Enterprise Blueprint

Multi-Tenant SaaS CRM Platform

A complete technical specification covering product requirements, system architecture, database schemas, service interfaces, security policies, and a fully sequenced implementation plan.

Prepared by	Specloop — Documentation Division
Document Type	Specloop-Generated Blueprint
Classification	Sample / Anonymized
Serial	SL-2026-0004
Date	April 2026

This document is an anonymized sample. The underlying platform, client details, and proprietary business logic have been redacted. The technical depth is real.

Executive Overview

What This Document Is and How It Was Created

This Enterprise Blueprint is a complete technical specification for a production-grade software platform. It is not a proposal, not a pitch deck, and not a rough estimate. It is the engineering document that a development team uses to build software with zero ambiguity.

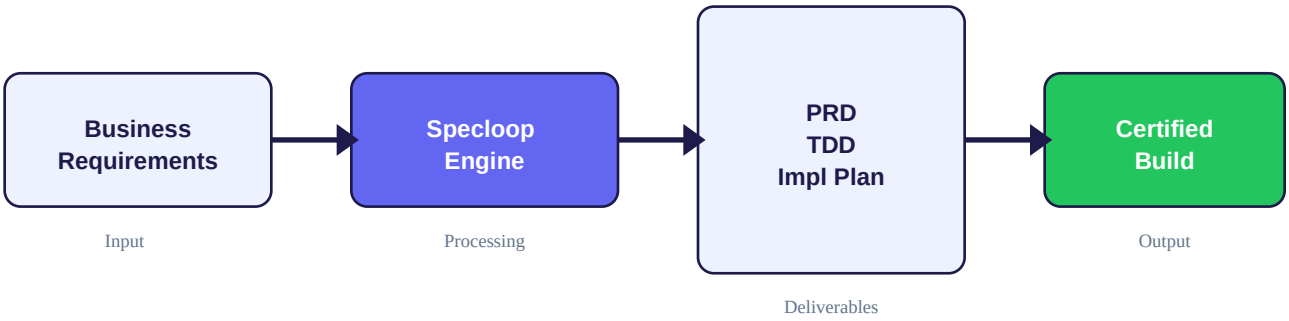
Fewer than 10% of development firms produce anything close to this level of pre-build documentation. Most projects begin with a 2-page proposal and a verbal handshake — then spend months discovering requirements that should have been defined on day one. The cost of that discovery is measured in missed deadlines, budget overruns, and features that don't work the way anyone expected.

Specloop eliminates that risk by replacing **guesswork** with a precise specification. Specloop takes your business requirements as input and generates three interlocking deliverables: a Product Requirements Document (PRD) that defines *what* the platform must do, a Technical Design Document (TDD) that defines *how* it is built, and an Implementation Plan that defines *when* and *in what order* every component is constructed and tested.

The result is a document where every acceptance criterion traces to a database schema, every schema traces to a service interface, and every service interface traces to a numbered task in the implementation plan. Nothing is left to interpretation. Nothing is 'figured out later.'

How Specloop Works

Specloop Process Flow



The three sections of this blueprint are designed to be read in sequence. The PRD establishes the requirements. The TDD translates those requirements into architecture. The Implementation Plan sequences the build so that foundational components are completed and tested before dependent features begin.

> ***This sample blueprint documents a Multi-Tenant SaaS CRM platform — a system with tenant isolation, role-based access control, pipeline management, webhook integrations, and a 43-point automated QA matrix. The technical depth is real; the client details have been anonymized.***

Table of Contents

Section 1

Product Requirements Document (PRD)

The Product Requirements Document defines **what** the platform must do. It is the authoritative source of truth for every feature, every validation rule, and every edge case.

Specloop analyzes your business logic and decomposes it into testable, numbered acceptance criteria. Each criterion is written as a SHALL statement — a binary, testable condition that either passes or fails. There is no room for 'it should kind of work.' Every requirement has a priority level, and every criterion maps to specific test cases in the Implementation Plan.

This PRD contains **20 requirements** with **160+ individual acceptance criteria** covering tenant provisioning, authentication, RBAC, contact and deal management, webhooks, search, reporting, and more. Nothing gets 'discovered' mid-build. If it's not in here, it's not in scope.

