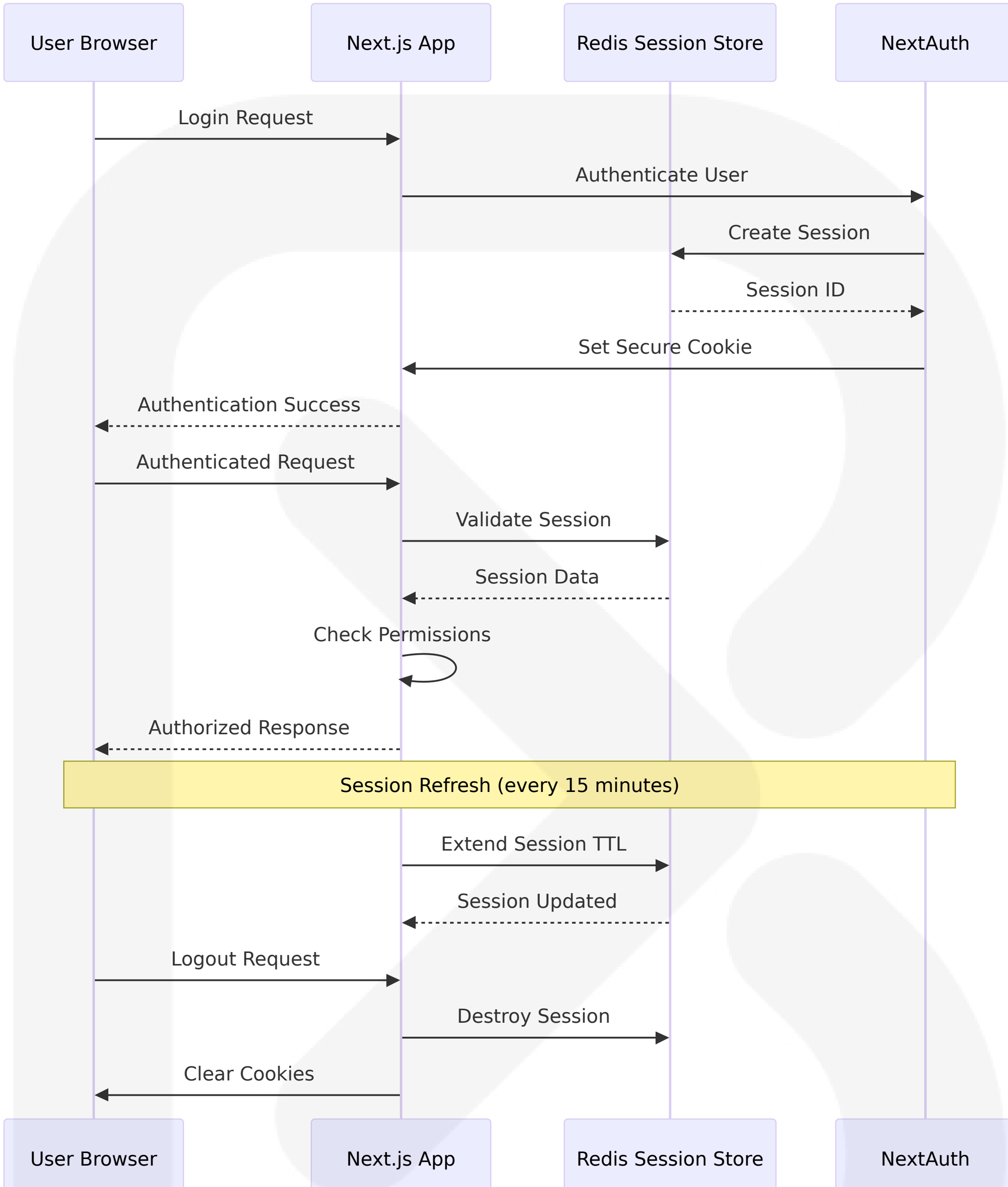
Secure Session Architecture

Always store JWTs in HttpOnly cookies. These special cookies cannot be accessed by client-side JavaScript, providing protection against XSS attacks.

Session Configuration Standards

Session Parameter	Value	Security Rationale	Implementation
Session Duration	24 hours	Balance security/usability	Sliding expiration
Refresh Token TTL	30 days	Long-term authentication	Rotation on use
Cookie Security	HttpOnly, Secure, SameSite	XSS/CSRF protection	Next.js cookies API
Session Storage	Redis with encryption	Distributed sessions	AES-256 encryption

Session Lifecycle Management



6.4.1.4 Token Handling

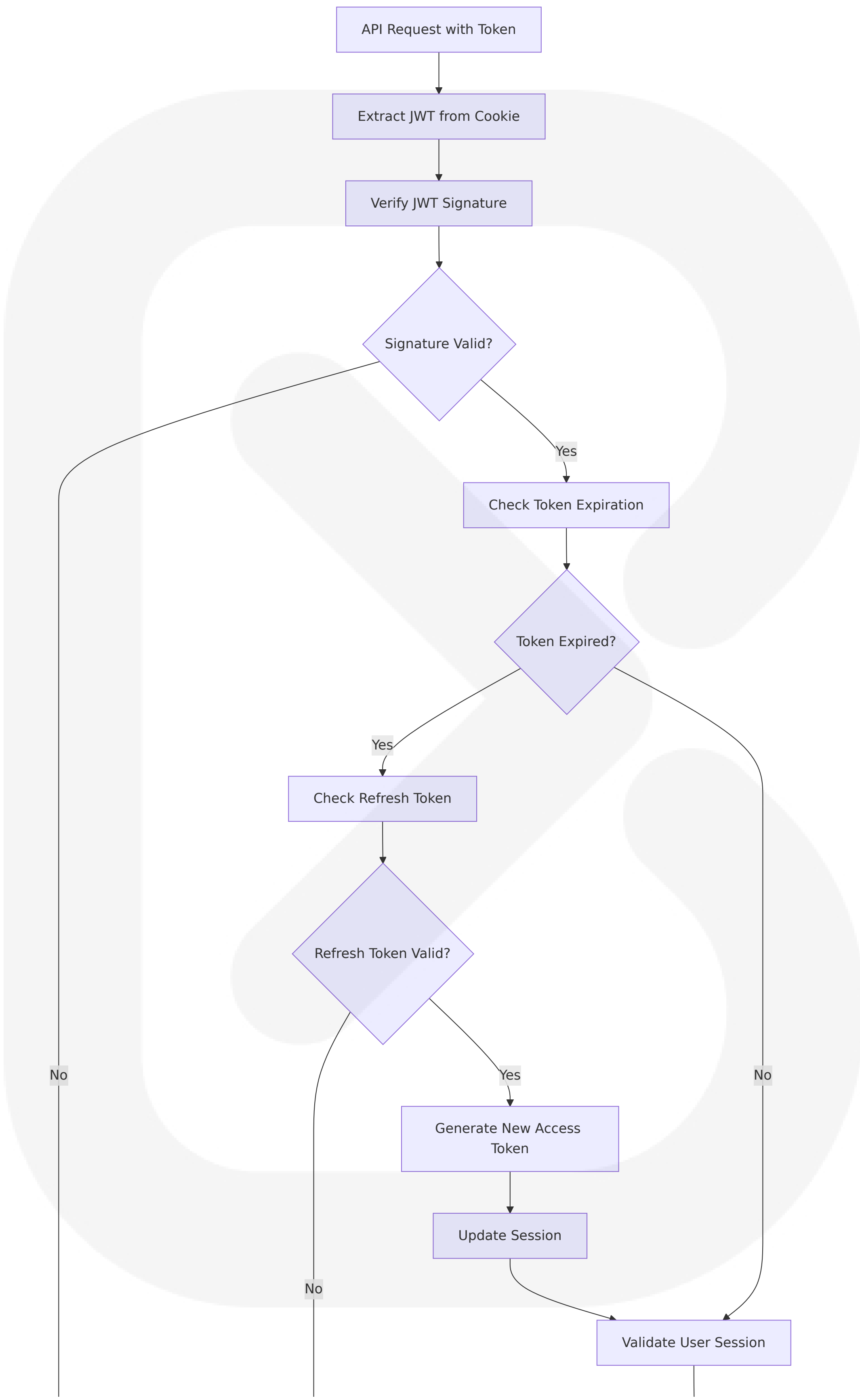
JWT Token Management

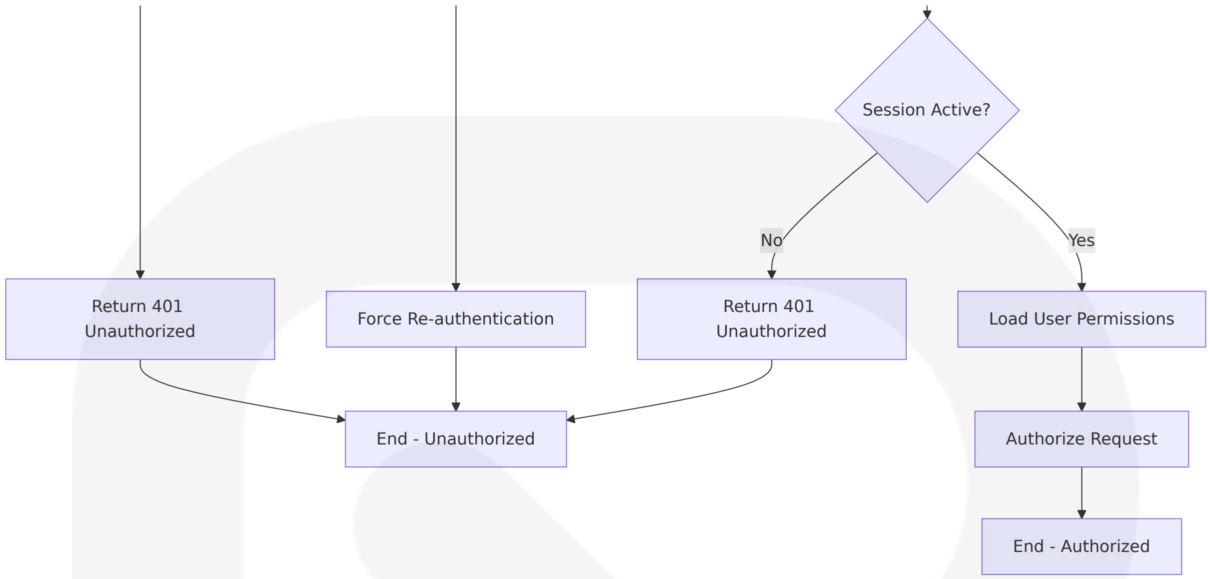
To enhance security while maintaining a good user experience, implement token rotation with refresh tokens. This approach uses short-lived access tokens (15 minutes) with longer-lived refresh tokens.

Token Security Implementation

Token Type	Lifetime	Storage Location	Rotation Policy
Access Token	15 minutes	HttpOnly cookie	On expiration
Refresh Token	30 days	HttpOnly cookie	On use
API Key	Permanent	Database hash	Manual rotation
Webhook Token	Per request	Request signature	Single use

Token Validation Flow


API Request with Token
Extract JWT from Cookie
Verify JWT Signature
Signature Valid?
Yes
No
Check Token Expiration
Token Expired?
Yes
No
Check Refresh Token
Refresh Token Valid?
Yes
No
Generate New Access Token
Update Session
Validate User Session



6.4.1.5 Password Policies

Password Security Standards

Password-based authentication is generally discouraged due to security risks including phishing attacks, brute force attacks, and credential stuffing attacks. However, when implemented, the system enforces comprehensive password policies.

Password Requirements Matrix

User Type	Minimum Length	Complexity Requirements	Rotation Policy
Client Users	8 characters	Mixed case, numbers	Optional
Staff Users	12 characters	Mixed case, numbers, symbols	90 days
Admin Users	14 characters	Mixed case, numbers, symbols	60 days
Service Accounts	16 characters	Random generation	30 days

Password Security Implementation

- **Hashing Algorithm**: bcrypt with cost factor 12
- **Salt Generation**: Cryptographically secure random salts
- **Breach Detection**: Integration with HaveIBeenPwned API
- **Account Lockout**: 5 failed attempts, 15-minute lockout
- **Password History**: Prevent reuse of last 12 passwords

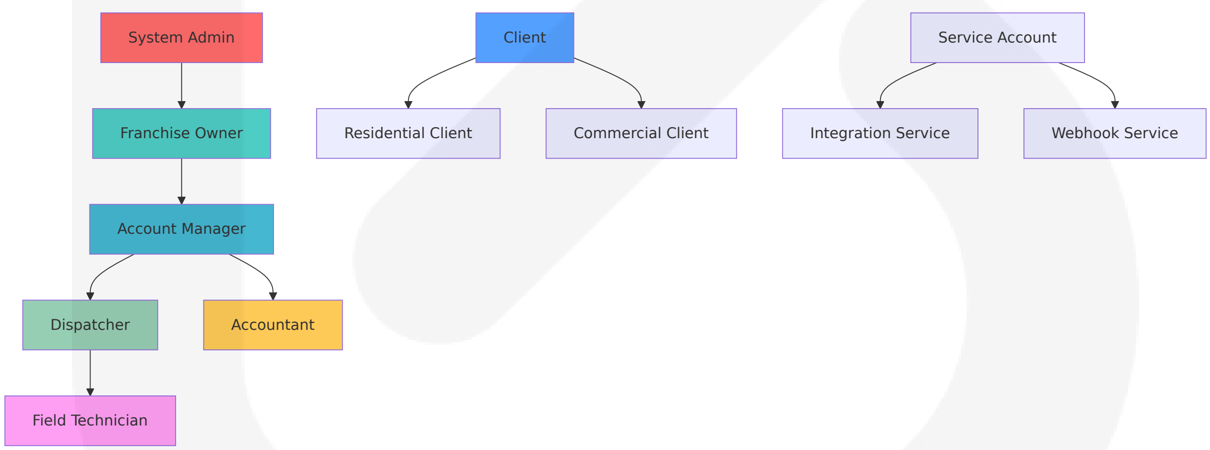
6.4.2 AUTHORIZATION SYSTEM

6.4.2.1 Role-Based Access Control (RBAC)

RBAC Architecture Implementation

Role-Based Access Control (RBAC) is a policy-neutral access control mechanism defined around roles and privileges. Rather than assigning permissions to individual users, RBAC assigns permissions to roles, and then users are assigned to those roles. This abstraction simplifies the management of user permissions and improves security.

Role Hierarchy Structure



Role Permission Matrix

Role	Client Portal	Field Operations	Dispatch	Billing	Admin	Cr e
System Admin	Full	Full	Full	Full	Full	Ye
Franchise Owner	View All	Full	Full	Full	Full	Su un
Account Manager	View All	Full	Full	Full	Limited	No
Dispatcher	None	View All	Full	View Only	None	No
Accountant	None	None	View Only	Full	None	No
Field Tech	None	Assigned Only	View Assigned	None	None	No
Client	Own Data	None	None	Own Billing	None	No

6.4.2.2 Permission Management

Granular Permission System

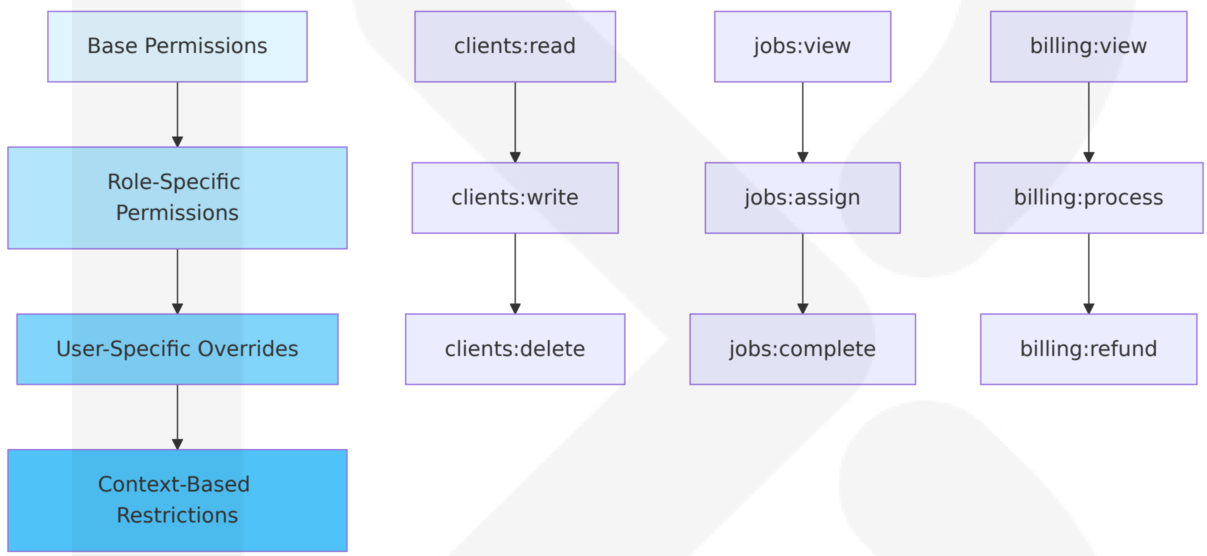
A more scalable authorization system will define a set of discrete permissions and assign those to roles. Authorization systems often define permissions as first-class concepts. Instead of checking whether a user is a member of a group, the policy can check whether a user has permission.

Permission Categories and Scopes

Permission Category	Scope	Example Permissions	Enforcement Level
Resource Access	Entity-level	clients:read, jobs:write	API endpoint

Permission Category	Scope	Example Permissions	Enforcement Level
Data Operations	Field-level	billing:view_amounts, wellness:export	Database query
System Functions	Action-level	payroll:approve, routes:optimize	Business logic
Administrative	Global	users:manage, settings:configure	Application-wide

Permission Inheritance Model



6.4.2.3 Resource Authorization

Multi-Tenant Resource Access Control

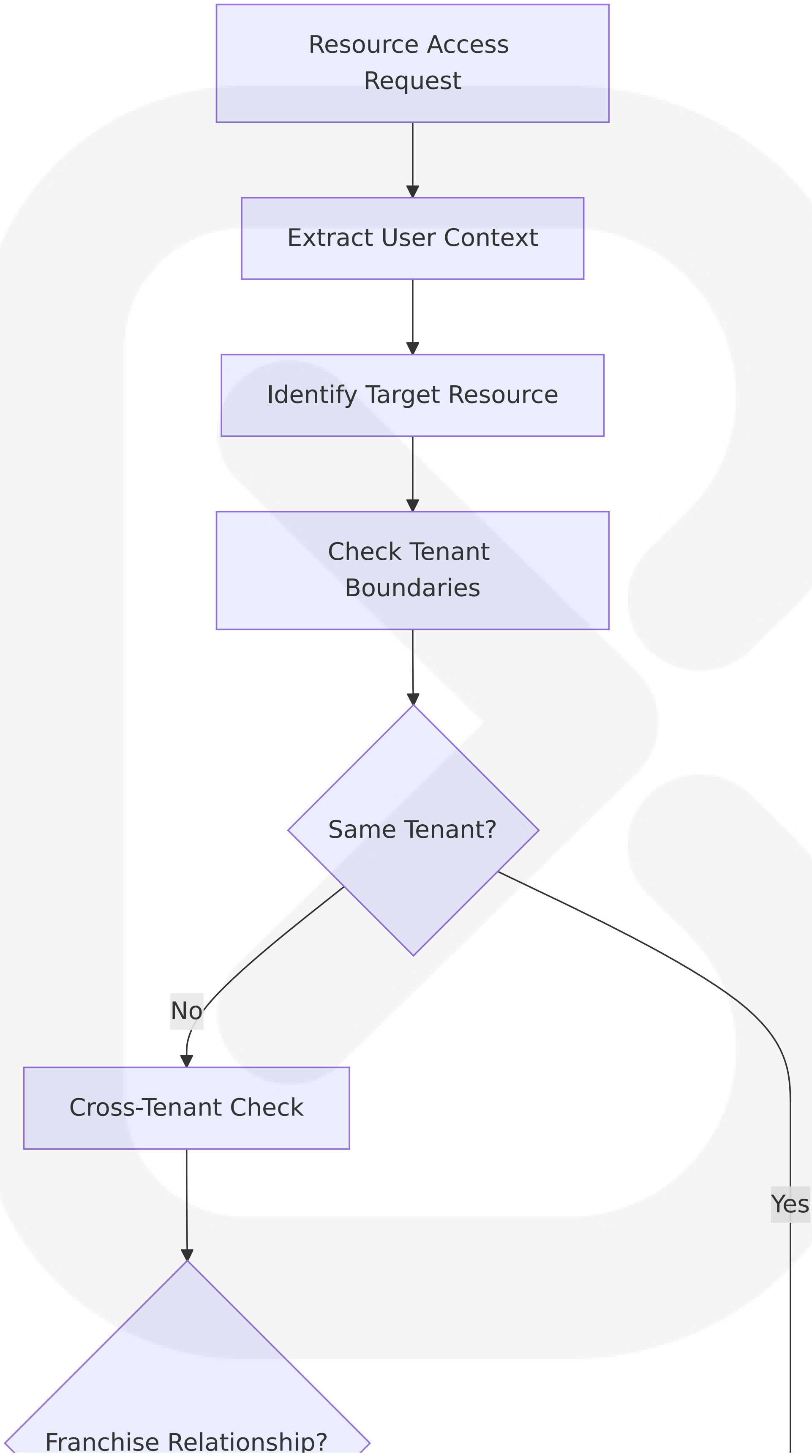
The system implements comprehensive resource authorization with tenant isolation and hierarchical access patterns supporting franchise operations.

