PROJECT VISION DOCUMENT V2

befoodi

Multi-Restaurant SaaS Platform

Constitutional Architecture Blueprint

Document Status:
LOCKED BASELINE — Single Source of Truth

Version: 2.0 (Complete Architectural Reset)
Date: February 2026
Authority: Principal SaaS Architect

SECTION 1: Vision & Product Philosophy

1.1 Core Mission

Build a production-grade, multi-tenant SaaS platform that eliminates operational friction for small to mid-size restaurants through QR-based ordering and real-time kitchen management. The platform must scale from 1 to 100+ tenants without architectural redesign, maintain strict data isolation, and provide audit-proof historical records for 7-10 years.

1.2 Product Identity

Product Name: befoodi

Tagline: QR ordering made simple

Contact: 8650373129, unknownuser200625@gmail.com

1.3 Architectural Philosophy

Security is a Database Property

Row-Level Security (RLS) is the absolute and only security boundary. Application-layer security is supplementary but never primary. Even if an attacker gains full control of the application layer, database-level RLS must prevent cross-tenant data exfiltration.

Production is the Source of Truth

The live production database state is the definitive reference for system health. Repository migration files are secondary artifacts that must be validated against the live environment. The Repository-as-Truth fallacy has been the root cause of silent deployment failures in previous cycles.

Verification Beats Generation

Generative AI drafts solutions; deterministic tests and browser-based verification artifacts prove correctness. Trust is the bottleneck in agentic workflows. No agent may claim completion without terminal output and browser screenshots verifying the change against a live environment.

Performance is a Constraint

Scalability is not a feature to be added later; it is a technical constraint that dictates every architectural choice today. The platform must natively support 100+ tenants with <50ms RLS overhead and sub-2-second menu load times.

SECTION 2: Core Architectural Principles

2.1 Non-Negotiable Rules

1. **RLS-First Security:** No table shall exist without Row-Level Security enabled and at least one restrictive policy covering all CRUD operations.
2. **Production as Supreme Truth:** The live database state is the definitive reference. Repository files must be validated against the live environment.
3. **Deterministic Rollbacks:** No migration shall be merged unless it includes a validated rollback script and has passed an idempotent test on a fresh local instance.
4. **Append-Only Compliance:** Deletion of historical audit logs or transactional records is forbidden for a minimum of 7-10 years. Use partitioned tables and background archival.
5. **Credential Isolation:** The service_role key is a high-risk asset. It must be locked in a secret vault and only accessed by authorized Edge Functions.
6. **Forward-Compatible Scale:** All schema designs must natively support sharding of heavy tables (orders/events) from Day 1 to ensure 100+ tenant scalability.
7. **No Manual Edits:** Direct modification of production data or schema via dashboard is a governance failure. All changes must follow the versioned migration pipeline.

2.2 Permanently Banned Patterns

- Using USING (true) for any table containing user or tenant data
- Reliance on auth.role() for custom RBAC logic; use custom JWT claims instead
- Manual production schema changes via Supabase Dashboard or SQL Editor
- Storing PII or sensitive tokens in the user_metadata field
- AI sessions exceeding 50,000 tokens without mandatory context reset
- Cross-tenant foreign keys or references that bypass the tenant_id isolation layer
- Use of unindexed columns in any RLS policy predicate
- service_role usage in frontend or public APIs

SECTION 3: Multi-Tenant Design Strategy

3.1 Isolation Model

The platform implements a Shared Database, Shared Schema architecture with strict Row-Level Security isolation. This provides the optimal balance of cost-efficiency and tenant density for 100+ restaurants while maintaining absolute data isolation.

Tenant Identification

Rule: Every table containing tenant-specific data MUST include a restaurant_id (or tenant_id) column with a foreign key to the tenants table.

Enforcement: The restaurant_id must be extracted server-side from JWT claims and never trusted if provided by the client.

RLS Policy Pattern

All RLS policies must use cached predicates to prevent the Filter Explosion bottleneck. The pattern is:

```
(restaurant_id = (SELECT auth.jwt() ->> 'restaurant_id')::uuid)
```

The (SELECT...) wrapper forces PostgreSQL to evaluate the identity once using initPlan, caching the result for the duration of the query rather than re-evaluating it for every row. This optimization reduces query latency by up to 61% for basic operations.