Glossary of Terms

Term	Definition
Tenant	An isolated organizational unit (company) within the platform.
RLS	Row-Level Security — PostgreSQL policies that restrict data access per-row.
RBAC	Role-Based Access Control — permissions model with admin, manager, member, viewer roles.
Edge Function	Server-side TypeScript function deployed on Supabase infrastructure.
PRD	Product Requirements Document — defines what the system must do.
TDD	Technical Design Document — defines how the system is built.
Specloop	Specloop's AI-powered engine for generating enterprise blueprints.
Pipeline	A configurable workflow of stages that deals progress through.
Stage	A step within a pipeline (e.g., 'Qualification', 'Negotiation', 'Closed Won').
Activity	A logged interaction — call, email, meeting, note — attached to a contact or deal.

Webhook	An HTTP callback triggered by system events, signed with HMAC-SHA256.
Custom Field	A tenant-defined field that extends a standard entity schema.
Audit Log	Immutable record of every create, update, and delete operation.
Import/Export	Bulk data operations via CSV with validation and rollback support.
QA Matrix	The 43-point automated test suite that validates every requirement.
Acceptance Criterion	A testable condition that must be true for a requirement to pass.
Checkpoint Gate	A milestone where all preceding tests must pass before proceeding.
Slug	A URL-safe tenant identifier (e.g., 'acme-corp') validated via regex.
HMAC-SHA256	Hash-based Message Authentication Code used for webhook signing.
Sliding Window	Rate limiting algorithm that tracks requests in a rolling time window.
Monetary Value	Currency amounts stored as integers (cents) to avoid floating-point errors.
Service Interface	A TypeScript type definition for a module's public API contract.
Migration	A versioned SQL script that modifies database schema.
Idempotent	An operation that produces the same result when applied multiple times.
Seed Data	Initial data loaded into the system for development and testing.

Requirements

REQ-001: Tenant Provisioning & Isolation [P0 — Critical]

REQ-001.01 System SHALL create a new tenant record with a unique UUID upon registration.

REQ-001.02 System SHALL generate a URL-safe slug from the company name, validated against ^[a-z0-9][a-z0-9-]{2,62}$.

REQ-001.03 System SHALL reject duplicate slugs with a 409 Conflict response.

REQ-001.04 System SHALL create default pipeline with stages: Lead, Qualified, Proposal, Negotiation, Closed Won, Closed Lost.

REQ-001.05 System SHALL provision tenant-specific RLS policies within 500ms of creation.

REQ-001.06 System SHALL set the creating user as tenant admin with full permissions.

REQ-001.07 System SHALL create an audit log entry for tenant creation.

REQ-001.08 System SHALL enforce complete data isolation — no cross-tenant data leakage under any query pattern.

REQ-001.09 System SHALL support soft-delete of tenants with 30-day recovery window.

REQ-002: Authentication & Session Management [P0 — Critical]

REQ-002.01 System SHALL support email/password authentication via Supabase Auth.

REQ-002.02 System SHALL support OAuth 2.0 with Google and Microsoft providers.

REQ-002.03 System SHALL enforce minimum password complexity: 8 chars, 1 uppercase, 1 number, 1 special.

REQ-002.04 System SHALL issue JWT tokens with 1-hour expiry and automatic refresh.

REQ-002.05 System SHALL include tenant_id and role in JWT claims.

REQ-002.06 System SHALL lock accounts after 5 failed login attempts for 15 minutes.

REQ-002.07 System SHALL support 'Remember me' with 30-day persistent sessions.

REQ-002.08 System SHALL log all authentication events to the audit log.

REQ-002.09 System SHALL support password reset via email with 1-hour token expiry.

REQ-002.10 System SHALL terminate all active sessions on password change.

REQ-002.11 System SHALL support multi-tenant user switching without re-authentication.

REQ-002.12 System SHALL enforce HTTPS-only session cookies with SameSite=Strict.

REQ-003: Role-Based Access Control (RBAC) [P0 — Critical]

REQ-003.01 System SHALL support four roles: admin, manager, member, viewer.

REQ-003.02 Admin SHALL have full CRUD on all entities within their tenant.

REQ-003.03 Manager SHALL have full CRUD on contacts, accounts, deals, and activities.

REQ-003.04 Member SHALL have CRUD on own records and read access to team records.

REQ-003.05 Viewer SHALL have read-only access to all entities.

REQ-003.06 System SHALL enforce RBAC at both API and database (RLS) levels.

REQ-003.07 System SHALL allow admins to assign and revoke roles.

REQ-003.08 System SHALL prevent the last admin from being demoted or removed.

REQ-003.09 System SHALL log all role changes to the audit log.

REQ-003.10 System SHALL evaluate permissions in under 5ms per request.