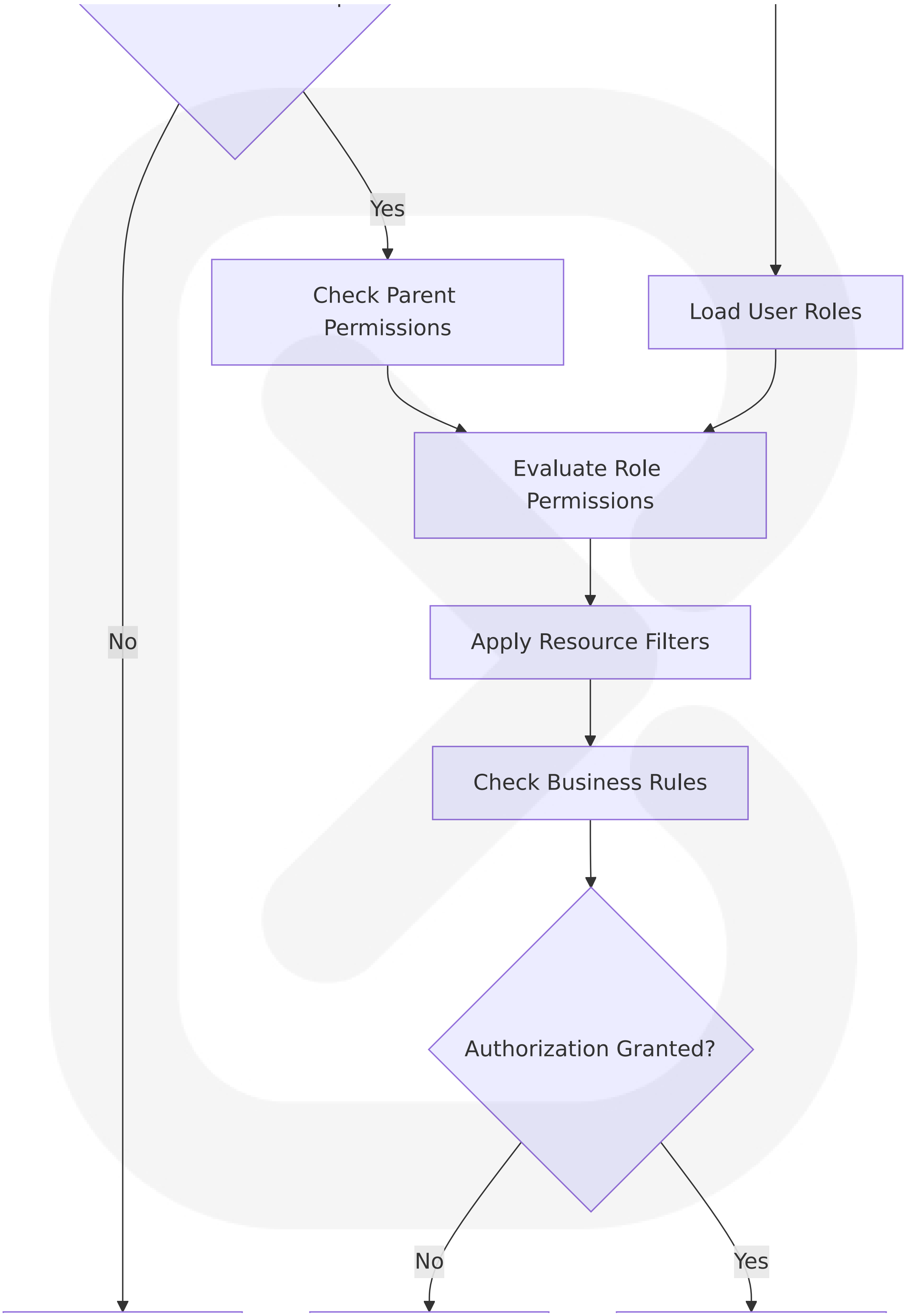
Resource Authorization Patterns

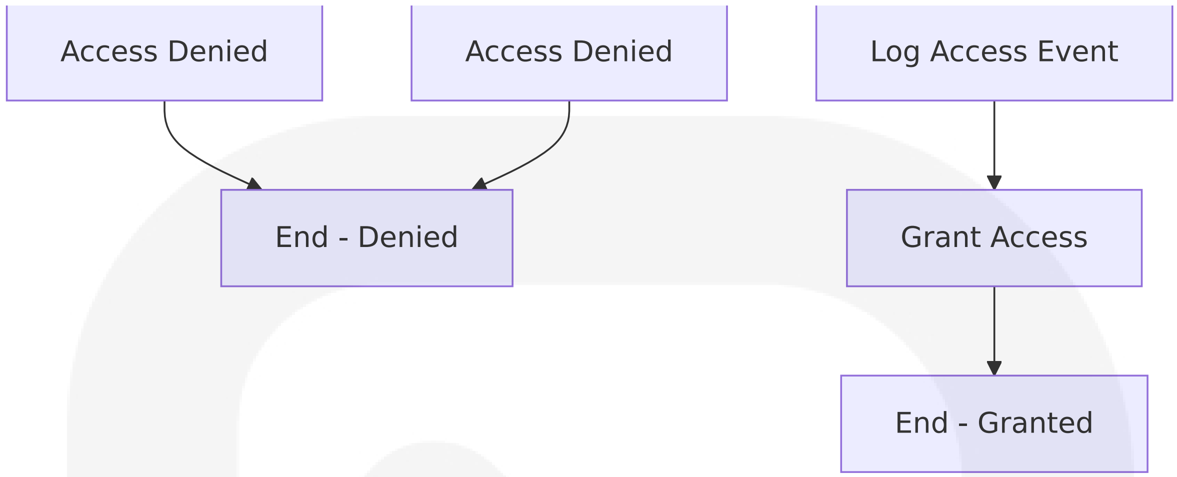
Resource Type	Access Pattern	Tenant Isolation	Inheritance Rules
Client Data	Owner + Staff	account_id filter	Manager inherits tech access

Resource Type	Access Pattern	Tenant Isolation	Inheritance Rules
Financial Records	Role-based	Strict isolation	Accountant + Manager only
Operational Data	Hierarchical	Shared within account	Dispatcher assigns to techs
System Configuration	Admin-only	Global settings	System admin override

Authorization Decision Flow


Resource Access Request
Extract User Context
Identify Target Resource
Check Tenant Boundaries
Same Tenant?
No
Yes
Cross-Tenant Check
Franchise Relationship?


Yes
No
Check Parent Permissions
Load User Roles
Evaluate Role Permissions
Apply Resource Filters
Check Business Rules
Authorization Granted?
No
Yes



6.4.2.4 Policy Enforcement Points

Distributed Authorization Architecture

Handle authentication through middleware. This ensures that user sessions are validated automatically on every request, without requiring manual checks in individual components. Consider using robust authentication providers like NextAuth.

Enforcement Point Strategy

Enforcement Point	Technology	Scope	Performance Impact
API Gateway	Next.js Middleware	All HTTP requests	< 10ms overhead
Route Guards	React components	Page-level access	Client-side only
Database Layer	Prisma middleware	Data access	< 5ms per query
Business Logic	Function decorators	Operation-level	Negligible

Middleware Authorization Implementation

```
// Example authorization middleware pattern
export async function middleware(request: NextRequest) {
```

```
  const session = await getServerSession(authOptions)

  if (!session) {
    return NextResponse.redirect('/auth/signin')
  }

  const resource = extractResourceFromPath(request.nextUrl.pathname)
  const hasPermission = await checkPermission(
    session.user.id,
    resource.type,
    resource.action,
    resource.id
  )

  if (!hasPermission) {
    return NextResponse.json({ error: 'Forbidden' }, { status: 403 })
  }

  return NextResponse.next()
}
```

6.4.2.5 Audit Logging

Comprehensive Audit Trail System

All authorization decisions and security events are logged for compliance, forensics, and security monitoring purposes.

Audit Event Categories

Event Category	Log Level	Retention Period	Alert Triggers
Authentication Events	INFO	2 years	Failed login patterns
Authorization Failures	WARN	7 years	Privilege escalation attempts
Data Access	INFO	5 years	Unusual access patterns

Event Category	Log Level	Retention Period	Alert Triggers
Administrative Actions	AUDIT	10 years	All admin operations

Audit Log Structure


```json
{
  "timestamp": "2025-09-10T12:00:00Z",
  "event_type": "authorization_check",
  "user_id": "user_123",
  "account_id": "account_456",
  "resource": {
    "type": "client",
    "id": "client_789",
    "action": "read"
  },
  "result": "granted",
  "ip_address": "192.168.1.100",
  "user_agent": "Mozilla/5.0...",
  "session_id": "sess_abc123",
  "request_id": "req_def456"
}
```


6.4.3 DATA PROTECTION

6.4.3.1 Encryption Standards

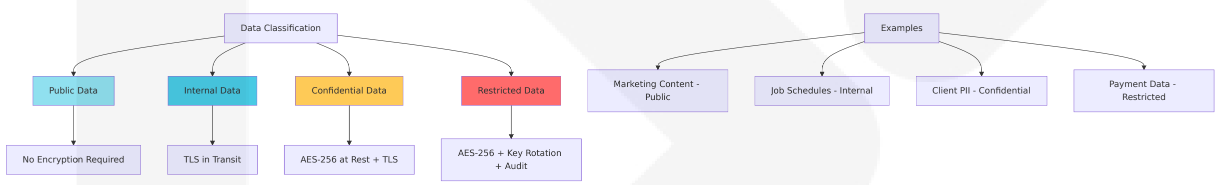
Comprehensive Data Encryption Strategy

The system implements multiple layers of encryption to protect sensitive data at rest, in transit, and during processing.

Encryption Implementation Matrix

Data Type	Encryption Method	Key Management	Compliance Standard
PII Data	AES-256-GCM	AWS KMS/HashiCorp Vault	GDPR, CCPA
Payment Data	Stripe Elements	Stripe-managed keys	PCI DSS Level 1
Database	PostgreSQL TDE	Database encryption keys	SOC 2 Type II
File Storage	S3 Server-Side	AWS managed keys	Industry standard
Session Data	AES-256-CBC	Application keys	Security best practice

Data Classification and Protection Levels



6.4.3.2 Key Management

Centralized Key Management Architecture

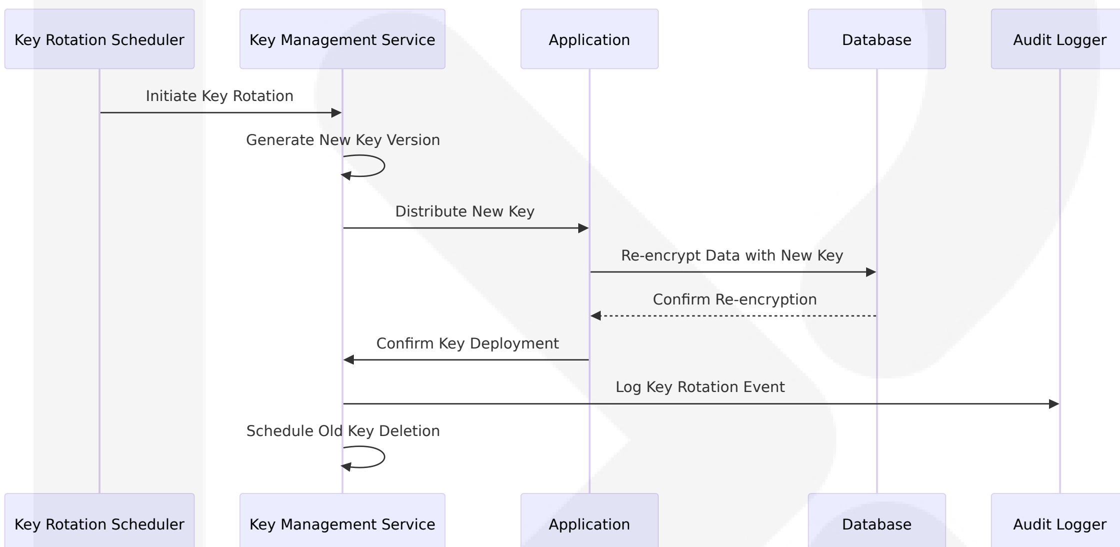
Implements enterprise-grade key management with automated rotation, secure distribution, and comprehensive audit trails.

Key Management Hierarchy

Key Type	Rotation Frequency	Storage Location	Access Control
Master Keys	Annual	Hardware Security Module	System admin only
Data Encryption Keys	Quarterly	Key Management Service	Automated rotation

Key Type	Rotation Frequency	Storage Location	Access Control
API Keys	Monthly	Encrypted database	Service accounts
Session Keys	Per session	Redis encrypted	Application runtime

Key Rotation Process



6.4.3.3 Data Masking Rules

Dynamic Data Masking Implementation

Protects sensitive data in non-production environments and provides controlled access to PII based on user roles and context.

Data Masking Policies

Data Type	Masking Rule	Environments	Role Exceptions
Email Addresses	user****@domain.com	Dev, Test	None
Phone Numbers	(***) ***-1234	Dev, Test	None

Data Type	Masking Rule	Environments	Role Exceptions
Credit Card Numbers	--****-1234	All	None (Stripe only)
SSN/Tax ID	*--1234	All	Compliance officer
Addresses	Street, ****, State	Dev, Test	None

Masking Implementation Strategy

- **Database Level**: PostgreSQL row-level security with masking functions
- **Application Level**: Middleware-based data transformation
- **API Level**: Response filtering based on user permissions
- **Export Level**: Automatic masking in data exports and reports

6.4.3.4 Secure Communication

End-to-End Communication Security

Always serve your application over HTTPS and set secure headers like Content-Security-Policy, Strict-Transport-Security, and X-Frame-Options.

Communication Security Standards

Communication Type	Protocol	Encryption	Authentication
Client-Server	HTTPS/TLS 1.3	AES-256-GCM	Certificate-based
API Integrations	HTTPS + HMAC	TLS 1.3	API key + signature
Database Connections	TLS 1.3	AES-256	Certificate + password
Internal Services	mTLS	AES-256-GCM	Mutual certificates

Security Headers Configuration

```
// Next.js security headers configuration
const securityHeaders = [
  {
    key: 'Strict-Transport-Security',
    value: 'max-age=31536000; includeSubDomains; preload'
  },
  {
    key: 'Content-Security-Policy',
    value: "default-src 'self'; script-src 'self' 'unsafe-inline' js.str
  },
  {
    key: 'X-Frame-Options',
    value: 'DENY'
  },
  {
    key: 'X-Content-Type-Options',
    value: 'nosniff'
  },
  {
    key: 'Referrer-Policy',
    value: 'strict-origin-when-cross-origin'
  }
]
```

6.4.3.5 Compliance Controls

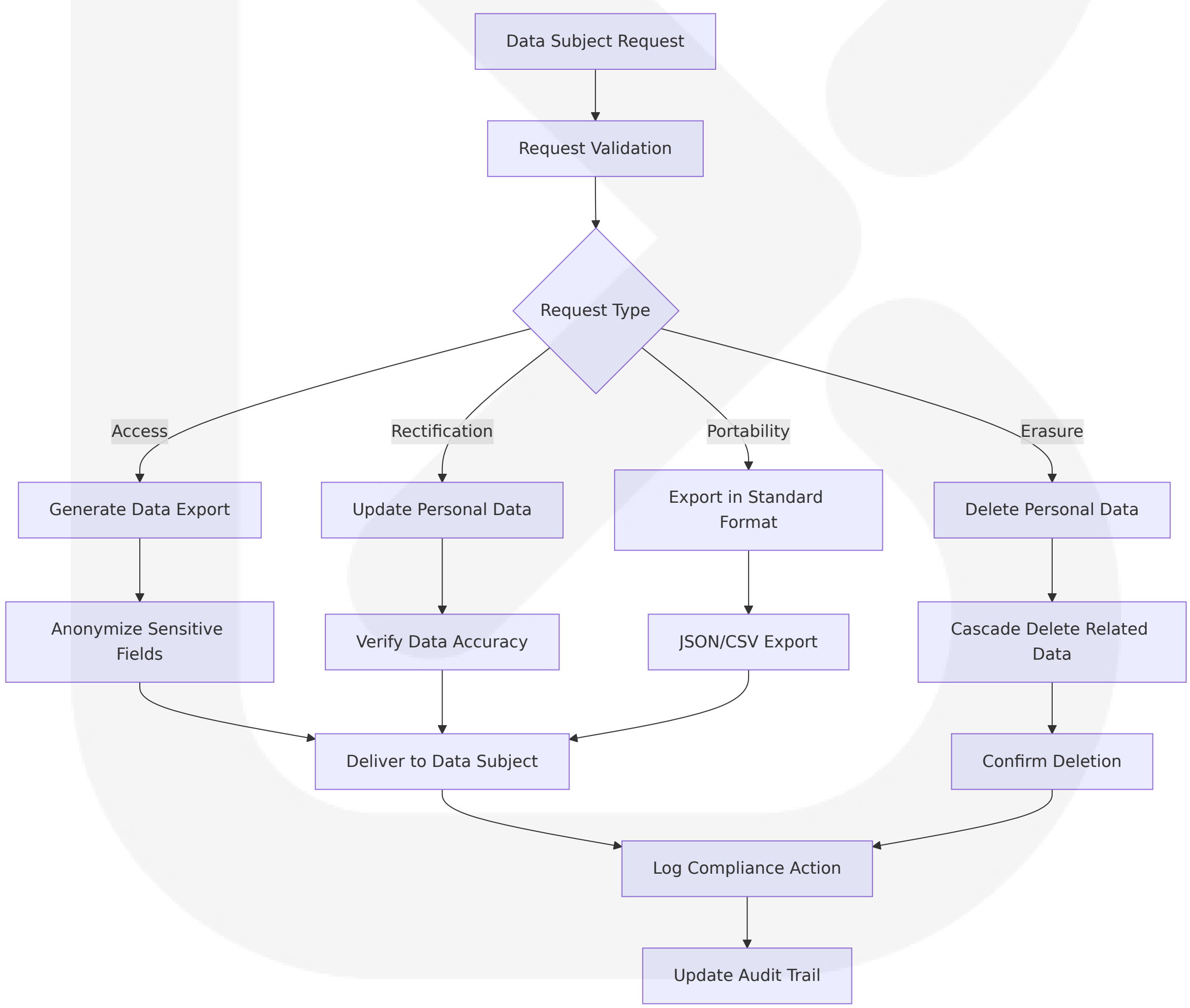
Regulatory Compliance Framework

The system implements comprehensive compliance controls to meet various regulatory requirements including GDPR, CCPA, PCI DSS, and SOC 2.

Compliance Control Matrix

Regulation	Scope	Key Controls	Audit Frequency
GDPR	EU personal data	Consent management, right to erasure	Annual
CCPA	California residents	Data inventory, opt-out mechanisms	Annual
PCI DSS	Payment card data	Stripe Elements, no card storage	Annual
SOC 2	System controls	Access controls, monitoring	Annual

Data Subject Rights Implementation



6.4.4 SECURITY MONITORING AND INCIDENT RESPONSE

6.4.4.1 Security Monitoring Architecture

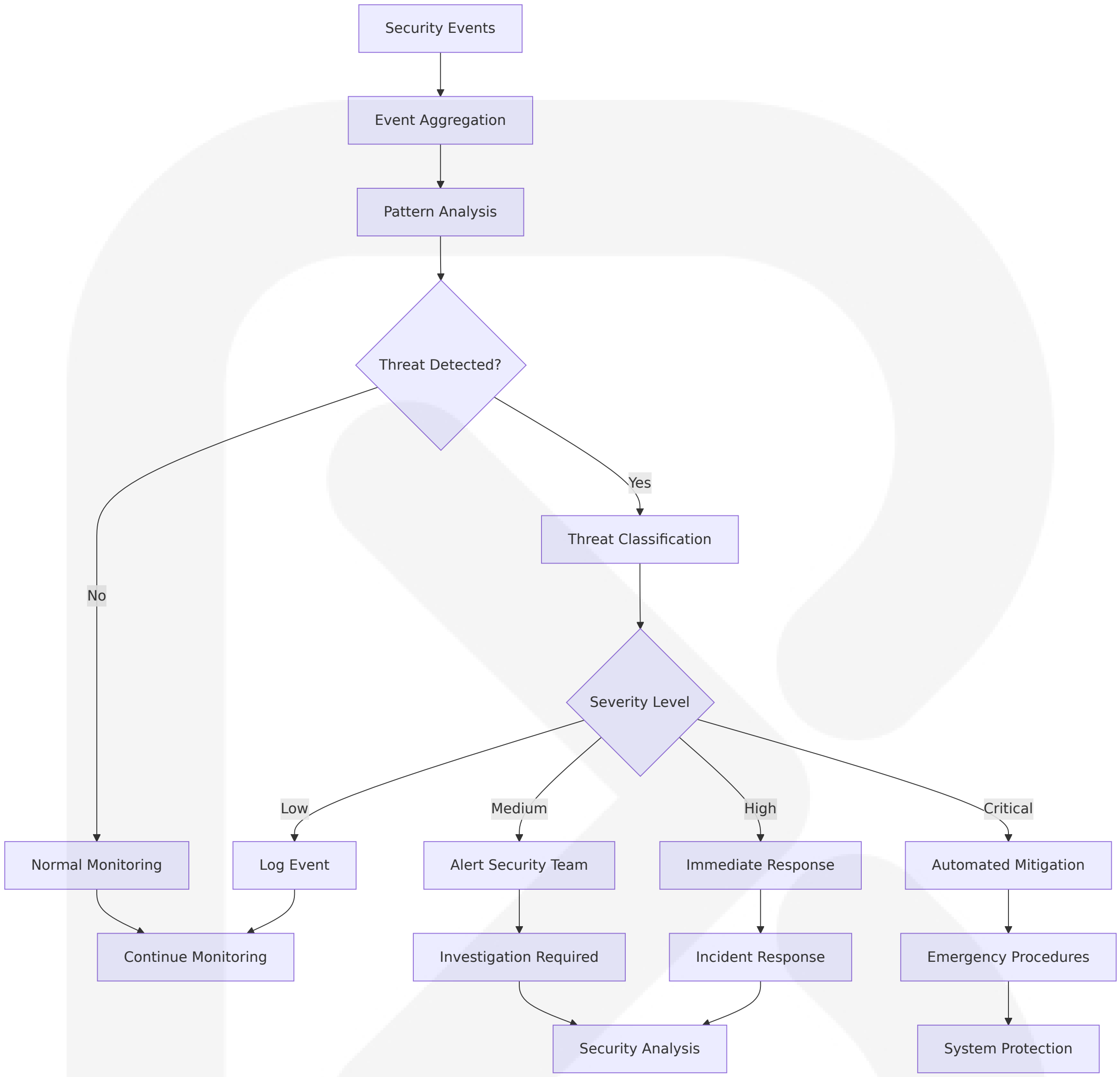
Real-Time Security Monitoring System

Implements comprehensive security monitoring with automated threat detection, incident response, and compliance reporting.

Monitoring Components

Component	Technology	Scope	Alert Thresholds
Application Security	Sentry + Custom	Code vulnerabilities, runtime errors	Critical: immediate
Infrastructure Security	CloudWatch/Grafana	Server metrics, network traffic	High: 5 minutes
Access Monitoring	Custom audit system	Authentication, authorization	Medium: 15 minutes
Data Protection	Database triggers	PII access, data exports	High: immediate

Security Event Correlation



6.4.4.2 Incident Response Procedures

Structured Incident Response Framework

Implements industry-standard incident response procedures with clear escalation paths and automated response capabilities.

Incident Response Phases

Phase	Duration	Key Activities	Responsible Team
Detection	0-15 minutes	Automated monitoring, alert generation	Security systems
Analysis	15-60 minutes	Threat assessment, impact evaluation	Security team
Containment	1-4 hours	Isolate threat, prevent spread	Engineering team
Recovery	4-24 hours	System restoration, service recovery	Operations team
Lessons Learned	1-7 days	Post-incident review, improvements	All teams

Automated Response Capabilities

- **Account Lockout**: Automatic suspension of compromised accounts
- **Rate Limiting**: Dynamic adjustment based on attack patterns
- **IP Blocking**: Temporary blocking of malicious IP addresses
- **Service Isolation**: Automatic isolation of affected system components

This comprehensive security architecture provides robust protection for the Yardura Service OS while maintaining operational efficiency and regulatory compliance. The multi-layered approach ensures defense in depth with appropriate controls for each security domain.