3.2 Performance Optimization

Strategic Indexing

Every column referenced in an RLS policy must be indexed, even if never used in application WHERE clauses. Without indexes on security predicates, every RLS check triggers a sequential scan, multiplying latency by active user count.

Standard Index: CREATE INDEX idx_orders_restaurant ON orders(restaurant_id);

Audit Log Index: Use BRIN (Block Range Indexing) for append-only tables partitioned by time.

Connection Pool Integrity

Configure the connection pooler (PgBouncer) with server_reset_query = 'DISCARD ALL' to prevent session variable contamination between tenant requests. This eliminates async context leaks where Tenant A inherits security context from Tenant B.

SECTION 4: Database & RLS Governance Model

4.1 RLS Policy Requirements

Every table must have four policies covering SELECT, INSERT, UPDATE, DELETE operations. The WITH CHECK clause is mandatory for INSERT and UPDATE to prevent unauthorized data creation.

Policy Template

```
CREATE POLICY "Tenants can read own data" ON orders
FOR SELECT
USING (restaurant_id = (SELECT auth.jwt() ->>
'restaurant_id')::uuid);
```

4.2 Drift Detection

Schema drift occurs when changes are made outside the CI/CD pipeline. The framework requires automated drift detection:

8. Build ephemeral database from Git migrations (Desired State)
9. Introspect production database via MCP (Live State)
10. Diff Desired vs Live using Supabase CLI
11. Fail build if discrepancies exist (missing indexes, altered RLS policies)

4.3 Security Advisors

Integrate the get_advisors MCP tool into CI pipeline to scan for:

- Tables without RLS enabled
- Overly permissive policies (USING true)
- Missing WITH CHECK clauses
- Exposed sensitive columns

Any High or Critical priority alert automatically blocks deployment until remediated.

SECTION 5: Auth & JWT Strategy (Strict Model)

5.1 JWT Claim Structure

All tenant-level membership data must be injected into JWT claims via Supabase Auth Hooks. This eliminates the Lookup Loop bottleneck where RLS policies must query database tables to determine tenant membership.

Required Claims

restaurant_id (UUID): Primary tenant identifier
app_role (string): User role within tenant (admin, staff, customer)
user_id (UUID): Unique user identifier from auth.users

5.2 Authentication Patterns

Restaurant Admin

Method: Restaurant ID + Password

Storage: auth.users table

JWT Claims: restaurant_id, app_role: 'admin'

Kitchen Staff

Method: PIN-only authentication with device pinning

Storage: staff table (NOT in auth.users)

JWT Claims: restaurant_id, app_role: 'staff', staff_id

Security: PIN valid only if request includes registered device_id

Customers

Method: Anonymous authentication

Storage: Temporary anonymous auth.users entry

JWT Claims: restaurant_id (from QR scan), app_role: 'customer', session_id

Scope: Limited to session-based table ordering only

5.3 SaaS Platform Admin

Method: Email + Password with MFA required

Authority: Platform-wide visibility, restaurant lifecycle management

Restriction: CANNOT access restaurant operational data (orders, menu, customer PII)
JWT Claims: is_platform_admin: true, no restaurant_id

SECTION 6: Soft Delete & Audit Integrity Rules

6.1 Soft Delete Pattern

Standard: Use deleted_at timestamp (not boolean is_deleted)

Rationale: Timestamps provide metadata and integrate better with RLS. Boolean updates create Dead Tuples in MVCC, causing index bloat.

Partial Index for Uniqueness

CREATE UNIQUE INDEX users_email_active ON users(email) WHERE deleted_at IS NULL;

This allows email reuse after soft deletion while maintaining referential integrity.

Cascading Soft Deletes

Database triggers must automatically update deleted_at for child records when parent tenant is deactivated. This prevents Ghost Records from appearing in reports.

6.2 Audit Log Strategy

Trigger-Based Logging

All transactional tables (orders, payments, menu_items) must have AFTER INSERT OR UPDATE OR DELETE triggers that write to an append-only audit_log table.

Immutability Enforcement

CREATE TRIGGER no_delete_audit BEFORE DELETE ON audit.change_log FOR EACH ROW EXECUTE FUNCTION audit.prevent_mutation();

This function returns NULL, canceling all deletion attempts. Only high-privilege aal2 authenticated purges can remove audit data per retention policy.