REQ-004: Contact Management [P0 — Critical]

REQ-004.01 System SHALL store contacts with: first_name, last_name, email, phone, company, title, source, status.

REQ-004.02 System SHALL enforce unique email per tenant.

REQ-004.03 System SHALL support custom fields (text, number, date, select, multi-select).

REQ-004.04 System SHALL support contact tagging with unlimited tags.

REQ-004.05 System SHALL track contact lifecycle: new, contacted, qualified, customer, churned.

REQ-004.06 System SHALL support bulk import via CSV with validation and error reporting.

REQ-004.07 System SHALL support bulk export to CSV with column selection.

REQ-004.08 System SHALL display full activity timeline on contact detail view.

REQ-004.09 System SHALL support contact merging with conflict resolution.

REQ-004.10 System SHALL support contact assignment to team members.

REQ-005: Account (Company) Management [P1 — High]

REQ-005.01 System SHALL store accounts with: name, domain, industry, size, revenue, address.

REQ-005.02 System SHALL support parent-child account hierarchies (max 5 levels).

REQ-005.03 System SHALL auto-link contacts to accounts via email domain matching.

REQ-005.04 System SHALL display account health score based on activity recency and deal value.

REQ-005.05 System SHALL show all associated contacts, deals, and activities on account detail.

REQ-005.06 System SHALL support account segmentation by industry, size, and custom fields.

REQ-005.07 System SHALL enforce unique account name per tenant.

REQ-006: Deal Management [P0 — Critical]

REQ-006.01 System SHALL store deals with: name, value, stage, probability, expected_close_date, owner.

REQ-006.02 System SHALL store monetary values as integers (cents) to prevent floating-point errors.

REQ-006.03 System SHALL associate deals with exactly one pipeline and one stage.

REQ-006.04 System SHALL track stage history with timestamps for velocity reporting.

REQ-006.05 System SHALL support weighted pipeline value: SUM(value * probability).

REQ-006.06 System SHALL enforce stage-specific required fields (configurable per pipeline).

REQ-006.07 System SHALL support deal cloning with optional field overrides.

REQ-006.08 System SHALL trigger webhook events on stage transitions.

REQ-006.09 System SHALL prevent moving deals to 'Closed Won' without a primary contact.

REQ-006.10 System SHALL support multi-currency with configurable base currency per tenant.

REQ-007: Pipeline Configuration [P1 — High]

REQ-007.01 System SHALL support multiple pipelines per tenant.

REQ-007.02 System SHALL allow custom stages with: name, order, probability, color, required_fields.

REQ-007.03 System SHALL enforce sequential stage ordering (no gaps in order values).

REQ-007.04 System SHALL prevent deletion of stages containing active deals.

REQ-007.05 System SHALL support drag-and-drop stage reordering.

REQ-007.06 System SHALL provide a default pipeline on tenant creation.

REQ-007.07 System SHALL support pipeline-level win/loss rate analytics.

REQ-007.08 System SHALL allow stage-specific automation triggers.

REQ-008: Activity Logging [P1 — High]

REQ-008.01 System SHALL support activity types: call, email, meeting, note, task.

REQ-008.02 System SHALL associate activities with contacts, accounts, and/or deals.

REQ-008.03 System SHALL track: type, subject, description, scheduled_at, completed_at, duration.

REQ-008.04 System SHALL support recurring activities with configurable frequency.

REQ-008.05 System SHALL provide activity feed with filters by type, date range, and entity.

REQ-008.06 System SHALL send reminders for upcoming scheduled activities.

REQ-008.07 System SHALL support activity templates for common interactions.

REQ-009: Webhook System [P1 — High]

REQ-009.01 System SHALL support configurable webhooks per tenant.

REQ-009.02 System SHALL sign payloads with HMAC-SHA256 using a per-webhook secret.

REQ-009.03 System SHALL support events: contact.created, contact.updated, deal.stage_changed, deal.won, deal.lost.

REQ-009.04 System SHALL retry failed deliveries with exponential backoff (max 5 retries).

REQ-009.05 System SHALL log delivery status: pending, delivered, failed, retrying.

REQ-009.06 System SHALL timeout webhook calls after 10 seconds.

REQ-009.07 System SHALL allow pausing and resuming individual webhooks.

REQ-009.08 System SHALL provide a test endpoint for webhook configuration validation.

REQ-010: Import / Export System [P1 — High]

REQ-010.01 System SHALL support CSV import for contacts, accounts, and deals.

REQ-010.02 System SHALL validate all rows before committing any data (atomic import).