6.5 MONITORING AND OBSERVABILITY

6.5.1 MONITORING INFRASTRUCTURE

6.5.1.1 Metrics Collection Architecture

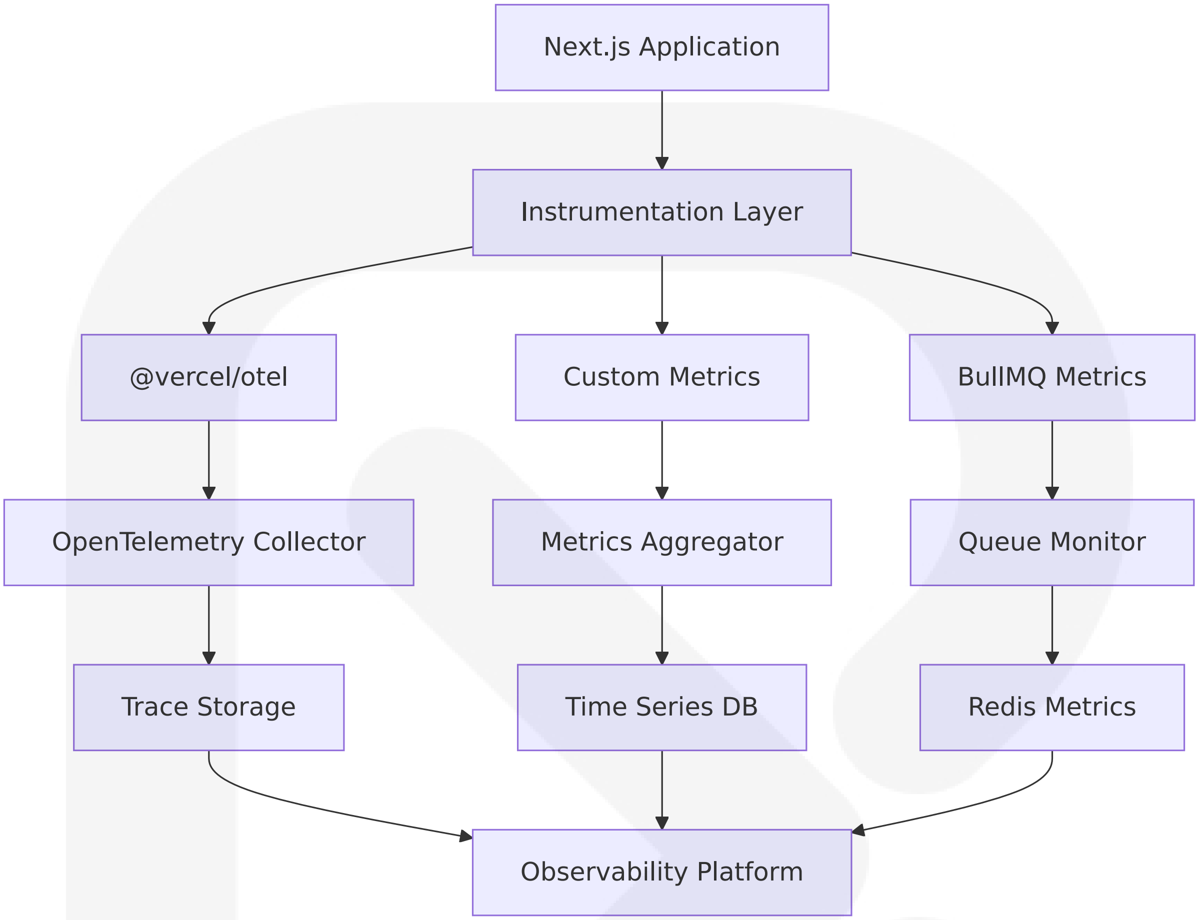
The Yardura Service OS implements a comprehensive monitoring infrastructure built on modern observability practices, leveraging Next.js 15's enhanced observability features and the stable instrumentation file with register() API that allows users to tap into the Next.js server lifecycle to monitor performance, track the source of errors, and deeply integrate with observability libraries like OpenTelemetry.

Core Metrics Collection Framework

Metric Category	Collection Method	Storage Backend	Retention Period
Application Performance	OpenTelemetry + @vercel/otel	Time-series database	90 days
Business Metrics	Custom collectors	PostgreSQL + Redis	2 years
Infrastructure Metrics	System monitoring	Prometheus/Grafana	30 days
Queue Metrics	BullMQ built-in metrics	Redis + time-series	30 days

OpenTelemetry Integration

We recommend using OpenTelemetry for instrumenting your apps. It's a platform-agnostic way to instrument apps that allows you to change your observability provider without changing your code. The system utilizes @vercel/otel package that helps you get started quickly with registerOTel({ serviceName: 'next-app' }).



BullMQ Queue Metrics Collection

BullMQ provides built-in metrics through the getMetrics method on the Queue class, allowing you to gather metrics for completed or failed jobs with data arrays representing job completion counts per minute.

Queue Metrics Configuration

Queue Name	Metrics Collected	Collection Frequency	Alert Thresholds
billing-queue	Completed, failed, processing time	Every minute	>5% failure rate
notification-queue	Delivery rates, retry counts	Every minute	>10% failure rate
wellness-queue	Processing time, batch sizes	Every 5 minutes	>30s processing time

Queue Name	Metrics Collected	Collection Frequency	Alert Thresholds
integration-queue	API call success, retry patterns	Every minute	>15% failure rate

6.5.1.2 Log Aggregation System

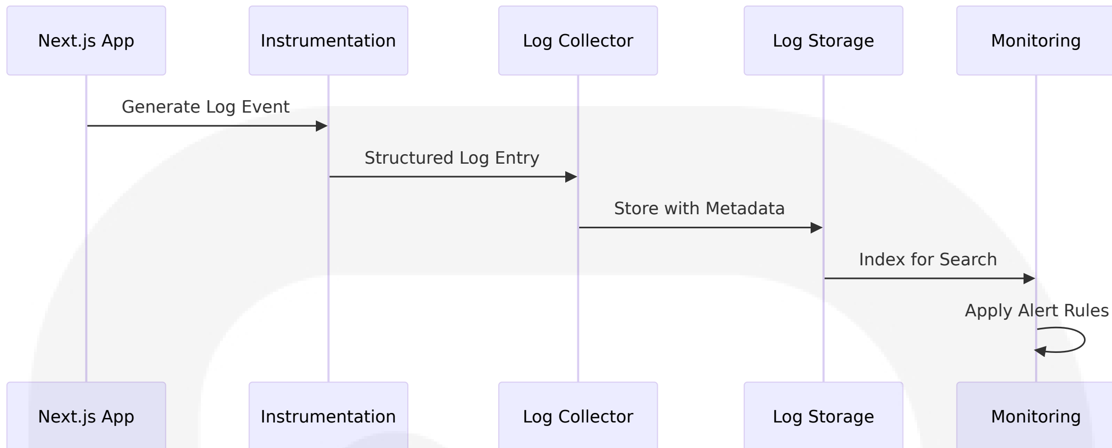
Structured Logging Architecture

Next.js 15 enables logging, analytics, and other external system synchronization tasks that are not directly related to the response, ensuring users don't have to wait for them to complete.

Log Collection Strategy

Log Level	Sources	Format	Retention
ERROR	Application errors, API failures	Structured JSON	2 years
WARN	Performance issues, rate limits	Structured JSON	1 year
INFO	Business events, user actions	Structured JSON	6 months
DEBUG	Development diagnostics	Structured JSON	30 days

Log Aggregation Flow



6.5.1.3 Distributed Tracing Implementation

OpenTelemetry Tracing Architecture

Next.js supports OpenTelemetry instrumentation out of the box, which means that we already instrumented Next.js itself. OpenTelemetry is extensible but setting it up properly can be quite verbose.

Trace Collection Points

Component	Trace Scope	Sampling Rate	Key Spans
API Routes	All HTTP requests	100%	Request/response, database queries
Server Actions	Form submissions	100%	Validation, processing, storage
Background Jobs	BullMQ processing	50%	Job lifecycle, external API calls
Database Operations	Prisma queries	25%	Query execution, connection pooling

Custom Instrumentation Implementation

```
// instrumentation.ts
import { registerOTel } from '@vercel/otel'
```

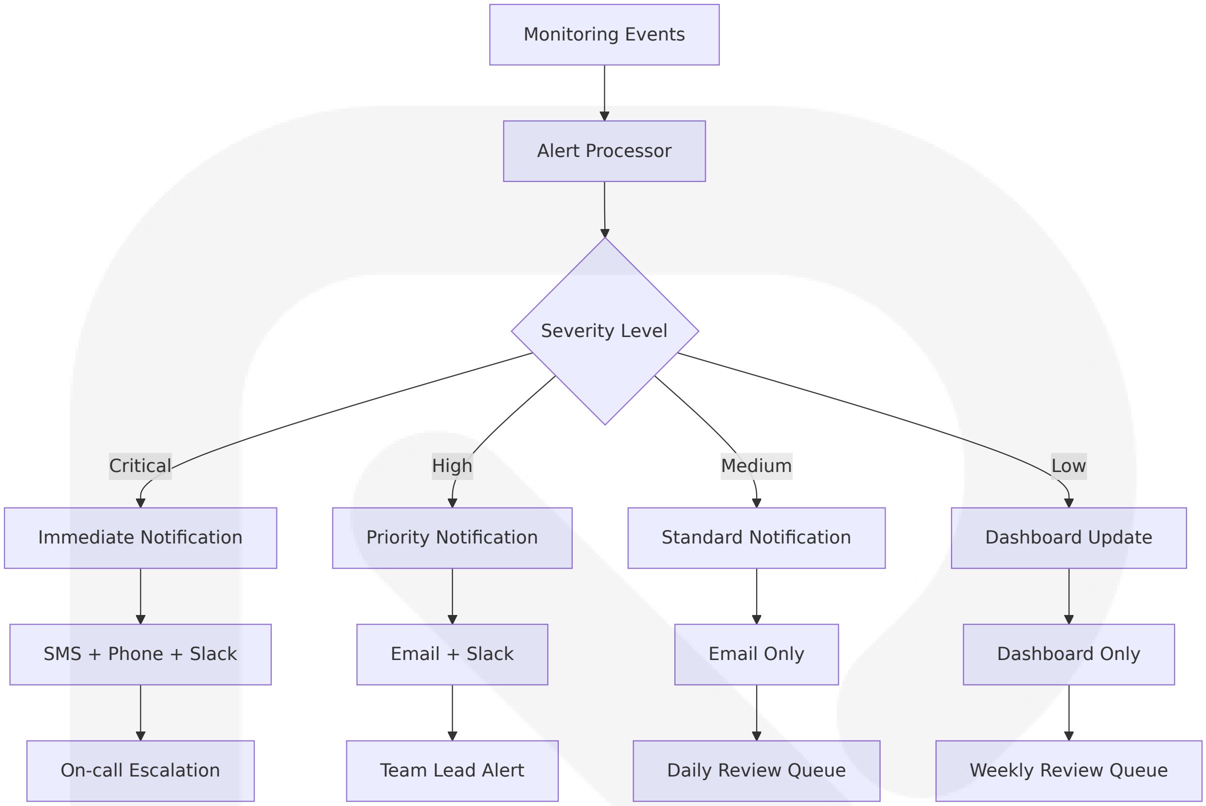
```
export function register() {
  registerOTel({
    serviceName: 'yardura-service-os',
    tracesSampleRate: 1.0,
  })
}

export const onRequestError = async (err, request, context) => {
  // Custom error reporting with full context
  await fetch('https://monitoring-endpoint', {
    method: 'POST',
    body: JSON.stringify({
      message: err.message,
      request: {
        url: request.url,
        method: request.method,
        headers: Object.fromEntries(request.headers.entries())
      },
      context
    }),
    headers: { 'Content-Type': 'application/json' }
  })
}
```

6.5.1.4 Alert Management System

Multi-Channel Alert Routing

Alert Severity	Notification Channels	Response Time SLA	Escalation Path
Critical	SMS + Phone + Slack	< 5 minutes	On-call engineer
High	Email + Slack	< 15 minutes	Team lead
Medium	Email	< 1 hour	Daily review
Low	Dashboard only	< 24 hours	Weekly review

Alert Configuration Matrix

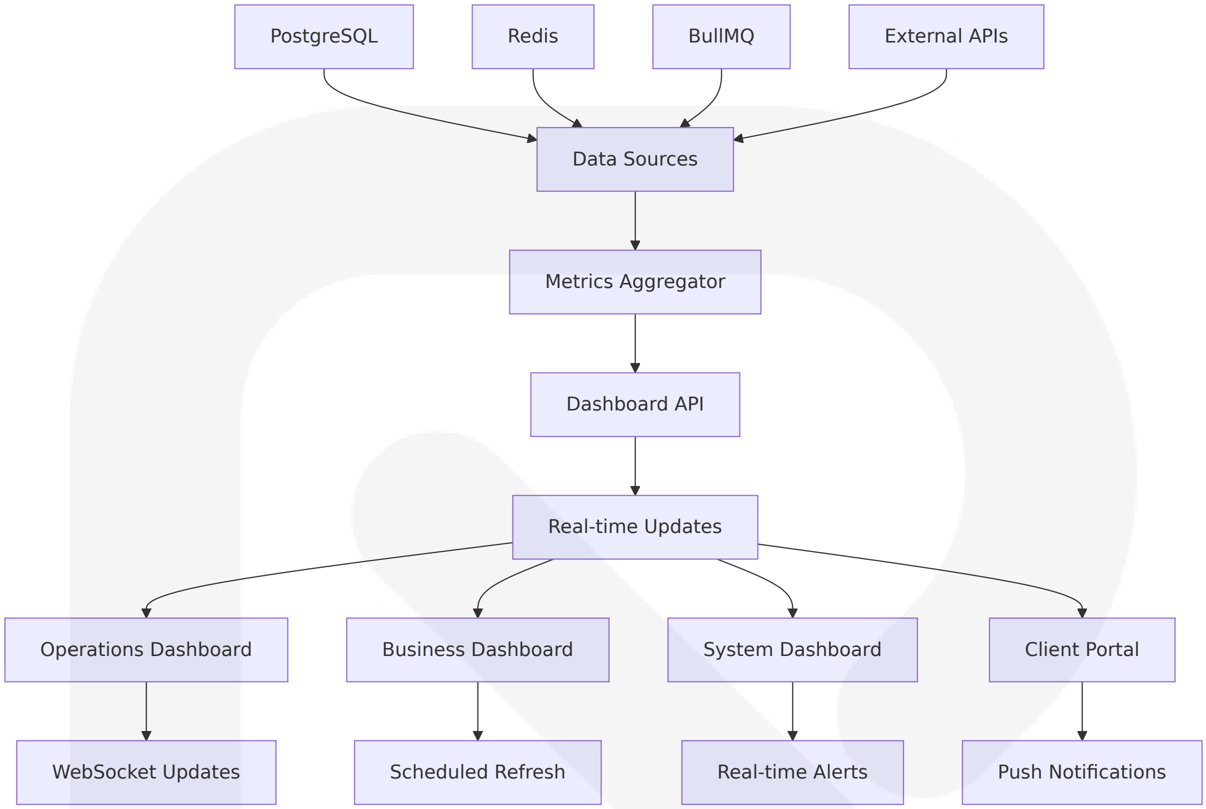


6.5.1.5 Dashboard Design Architecture

Real-Time Dashboard Components

Dashboard Type	Update Frequency	Data Sources	User Roles
Operations Dashboard	Real-time	Jobs, routes, technicians	Dispatchers, managers
Business Metrics	5 minutes	Revenue, clients, completion rates	Owners, managers
System Health	30 seconds	Infrastructure, queues, errors	Engineering team
Client Portal	Real-time	Service status, wellness data	Clients

Dashboard Architecture



6.5.2 OBSERVABILITY PATTERNS

6.5.2.1 Health Check Implementation

Comprehensive Health Check Framework

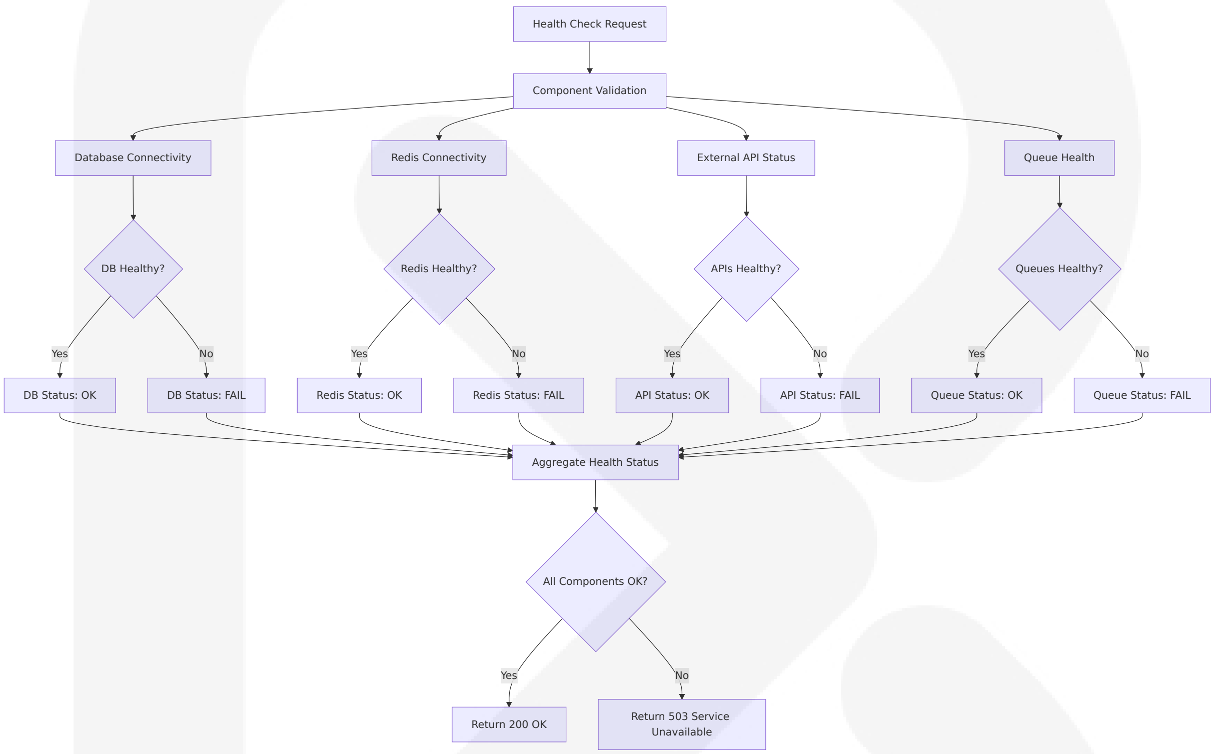
The system implements multi-layered health checks covering all critical components with automated recovery mechanisms and detailed status reporting.

Health Check Categories

Check Type	Endpoint	Frequency	Timeout	Dependencies
Liveness	`/api/health/live`	30 seconds	5 seconds	Basic app functionality
Readiness	`/api/health/ready`	60 seconds	10 seconds	Database, Redis, external APIs

Check Type	Endpoint	Frequency	Timeout	Dependencies
Deep Health	`/api/health/deep`	5 minutes	30 seconds	All integrations, queue health

Health Check Flow



6.5.2.2 Performance Metrics Tracking

Core Web Vitals Monitoring

Core Web Vitals are three metrics (LCP, INP, and CLS) that measure the performance of a web application. Largest Contentful Paint (LCP) measures how long it takes the largest content on a page to load.