JSONB Diff Optimization

Store only changed fields using jsonb_diff algorithm to reduce storage footprint by up to 80% for tables with many columns but sparse updates.

Partitioning

Partition audit logs by month using RANGE partitioning on executed_at timestamp. This enables zero-downtime archival where old partitions are detached and moved to cold storage without locking the live table.

SECTION 7: Admin Isolation Rules (Non-God Model)

7.1 Tenant-Scoped Admin Authority

Restaurant Admin has complete authority within their tenant scope but zero visibility into other tenants. The God Mode is logically restricted by RLS policies that filter by restaurant_id from JWT claims.

7.2 Platform Admin Restrictions

SaaS Platform Admins are strictly prohibited from viewing:

- Restaurant operational data (orders, menu, pricing)
- Customer PII (names, contact information)
- Financial transactions

Platform Admins are limited to:

- Restaurant onboarding and approval
- Activation code generation
- Restaurant status changes (PENDING → ACTIVE → SUSPENDED → CLOSED)
- Platform-level statistics (total restaurants, aggregate order volume)

7.3 service_role Key Governance

Ban: service_role key is BANNED from all client-side code and public APIs.
Permitted Use: Edge Functions with SECURITY DEFINER helper functions that verify admin identity via JWT claims.
Emergency Access: Manual production interventions must be logged in append-only audit table with full context (who, when, why, what changed).

SECTION 8: Anti-Spam & Session Protection Model

8.1 Restaurant Registration Controls

Manual approval by SaaS Admin prevents spam registrations during MVP phase. This is a process decision, not an architectural constraint. The system scales from manual (1-100 restaurants) to automated (100,000+ restaurants) without redesign.

Activation Code Model

Properties:

- Single-use only
- Time-limited (expires after 72 hours)
- Restaurant-specific (bound to restaurant_id)
- Cryptographically random (32-character hex)

8.2 Anonymous Session Security

Customer anonymous auth is scoped to:

- Single table session (identified by session_id in JWT)
- Read access to public menu for their restaurant only
- Write access to orders table with automatic restaurant_id injection
- Session expires after 24 hours or admin closes session

8.3 Rate Limiting

Implement per-tenant rate limits to prevent Noisy Neighbor effects:

- API requests: 1000/minute per restaurant
- Order creation: 100/minute per session
- Menu updates: 10/minute per admin

SECTION 9: AI Orchestration Model (ChatGPT + Claude + Antigravity + MCP)

9.1 Sharded Context Strategy

To survive the Context-Capability Paradox, agent orchestration follows a Thin Agent / Fat Platform model:

- **ChatGPT (System Orchestrator):** Maintains global state machine, enforces architectural consistency, reviews all artifacts
- **Claude (Technical Implementer):** Generates code, writes documentation, performs TDD implementation
- **Google Antigravity (Execution Surface):** Browser-integrated verification, visual proof generation, console error detection
- **MCP (Universal Remote):** Production database inspection, grounded reasoning, drift detection

9.2 16-Phase State Machine

Every complex feature runs through standardized lifecycle:

12. Setup: Initialize context with CLAUDE.md principles
13. Discovery: MCP-based inspection of current codebase and production schema
14. Design: Generate ADR and implementation plan
15. Refinement: Human-in-the-loop review
16. Implementation: Execute in atomic steps
17. Validation: Automated tests + browser verification
18. Reporting: Generate artifacts (recordings, walkthroughs)
19. Completion: Final human approval

9.3 Artifact-Driven Trust

Agents must communicate via Artifacts, not raw logs:

- **Task Lists:** Scope visibility and planning
- **Implementation Plans:** Pre-code validation
- **Browser Recordings:** End-to-end proof of UI/UX flows
- **Diff Views:** Identification of unintended code changes
- **Performance Tabs:** Screenshot proof of <2s menu load times

9.4 Context Management Rules

- Agents limited to <150 lines per context window
- Hard block execution if context usage exceeds 85%
- Force session reset, re-inject only MANIFEST.yaml state
- Leaf node execution: sub-agents forbidden from spawning other agents

SECTION 10: DevOps & Migration Discipline

10.1 Migration Pipeline

All database changes follow strict versioned pipeline:

20. **Drafting:** Generate migration via supabase db diff or manual SQL in supabase/migrations/
21. **Naming:** YYYYMMDDHHmmss_description.sql format
22. **Validation:** Must include rollback plan and pass db reset test
23. **Shadow DB Audit:** CI introspects live production, diffs against codebase
24. **Gated Deployment:** Requires two-stage approval and staging verification at scale

10.2 Local-First Development Ritual

Mandatory developer workflow:

25. **Sync:** Pull latest production schema via MCP
26. **Develop:** Implement changes using local Supabase stack (supabase start)
27. **Test:** Run supabase db reset to ensure migrations are idempotent
28. **Verify:** Execute supabase test db for pgTAP unit tests
29. **Promote:** Push to preview branch for staging verification

10.3 Production Verification Rituals

- **Local Reset:** Prove migrations run in order on fresh database
- **Staging Smoke Test:** Apply to restored clone of production to identify performance regressions
- **Rollback Dry-Run:** Test revert script in staging environment
- **Post-Deployment Health Check:** Verify RLS policies active, indexes present, performance within SLA

SECTION 11: Environment Separation Rules (Local / Staging / Production)

11.1 Environment Architecture

Local Development

Purpose: Rapid iteration and feature development

Database: Supabase CLI local instance

Data: Synthetic test data only

Reset: Frequent (after each migration test)

Staging

Purpose: Pre-production validation at scale

Database: Restored clone of production (sanitized PII)

Data: Production-like volume and complexity

Refresh: Weekly from production snapshot

Production

Purpose: Live customer-facing system

Access: Read-only for developers, write access via CI/CD only

Monitoring: Real-time drift detection, performance alerts, security advisors

11.2 Credential Isolation

Each environment must use distinct credentials:

- Separate Supabase projects
- Unique anon keys and service_role keys
- Environment-specific JWT secrets
- No production credentials in git or CI logs

SECTION 12: Documentation Governance Rules

12.1 Single Source of Truth Hierarchy

Documentation is sharded to maintain clarity for AI agents:

30. **Architecture Baseline:** docs/arc42/* (short, link-heavy, no inline code)
31. **Developer Standards:** docs/developer-guide/* (templates and naming conventions)
32. **Product Blueprints:** docs/product/* (epics, stories, user flows)
33. **Project Anchor:** CLAUDE.md (200-line summary read at every session start)
34. **Snapshots:** docs/snapshots/* (MCP-generated production state, never edited manually)

12.2 Architectural Decision Records (ADRs)

Every significant technical choice must be recorded in an ADR stored in docs/decisions/. ADRs are immutable; if a decision needs changing, a new ADR supersedes the previous one.

ADR Template:

- Title: Short noun phrase
- Status: Proposed, Accepted, Deprecated, Superseded
- Context: Problem statement
- Decision: The change being proposed
- Consequences: Trade-offs and implications

12.3 Documentation as Code

Documentation must be versioned alongside code. A schema change without corresponding documentation update is an invalid commit. Use pre-commit hooks to enforce:

- Migration files must have matching ADR or changelog entry
- API changes must update OpenAPI spec
- New tables must have RLS policy documentation

SECTION 13: Anti-Pattern List (What We Will NEVER Do Again)

13.1 Database & Schema Anti-Patterns

- Repository-as-Truth Fallacy: Assuming migration files match production state
- Premature RLS Optimization: Implementing security policies before schema stabilizes
- Inconsistent Soft Delete Strategy: Mixing is_deleted booleans with deleted_at timestamps
- Missing Indexes on RLS Predicates: Causing sequential scans on every query
- Global Unique Constraints: Allowing tenant reconnaissance via email existence tests

13.2 Security Anti-Patterns

- Forgetting to Enable RLS: 83% of Supabase breaches involve this mistake
- Application-Layer Security as Primary Defense: Trusting frontend filtering
- God-Mode Admin Design: Over-privileged service_role usage in APIs
- Weak Staff Authentication: PIN-only without device pinning
- Unscoped Anonymous Access: Allowing customers to see all restaurants