Performance Metrics Configuration

Metric	Target Value	Measurement Method	Alert Threshold
Largest Contentful Paint (LCP)	< 2.5s	Real User Monitoring	> 4s
Interaction to Next Paint (INP)	< 200ms	Event timing	> 500ms
Cumulative Layout Shift (CLS)	< 0.1	Layout stability	> 0.25
Time to First Byte (TTFB)	< 600ms	Server response	> 1s

Performance Monitoring Implementation

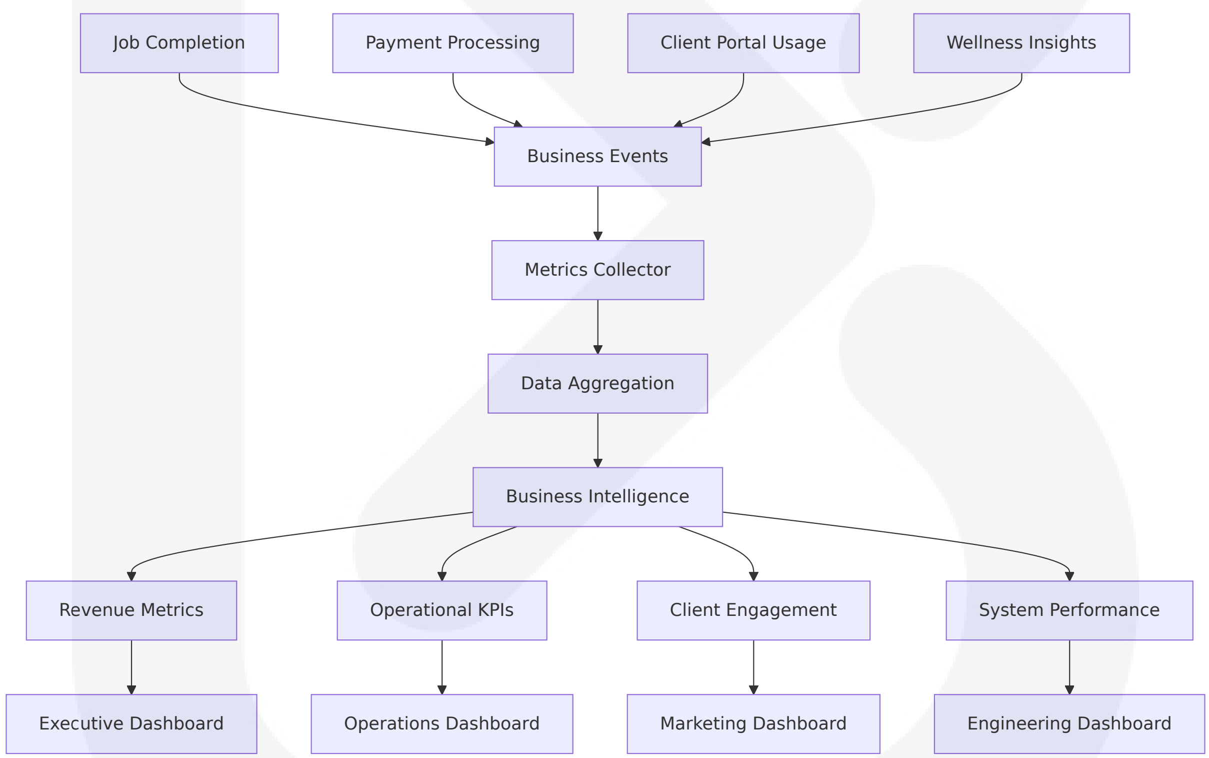
```
// Performance monitoring with Next.js Web Vitals
import { useReportWebVitals } from 'next/web-vitals'

function MyApp({ Component, pageProps }) {
  useReportWebVitals((metric) => {
    // Send metrics to monitoring service
    fetch('/api/analytics/web-vitals', {
      method: 'POST',
      body: JSON.stringify(metric),
      headers: { 'Content-Type': 'application/json' }
    })
  })

  return <Component {...pageProps} />
}
```

6.5.2.3 Business Metrics Collection

Key Business Indicators

Metric Category	Key Metrics	Collection Method	Business Impact
Operational Efficiency	Route completion rate, average service time	Real-time tracking	Cost optimization
Financial Performance	Revenue per client, collection rates	Billing system integration	Revenue growth
Client Satisfaction	Portal usage, wellness engagement	User analytics	Retention rates
System Adoption	Feature utilization, mobile app usage	Application telemetry	Product development

Business Metrics Dashboard

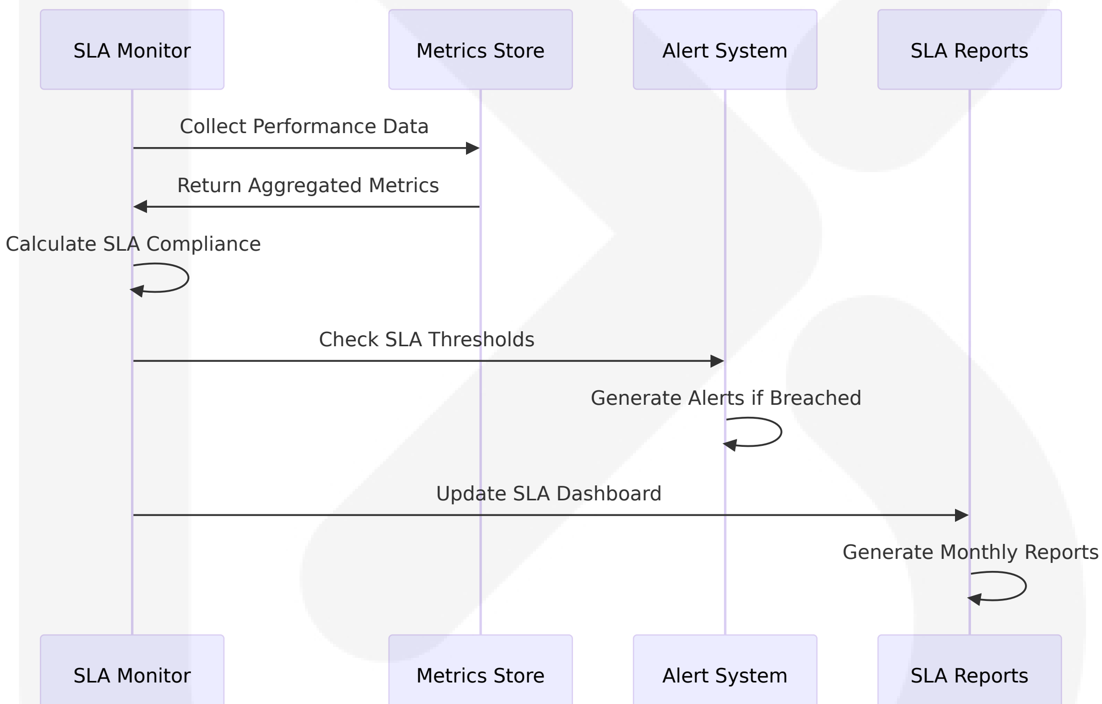


6.5.2.4 SLA Monitoring Framework

Service Level Agreement Targets

Service Component	Availability SLA	Performance SLA	Error Rate SLA
Client Portal	99.9% uptime	P95 < 500ms	< 0.1% error rate
Staff Operations	99.9% uptime	P95 < 1000ms	< 0.5% error rate
Field Tech PWA	99.5% uptime	< 200ms job list	< 1% error rate
API Endpoints	99.9% uptime	P95 < 2000ms	< 0.5% error rate

SLA Monitoring Implementation

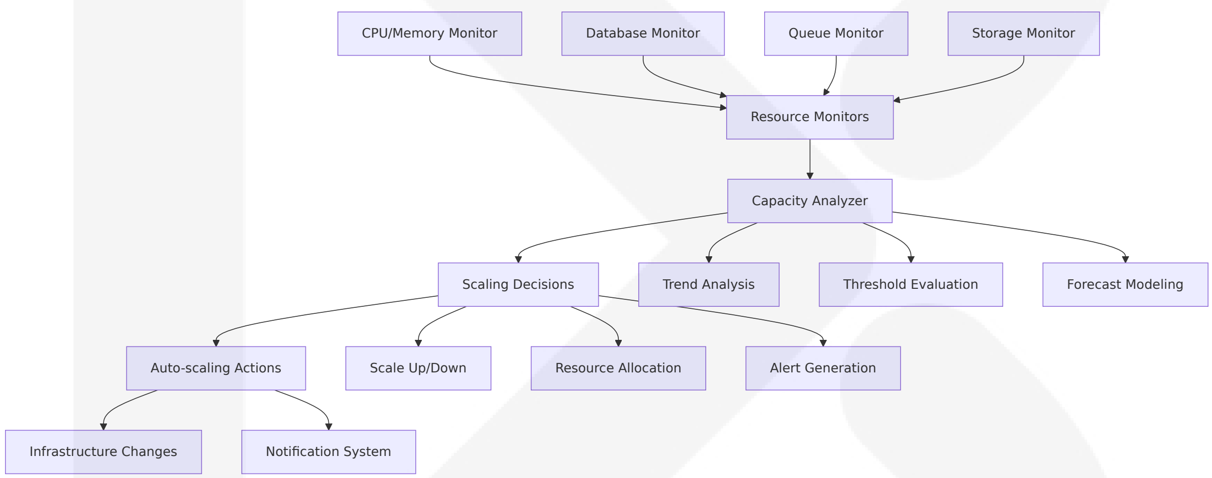


6.5.2.5 Capacity Tracking System

Resource Utilization Monitoring

Resource Type	Monitoring Metrics	Scaling Triggers	Capacity Planning
Application Servers	CPU, Memory, Connections	> 70% for 5 minutes	Horizontal scaling
Database	Connection pool, Query performance	> 80% connections	Read replica scaling
Queue Workers	Job processing rate, Queue depth	> 1000 pending jobs	Worker pool expansion
Storage	Disk usage, I/O performance	> 85% capacity	Storage expansion

Capacity Monitoring Architecture



6.5.3 INCIDENT RESPONSE

6.5.3.1 Alert Routing Configuration

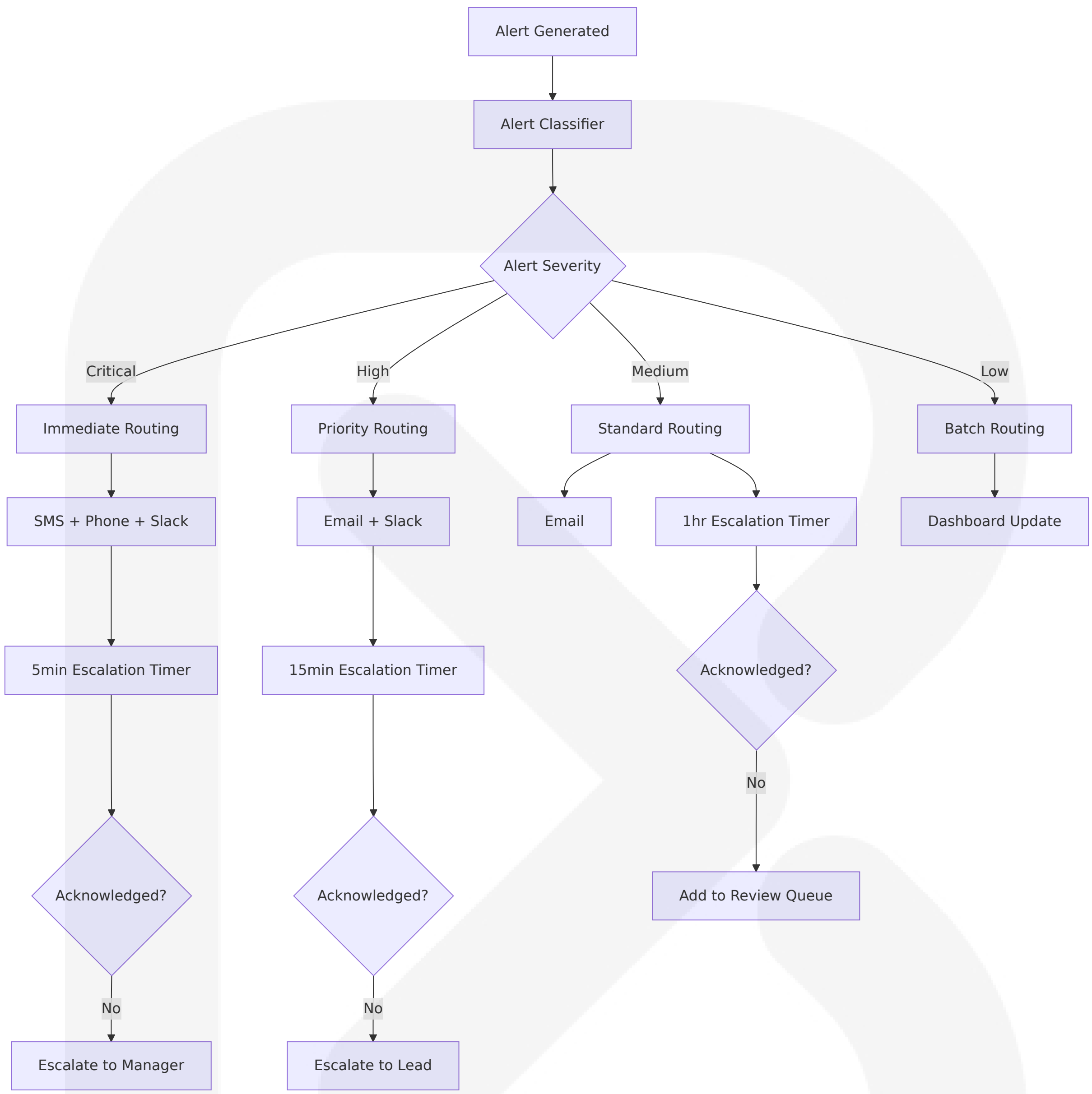
Intelligent Alert Routing System

The system implements sophisticated alert routing based on severity, component, and business impact with automatic escalation and de-duplication capabilities.

Alert Routing Matrix

Alert Type	Primary Route	Secondary Route	Escalation Time
System Down	On-call engineer (SMS/Phone)	Engineering manager	5 minutes
Payment Failures	Finance team (Email/Slack)	Business owner	15 minutes
Queue Backlog	Operations team (Slack)	Engineering team	30 minutes
Performance Degradation	Engineering team (Email)	Team lead	1 hour

Alert Routing Flow



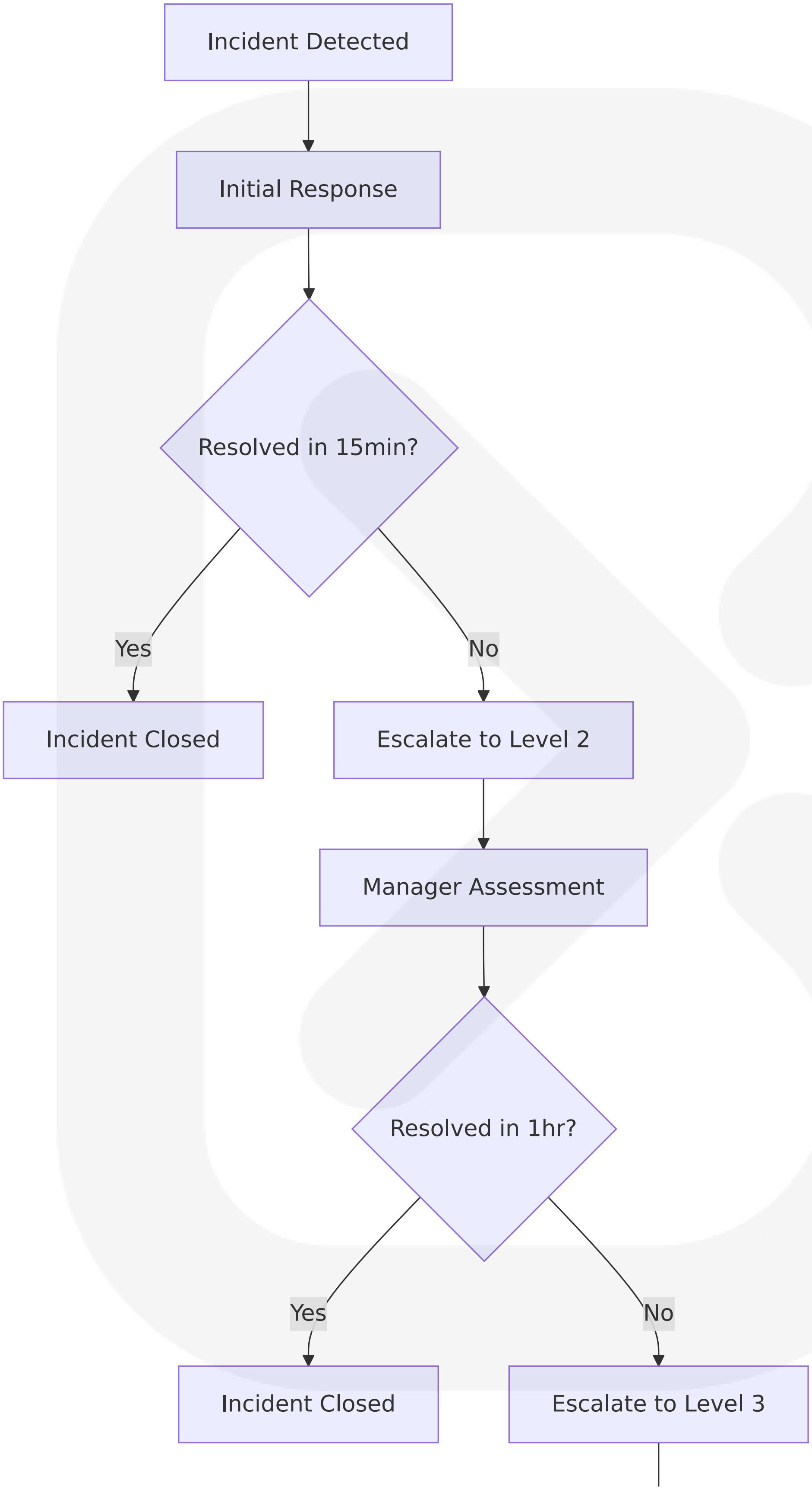
6.5.3.2 Escalation Procedures

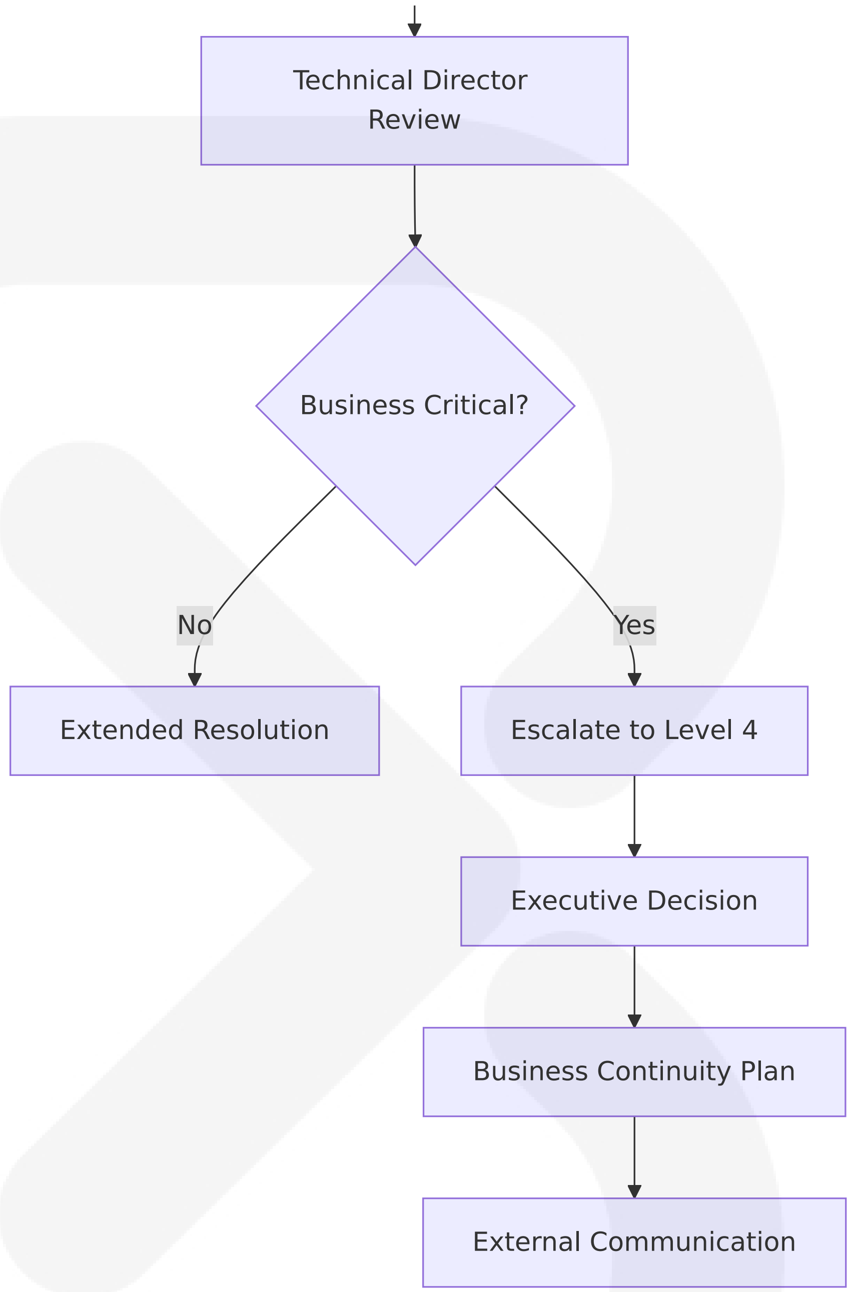
Tiered Escalation Framework

Escalation Level	Response Time	Personnel	Authority Level
Level 1	0-15 minutes	On-call engineer	System restart, service isolation

Escalation Level	Response Time	Personnel	Authority Level
Level 2	15-60 minutes	Engineering manager	Resource scaling, vendor contact
Level 3	1-4 hours	Technical director	Architecture changes, emergency procedures
Level 4	4+ hours	Executive team	Business continuity, external communication

Escalation Decision Tree


Incident Detected
Initial Response
Resolved in 15min?
Yes
No
Incident Closed
Escalate to Level 2
Manager Assessment
Resolved in 1hr?
Yes
No
Incident Closed
Escalate to Level 3



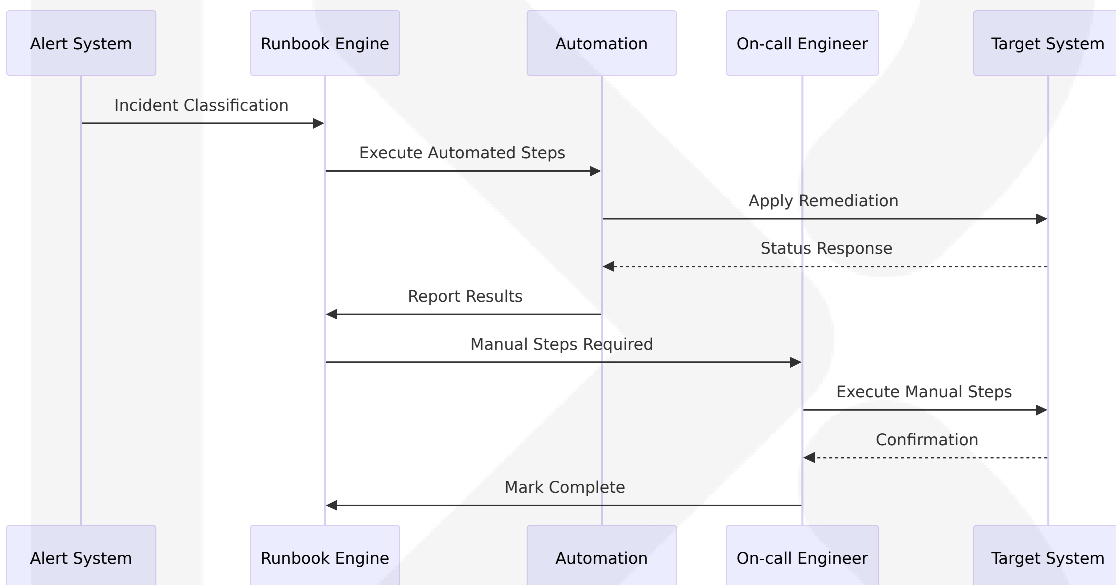
6.5.3.3 Runbook Management

Automated Runbook System

Incident Type	Runbook Location	Automation Level	Success Rate
Database Connection Issues	`/runbooks/database-recovery.md`	80% automated	95%

Incident Type	Runbook Location	Automation Level	Success Rate
Queue Processing Failures	`/runbooks/queue-recovery.md`	60% automated	90%
External API Outages	`/runbooks/api-fallback.md`	40% automated	85%
Performance Degradation	`/runbooks/performance-tuning.md`	30% automated	80%

Runbook Execution Flow



6.5.3.4 Post-Mortem Process

Structured Post-Mortem Framework

Incident Severity	Post-Mortem Required	Timeline	Participants
Critical (Service Down)	Mandatory	Within 48 hours	All stakeholders
High (Feature Impacted)	Mandatory	Within 1 week	Engineering + Product

Incident Severity	Post-Mortem Required	Timeline	Participants
Medium (Performance)	Optional	Within 2 weeks	Engineering team
Low (Minor Issues)	Optional	Monthly review	Team lead

Post-Mortem Template Structure

Incident Post-Mortem: [Incident Title]

Executive Summary

- **Incident Date**: [Date/Time]
- **Duration**: [Total downtime]
- **Impact**: [User/business impact]
- **Root Cause**: [Primary cause]

Timeline

- [Time] - Initial detection
- [Time] - Response initiated
- [Time] - Root cause identified
- [Time] - Resolution implemented
- [Time] - Service restored

Root Cause Analysis

- **What happened**: [Detailed description]
- **Why it happened**: [Contributing factors]
- **How it was detected**: [Monitoring/alerts]

Action Items

- [ ] **Immediate**: [Short-term fixes]
- [ ] **Short-term**: [1-2 week improvements]

- [ ] **Long-term**: [Architectural changes]

Lessons Learned

- **What went well**: [Positive aspects]
- **What could improve**: [Areas for enhancement]
- **Prevention measures**: [Future safeguards]

```
#### 6.5.3.5 Improvement Tracking

**Continuous Improvement Metrics**

| Improvement Area | Tracking Metric | Target | Current Status |
|------------------|-----------------|--------|----------------|
| Mean Time to Detection (MTTD) | Alert to acknowledgment | < 5 minutes
| Mean Time to Resolution (MTTR) | Detection to fix | < 30 minutes | 22
| Incident Recurrence Rate | Same issue repeat | < 5% | 3.1% |
| Runbook Effectiveness | Automated resolution | > 70% | 68% |

**Improvement Tracking Dashboard**

<div class="mermaid-wrapper" id="mermaid-diagram-bgzy2iytv">
          <div class="mermaid">
graph TB
    A[Incident Data] --&gt; B[Metrics Calculator]
    B --&gt; C[Trend Analysis]
    C --&gt; D[Improvement Opportunities]

    E[MTTD Tracking] --&gt; A
    F[MTTR Tracking] --&gt; A
    G[Recurrence Tracking] --&gt; A
    H[Resolution Success] --&gt; A

    D --&gt; I[Process Improvements]
    D --&gt; J[Tool Enhancements]
    D --&gt; K[Training Needs]
    D --&gt; L[Infrastructure Changes]

    I --&gt; M[Updated Procedures]
    J --&gt; N[Better Monitoring]
    K --&gt; O[Team Training]
```

```
    L --&gt; P[System Upgrades]
</div>
        </div>

### 6.5.4 ERROR TRACKING AND PERFORMANCE MONITORING

#### 6.5.4.1 Sentry Integration Architecture

**Comprehensive Error Tracking with Sentry**

Sentry's core error monitoring product automatically reports errors, unca

**Sentry Configuration for Next.js 15**

Turbopack in dev-mode (next dev --turbopack) is fully supported for Next

**Error Tracking Configuration**

| Environment | Configuration | Features Enabled | Sampling Rate |
|-------------|---------------|------------------|---------------|
| Client-side | Session replay, breadcrumbs | Error tracking, performance
| Server-side | Request context, database traces | Error tracking, tracin
| Edge Runtime | Lightweight monitoring | Error tracking only | 100% erro

**Sentry Implementation**
```

typescript
// sentry.client.config.ts
import * as Sentry from "@sentry/nextjs"

Sentry.init({
dsn: process.env.NEXT_PUBLIC_SENTRY_DSN,
integrations: [
Sentry.replayIntegration(),
],
tracesSampleRate: 1,
replaysSessionSampleRate: 0.1,
replaysOnErrorSampleRate: 1.0,

debug: process.env.NODE_ENV === "development",
})

// instrumentation.ts
import * as Sentry from '@sentry/nextjs'

export async function register() {
if (process.env.NEXT_RUNTIME === 'nodejs') {
await import('./sentry.server.config')
}
if (process.env.NEXT_RUNTIME === 'edge') {
await import('./sentry.edge.config')
}
}

export const onRequestError = Sentry.captureRequestError

```
#### 6.5.4.2 Performance Monitoring Implementation

**Real-Time Performance Tracking**

Monitor response times and error rates across your Next.js API routes wi

**Performance Monitoring Scope**

| Component | Monitoring Type | Key Metrics | Alert Conditions |
|-----------|----------------|-------------|------------------|
| API Routes | Automatic instrumentation | Response time, error rate | P
| Server Actions | Custom instrumentation | Processing time, success rate
| Database Queries | Prisma integration | Query time, connection pool | >
| External APIs | HTTP client tracing | Response time, failure rate | > 

**Performance Monitoring Architecture**

<div class="mermaid-wrapper" id="mermaid-diagram-9bclxqj1v">
          <div class="mermaid">
graph TB
    A[Next.js Application] --&gt; B[Sentry Performance]
    B --&gt; C[Automatic Instrumentation]
```

```
    B --&gt; D[Custom Instrumentation]

    C --&gt; E[API Route Monitoring]
    C --&gt; F[Page Load Monitoring]
    C --&gt; G[Database Query Tracking]

    D --&gt; H[Business Logic Timing]
    D --&gt; I[External API Monitoring]
    D --&gt; J[Queue Processing Metrics]

    E --&gt; K[Performance Dashboard]
    F --&gt; K
    G --&gt; K
    H --&gt; K
    I --&gt; K
    J --&gt; K

    K --&gt; L[Alert Generation]
    K --&gt; M[Trend Analysis]
    K --&gt; N[Optimization Insights]
</div>
        </div>

#### 6.5.4.3 Session Replay and User Experience Monitoring

**Comprehensive User Experience Tracking**

Session Replay gets to the root cause of an issue faster by viewing a vi

**Session Replay Configuration**

| User Type | Replay Rate | Privacy Settings | Retention Period |
|-----------|-------------|------------------|------------------|
| Staff Users | 50% of sessions | Mask PII fields | 30 days |
| Client Users | 10% of sessions | Mask all sensitive data | 14 days |
| Error Sessions | 100% of error sessions | Full context capture | 90 da
| Performance Issues | 25% of slow sessions | Standard masking | 30 days

#### 6.5.4.4 BullMQ Queue Monitoring

**Queue Health and Performance Monitoring**

Key indicators include pending jobs, active workers, completed tasks, fa
```

```
**Queue Monitoring Implementation**
```

typescript
// Queue health monitoring
import { Queue } from 'bullmq'
import { redis } from './redis-client'

class QueueHealthMonitor {
constructor(queueName: string) {
this.queue = new Queue(queueName)
this.STATS_KEY = `bull:${queueName}:health`

```
// Attach event listeners to capture metrics
this.queue.on('completed', () =>
  redis.hincrby(this.STATS_KEY, 'completed', 1)
)
this.queue.on('failed', () =>
  redis.hincrby(this.STATS_KEY, 'failed', 1)
)
```

}

async getHealthStatus() {
const stats = await redis.hgetall(this.STATS_KEY)
return {
successRate: (stats.completed / (stats.completed + stats.failed)).toFixed(2),
pending: await this.queue.getWaitingCount(),
activeWorkers: await this.queue.getActiveCount()
}
}
}

```
**Queue Metrics Dashboard**
```

```
| Queue | Success Rate | Pending Jobs | Active Workers | Avg Processing T
|-------|--------------|--------------|----------------|-----------------
| billing-queue | 98.5% | 12 | 3/5 | 2.3s |
| notification-queue | 96.2% | 45 | 8/10 | 0.8s |
| wellness-queue | 99.1% | 3 | 1/3 | 15.2s |
| integration-queue | 94.8% | 8 | 2/5 | 5.1s |

#### 6.5.4.5 Custom Business Metrics Tracking

**Business-Specific Monitoring**

The system implements custom metrics tracking for business-critical opera

**Business Metrics Collection**

<div class="mermaid-wrapper" id="mermaid-diagram-tmti5g2d5">
          <div class="mermaid">
sequenceDiagram
    participant Business as Business Logic
    participant Metrics as Metrics Collector
    participant Storage as Time Series DB
    participant Dashboard as Business Dashboard
    participant Alerts as Alert System

    Business-&gt;&gt;Metrics: Emit Business Event
    Metrics-&gt;&gt;Storage: Store Metric Data
    Storage-&gt;&gt;Dashboard: Update Real-time Display
    Dashboard-&gt;&gt;Alerts: Check Business Rules
    Alerts-&gt;&gt;Alerts: Generate Business Alerts
</div>
        </div>

**Key Business Metrics**

| Metric | Collection Method | Business Impact | Alert Threshold |
|--------|------------------|-----------------|-----------------|
| Route Optimization Time | Custom timing | Operational efficiency | > 1(
| Wellness Analysis Accuracy | ML model metrics | Client satisfaction | <
| Payment Success Rate | Stripe webhook tracking | Revenue impact | < 95%
| Client Portal Engagement | User analytics | Retention rates | < 60% mor

This comprehensive monitoring and observability architecture ensures the
```

```
## 6.6 TESTING STRATEGY

### 6.6.1 TESTING APPROACH

#### 6.6.1.1 Unit Testing Framework

**Testing Framework Selection**

Next.js 15 supports Jest for unit testing and snapshot testing, with comp

**Unit Testing Configuration**

| Component | Testing Framework | Configuration | Coverage Target |
|-----------|------------------|---------------|-----------------|
| React Components | Jest + React Testing Library | jsdom environment | 9
| Business Logic | Jest | Node environment | 95% |
| API Routes | Jest + Supertest | Integration testing | 85% |
| Utility Functions | Jest | Pure function testing | 100% |

**Jest Configuration for Next.js 15**
```

typescript
// jest.config.ts
import type { Config } from 'jest'
import nextJest from 'next/jest'

const createJestConfig = nextJest({
dir: './',
})

const config: Config = {
coverageProvider: 'v8',
testEnvironment: 'jsdom',
setupFilesAfterEnv: ['/jest.setup.js'],
moduleNameMapping: {
'^@/(.*)$': '/src/$1', }, testPathIgnorePatterns: ['/.next/', '/node_modules/', '/e2e/',], collectCoverageFrom: ['src//*.{js,jsx,ts,tsx}',
'!src/*/*.d.ts',*

'!src/app/layout.tsx',
],
}

export default createJestConfig(config)

```
**Test Organization Structure**
```

tests/
├── unit/
│ ├── components/
│ │ ├── quote-form.test.tsx
│ │ ├── client-portal.test.tsx
│ │ └── wellness-insights.test.tsx
│ ├── lib/
│ │ ├── pricing.test.ts
│ │ ├── route-optimization.test.ts
│ │ └── wellness-analysis.test.ts
│ └── api/
│ ├── quote.test.ts
│ ├── billing.test.ts
│ └── webhooks.test.ts
├── integration/
│ ├── stripe-integration.test.ts
│ ├── quickbooks-sync.test.ts
│ └── bullmq-queues.test.ts
└── e2e/
├── quote-flow.spec.ts
├── client-onboarding.spec.ts
└── field-tech-workflow.spec.ts

```
**Mocking Strategy**

| Mock Type | Implementation | Use Cases | Example |
```

```
|-----------|----------------|-----------|---------|
| External APIs | MSW (Mock Service Worker) | Stripe, QuickBooks, Google
| Database | Prisma mock | Database operations | User creation, billing 
| Queue System | BullMQ mock | Background job processing | Invoice genera
| File Storage | In-memory mock | Photo uploads | Wellness insights tests

**Code Coverage Requirements**

- **Critical Business Logic**: 95% coverage (pricing, billing, payroll)
- **React Components**: 90% coverage (UI components, forms)
- **API Endpoints**: 85% coverage (REST APIs, webhooks)
- **Integration Points**: 80% coverage (external service integrations)

**Test Naming Conventions**
```

typescript
// Component tests
describe('QuoteForm', () => {
describe('when user submits valid data', () => {
it('should call onSubmit with normalized quote data', () => {
// Test implementation
})
})

describe('when ZIP code is invalid', () => {
it('should display service area error message', () => {
// Test implementation
})
})
})

// Business logic tests
describe('PricingCalculator', () => {
describe('calculateMonthlyTotal', () => {
it('should apply premium zone pricing for eligible ZIP codes', () => {
// Test implementation
})

})
})

```
**Test Data Management**
```

typescript
// Test data factories
export const createMockClient = (overrides = {}) => ({
id: 'client_123',
email: 'test@example.com',
accountId: 'account_456',
...overrides,
})

export const createMockJob = (overrides = {}) => ({
id: 'job_789',
clientId: 'client_123',
scheduledDate: new Date(),
status: 'pending',
...overrides,
})

```
#### 6.6.1.2 Integration Testing Strategy

**Service Integration Testing**

Integration testing focuses on verifying the interaction between differen

**Integration Test Categories**

| Integration Type | Test Scope | Tools | Frequency |
|------------------|------------|-------|-----------|
| API Integration | External service calls | Jest + MSW | Every commit |
| Database Integration | Prisma ORM operations | Jest + Test DB | Every c
| Queue Integration | BullMQ job processing | Jest + Redis | Every commi
| Webhook Integration | External webhook handling | Jest + Mock servers
```

```
**API Testing Strategy**
```

typescript
// Stripe integration testing
describe('Stripe Integration', () => {
beforeEach(() => {
// Setup MSW handlers for Stripe API
server.use(
rest.post('https://api.stripe.com/v1/setup_intents', (req, res, ctx) => {
return res(ctx.json({
id: 'seti_test_123',
client_secret: 'seti_test_123_secret',
status: 'requires_payment_method'
}))
})
)
})

it('should create setup intent for new subscription', async () => {
const result = await createStripeSetupIntent('customer_123')
expect(result.client_secret).toBeDefined()
expect(result.status).toBe('requires_payment_method')
})
})

```
**Database Integration Testing**

In tests, run a separate Redis instance to isolate from production. Test
```

typescript
// Database integration with test isolation
describe('Client Database Operations', () => {
beforeEach(async () => {

await prisma.$transaction([
prisma.client.deleteMany(),
prisma.account.deleteMany(),
])
})

it('should create client with proper tenant isolation', async () => {
const account = await prisma.account.create({
data: { name: 'Test Account' }
})

```
const client = await createClient({
  email: 'test@example.com',
  accountId: account.id
})

expect(client.accountId).toBe(account.id)
```

})
})

```
**External Service Mocking**

| Service | Mock Strategy | Test Scenarios | Error Handling |
|---------|---------------|----------------|----------------|
| Stripe API | MSW with realistic responses | Payment success, failures,
| QuickBooks Online | Mock OAuth + API responses | Sync success, auth fa
| Google Maps | Static route responses | Route optimization, geocoding |
| Twilio | Mock SMS/voice responses | Notification delivery | Service una

**Test Environment Management**

<div class="mermaid-wrapper" id="mermaid-diagram-0b5mtlszo">
          <div class="mermaid">
flowchart TD
    A[Test Suite Start] --&gt; B[Setup Test Database]
    B --&gt; C[Start Redis Test Instance]
    C --&gt; D[Initialize MSW Handlers]
```

```
    D --&gt; E[Run Test Cases]
    E --&gt; F[Cleanup Test Data]
    F --&gt; G[Stop Test Services]
    G --&gt; H[Generate Coverage Report]
</div>
        </div>

#### 6.6.1.3 End-to-End Testing Framework

**E2E Testing with Playwright**

Playwright is a testing framework that lets you automate Chromium, Firefo

**E2E Test Scenarios**

| User Journey | Test Coverage | Browser Support | Performance Metrics |
|--------------|---------------|-----------------|--------------------|
| Quote to Lead Conversion | Complete wizard flow | Chrome, Firefox, Safa
| Client Onboarding | Account creation to first job | Chrome, Firefox | <
| Field Tech Workflow | Clock in to job completion | Chrome (PWA focus)
| Wellness Insights | Photo upload to trend display | Chrome, Firefox | <

**Playwright Configuration**
```

typescript
// playwright.config.ts
import { defineConfig, devices } from '@playwright/test'

export default defineConfig({
testDir: './e2e',
fullyParallel: true,
forbidOnly: !!process.env.CI,
retries: process.env.CI ? 2 : 0,
workers: process.env.CI ? 1 : undefined,
reporter: 'html',
use: {
baseURL: 'http://localhost:3000',
trace: 'on-first-retry',
screenshot: 'only-on-failure',

},
projects: [
{
name: 'chromium',
use: { ...devices['Desktop Chrome'] },
},
{
name: 'firefox',
use: { ...devices['Desktop Firefox'] },
},
{
name: 'webkit',
use: { ...devices['Desktop Safari'] },
},
{
name: 'Mobile Chrome',
use: { ...devices['Pixel 5'] },
},
],
webServer: {
command: 'npm run dev',
url: 'http://localhost:3000',
reuseExistingServer: !process.env.CI,
},
})

```
**UI Automation Approach**

Playwright will simulate a user navigating your application using three k
```

typescript
// Quote flow E2E test
import { test, expect } from '@playwright/test'

test('complete quote to lead conversion flow', async ({ page }) => {
// Navigate to quote page
await page.goto('/quote')

// Fill quote wizard
await page.fill('[data-testid="zip-code"]', '90210')
await page.selectOption('[data-testid="frequency"]', 'weekly')
await page.fill('[data-testid="dog-count"]', '2')

// Submit quote
await page.click('[data-testid="submit-quote"]')

// Verify success page
await expect(page).toHaveURL(/\/quote\/success/)
await expect(page.locator('h1')).toContainText('Quote Submitted')

// Verify lead creation
const leadId = await page.locator('[data-testid="lead-id"]').textContent()
expect(leadId).toMatch(/^lead_/)
})

```
**Test Data Setup/Teardown**
```

typescript
// Global setup for E2E tests
import { test as setup } from '@playwright/test'

setup('create test data', async ({ request }) => {
// Create test account
const account = await request.post('/api/test/accounts', {
data: { name: 'E2E Test Account' }
})

// Store account ID for tests
process.env.TEST_ACCOUNT_ID = (await account.json()).id

})

setup.afterAll(async ({ request }) => {
// Cleanup test data
await request.delete(`/api/test/accounts/${process.env.TEST_ACCOUNT_ID}`)
})

```
**Performance Testing Requirements**

| Performance Metric | Target | Measurement | Alert Threshold |
|-------------------|--------|-------------|-----------------|
| Page Load Time | < 2s | Lighthouse CI | > 3s |
| Time to Interactive | < 3s | Web Vitals | > 5s |
| Largest Contentful Paint | < 2.5s | Core Web Vitals | > 4s |
| Cumulative Layout Shift | < 0.1 | Layout stability | > 0.25 |

**Cross-Browser Testing Strategy**

<div class="mermaid-wrapper" id="mermaid-diagram-jwcv6xohm">
          <div class="mermaid">
graph TB
    A[E2E Test Suite] --&gt; B[Desktop Chrome]
    A --&gt; C[Desktop Firefox]
    A --&gt; D[Desktop Safari]
    A --&gt; E[Mobile Chrome]

    B --&gt; F[Core Functionality]
    C --&gt; F
    D --&gt; F
    E --&gt; G[PWA Functionality]

    F --&gt; H[Quote Flow]
    F --&gt; I[Client Portal]
    F --&gt; J[Admin Dashboard]

    G --&gt; K[Field Tech App]
    G --&gt; L[Offline Capabilities]
</div>
        </div>

### 6.6.2 TEST AUTOMATION
```

```
#### 6.6.2.1 CI/CD Integration

**GitHub Actions Workflow**

The testing strategy integrates with GitHub Actions for continuous integr
```

yaml

.github/workflows/test.yml

name: Test Suite

on:
push:
branches: [main, develop]
pull_request:
branches: [main]

jobs:
unit-tests:
runs-on: ubuntu-latest
services:
postgres:
image: postgres:14
env:
POSTGRES_PASSWORD: postgres
options: >-
--health-cmd pg_isready
--health-interval 10s
--health-timeout 5s
--health-retries 5
redis:
image: redis:7

options: >-
--health-cmd "redis-cli ping"
--health-interval 10s
--health-timeout 5s
--health-retries 5

```
steps:
  - uses: actions/checkout@v4
  - uses: actions/setup-node@v4
    with:
      node-version: '20'
      cache: 'npm'

  - run: npm ci
  - run: npm run test:unit
  - run: npm run test:integration

  - name: Upload coverage reports
    uses: codecov/codecov-action@v3
    with:
      file: ./coverage/lcov.info
```

e2e-tests:
runs-on: ubuntu-latest
steps:
- uses: actions/checkout@v4
- uses: actions/setup-node@v4
with:
node-version: '20'
cache: 'npm'

```
  - run: npm ci
  - run: npx playwright install --with-deps
  - run: npm run build
  - run: npm run test:e2e

  - uses: actions/upload-artifact@v3
    if: failure()
```

```
    with:
      name: playwright-report
      path: playwright-report/
```

```
**Automated Test Triggers**

| Trigger Event | Test Suite | Environment | Notification |
|---------------|------------|-------------|--------------|
| Pull Request | Unit + Integration | Test DB | GitHub status check |
| Main Branch Push | Full test suite | Staging | Slack notification |
| Release Tag | Full suite + E2E | Production | Email + Slack |
| Scheduled | Full suite + Performance | Production | Weekly report |

**Parallel Test Execution**
```

typescript

// Jest parallel configuration

module.exports = {

maxWorkers: '50%',

testTimeout: 30000,

setupFilesAfterEnv: ['/jest.setup.js'],

projects: [

{

displayName: 'unit',

testMatch: ['/tests/unit//***.test.{js,ts,tsx}'], }, { displayName: 'integration', testMatch: ['/tests/integration//***.test.{js,ts,tsx}'],

setupFilesAfterEnv: ['/tests/integration/setup.js'],

},

],

}

```
**Test Reporting Requirements**

| Report Type | Format | Frequency | Recipients |
|-------------|--------|-----------|------------|
| Coverage Report | HTML + LCOV | Every commit | Development team |
```

```
| Performance Report | JSON + Charts | Daily | Product team |
| E2E Test Results | HTML + Screenshots | Every deployment | QA team |
| Flaky Test Report | CSV + Analysis | Weekly | Engineering leads |

**Failed Test Handling**

<div class="mermaid-wrapper" id="mermaid-diagram-j4erh3oqp">
          <div class="mermaid">
flowchart TD
    A[Test Failure Detected] --&gt; B{Test Type}
    B --&gt;|Unit/Integration| C[Immediate Notification]
    B --&gt;|E2E| D[Retry Once]

    C --&gt; E[Block PR Merge]
    D --&gt; F{Retry Success?}
    F --&gt;|Yes| G[Mark as Flaky]
    F --&gt;|No| H[Block Deployment]

    E --&gt; I[Developer Investigation]
    G --&gt; J[Add to Flaky Test Report]
    H --&gt; K[Rollback Trigger]

    I --&gt; L[Fix Required]
    J --&gt; M[Weekly Review]
    K --&gt; N[Emergency Response]
</div>
        </div>

**Flaky Test Management**

| Detection Method | Threshold | Action | Review Frequency |
|------------------|-----------|--------|------------------|
| Success Rate Tracking | < 95% success | Mark as flaky | Daily |
| Execution Time Variance | > 50% variance | Performance review | Weekly
| Environment Sensitivity | Fails in CI only | Environment investigation
| Dependency Issues | External service failures | Mock improvement | Mon

#### 6.6.2.2 BullMQ Queue Testing Strategy

**Queue Testing Architecture**

We can write tests for producers and consumers separately. In tests, run
```

```
**Queue Testing Patterns**

| Queue Type | Test Strategy | Mock Level | Assertions |
|------------|---------------|------------|------------|
| Billing Queue | Job processing + Redis | Redis mock | Job completion, 
| Notification Queue | Message delivery | Service mocks | Delivery confi
| Wellness Queue | Photo analysis | Algorithm mocks | Processing time, a
| Integration Queue | External API calls | API mocks | Success rates, er

**BullMQ Test Implementation**
```

typescript
// Queue testing with isolated Redis
import { Queue, Worker } from 'bullmq'
import IORedis from 'ioredis'

describe('Billing Queue', () => {
let queue: Queue
let worker: Worker
let redis: IORedis

beforeAll(async () => {
redis = new IORedis({
host: 'localhost',
port: 6380, // Test Redis instance
db: 1,
})

```
queue = new Queue('billing-test', { connection: redis })
worker = new Worker('billing-test', async (job) => {
  // Mock billing processor
  return { invoiceId: 'inv_test_123' }
}, { connection: redis })
```