13.3 AI Workflow Anti-Patterns

- Context Pollution: Mixing blueprint documents with production snapshots
- Documentation Drift: Multiple AI tools using different architectural intents
- Context Exhaustion: Sessions exceeding 50,000 tokens without reset
- Probabilistic Reasoning: AI claiming completion without deterministic verification
- Manual Override Syndrome: Hotfixes bypassing version control

13.4 DevOps Anti-Patterns

- Manual Production Edits: Shadow logic not reflected in architecture
- Missing Rollback Plans: Migrations without tested revert scripts
- No Drift Detection: Failing to monitor production vs codebase divergence
- Credentials in Git: Production secrets in repository or CI logs

13.5 Performance Anti-Patterns

- Analytical Queries on OLTP Database: Long-running reports blocking kitchen operations
- Complex Subqueries in RLS: 5x-11x performance degradation
- No Connection Pool Reset: Session variable contamination between tenants
- Unpartitioned Large Tables: Sequential scans on millions of rows

SECTION 14: Future Scaling Strategy

14.1 Hybrid Isolation Model

The architecture supports seamless transition from shared to dedicated infrastructure:
Standard Tenants (1-100): Shared schema with strict RLS isolation
Enterprise Tenants (High-Volume): Sharded to dedicated Supabase projects to eliminate Noisy Neighbor effects
Registry Shard: Tenant metadata determines project region at provisioning for GDPR/HIPAA compliance

14.2 OLAP Separation

Analytical processing must be completely separated from transactional operations:
Transactional (OLTP): High-frequency operations served by indexed tables with <50ms RLS overhead
Analytical (OLAP): Long-running reports use materialized views or read-replicas
Rule: No analytical query runs against primary transactional database during Kitchen Rush hours

14.3 Data Lifecycle Management

Hot Storage (0-90 days): Active transactional data in primary database
Warm Storage (91 days - 48 months): Historical data in time-partitioned tables for reporting
Cold Storage (48 months - 7-10 years): Archived data in object storage for compliance
Automated Lifecycle: Background reaper service moves data through tiers based on age

14.4 Global Data Residency

Support multi-region deployment with data sovereignty:
EU data stays in Frankfurt
US data stays in US regions
APAC data stays in Singapore/Tokyo
RLS policies and audit logs remain within regional project

SECTION 15: Risk Map & Failure Points

15.1 Performance Risks

RLS Evaluation Overhead: Complex subqueries can multiply query costs by 5x-11x. Mitigation: Use cached predicates and direct column matches.
Noisy Neighbor: One tenant's heavy queries degrade performance for all. Mitigation: Per-tenant rate limiting and query timeout enforcement.
Connection Pool Exhaustion: High tenant count can exhaust connections. Mitigation: Transaction-level pooling with PgBouncer.
Index Fragmentation: High-frequency updates cause bloat. Mitigation: Regular VACUUM and REINDEX operations.

15.2 Security Risks

Silent RLS Failures: Improperly written policies return zero rows instead of errors. Mitigation: Comprehensive test suite with negative test cases.
Optimizer Statistics Leakage: PostgreSQL optimizer can occasionally leak sampled data from hidden rows. Mitigation: Disable query plan caching for sensitive tables.
Table Ownership Changes: Unauthorized changes to table ownership can bypass RLS. Mitigation: Automated drift detection in CI.
Credential Exposure: service_role key in client code enables total database takeover. Mitigation: Banned from frontend, restricted to Edge Functions only.

15.3 Operational Risks

Schema Drift: Manual hotfixes create divergence between code and production. Mitigation: Automated CI drift detection blocks deployment.
Migration Rollback Failures: Untested rollback scripts cause downtime. Mitigation: Mandatory rollback testing in staging.
Documentation Drift: AI agents using outdated architecture. Mitigation: Documentation as code with version locking.
Context Exhaustion: AI sessions lose accuracy above 50,000 tokens. Mitigation: Hard context limits with forced resets.

15.4 Business Continuity Risks

Data Loss: Database corruption or ransomware. Mitigation: Point-in-Time Recovery with 30-day retention, daily snapshots.
Regional Outage: Cloud provider failure. Mitigation: Multi-region failover capability, cross-region replication.
Key Person Risk: Loss of architectural knowledge. Mitigation: Comprehensive ADRs, architectural documentation as code.