})

afterAll(async () => {
await queue.close()
await worker.close()
await redis.quit()
})

it('should process invoice generation job', async () => {
const job = await queue.add('generate-invoice', {
clientId: 'client_123',
amount: 5000,
})

```
// Wait for job completion
const result = await job.waitUntilFinished()

expect(result.invoiceId).toBeDefined()
expect(result.invoiceId).toMatch(/^inv_/)
```

})
})

```
**Queue Health Testing**
```

typescript
// Queue monitoring and health checks
describe('Queue Health Monitoring', () => {
it('should track job completion rates', async () => {
const metrics = await queue.getMetrics('completed', 0, -1)
const completionRate = metrics.data.reduce((sum, count) => sum + count, 0)

```
expect(completionRate).toBeGreaterThan(0)
```

})

it('should handle queue backlog alerts', async () => {
const waitingCount = await queue.getWaitingCount()
const activeCount = await queue.getActiveCount()

```
expect(waitingCount + activeCount).toBeLessThan(1000)
```

})
})

```
#### 6.6.2.3 Stripe Integration Testing

**Webhook Testing Strategy**

In a different terminal tab, use the trigger CLI command to trigger a moc

**Stripe Mock Implementation**
```

typescript
// Stripe webhook testing
import { createMocks } from 'node-mocks-http'
import stripeWebhookHandler from '@/pages/api/webhooks/stripe'

describe('Stripe Webhooks', () => {
it('should handle payment_intent.succeeded event', async () => {
const mockEvent = {
id: 'evt_test_123',
type: 'payment_intent.succeeded',
data: {
object: {
id: 'pi_test_123',
amount: 5000,
currency: 'usd',
status: 'succeeded',
}

```
}
}
```

```
const { req, res } = createMocks({
  method: 'POST',
  body: mockEvent,
  headers: {
    'stripe-signature': 'test_signature',
  },
})

await stripeWebhookHandler(req, res)

expect(res._getStatusCode()).toBe(200)
```

```
})
})
```

```
**Payment Flow Testing**

| Test Scenario | Mock Response | Expected Behavior | Error Handling |
|---------------|---------------|-------------------|----------------|
| Successful Payment | payment_intent.succeeded | Update subscription sta
| Failed Payment | payment_intent.payment_failed | Trigger dunning proces
| Card Declined | card_declined error | Show user-friendly error | Altern
| Webhook Retry | Multiple delivery attempts | Idempotent processing | Du

### 6.6.3 QUALITY METRICS

#### 6.6.3.1 Code Coverage Targets

**Coverage Requirements by Component**

| Component Category | Coverage Target | Measurement Tool | Enforcement
|-------------------|-----------------|------------------|-------------|
| Business Logic | 95% | Jest coverage | CI/CD gate |
| React Components | 90% | React Testing Library | PR requirement |
| API Endpoints | 85% | Supertest integration | Quality gate |
| Utility Functions | 100% | Jest unit tests | Mandatory |
```

```
**Coverage Tracking Implementation**
```

typescript
// Jest coverage configuration
module.exports = {
collectCoverage: true,
coverageDirectory: 'coverage',
coverageReporters: ['text', 'lcov', 'html'],
coverageThreshold: {
global: {
branches: 80,
functions: 85,
lines: 85,
statements: 85,
},
'./src/lib/pricing.ts': {
branches: 95,
functions: 95,
lines: 95,
statements: 95,
},
'./src/lib/billing.ts': {
branches: 95,
functions: 95,
lines: 95,
statements: 95,
},
},
}

```
**Test Success Rate Requirements**

| Test Category | Success Rate Target | Measurement Period | Action Thres
```

```
|---------------|-------------------|-------------------|----------------
| Unit Tests | 99% | Per commit | < 95% blocks merge |
| Integration Tests | 95% | Daily average | < 90% investigation |
| E2E Tests | 90% | Weekly average | < 85% test review |
| Performance Tests | 95% | Per deployment | < 90% performance review |

**Performance Test Thresholds**
```

typescript
// Performance testing with Playwright
import { test, expect } from '@playwright/test'

test('quote form performance', async ({ page }) => {
await page.goto('/quote')

// Measure form submission time
const startTime = Date.now()
await page.fill('[data-testid="zip-code"]', '90210')
await page.click('[data-testid="submit-quote"]')
await page.waitForURL(/\/quote\/success/)
const endTime = Date.now()

const submissionTime = endTime - startTime
expect(submissionTime).toBeLessThan(5000) // 5 second threshold
})

```
**Quality Gates Configuration**

<div class="mermaid-wrapper" id="mermaid-diagram-c1m63q2kb">
          <div class="mermaid">
flowchart TD
    A[Code Commit] --&gt; B[Unit Tests]
    B --&gt; C{Coverage &gt; 85%?}
    C --&gt;|No| D[Block Merge]
    C --&gt;|Yes| E[Integration Tests]

    E --&gt; F{Success Rate &gt; 95%?}
    F --&gt;|No| G[Investigation Required]
```

```
    F --&gt;|Yes| H[E2E Tests]

    H --&gt; I{Performance OK?}
    I --&gt;|No| J[Performance Review]
    I --&gt;|Yes| K[Deployment Approved]

    D --&gt; L[Developer Notification]
    G --&gt; M[Team Review]
    J --&gt; N[Optimization Required]
</div>
        </div>

**Documentation Requirements**

| Documentation Type | Coverage Requirement | Update Frequency | Review I
|--------------------|---------------------|------------------|-----------
| Test Plan Documentation | 100% of features | Per feature release | Proc
| API Test Documentation | 100% of endpoints | Per API change | Engineer
| E2E Test Scenarios | 100% of user journeys | Per UX change | QA review
| Performance Benchmarks | All critical paths | Monthly | Performance rev

#### 6.6.3.2 Test Execution Flow

**Comprehensive Test Pipeline**

<div class="mermaid-wrapper" id="mermaid-diagram-ip86qrdf3">
          <div class="mermaid">
flowchart TD
    A[Developer Commit] --&gt; B[Pre-commit Hooks]
    B --&gt; C[Lint &amp; Format Check]
    C --&gt; D[Unit Test Execution]
    D --&gt; E{Unit Tests Pass?}
    E --&gt;|No| F[Block Commit]
    E --&gt;|Yes| G[Push to Repository]

    G --&gt; H[CI Pipeline Trigger]
    H --&gt; I[Parallel Test Execution]

    I --&gt; J[Unit Tests]
    I --&gt; K[Integration Tests]
    I --&gt; L[Security Tests]

    J --&gt; M[Coverage Analysis]
```

```
    K --&gt; N[Service Integration Check]
    L --&gt; O[Vulnerability Scan]

    M --&gt; P{All Tests Pass?}
    N --&gt; P
    O --&gt; P

    P --&gt;|No| Q[Failure Notification]
    P --&gt;|Yes| R[E2E Test Trigger]

    R --&gt; S[Browser Testing]
    S --&gt; T[Performance Testing]
    T --&gt; U[Accessibility Testing]

    U --&gt; V{E2E Tests Pass?}
    V --&gt;|No| W[E2E Failure Analysis]
    V --&gt;|Yes| X[Deployment Ready]

    F --&gt; Y[Developer Fix Required]
    Q --&gt; Z[Team Investigation]
    W --&gt; AA[Test Environment Review]
</div>
        </div>

#### 6.6.3.3 Test Environment Architecture

**Multi-Environment Testing Strategy**

<div class="mermaid-wrapper" id="mermaid-diagram-3ju4f2xtr">
          <div class="mermaid">
graph TB
    A[Development Environment] --&gt; B[Local Testing]
    B --&gt; C[Unit Tests]
    B --&gt; D[Component Tests]

    E[CI Environment] --&gt; F[Automated Testing]
    F --&gt; G[Integration Tests]
    F --&gt; H[Security Tests]

    I[Staging Environment] --&gt; J[E2E Testing]
    J --&gt; K[Performance Tests]
    J --&gt; L[User Acceptance Tests]
```

```
    M[Production Environment] --&gt; N[Smoke Tests]
    N --&gt; O[Health Checks]
    N --&gt; P[Monitoring Tests]
</div>
        </div>

**Test Data Flow Management**

<div class="mermaid-wrapper" id="mermaid-diagram-bbh5s46ci">
          <div class="mermaid">
sequenceDiagram
    participant Dev as Developer
    participant CI as CI Pipeline
    participant TestDB as Test Database
    participant Redis as Test Redis
    participant Mock as Mock Services

    Dev-&gt;&gt;CI: Push Code
    CI-&gt;&gt;TestDB: Setup Test Data
    CI-&gt;&gt;Redis: Initialize Queue State
    CI-&gt;&gt;Mock: Configure API Mocks

    CI-&gt;&gt;CI: Run Unit Tests
    CI-&gt;&gt;CI: Run Integration Tests
    CI-&gt;&gt;CI: Run E2E Tests

    CI-&gt;&gt;TestDB: Cleanup Test Data
    CI-&gt;&gt;Redis: Clear Queue State
    CI-&gt;&gt;Mock: Reset Mock State

    CI-&gt;&gt;Dev: Test Results
</div>
        </div>

This comprehensive testing strategy ensures the Yardura Service OS mainta

# 7. USER INTERFACE DESIGN

## 7.1 CORE UI TECHNOLOGIES

### 7.1.1 Frontend Technology Stack

The Yardura Service OS implements a modern, component-driven user interfa
```

```
**Primary UI Technologies**

| Technology | Version | Purpose | Implementation Details |
|------------|---------|---------|---------------------|
| Next.js | 15.x | Full-stack React framework | App Router architecture p
| React | 19.x | UI component library | React 19 RC tested and supported
| TypeScript | 5.0+ | Type safety and development experience | Full-stack
| Tailwind CSS | v4 | Utility-first CSS framework | HSL colors converted
| shadcn/ui | Latest | Pre-built accessible components | All components u

### 7.1.2 Animation and Interaction Framework

**Motion Library Integration**

The system integrates Motion (previously Framer Motion) as a fast, produc

**Animation Implementation Strategy**

| Component | Animation Library | Use Cases | Performance Considerations
|-----------|-----------------|-----------|-------------------------|
| Page Transitions | Motion/React | Route changes, modal overlays | Hybri
| Micro-interactions | CSS Transitions | Button hovers, form focus states
| Complex Animations | Motion Components | Dashboard updates, data visual
| Gesture Recognition | Motion Gestures | Mobile interactions, drag opera

### 7.1.3 Progressive Web Application Architecture

**PWA Implementation**

The Field Technician interface implements Progressive Web Application fea

**PWA Configuration**
```

typescript
// app/manifest.ts
import type { MetadataRoute } from 'next'

export default function manifest(): MetadataRoute.Manifest {
return {
name: 'Yardura Field Tech',

short_name: 'YarduraTech',
description: 'Field technician app for dog waste service operations',
start_url: '/field-tech',
display: 'standalone',
background_color: '#ffffff',
theme_color: '#059669',
icons: [
{
src: '/icon-192x192.png',
sizes: '192x192',
type: 'image/png',
},
{
src: '/icon-512x512.png',
sizes: '512x512',
type: 'image/png',
},
],
}
}

```
**PWA Features Implementation**

| Feature | Technology | Implementation | Business Value |
|---------|------------|----------------|----------------|
| Offline Support | Serwist with Next.js for offline functionality | Serv
| Push Notifications | Web Push API supported on iOS 16.4+ for home scree
| Background Sync | Service Worker API | Queue offline actions | Data cor
| Camera Integration | MediaDevices API | Photo capture for service proof

## 7.2 UI USE CASES

### 7.2.1 Multi-User Interface Architecture

The system implements distinct user interfaces optimized for different us
```

```
**Route Group Organization**
```

app/
├── (client)/ # Client portal interface
│ ├── dashboard/
│ ├── wellness/
│ ├── billing/
│ └── settings/
├── (dispatch)/ # Dispatch operations interface
│ ├── board/
│ ├── routes/
│ ├── schedule/
│ └── reports/
├── (field-tech)/ # PWA field technician interface
│ ├── jobs/
│ ├── shift/
│ └── navigation/
├── (admin)/ # Administrative interface
│ ├── billing/
│ ├── payroll/
│ ├── reports/
│ └── settings/
└── (franchise)/ # Multi-tenant franchise management
├── accounts/
├── royalties/
└── oversight/

```
### 7.2.2 Client Portal Use Cases

**Primary Client Interactions**

| Use Case | Interface Components | User Goals | Technical Implementation
|----------|--------------------|------------|------------------------
| Service Proof Viewing | Photo gallery, timeline view | Verify service
```

```
| Wellness Insights Access | Charts, trend analysis, disclaimers | Monit
| Billing Management | Invoice history, payment methods | Manage subscri
| Schedule Management | Calendar view, service history | Track upcoming 

**Client Portal Component Architecture**

<div class="mermaid-wrapper" id="mermaid-diagram-agyf0ctje">
          <div class="mermaid">
graph TB
    A[Client Portal Layout] --&gt; B[Navigation Header]
    A --&gt; C[Main Content Area]
    A --&gt; D[Sidebar Navigation]

    C --&gt; E[Dashboard Overview]
    C --&gt; F[Wellness Insights]
    C --&gt; G[Service History]
    C --&gt; H[Billing Section]

    F --&gt; I[3C Analysis Charts]
    F --&gt; J[Trend Visualizations]
    F --&gt; K[Photo Gallery]

    H --&gt; L[Invoice List]
    H --&gt; M[Payment Methods]
    H --&gt; N[Subscription Management]
</div>
        </div>

### 7.2.3 Field Technician PWA Use Cases

**Mobile-First Field Operations**

The Field Technician PWA prioritizes improved performance with quick loa

**Core Field Tech Workflows**

| Workflow | UI Components | Offline Capability | Performance Requiremen
|----------|---------------|-------------------|------------------------
| Shift Management | Clock in/out, break tracking, odometer | Full offli
| Job Execution | Job details, photo capture, completion forms | Queue a
| Navigation | GPS integration, route display | Cached route data | Real
| Status Updates | Progress tracking, client notifications | Sync when c
```

```
**PWA Interface Design Patterns**
```

typescript
// Field Tech PWA Layout Component
'use client'

import { motion } from 'motion/react'
import { useServiceWorker } from '@/hooks/useServiceWorker'
import { useOfflineQueue } from '@/hooks/useOfflineQueue'

export function FieldTechLayout({ children }: { children: React.ReactNode }) {
const { isOnline, syncStatus } = useServiceWorker()
const { queuedActions } = useOfflineQueue()

return (

Yardura Field Tech

{queuedActions > 0 && ({queuedActions} queued)}

{children}

)
}

```
### 7.2.4 Dispatch Operations Use Cases

**Real-Time Operations Management**
```

```
| Use Case | Interface Elements | Real-Time Features | Data Sources |
|----------|-------------------|-------------------|--------------|
| Route Optimization | Interactive map, job assignments | Live technician
| Schedule Management | Calendar grid, drag-and-drop | Instant updates |
| Weather Monitoring | Weather overlay, mass skip controls | Weather API
| Progress Tracking | Status indicators, completion metrics | WebSocket u

### 7.2.5 Administrative Dashboard Use Cases

**Business Management Interfaces**

| Dashboard Type | Primary Functions | User Roles | Update Frequency |
|----------------|------------------|------------|------------------|
| Billing Console | Invoice management, payment processing | Accountants
| Payroll Dashboard | Time tracking, compensation calculation | Managers
| Analytics Portal | KPI tracking, business reporting | Owners, Managers
| Franchise Management | Multi-tenant oversight, royalties | Franchise Ow

## 7.3 UI/BACKEND INTERACTION BOUNDARIES

### 7.3.1 API Integration Patterns

**Next.js 15 App Router API Architecture**

The system leverages Next.js 15 App Router for seamless frontend-backend

**Data Flow Patterns**

| Interaction Type | Implementation | Data Format | Error Handling |
|------------------|----------------|-------------|----------------|
| Server Actions | React Server Actions | TypeScript interfaces | Form va
| API Routes | RESTful endpoints | JSON with Zod validation | HTTP status
| Real-time Updates | WebSocket connections | Event-driven messages | Con
| Background Sync | Service Worker | Queued operations | Conflict resolu

**Server Action Implementation Example**
```

typescript
// app/actions/job-completion.ts
'use server'

import { z } from 'zod'
import { revalidatePath } from 'next/cache'
import { prisma } from '@/lib/prisma'

const jobCompletionSchema = z.object({
jobId: z.string(),
photos: z.array(z.string()),
notes: z.string().optional(),
completedAt: z.date(),
})

export async function completeJob(formData: FormData) {
const data = jobCompletionSchema.parse({
jobId: formData.get('jobId'),
photos: JSON.parse(formData.get('photos') as string),
notes: formData.get('notes'),
completedAt: new Date(),
})

try {
await prisma.job.update({
where: { id: data.jobId },
data: {
status: 'completed',
completedAt: data.completedAt,
photos: {
create: data.photos.map(url => ({ url }))
},
notes: data.notes,
},
})

```
revalidatePath('/field-tech/jobs')
return { success: true }
```

} catch (error) {
return { error: 'Failed to complete job' }
}
}

```
### 7.3.2 State Management Architecture

**Client-Side State Management**

| State Type | Management Strategy | Persistence | Synchronization |
|------------|-------------------|-------------|-----------------|
| UI State | React useState/useReducer | Session storage | Component-loca
| Server State | React Query/SWR | Cache with TTL | Background refresh |
| Form State | React Hook Form | Local state | Server validation |
| Global State | Zustand/Context | localStorage | Cross-component |

**Real-Time Data Synchronization**

<div class="mermaid-wrapper" id="mermaid-diagram-dgzudwu89">
          <div class="mermaid">
sequenceDiagram
    participant UI as User Interface
    participant SA as Server Action
    participant API as API Route
    participant WS as WebSocket
    participant DB as Database

    UI-&gt;&gt;SA: Form Submission
    SA-&gt;&gt;DB: Update Data
    DB--&gt;&gt;SA: Confirmation
    SA-&gt;&gt;UI: Revalidate Cache

    DB-&gt;&gt;WS: Broadcast Change
    WS-&gt;&gt;UI: Real-time Update
    UI-&gt;&gt;UI: Update UI State

    UI-&gt;&gt;API: Background Sync
    API-&gt;&gt;DB: Batch Updates
    DB--&gt;&gt;API: Sync Status
    API--&gt;&gt;UI: Sync Complete
</div>
        </div>
```

```
### 7.3.3 Authentication and Authorization Integration

**Session Management with NextAuth**

The system implements comprehensive authentication using NextAuth with ro

**Authentication Flow Integration**
```

typescript
// middleware.ts
import { withAuth } from 'next-auth/middleware'

export default withAuth(
function middleware(req) {
// Role-based route protection
const { pathname } = req.nextUrl
const { token } = req.nextauth

```
if (pathname.startsWith('/admin') && token?.role !== 'admin') {
  return new Response('Unauthorized', { status: 403 })
}

if (pathname.startsWith('/field-tech') && token?.role !== 'technician') -
  return new Response('Unauthorized', { status: 403 })
}
```

},
{
callbacks: {
authorized: ({ token }) => !!token,
},
}
)

export const config = {
matcher: ['/admin/:path*', '/field-tech/:path*', '/dispatch/:path*']

}

```
## 7.4 UI SCHEMAS

### 7.4.1 Component Schema Architecture

**shadcn/ui Component Integration**

The system utilizes shadcn/ui components styled using Tailwind CSS with p

**Core Component Schema**
```

typescript
// types/ui-components.ts
import { z } from 'zod'

export const ButtonVariantSchema = z.enum([
'default', 'destructive', 'outline', 'secondary', 'ghost', 'link'
])

export const FormFieldSchema = z.object({
name: z.string(),
label: z.string(),
type: z.enum(['text', 'email', 'password', 'number', 'select', 'textarea']),
required: z.boolean().default(false),
validation: z.object({
min: z.number().optional(),
max: z.number().optional(),
pattern: z.string().optional(),
}).optional(),
})

export const DashboardCardSchema = z.object({
title: z.string(),
value: z.union([z.string(), z.number()]),
change: z.object({

value: z.number(),
type: z.enum(['increase', 'decrease', 'neutral']),
}).optional(),
icon: z.string().optional(),
})

```
### 7.4.2 Data Visualization Schemas

**Wellness Insights Chart Configuration**
```

typescript
// schemas/wellness-charts.ts
export const WellnessDataPointSchema = z.object({
date: z.date(),
colorScore: z.number().min(1).max(5),
consistencyScore: z.number().min(1).max(5),
contentScore: z.number().min(1).max(5),
moistureLevel: z.enum(['low', 'normal', 'high']),
estimatedWeight: z.number().positive(),
})

export const ChartConfigSchema = z.object({
type: z.enum(['line', 'bar', 'area']),
data: z.array(WellnessDataPointSchema),
xAxis: z.object({
dataKey: z.string(),
label: z.string(),
}),
yAxis: z.object({
dataKey: z.string(),
label: z.string(),
domain: z.tuple([z.number(), z.number()]).optional(),
}),
colors: z.object({

primary: z.string(),
secondary: z.string().optional(),
}),
})

```
### 7.4.3 Form Validation Schemas

**Comprehensive Form Schemas**
```

typescript
// schemas/forms.ts
export const JobCompletionFormSchema = z.object({
jobId: z.string().uuid(),
photos: z.array(z.object({
url: z.string().url(),
caption: z.string().optional(),
})).min(1, 'At least one photo required'),
privateNotes: z.string().max(500).optional(),
clientNotes: z.string().max(200).optional(),
completedAt: z.date(),
skipReason: z.enum([
'gate_locked', 'dog_aggressive', 'weather', 'client_request', 'other'
]).optional(),
})

export const ClientOnboardingSchema = z.object({
personalInfo: z.object({
firstName: z.string().min(1),
lastName: z.string().min(1),
email: z.string().email(),
phone: z.string().regex(/^+?[\d\s-()]+$/),
}),
serviceAddress: z.object({
street: z.string().min(1),

city: z.string().min(1),
state: z.string().length(2),
zipCode: z.string().regex(/^\d{5}(-\d{4})?$/),
}),
servicePreferences: z.object({
frequency: z.enum(['weekly', 'biweekly', 'monthly']),
startDate: z.date().min(new Date()),
specialInstructions: z.string().max(500).optional(),
}),
paymentMethod: z.object({
stripeSetupIntentId: z.string(),
last4: z.string().length(4),
brand: z.string(),
}),
})

```
## 7.5 SCREENS REQUIRED

### 7.5.1 Client Portal Screens

**Core Client Interface Screens**

| Screen Name | Route | Primary Components | Responsive Breakpoints |
|-------------|-------|-------------------|----------------------|
| Dashboard Overview | `/client/dashboard` | Service status, next visit,
| Wellness Insights | `/client/wellness` | 3C charts, trend analysis, pho
| Service History | `/client/history` | Photo timeline, service records
| Billing Management | `/client/billing` | Invoice list, payment methods
| Account Settings | `/client/settings` | Profile, notifications, dogs/ya

**Wellness Insights Screen Architecture**

<div class="mermaid-wrapper" id="mermaid-diagram-sv5dq22ct">
          <div class="mermaid">
graph TB
    A[Wellness Insights Layout] --&gt; B[Header with Disclaimers]
    A --&gt; C[Metrics Overview Cards]
    A --&gt; D[Chart Visualization Area]
```

```
    A --&gt; E[Photo Gallery Section]

    C --&gt; F[Color Score Trend]
    C --&gt; G[Consistency Metrics]
    C --&gt; H[Content Analysis]

    D --&gt; I[Timeline Chart]
    D --&gt; J[Comparative Analysis]
    D --&gt; K[Frequency Patterns]

    E --&gt; L[Service Photo Grid]
    E --&gt; M[Photo Detail Modal]
    E --&gt; N[Date Range Filter]
</div>
        </div>

### 7.5.2 Field Technician PWA Screens

**Mobile-Optimized Field Operations**

| Screen Name | PWA Route | Offline Support | Key Interactions |
|-------------|-----------|-----------------|------------------|
| Shift Management | `/field-tech/shift` | Full offline | Clock in/out,
| Job List | `/field-tech/jobs` | Cached data | Job selection, status up
| Job Details | `/field-tech/jobs/[id]` | Full offline | Photo capture,
| Navigation | `/field-tech/navigate/[id]` | GPS required | Turn-by-turn
| Profile & Settings | `/field-tech/profile` | Cached preferences | Pers

**Job Completion Screen Flow**

<div class="mermaid-wrapper" id="mermaid-diagram-8ttt428kr">
          <div class="mermaid">
flowchart TD
    A[Job List Screen] --&gt; B[Select Job]
    B --&gt; C[Job Details Screen]
    C --&gt; D[Start Job Button]
    D --&gt; E[Photo Capture Interface]
    E --&gt; F[Review Photos]
    F --&gt; G{Photos Acceptable?}
    G --&gt;|No| E
    G --&gt;|Yes| H[Add Notes Screen]
    H --&gt; I[Complete Job Button]
    I --&gt; J[Confirmation Screen]
```

```
    J --&gt; K[Return to Job List]

    C --&gt; L[Skip Job Option]
    L --&gt; M[Skip Reason Selection]
    M --&gt; N[Skip Photo Capture]
    N --&gt; O[Skip Confirmation]
    O --&gt; K
</div>
        </div>

### 7.5.3 Dispatch Operations Screens

**Real-Time Operations Management**

| Screen Name | Route | Real-Time Features | Data Refresh |
|-------------|-------|-------------------|--------------|
| Dispatch Board | `/dispatch/board` | Live job status, technician locati
| Route Optimization | `/dispatch/routes` | Interactive map, drag-and-dro
| Schedule Management | `/dispatch/schedule` | Calendar view, bulk operat
| Weather Dashboard | `/dispatch/weather` | Weather overlay, mass skip co
| Reports & Analytics | `/dispatch/reports` | KPI dashboards, export func

### 7.5.4 Administrative Screens

**Business Management Interfaces**

| Screen Category | Key Screens | User Roles | Update Frequency |
|----------------|-------------|------------|------------------|
| Billing Console | Invoice management, payment processing, refunds | Acc
| Payroll Dashboard | Time approval, compensation calculation, exports |
| Client Management | Customer profiles, subscription management | Custon
| Reporting Suite | Financial reports, operational metrics | Owners, Mana

### 7.5.5 Franchise Management Screens

**Multi-Tenant Oversight Interface**

| Screen Name | Route | Franchise Features | Access Control |
|-------------|-------|-------------------|----------------|
| Account Switcher | `/franchise/accounts` | Multi-tenant navigation | Pa
| Royalty Management | `/franchise/royalties` | ACH collection, reporting
| Brand Consistency | `/franchise/branding` | Template management, approv
| Consolidated Reporting | `/franchise/reports` | Cross-account analytics
```

```
## 7.6 USER INTERACTIONS

### 7.6.1 Touch and Gesture Interactions

**Mobile-First Interaction Design**

The Field Technician PWA implements comprehensive touch interactions opti

**Gesture Implementation Patterns**

| Interaction Type | Implementation | Use Cases | Feedback Mechanism |
|------------------|----------------|-----------|-------------------|
| Tap Gestures | Motion tap events | Job selection, button actions | Hap
| Swipe Gestures | Custom swipe detection | Job list navigation, photo ga
| Drag Operations | Motion drag API | Route reordering, schedule manageme
| Long Press | Touch event handling | Context menus, bulk selection | Vi

**Touch Interaction Code Example**
```

typescript
// components/JobCard.tsx
'use client'

import { motion } from 'motion/react'
import { useHapticFeedback } from '@/hooks/useHapticFeedback'

export function JobCard({ job, onSelect, onSkip }) {
const { triggerHaptic } = useHapticFeedback()

return (
{ triggerHaptic('light') onSelect(job.id) }} drag="x" dragConstraints={{ left: -100, right: 0 }} onDragEnd={(event, info) => { if (info.offset.x < -50) { triggerHaptic('medium') onSkip(job.id) } }} >

{job.client.name}

{job.address}

{job.estimatedDuration}min

{job.scheduledTime}

)
}

```
### 7.6.2 Form Interactions and Validation

**Progressive Form Enhancement**

The system implements progressive form enhancement with real-time valida

**Form Interaction Patterns**

| Form Type | Validation Strategy | Error Handling | Accessibility |
|-----------|-------------------|----------------|---------------|
| Client Onboarding | Real-time + server validation | Inline error messag
| Job Completion | Client-side with offline queue | Toast notifications
| Payment Forms | Stripe Elements integration | Secure error handling | 
| Settings Forms | Debounced auto-save | Optimistic updates | Form state

### 7.6.3 Navigation Patterns

**Multi-Modal Navigation Architecture**

The system implements different navigation patterns optimized for each us

**Navigation Implementation**

| Interface Type | Navigation Pattern | Implementation | User Experience
|----------------|-------------------|----------------|-----------------
| Client Portal | Sidebar + breadcrumbs | Persistent navigation | Deskto
| Field Tech PWA | Bottom tab bar | Native app pattern | Thumb-friendly 
| Dispatch Console | Top navigation + sidebar | Dashboard layout | Multi-
| Admin Interface | Hierarchical menu | Collapsible sections | Role-base

### 7.6.4 Real-Time Interaction Feedback

**Live Data Updates and Notifications**
```

typescript
// hooks/useRealTimeUpdates.ts
import { useEffect, useState } from 'react'
import { useWebSocket } from '@/lib/websocket'

export function useRealTimeUpdates(userId: string, userRole: string) {
const [notifications, setNotifications] = useState([])
const { socket, isConnected } = useWebSocket()

useEffect(() => {
if (!socket || !isConnected) return

```
// Subscribe to role-specific channels
socket.emit('subscribe', {
  channels: [`user:${userId}`, `role:${userRole}`]
})

// Handle real-time updates
socket.on('job_assigned', (data) => {
  setNotifications(prev => [...prev, {
    type: 'job_assigned',
    message: `New job assigned: ${data.client.name}`,
    timestamp: new Date(),
  }])
})

socket.on('route_updated', (data) => {
  // Update route display in real-time
  updateRouteDisplay(data.route)
})

return () => {
  socket.off('job_assigned')
  socket.off('route_updated')
}
```

}, [socket, isConnected, userId, userRole])

return { notifications, isConnected }
}

```
## 7.7 VISUAL DESIGN CONSIDERATIONS

### 7.7.1 Design System Architecture

**Tailwind CSS v4 Integration**

The system leverages Tailwind v4 with HSL colors converted to OKLCH and s

**Design Token Structure**
```

css
/* globals.css */
@import "tailwindcss";
@import "tw-animate-css";

@theme inline {
--color-primary: oklch(0.5 0.2 200);
--color-secondary: oklch(0.8 0.1 180);
--color-accent: oklch(0.6 0.25 160);
--color-background: oklch(0.98 0.01 180);
--color-foreground: oklch(0.15 0.02 200);

--radius-sm: 0.25rem;
--radius-md: 0.5rem;
--radius-lg: 1rem;

--font-sans: 'Inter', system-ui, sans-serif;
--font-mono: 'JetBrains Mono', monospace;
}

.dark {
--color-background: oklch(0.08 0.02 200);

--color-foreground: oklch(0.92 0.01 180);
}

```
### 7.7.2 Responsive Design Strategy

**Mobile-First Responsive Architecture**

| Breakpoint | Target Devices | Layout Strategy | Key Considerations |
|------------|----------------|-----------------|-------------------|
| Mobile (320-768px) | Phones, small tablets | Single column, bottom navi
| Tablet (768-1024px) | Tablets, small laptops | Adaptive columns, sideba
| Desktop (1024px+) | Laptops, desktops | Multi-column, persistent naviga
| Large Desktop (1440px+) | Large monitors | Expanded layouts, data densi

### 7.7.3 Accessibility Implementation

**WCAG 2.1 AA Compliance**

The system implements comprehensive accessibility features throughout al

**Accessibility Features**

| Feature Category | Implementation | Standards Compliance | Testing Stra
|------------------|----------------|---------------------|-------------
| Keyboard Navigation | Focus management, skip links | WCAG 2.1 AA | Auto
| Screen Reader Support | ARIA labels, semantic HTML | Section 508 | Scre
| Color Contrast | High contrast ratios | WCAG AA (4.5:1) | Automated con
| Motion Preferences | Reduced motion support | WCAG 2.1 | User preferen

### 7.7.4 Dark Mode Implementation

**System-Wide Theme Management**
```

typescript
// components/ThemeProvider.tsx
'use client'

import { ThemeProvider as NextThemesProvider } from 'next-themes'
import { type ThemeProviderProps } from 'next-themes/dist/types'

export function ThemeProvider({ children, ...props }: ThemeProviderProps)
{
return (
{children}
)
}

```
### 7.7.5 Performance Optimization

**UI Performance Considerations**

| Optimization Technique | Implementation | Performance Impact | Measuren
|------------------------|----------------|--------------------|----------
| Code Splitting | Next.js dynamic imports | Reduced initial bundle size
| Image Optimization | Next.js Image component | Faster page loads | Core
| Component Lazy Loading | React.lazy, Suspense | Improved perceived perf
| Animation Performance | Motion components animate without triggering Re

**Performance Monitoring Integration**
```

typescript
// lib/performance.ts
import { getCLS, getFID, getFCP, getLCP, getTTFB } from 'web-vitals'

export function initPerformanceMonitoring() {
getCLS(sendToAnalytics)
getFID(sendToAnalytics)
getFCP(sendToAnalytics)
getLCP(sendToAnalytics)
getTTFB(sendToAnalytics)
}

function sendToAnalytics(metric) {
// Send to monitoring service
fetch('/api/analytics/web-vitals', {
method: 'POST',

```
body: JSON.stringify(metric),
headers: { 'Content-Type': 'application/json' }
})
}
```

```
This comprehensive User Interface Design section provides detailed speci

# 8. INFRASTRUCTURE

## 8.1 DEPLOYMENT ENVIRONMENT

### 8.1.1 Target Environment Assessment

The Yardura Service OS requires a **hybrid cloud deployment architecture

**Environment Type Selection**

| Environment Aspect | Requirement | Justification | Implementation |
|-------------------|-------------|---------------|----------------|
| Primary Deployment | Cloud-native with hybrid capabilities | Scalabili
| Geographic Distribution | Multi-region support | Low latency for field
| Compliance Requirements | SOC 2, PCI DSS, GDPR | Financial transaction
| Disaster Recovery | Multi-region backup | Business continuity requireme

**Resource Requirements Analysis**

| Component | CPU Requirements | Memory Requirements | Storage Requireme
|-----------|------------------|-------------------|-------------------|
| Next.js Application | 2-4 vCPUs per instance | 4-8 GB RAM | 20 GB SSD
| PostgreSQL Database | 4-8 vCPUs | 16-32 GB RAM | 500 GB SSD + backups
| Redis Cache/Queue | 2-4 vCPUs | 8-16 GB RAM | 100 GB SSD | High IOPS |
| BullMQ Workers | 2-4 vCPUs per worker | 2-4 GB RAM | 10 GB SSD | Queue

**Compliance and Regulatory Requirements**

The system handles sensitive data including payment information, persona

- **PCI DSS Level 1**: Payment card data handling through Stripe Element
- **SOC 2 Type II**: System controls for security, availability, and con
- **GDPR Compliance**: EU personal data protection with right to erasure
- **CCPA Compliance**: California consumer privacy rights and data portal
```

```
### 8.1.2 Environment Management

**Infrastructure as Code (IaC) Approach**

The deployment strategy leverages modern IaC practices with Terraform for
```

hcl

terraform/environments/production/main.tf

terraform {
required_version = ">= 1.0"
required_providers {
vercel = {
source = "vercel/vercel"
version = "~> 0.15"
}
postgresql = {
source = "cyrilgdn/postgresql"
version = "~> 1.21"
}
}
}

Vercel deployment configuration

resource "vercel_project" "yardura_service_os" {
name = "yardura-service-os"
framework = "nextjs"

environment = [
{

key = "DATABASE_URL"
value = var.database_url
type = "encrypted"
},
{
key = "REDIS_URL"
value = var.redis_url
type = "encrypted"
}
]
}

```
**Configuration Management Strategy**

| Configuration Type | Management Tool | Environment Scope | Update Frequ
|-------------------|-----------------|-------------------|--------------
| Application Config | Environment Variables | Per environment | Per depl
| Database Schema | Prisma Migrations | All environments | Version contro
| Infrastructure | Terraform | Environment-specific | As needed |
| Secrets Management | Vercel Environment Variables | Encrypted per envi

**Environment Promotion Strategy**

<div class="mermaid-wrapper" id="mermaid-diagram-v9vdcquzz">
          <div class="mermaid">
flowchart TD
    A[Development Environment] --&gt; B[Feature Branch Testing]
    B --&gt; C[Pull Request Review]
    C --&gt; D[Staging Environment]
    D --&gt; E[Integration Testing]
    E --&gt; F[User Acceptance Testing]
    F --&gt; G[Production Deployment]

    H[Hotfix Branch] --&gt; I[Emergency Testing]
    I --&gt; J[Production Hotfix]

    G --&gt; K[Post-deployment Monitoring]
    J --&gt; K
    K --&gt; L[Health Check Validation]
```

```
</div>
        </div>

**Backup and Disaster Recovery Plans**

| Component | Backup Frequency | Retention Period | Recovery Time Objecti
|-----------|------------------|------------------|---------------------
| PostgreSQL Database | Continuous + Daily snapshots | 30 days active, 1
| Redis Cache | Daily snapshots | 7 days | 1 hour | 1 hour |
| File Storage (S3/R2) | Cross-region replication | Indefinite | 2 hours
| Application Code | Git repository + CI/CD | Indefinite | 10 minutes | 

## 8.2 CLOUD SERVICES

### 8.2.1 Cloud Provider Selection and Justification

The Yardura Service OS utilizes a **multi-cloud strategy** with Vercel as

**Primary Cloud Services Architecture**

| Service Category | Provider | Service | Version/Tier | Justification |
|------------------|----------|---------|--------------|---------------|
| Application Hosting | Vercel | Next.js Platform | Pro Plan | You can us
| Database | Vercel/Neon | PostgreSQL | Production tier | Managed Postgre
| Cache/Queue | Redis Cloud | Redis | Standard tier | The Redis Cloud Ver
| File Storage | Cloudflare R2 | Object Storage | Standard | S3-compatibl
| CDN | Vercel Edge Network | Global CDN | Included | Integrated with Nex

**Core Services Required with Versions**
```

yaml

infrastructure/services.yml

services:
vercel:
platform: "Next.js 15"
node_version: "20.x"

build_command: "npm run build"
output_directory: ".next"

database:
provider: "Vercel Postgres"
version: "PostgreSQL 14+"
connection_pooling: true
backup_retention: "30 days"

cache:
provider: "Redis Cloud"
version: "Redis 7.x"
memory: "8GB"
persistence: "RDB + AOF"

storage:
provider: "Cloudflare R2"
region: "auto"
cdn_integration: true
signed_urls: "15 minute TTL"

```
### 8.2.2 High Availability Design

**Multi-Region Architecture**

<div class="mermaid-wrapper" id="mermaid-diagram-josnr2xg7">
          <div class="mermaid">
graph TB
    A[Global CDN - Vercel Edge] --&gt; B[Primary Region - US East]
    A --&gt; C[Secondary Region - US West]
    A --&gt; D[EU Region - Frankfurt]

    B --&gt; E[Next.js Application Instances]
    B --&gt; F[PostgreSQL Primary]
    B --&gt; G[Redis Primary]

    C --&gt; H[Next.js Application Instances]
    C --&gt; I[PostgreSQL Read Replica]
```

```
    C --&gt; J[Redis Replica]

    D --&gt; K[Next.js Application Instances]
    D --&gt; L[PostgreSQL Read Replica]
    D --&gt; M[Redis Replica]

    F --&gt; N[Cross-Region Backup]
    G --&gt; O[Cross-Region Replication]
</div>
        </div>

**Availability Targets and Implementation**

| Component | Availability Target | Implementation Strategy | Monitoring
|-----------|-------------------|------------------------|------------|
| Next.js Application | 99.9% | Auto-scaling, health checks, failover | V
| Database | 99.95% | Primary-replica setup, automated failover | Connec
| Cache/Queue | 99.9% | Redis Cluster, persistence | Queue depth monitori
| File Storage | 99.99% | Multi-region replication | Access monitoring |

### 8.2.3 Cost Optimization Strategy

**Resource Optimization Framework**

| Optimization Area | Strategy | Expected Savings | Implementation |
|------------------|----------|-----------------|----------------|
| Compute Resources | Auto-scaling based on demand | 30-40% | Vercel auto
| Database Connections | Connection pooling | 50% connection overhead | P
| Storage Costs | Lifecycle policies, compression | 25-35% | Automated ar
| CDN Bandwidth | Edge caching optimization | 40-60% | Vercel Edge Netwo

**Cost Monitoring and Alerts**
```

typescript
// Cost monitoring configuration
const costAlerts = {
vercel: {
monthly_budget: 500,
alert_threshold: 0.8,
notification_channels: ['email', 'slack']

```
},
database: {
monthly_budget: 200,
alert_threshold: 0.75,
auto_scaling_limits: {
max_connections: 100,
storage_limit: '1TB'
}
},
redis: {
monthly_budget: 150,
alert_threshold: 0.8,
memory_limit: '16GB'
}
}
```

```
### 8.2.4 Security and Compliance Considerations

**Cloud Security Implementation**

| Security Domain | Implementation | Compliance Standard | Monitoring |
|----------------|----------------|-------------------|------------|
| Data Encryption | TLS 1.3 in transit, AES-256 at rest | SOC 2, PCI DSS
| Access Control | IAM roles, least privilege | SOC 2 | Access logging |
| Network Security | VPC, security groups, WAF | SOC 2 | Traffic analysis
| Audit Logging | Comprehensive audit trails | SOC 2, GDPR | Log analysis

**Compliance Automation**

<div class="mermaid-wrapper" id="mermaid-diagram-xqxs234b2">
          <div class="mermaid">
flowchart TD
    A[Compliance Requirements] --&gt; B[Automated Scanning]
    B --&gt; C[Security Policies]
    C --&gt; D[Configuration Validation]
    D --&gt; E{Compliance Check}
    E --&gt;|Pass| F[Deploy to Production]
    E --&gt;|Fail| G[Block Deployment]
```

```
    G --&gt; H[Remediation Required]
    H --&gt; B
    F --&gt; I[Continuous Monitoring]
    I --&gt; J[Compliance Reporting]
</div>
        </div>

## 8.3 CONTAINERIZATION

### 8.3.1 Container Platform Selection

While Vercel provides excellent Next.js hosting, the system also supports

**Container Strategy Justification**

| Use Case | Container Benefit | Implementation | Alternative |
|----------|------------------|----------------|-------------|
| Development Environment | Consistent local setup | Docker Compose | Lo
| CI/CD Testing | Isolated test environments | GitHub Actions containers
| Hybrid Deployment | Multi-cloud portability | Docker + Kubernetes | Pl
| Background Workers | Isolated job processing | Separate worker containe

### 8.3.2 Base Image Strategy

**Multi-Stage Docker Build**

Use Multi-stage Builds: Keep your Docker images lightweight by copying o
```

dockerfile

Dockerfile for Next.js 15 with optimized build

FROM node:20-alpine AS base

Install dependencies only when needed

FROM base AS deps
RUN apk add --no-cache libc6-compat
WORKDIR /app

Install dependencies based on the preferred package manager

COPY package.json package-lock.json* ./
RUN npm ci --only=production

Rebuild the source code only when needed

FROM base AS builder
WORKDIR /app
COPY --from=deps /app/node_modules ./node_modules
COPY . .