SECTION 16: Operational Automation with Antigravity + MCP + Supabase

16.1 MCP Production Inspection Loop

Model Context Protocol provides AI agents secure, read-only access to live production:
Grounded Reasoning: Agents fetch current table structure and active RLS policies before suggesting changes
Drift Detection: Compare live database against blueprint.md, trigger Red Alert for human remediation
Security Advisors: Automated scanning for tables without RLS or overly permissive policies

16.2 Antigravity Verification Pipeline

Browser-integrated verification eliminates trust gaps:
Performance Verification: Screenshot of browser performance tab proving <2s menu load
Visual Proof: Recording of user flows demonstrating feature completion
Console Error Detection: Automatic analysis of 403 Forbidden errors to suggest RLS policy fixes
Self-Healing: Agent empowered to fix detected issues based on live system logs

16.3 Automated Health Checks

Post-deployment verification suite runs automatically:
RLS Policy Verification: Check all tables have enabled RLS
Index Verification: Confirm all RLS predicate columns are indexed
Performance Benchmark: Execute standard queries, fail if >50ms RLS overhead
Audit Log Integrity: Verify triggers are active and logs are append-only
Connection Pool Status: Check for session variable contamination

SECTION 17: Human Error Reduction Model

17.1 Pre-Commit Hooks

Automated validation before code reaches repository:
Migration files must have matching ADR or changelog entry
New tables must include RLS policy template
service_role key usage in non-Edge Function code blocks commit
Hardcoded credentials or production URLs prevent commit

17.2 CI/CD Gates

Automated quality gates in continuous integration:
Schema Drift Detection: Fail build if production diverges from codebase
Security Advisor Scan: Block deployment on High/Critical findings
Performance Regression: Fail if RLS benchmark exceeds 50ms
Test Coverage: Require 80% coverage for transactional logic
Documentation Sync: Verify ADRs match schema changes

17.3 Blameless Post-Mortems

When incidents occur, focus on system improvement not individual blame:
Document root cause with ADR
Implement automated prevention (pre-commit hook, CI gate, runtime check)
Share learnings across team via changelog
Update AI agent instructions to prevent recurrence

SECTION 18: Version Control & Branch Discipline

18.1 Branch Strategy

main: Production-ready code only, requires two approvals
staging: Pre-production testing, mirrors production environment
feature/*: Individual feature branches, merge to staging first
hotfix/*: Emergency production fixes, requires post-deployment ADR

18.2 Commit Message Standards

Follow Conventional Commits specification:

feat: New feature for the user
fix: Bug fix for the user
docs: Documentation only changes
refactor: Code change that neither fixes a bug nor adds a feature
perf: Performance improvement
test: Adding missing tests or correcting existing tests

18.3 Code Review Standards

All pull requests must address:
Security: Are RLS policies correct? Is service_role usage justified?
Performance: Will this scale to 100+ tenants? Are indexes present?
Testability: Can this be tested in isolation? Are test cases included?
Documentation: Is the ADR updated? Does CLAUDE.md reflect changes?
Rollback: Can this change be reverted safely?

SECTION 19: Security Enforcement Hierarchy

19.1 Defense in Depth Layers

Security implemented at multiple layers (database is authoritative):
Database Layer (Primary): Row-Level Security policies enforce tenant isolation
Edge Function Layer: SECURITY DEFINER functions verify JWT claims before mutations
API Layer: PostgREST automatically applies RLS, no custom middleware needed
Application Layer: UI/UX improvements only, never trusted for security decisions

19.2 Authentication Assurance Levels

Different operations require different authentication strength:
aal0 (No Auth): Public menu viewing only
aal1 (Standard Auth): Password or PIN for regular operations
aal2 (MFA Required): TOTP for high-risk operations (delete restaurant, access audit logs, billing changes)

19.3 Audit Trail Requirements

All security-relevant actions must be logged:
Authentication attempts (success and failure)
Permission changes (role assignments, RLS policy modifications)

Data access (who viewed sensitive PII)
Schema changes (DDL operations)
Admin actions (restaurant activation, suspension)

SECTION 20: Deployment Discipline & Rollback Protocol

20.1 Deployment Windows

Standard Deployments: Tuesday-Thursday, 10am-2pm (non-peak hours)
Emergency Hotfixes: Any time, requires post-deployment retrospective
Banned Windows: Friday-Sunday (weekend), lunch/dinner rush hours (11am-2pm, 5pm-9pm)