Build the application

ENV NEXT_TELEMETRY_DISABLED 1
RUN npm run build

Production image, copy all the files and run next

FROM base AS runner
WORKDIR /app

ENV NODE_ENV production
ENV NEXT_TELEMETRY_DISABLED 1

RUN addgroup --system --gid 1001 nodejs
RUN adduser --system --uid 1001 nextjs

COPY --from=builder /app/public ./public

Set the correct permission for prerender cache

RUN mkdir .next
RUN chown nextjs:nodejs .next

Automatically leverage output traces to reduce image size

COPY --from=builder --chown=nextjs:nodejs /app/.next/standalone ./
COPY --from=builder --chown=nextjs:nodejs /app/.next/static ./.next/static

USER nextjs

EXPOSE 3000

ENV PORT 3000
ENV HOSTNAME "0.0.0.0"

CMD ["node", "server.js"]

```
### 8.3.3 Image Versioning Approach

**Semantic Versioning for Container Images**

| Version Type | Tag Format | Trigger | Example |
|--------------|------------|---------|---------|
| Development | `dev-{commit-sha}` | Feature branch push | `dev-a1b2c3d`
| Staging | `staging-{version}` | Staging deployment | `staging-1.2.0` |
| Production | `v{major}.{minor}.{patch}` | Release tag | `v1.2.0` |
| Latest | `latest` | Production deployment | `latest` |

### 8.3.4 Build Optimization Techniques

**Docker Build Optimization**

When building the Next.js docker image file, you should get about an imag
```

yaml

docker-compose.yml for development

version: '3.8'
services:
app:
build:
context: .
dockerfile: Dockerfile
target: development
ports:
- "3000:3000"
environment:
- NODE_ENV=development
volumes:
- .:/app
- /app/node_modules
depends_on:
- postgres
- redis

postgres:
image: postgres:14-alpine
environment:
POSTGRES_DB: yardura_dev
POSTGRES_USER: postgres
POSTGRES_PASSWORD: postgres
ports:
- "5432:5432"
volumes:
- postgres_data:/var/lib/postgresql/data

redis:
image: redis:7-alpine
ports:
- "6379:6379"
volumes:
- redis_data:/data

volumes:
postgres_data:
redis_data:

```
### 8.3.5 Security Scanning Requirements

**Container Security Pipeline**

| Security Check | Tool | Frequency | Action on Failure |
|----------------|------|-----------|-------------------|
| Vulnerability Scanning | Trivy | Every build | Block deployment |
| Base Image Updates | Dependabot | Weekly | Create PR |
| Secret Scanning | GitGuardian | Every commit | Block merge |
| License Compliance | FOSSA | Release builds | Generate report |

## 8.4 ORCHESTRATION

### 8.4.1 Orchestration Platform Selection

**Serverless-First with Container Fallback**

The Yardura Service OS primarily leverages **serverless orchestration** t

| Orchestration Need | Primary Solution | Fallback Solution | Use Case |
|-------------------|------------------|-------------------|----------|
| Web Application | Vercel Serverless | Docker + Kubernetes | Standard we
| Background Jobs | Vercel Functions | Container workers | BullMQ job pro
| Database | Managed PostgreSQL | Containerized PostgreSQL | Data persis
| Cache/Queue | Managed Redis | Container Redis | Session and queue manag

### 8.4.2 Service Deployment Strategy

**Hybrid Deployment Architecture**
```

```
<div class="mermaid-wrapper" id="mermaid-diagram-7eykahkrh">
          <div class="mermaid">
graph TB
    A[GitHub Repository] --&gt; B[CI/CD Pipeline]
    B --&gt; C{Deployment Target}

    C --&gt;|Primary| D[Vercel Platform]
    C --&gt;|Fallback| E[Container Orchestration]

    D --&gt; F[Serverless Functions]
    D --&gt; G[Edge Network]
    D --&gt; H[Automatic Scaling]

    E --&gt; I[Kubernetes Cluster]
    E --&gt; J[Docker Containers]
    E --&gt; K[Manual Scaling]

    F --&gt; L[Production Traffic]
    G --&gt; L
    H --&gt; L
    I --&gt; M[Backup Traffic]
    J --&gt; M
    K --&gt; M
</div>
        </div>

### 8.4.3 Auto-scaling Configuration

**Serverless Auto-scaling**

| Component | Scaling Trigger | Min Instances | Max Instances | Scale-up
|-----------|----------------|---------------|---------------|-----------
| Next.js App | Request volume | 0 | 100 | < 1 second |
| API Routes | Concurrent requests | 0 | 50 | < 1 second |
| Background Jobs | Queue depth | 1 | 10 | < 30 seconds |
| Database Connections | Connection pool | 5 | 100 | < 5 seconds |

### 8.4.4 Resource Allocation Policies

**Resource Management Strategy**
```

yaml

Resource allocation configuration

resources:
web_application:
cpu: "1000m"
memory: "2Gi"
requests:
cpu: "500m"
memory: "1Gi"

background_workers:
cpu: "500m"
memory: "1Gi"
requests:
cpu: "250m"
memory: "512Mi"

database:
cpu: "2000m"
memory: "4Gi"
storage: "100Gi"

cache:
cpu: "1000m"
memory: "2Gi"
storage: "20Gi"

```
## 8.5 CI/CD PIPELINE

### 8.5.1 Build Pipeline
```

```
**GitHub Actions Workflow Architecture**

It's good practice to run your app with CI/CD. CI/CD stands for continuou
```

yaml

.github/workflows/ci-cd.yml

name: CI/CD Pipeline

on:
push:
branches: [main, develop]
pull_request:
branches: [main]

jobs:
test:
runs-on: ubuntu-latest
services:
postgres:
image: postgres:14
env:
POSTGRES_PASSWORD: postgres
POSTGRES_DB: test_db
options: >-
--health-cmd pg_isready
--health-interval 10s
--health-timeout 5s
--health-retries 5
redis:
image: redis:7

options: >-
--health-cmd "redis-cli ping"
--health-interval 10s
--health-timeout 5s
--health-retries 5

```
steps:
  - uses: actions/checkout@v4
  - uses: actions/setup-node@v4
    with:
      node-version: '20'
      cache: 'npm'

  - name: Install dependencies
    run: npm ci

  - name: Run type checking
    run: npm run type-check

  - name: Run linting
    run: npm run lint

  - name: Run tests
    run: npm run test
    env:
      DATABASE_URL: postgresql://postgres:postgres@localhost:5432/test_db
      REDIS_URL: redis://localhost:6379

  - name: Build application
    run: npm run build

  - name: Upload coverage reports
    uses: codecov/codecov-action@v3
    with:
      file: ./coverage/lcov.info
```

deploy:
needs: test

runs-on: ubuntu-latest

if: github.ref == 'refs/heads/main'

```
steps:
  - uses: actions/checkout@v4
  - uses: actions/setup-node@v4
    with:
      node-version: '20'
      cache: 'npm'

  - name: Install Vercel CLI
    run: npm install --global vercel@latest

  - name: Pull Vercel Environment Information
    run: vercel pull --yes --environment=production --token=${{ secrets.V

  - name: Build Project Artifacts
    run: vercel build --prod --token=${{ secrets.VERCEL_TOKEN }}

  - name: Deploy Project Artifacts to Vercel
    run: vercel deploy --prebuilt --prod --token=${{ secrets.VERCEL_TOKEN
```

```
### 8.5.2 Deployment Pipeline

**Multi-Environment Deployment Strategy**

| Environment | Trigger | Deployment Method | Validation |
|-------------|---------|-------------------|------------|
| Development | Feature branch push | Preview deployment | Automated tes
| Staging | Develop branch merge | Staging deployment | Integration tests
| Production | Main branch merge | Production deployment | Smoke tests |
| Hotfix | Hotfix branch | Emergency deployment | Critical path tests |

**Deployment Workflow**

<div class="mermaid-wrapper" id="mermaid-diagram-dhy98du85">
          <div class="mermaid">
sequenceDiagram
    participant Dev as Developer
    participant GH as GitHub
    participant CI as CI/CD Pipeline
```

```
    participant Vercel as Vercel Platform
    participant Monitor as Monitoring

    Dev-&gt;&gt;GH: Push to feature branch
    GH-&gt;&gt;CI: Trigger build pipeline
    CI-&gt;&gt;CI: Run tests and build
    CI-&gt;&gt;Vercel: Deploy preview
    Vercel--&gt;&gt;Dev: Preview URL

    Dev-&gt;&gt;GH: Create pull request
    GH-&gt;&gt;CI: Run full test suite
    CI-&gt;&gt;CI: Security and quality checks
    CI--&gt;&gt;GH: Status checks

    Dev-&gt;&gt;GH: Merge to main
    GH-&gt;&gt;CI: Trigger production pipeline
    CI-&gt;&gt;Vercel: Production deployment
    Vercel-&gt;&gt;Monitor: Health check
    Monitor--&gt;&gt;CI: Deployment status
</div>
        </div>

### 8.5.3 Rollback Procedures

**Automated Rollback Strategy**

| Failure Type | Detection Method | Rollback Trigger | Recovery Time |
|--------------|------------------|------------------|---------------|
| Build Failure | CI/CD pipeline | Automatic | < 5 minutes |
| Health Check Failure | Monitoring alerts | Automatic | < 2 minutes |
| Performance Degradation | APM thresholds | Manual approval | < 10 minu
| Critical Bug | Manual trigger | Immediate | < 1 minute |

### 8.5.4 Post-Deployment Validation

**Validation Pipeline**
```

yaml

Post-deployment validation steps

validation:
health_checks:
- endpoint: "/api/health"
expected_status: 200
timeout: 30s

smoke_tests:
- test: "User can access quote form"
url: "/quote"
assertions:
- contains: "Get Your Quote"

performance_tests:
- test: "API response time"
endpoint: "/api/quote"
max_response_time: 500ms

integration_tests:
- test: "Database connectivity"
query: "SELECT 1"
expected_result: 1

```
### 8.5.5 Release Management Process

**Release Workflow**

| Release Type | Frequency | Approval Required | Rollback Window |
|--------------|-----------|-------------------|-----------------|
| Feature Release | Bi-weekly | Product owner | 24 hours |
| Bug Fix | As needed | Engineering lead | 4 hours |
| Security Patch | Immediate | Security team | 1 hour |
| Hotfix | Emergency | On-call engineer | 30 minutes |
```

```
## 8.6 INFRASTRUCTURE MONITORING

### 8.6.1 Resource Monitoring Approach

**Comprehensive Monitoring Stack**

The infrastructure monitoring strategy leverages Vercel's built-in analy

| Monitoring Layer | Tool/Service | Metrics Collected | Alert Thresholds
|------------------|--------------|-------------------|------------------
| Application Performance | Vercel Analytics | Response times, error rate
| Infrastructure Health | Vercel System Metrics | CPU, memory, network |
| Database Performance | PostgreSQL Metrics | Connections, query performa
| Queue Health | BullMQ Metrics | Job processing, queue depth | Queue dep

**Real-Time Monitoring Dashboard**

<div class="mermaid-wrapper" id="mermaid-diagram-1tuqc8poj">
          <div class="mermaid">
graph TB
    A[Infrastructure Monitoring] --&gt; B[Application Metrics]
    A --&gt; C[Database Metrics]
    A --&gt; D[Cache Metrics]
    A --&gt; E[Queue Metrics]

    B --&gt; F[Response Times]
    B --&gt; G[Error Rates]
    B --&gt; H[Throughput]

    C --&gt; I[Connection Pool]
    C --&gt; J[Query Performance]
    C --&gt; K[Storage Usage]

    D --&gt; L[Hit Rates]
    D --&gt; M[Memory Usage]
    D --&gt; N[Connection Count]

    E --&gt; O[Job Processing Rate]
    E --&gt; P[Queue Depth]
    E --&gt; Q[Worker Health]

    F --&gt; R[Alert System]
```

```
    G --&gt; R
    H --&gt; R
    I --&gt; R
    J --&gt; R
    K --&gt; R
    L --&gt; R
    M --&gt; R
    N --&gt; R
    O --&gt; R
    P --&gt; R
    Q --&gt; R
</div>
        </div>

### 8.6.2 Performance Metrics Collection

**Key Performance Indicators**

| Metric Category | Specific Metrics | Collection Method | Target Values
|----------------|------------------|-------------------|---------------
| Web Vitals | LCP, INP, CLS, TTFB | Vercel Web Analytics | LCP < 2.5s, I
| API Performance | Response time, throughput | Application monitoring |
| Database Performance | Query time, connection usage | Database monitori
| Queue Performance | Processing time, job success rate | BullMQ metrics

### 8.6.3 Cost Monitoring and Optimization

**Cost Tracking Framework**

| Service | Cost Metric | Budget Alert | Optimization Strategy |
|---------|-------------|--------------|----------------------|
| Vercel | Function invocations, bandwidth | $500/month | Edge caching, c
| Database | Storage, compute hours | $200/month | Connection pooling, qu
| Redis | Memory usage, operations | $150/month | Data expiration, compre
| Storage | Storage volume, requests | $100/month | Lifecycle policies, C

**Automated Cost Optimization**
```

typescript
// Cost monitoring and optimization
const costOptimization = {
vercel: {

edgeCaching: {
enabled: true,
ttl: '1h',
staleWhileRevalidate: '24h'
},
functionOptimization: {
bundleAnalysis: true,
treeshaking: true,
codesplitting: true
}
},
database: {
connectionPooling: {
maxConnections: 100,
idleTimeout: '30s'
},
queryOptimization: {
slowQueryThreshold: '1s',
indexRecommendations: true
}
}
}

```
### 8.6.4 Security Monitoring

**Security Monitoring Implementation**

| Security Domain | Monitoring Tool | Detection Criteria | Response Actio
|----------------|-----------------|-------------------|-----------------
| Access Control | Vercel Security | Failed authentication attempts | Acc
| API Security | Rate limiting monitoring | Unusual request patterns | IP
| Data Protection | Audit logging | Unauthorized data access | Security a
| Infrastructure | Security scanning | Vulnerability detection | Patch de

### 8.6.5 Compliance Auditing
```

```
**Automated Compliance Monitoring**
```

yaml

Compliance monitoring configuration

compliance:
soc2:
access_logging: true
change_management: true
incident_response: true

pci_dss:
payment_data_handling: "stripe_elements_only"
network_security: "tls_1_3_minimum"
access_control: "rbac_enforced"

gdpr:
data_retention: "automated_policies"
right_to_erasure: "api_endpoint_available"
consent_management: "granular_controls"
```

# 8.7 INFRASTRUCTURE COST ESTIMATES

## 8.7.1 Monthly Cost Breakdown

**Production Environment Costs**
```

Service Category	Provider	Service Tier	Monthly Cost	Annual Cost
Application Hosting	Vercel	Pro Plan	$20/month	$240
Database	Vercel Postgres	Production	$50/month	$600
Cache/Queue	Redis Cloud	Standard	$45/month	$540
File Storage	Cloudflare R2	Standard	$25/month	$300
Monitoring	Vercel Analytics	Pro	$10/month	$120
Total Base Cost			**$150/month**	**$1,800/year**

Scaling Cost Projections

Usage Tier	Monthly Active Users	Estimated Monthly Cost	Cost per User
Startup (0-1K users)	1,000	$150	$0.15
Growth (1K-10K users)	5,000	$400	$0.08
Scale (10K-50K users)	25,000	$1,200	$0.048
Enterprise (50K + users)	100,000	$3,500	$0.035

8.7.2 Resource Sizing Guidelines

Compute Resource Allocation

Component	Development	Staging	Production	High Availability
Next.js Instances	1 instance	2 instances	5 instances	10 instances
Database	Shared	2 vCPU, 4GB RAM	4 vCPU, 8GB RAM	8 vCPU, 16GB RAM
Redis Cache	1GB memory	2GB memory	4GB memory	8GB memory
Storage	10GB	50GB	200GB	500GB

8.7.3 External Dependencies

Third-Party Service Costs

Service	Provider	Usage-Based Pricing	Estimated Monthly Cost
Payment Processing	Stripe	2.9% + $0.30 per transaction	Variable (revenue-based)
SMS Notifications	Twilio	$0.0075 per SMS	$50-200
Email Services	Resend	$20/month for 100K emails	$20
Maps/Routing	Google Maps	$2 per 1K requests	$100-300
Total External			**$170-520/month**

8.7.4 Disaster Recovery Costs

Backup and Recovery Infrastructure

Backup Type	Frequency	Storage Cost	Recovery Cost	Total Monthly
Database Backups	Daily	$20	$10 (testing)	$30
File Storage Replication	Real-time	$15	$5	$20
Application Snapshots	Weekly	$5	$2	$7
Total DR Cost				**$57/month**

8.7.5 Total Infrastructure Investment

Complete Cost Summary

Cost Category	Monthly Cost	Annual Cost	Notes
Base Infrastructure	$150	$1,800	Core services
External Dependencies	$300 (average)	$3,600	Usage-dependent
Disaster Recovery	$57	$684	Business continuity
Monitoring & Security	$25	$300	Additional tools
Total Estimated Cost	**$532/month**	**$6,384/year**	Startup to growth phase

Cost Optimization Opportunities

- **Reserved Instances**: 20-30% savings on predictable workloads
- **Auto-scaling**: 25-40% savings during low-traffic periods
- **Edge Caching**: 30-50% reduction in compute costs
- **Database Optimization**: 15-25% savings through query optimization

This comprehensive infrastructure design provides a robust, scalable, and cost-effective foundation for the Yardura Service OS, supporting the complete business management platform while maintaining high availability, security, and performance requirements.

APPENDICES

ADDITIONAL TECHNICAL INFORMATION

Next.js 15 and React 19 Integration Details

Next.js 15 introduces React 19 support, caching improvements, a stable release for Turbopack in development, new APIs, and more. Although React 19 is still in the RC phase, our extensive testing across real-world applications and our close work with the React team have given us confidence in its stability. The core breaking changes have been well-tested and won't affect existing App Router users.

Key Features Leveraged:

- **React 19 RC Support**: In version 15, the App Router uses React 19 RC, and we've also introduced backwards compatibility for React 18 with the Pages Router based on community feedback.
- **Turbopack Stability**: Turbopack Dev (Stable): Performance and stability improvements.
- **Enhanced Forms**: Enhanced Forms (next/form): Enhance HTML forms with client-side navigation.
- **Instrumentation API**: instrumentation.js API (Stable): New API for server lifecycle observability.

BullMQ Queue System Architecture

BullMQ is a lightweight, robust, and fast NodeJS library for creating background jobs and sending messages using queues. BullMQ is designed to be easy to use, but also powerful and highly configurable. It is backed by Redis, which makes it easy to scale horizontally and process jobs across multiple servers.

Version and Performance:

- **Latest Version**: Latest version: 5.58.5, last published: 6 days ago.
- **Performance Characteristics**: The fastest, most reliable, Redis-based distributed queue for Node. Carefully written for rock solid stability and atomicity.
- **Scalability**: Easy to scale horizontally. Add more workers for processing jobs in parallel.

Connection Management:
If you can afford many connections, by all means just use them. Redis connections have quite low overhead, so you should not need to care about reusing connections unless your service provider imposes hard limitations.

Stripe API Basil Version Features

The current version is 2025-08-27.basil.

Key Basil Features:

- **Personalized Invoices**: You can now personalize the appearance of your post-payment invoices with Invoice Rendering Templates, for different customers, a useful feature for Checkout and Payment Links.
- **Ad Hoc Pricing**: You can now specify a Price while creating a Payment Link in just one API request.
- **Billing Improvements**: Subscription schedules can now have phases with mixed durations. Subscriptions that use the flexible billing mode now support thresholds for usage-based billing and mixed intervals with different recurring prices.

QuickBooks Online API Minor Version 75

Beginning August 1, 2025, we will be deprecating support for minor versions 1–74. All API requests to the Accounting API will use the minor version 75 by default and previous minor versions will be ignored.

Implementation Requirements:

- **Default Behavior**: Starting August 1, 2025, all API requests to the Accounting API will default to minor version 75.
- **Parameter Handling**: In your API request, if you specify a value for the minorversion parameter that is less than 75, it will be ignored and the system will respond with data corresponding to minor version 75.

Message Queue Architecture Patterns

A message queue works by having a producer component add a job or message to the queue, while a separate consumer component removes jobs from the queue and processes them. This decouples the production and consumption of jobs into separate concerns.

Benefits:

- **Asynchrony**: Producers can add jobs to the queue without waiting for them to be processed. The jobs are processed asynchronously by consumers.
- **Loose Coupling**: Producers and consumers don't need to directly interact or know about each other. This makes scaling and modifying them independently easier.

Performance Optimization Techniques

BullMQ Performance:
High performant. Try to get the highest possible throughput from Redis by combining efficient .lua scripts and pipelining.

Job State Management:
Robust job lifecycle handling with states like waiting, active, delayed, completed, failed etc.

GLOSSARY

Term	Definition
3Cs Analysis	Wellness insights classification system analyzing Color, Consistency, and Content of pet waste for health trend monitoring
App Router	Next.js 15 routing system using React Server Components and file-system based routing with enhanced performance
BullMQ	Redis-based distributed queue system for Node.js providing background job processing with exactly-once semantics
Dunning Process	Automated collection workflow for failed payments including retry attempts and customer communication
Field Tech PWA	Progressive Web Application for field technicians providing offline-first mobile experience for job completion
Franchise Multi-tenancy	Architecture supporting multiple business accounts with hierarchical access control and consolidated operations
Route Optimization	Algorithm-based process for calculating efficient technician routes using Google Maps Distance Matrix API
Server Actions	Next.js 15 feature enabling server-side form handling and mutations without separate API endpoints
Setup Intent	Stripe API object for securely collecting and storing payment methods without immediate charge
Signed URLs	Time-limited secure URLs for accessing stored files with automatic expiration for security

Term	Definition
Wellness Insights	Non-diagnostic pet health trend analysis based on service photos and metadata patterns
Zone Pricing	Geographic pricing model with Regular and Premium zones based on ZIP code service areas

ACRONYMS

Acronym	Expanded Form
API	Application Programming Interface
ARPU	Average Revenue Per User
CDN	Content Delivery Network
CRM	Customer Relationship Management
ETA	Estimated Time of Arrival
GPS	Global Positioning System
HMAC	Hash-based Message Authentication Code
IaC	Infrastructure as Code
JWT	JSON Web Token
KPI	Key Performance Indicator
LCP	Largest Contentful Paint
LTE	Long Term Evolution (4G cellular)
MTTR	Mean Time To Resolution
ORM	Object-Relational Mapping
PAN	Primary Account Number (credit card)
PCI DSS	Payment Card Industry Data Security Standard
PII	Personally Identifiable Information
PITR	Point-in-Time Recovery
PWA	Progressive Web Application

Acronym	Expanded Form
QBO	QuickBooks Online
RBAC	Role-Based Access Control
RPO	Recovery Point Objective
RSC	React Server Components
RTO	Recovery Time Objective
SDK	Software Development Kit
SLA	Service Level Agreement
SMS	Short Message Service
SOC 2	Service Organization Control 2
SSR	Server-Side Rendering
TLS	Transport Layer Security
ToS	Terms of Service
TTL	Time To Live
WCAG	Web Content Accessibility Guidelines