20.2 Deployment Checklist

Pre-deployment verification:
All CI checks passing (drift detection, security advisors, tests)
Code review approved by two team members
Migration tested in staging with production-volume data
Rollback script validated in staging
Backup snapshot created immediately before deployment
Monitoring dashboards ready for anomaly detection

20.3 Rollback Protocol

If deployment issues detected within 30 minutes:
Immediate Rollback: Execute validated rollback script within 5 minutes
Health Check: Verify all tenants operational, RLS policies active
Incident Report: Create ADR documenting failure cause and prevention
Remediation: Fix issue in feature branch, re-test in staging
Post-Mortem: Blameless review within 48 hours

20.4 Disaster Recovery

Monthly disaster recovery drill:
Restore production database from PITR snapshot to fresh Supabase project
Verify RLS isolation and data integrity across all tenants
Prove Recovery Time Objective (RTO) < 1 hour
Prove Recovery Point Objective (RPO) < 5 minutes
Document any gaps and update runbook

LESSONS EXTRACTED FROM PHASE-4 MISTAKES

This section documents concrete mistakes from previous implementation cycles and how V2 prevents them structurally.

Mistake 1: Repository-as-Truth Fallacy

What Happened: Architects assumed migration files in the repository matched production schema, leading to silent deployment failures when indexes were missing or constraints were outdated.
V2 Prevention: Automated CI drift detection. Pipeline introspects live production via MCP, diffs against codebase, and fails build if discrepancies exist. Production is now the definitive source of truth.

Mistake 2: Forgetting to Enable RLS

What Happened: New tables defaulted to publicly accessible state. 83% of Supabase breaches involve RLS misconfigurations.
V2 Prevention: Security Advisors integrated into CI pipeline. Any table without RLS or with overly permissive policies (USING true) automatically blocks deployment.

Mistake 3: Application-Layer Security as Primary Defense

What Happened: Middleware and frontend filtering were treated as primary defense, leading to God-mode bypasses when users accessed the database via direct REST or GraphQL endpoints.
V2 Prevention: RLS is the only security boundary. All queries automatically filtered by database-level policies. Application layer provides UX improvements only.

Mistake 4: Documentation Chaos

What Happened: Mixing blueprint documents with production snapshots caused AI agents to hallucinate based on outdated architectural intents. Multiple versions of truth created fragmentation.
V2 Prevention: Single Source of Truth Hierarchy with strict separation. Blueprints in /docs/architecture, production snapshots in /docs/snapshots. Documentation as code with version locking.

Mistake 5: Context Exhaustion in AI Sessions

What Happened: As projects grew, comprehensive instructions exhausted context windows, causing AI agents to forget constraints and produce code that deviated from established conventions.
V2 Prevention: Sharded context strategy with hard limits. Agents restricted to <150 lines per context window. Hard block at 85% usage forcing session reset with only MANIFEST.yaml re-injection.

Mistake 6: Manual Production Hotfixes

What Happened: Emergency fixes via Supabase Dashboard bypassed version control, creating Shadow Logic not reflected in architecture. Next migration overwrote critical production changes.
V2 Prevention: Manual production edits are banned. All changes follow versioned migration pipeline. Drift detection identifies unauthorized changes. Hotfixes require immediate post-deployment ADR.

Mistake 7: Inconsistent Soft Delete Strategy

What Happened: Mixed use of is_deleted booleans with deleted_at timestamps caused data fragmentation, broke unique constraints, and compromised referential integrity.
V2 Prevention: Standardized deleted_at timestamp pattern with partial indexes for uniqueness. Cascading soft deletes via database triggers maintain referential integrity.

Mistake 8: Missing Indexes on RLS Predicates

What Happened: Indexed for business logic but ignored RLS predicate columns. Every security check triggered sequential scan, multiplying latency by active user count.
V2 Prevention: Strategic indexing requirement: every column in RLS policy must be indexed. Automated performance benchmarks fail if RLS overhead exceeds 50ms.

Mistake 9: Analytical Queries on OLTP Database

What Happened: Long-running reports executed against active order tables during Kitchen Rush hours. Share-locks prevented staff from marking orders ready, causing UI freezes.
V2 Prevention: Strict OLTP/OLAP separation. Analytical queries use materialized views or read-replicas. Rule: no analytical query runs against primary database during peak hours.

Mistake 10: Over-Privileged Platform Admin

What Happened: Platform admins had God-mode access to all restaurant data including orders, menu, pricing, and customer PII. Created liability during security audits.

V2 Prevention: Platform Admin strictly prohibited from operational data. Limited to tenant lifecycle management only (onboarding, activation, suspension). RLS policies enforce isolation at database level.

IMPLEMENTATION ROADMAP

This roadmap provides the execution flow for implementing the V2 Architecture from ground zero to production-ready platform.

Phase 1: Foundation (Weeks 1-2)

Set up environment separation (Local, Staging, Production)
Configure Supabase projects with distinct credentials
Establish documentation hierarchy (docs/architecture, docs/decisions, CLAUDE.md)
Create first ADRs documenting architectural decisions
Set up pre-commit hooks and CI/CD pipeline

Phase 2: Core Schema (Weeks 3-4)

Design tenant isolation model (restaurant_id everywhere)
Create core tables: tenants, users, staff, menu, categories, products
Implement standardized soft delete with deleted_at timestamp
Create audit log infrastructure with trigger-based logging
Set up partition strategy for time-series data

Phase 3: RLS Security (Weeks 5-6)

Enable RLS on all tables
Create policies using cached predicate pattern
Index all RLS predicate columns
Configure Security Advisors in CI pipeline
Create comprehensive test suite with negative test cases
Performance benchmark: verify <50ms RLS overhead

Phase 4: Authentication & Authorization (Weeks 7-8)

Implement JWT claim structure (restaurant_id, app_role)
Create Auth Hooks for claim injection
Build Restaurant Admin authentication (Restaurant ID + Password)
Build Kitchen Staff authentication (PIN + device pinning)
Build Customer anonymous authentication

Build Platform Admin authentication with MFA

Phase 5: Business Logic (Weeks 9-11)

Implement restaurant onboarding workflow
Create activation code model
Build menu management (CRUD operations)
Implement table session management
Create order processing workflow
Build Kitchen Display System (KDS) real-time updates

Phase 6: Performance & Monitoring (Weeks 12-13)

Implement OLTP/OLAP separation
Create materialized views for reporting
Set up connection pooling with PgBouncer
Configure per-tenant rate limiting
Implement automated drift detection
Set up real-time monitoring dashboards

Phase 7: AI Orchestration (Weeks 14-15)

Configure MCP production inspection
Set up Antigravity verification pipeline
Implement sharded context strategy
Create 16-phase state machine for complex features
Build artifact-driven trust system

Phase 8: Testing & Validation (Weeks 16-17)

Load test with 100+ simulated tenants
Security penetration testing
RLS policy audit (verify no cross-tenant leakage)
Performance verification (<2s menu load, <50ms RLS overhead)
Disaster recovery drill

Phase 9: Production Preparation (Weeks 18-19)

Create runbooks for common operations
Document emergency procedures
Set up automated backups with PITR

Configure alerting for anomalies
Conduct team training on V2 architecture

Phase 10: Launch & Monitor (Week 20+)

Soft launch with pilot restaurants
Monitor performance metrics and error rates
Gather feedback and create improvement ADRs
Scale to 100+ tenants gradually
Continuous improvement based on real-world data

FINAL DECLARATION

This Project Vision Document V2 represents the constitutional architecture for befoodi. It is the culmination of lessons learned, mistakes corrected, and principles established through rigorous analysis of multi-tenant SaaS failures.

Core Principles

Security is a Database Property
Production is the Source of Truth
Verification Beats Generation
Performance is a Constraint
Discipline is the Moat

Rules of Conformance

RLS is the only security boundary
Production state is the definitive reference
Documentation must be versioned with code
All changes follow the migration pipeline
AI agents require deterministic verification
Scalability is designed from Day 1

Conformance to this baseline is mandatory. Deviations are architectural errors. Manual overrides are governance failures. This framework ensures a secure, scalable, and investor-ready platform capable of supporting the next 100+ tenants with zero architectural degradation.

Phase-4 V2 Architecture — Locked Baseline

February 2026