REQ-010.03 System SHALL report row-level errors with line numbers and field details.

REQ-010.04 System SHALL support field mapping (CSV column → entity field).

REQ-010.05 System SHALL detect and skip duplicate records based on configurable keys.

REQ-010.06 System SHALL support CSV export with selectable columns and filters.

REQ-010.07 System SHALL limit import batch size to 10,000 rows per operation.

REQ-010.08 System SHALL create an audit log entry for each import/export operation.

REQ-011: Reporting & Analytics [P1 — High]

REQ-011.01 System SHALL provide dashboard with: total contacts, active deals, pipeline value, win rate.

REQ-011.02 System SHALL support pipeline velocity report: average time per stage.

REQ-011.03 System SHALL support activity report: activities by type, user, and date range.

REQ-011.04 System SHALL support revenue forecast based on weighted pipeline.

REQ-011.05 System SHALL support custom date range selection for all reports.

REQ-011.06 System SHALL cache report data with 5-minute TTL.

REQ-011.07 System SHALL support CSV export of any report.

REQ-012: Search & Filtering [P1 — High]

REQ-012.01 System SHALL support full-text search across contacts, accounts, and deals.

REQ-012.02 System SHALL return results in under 200ms for datasets up to 100,000 records.

REQ-012.03 System SHALL support compound filters: field + operator + value, with AND/OR logic.

REQ-012.04 System SHALL support saved filters (views) per user.

REQ-012.05 System SHALL highlight matching terms in search results.

REQ-012.06 System SHALL support search within custom fields.

REQ-013: Custom Fields [P2 — Medium]

REQ-013.01 System SHALL support field types: text, number, date, select, multi-select, boolean.

REQ-013.02 System SHALL allow custom fields on contacts, accounts, and deals.

REQ-013.03 System SHALL enforce field validation rules (required, min/max, regex pattern).

REQ-013.04 System SHALL support field ordering on forms.

REQ-013.05 System SHALL include custom fields in search, filtering, and export.

REQ-013.06 System SHALL limit custom fields to 50 per entity type per tenant.

REQ-014: Audit Logging [P0 — Critical]

REQ-014.01 System SHALL log all create, update, and delete operations.

REQ-014.02 System SHALL record: user_id, tenant_id, entity_type, entity_id, action, old_value, new_value, timestamp.

REQ-014.03 System SHALL make audit logs immutable (no update or delete).

REQ-014.04 System SHALL support audit log search by entity, user, action, and date range.

REQ-014.05 System SHALL retain audit logs for minimum 2 years.

REQ-014.06 System SHALL support audit log export to CSV.

REQ-015: Rate Limiting [P1 — High]

REQ-015.01 System SHALL enforce per-tenant rate limits using sliding window algorithm.

REQ-015.02 System SHALL default to 1,000 requests per minute per tenant.

REQ-015.03 System SHALL return 429 Too Many Requests with Retry-After header.

REQ-015.04 System SHALL allow configurable limits per tenant (admin override).

REQ-015.05 System SHALL exempt health check and status endpoints.

REQ-016: Email Integration [P2 — Medium]

REQ-016.01 System SHALL support sending emails via configured SMTP or Resend API.

REQ-016.02 System SHALL track email open and click events.

REQ-016.03 System SHALL support email templates with variable substitution.

REQ-016.04 System SHALL log all sent emails as activities on the associated contact.

REQ-016.05 System SHALL enforce sending limits: 500 emails per day per tenant.

REQ-017: Data Validation & Integrity [P0 — Critical]

REQ-017.01 System SHALL validate all inputs at both client and server layers.

REQ-017.02 System SHALL use Zod schemas for runtime type validation.

REQ-017.03 System SHALL enforce referential integrity via foreign keys.
REQ-017.04 System SHALL prevent orphaned records on cascading deletes.
REQ-017.05 System SHALL validate email format using RFC 5322 compliant regex.
REQ-017.06 System SHALL sanitize all text inputs against XSS attacks.
REQ-017.07 System SHALL enforce maximum field lengths at database level.

REQ-018: Performance Requirements [P0 — Critical]

REQ-018.01 API responses SHALL complete in under 200ms at P95.
REQ-018.02 Page load (Time to Interactive) SHALL be under 2 seconds.
REQ-018.03 System SHALL support 100 concurrent users per tenant without degradation.
REQ-018.04 Database queries SHALL use indexes for all filtered and sorted columns.
REQ-018.05 System SHALL implement connection pooling with max 20 connections per tenant.
REQ-018.06 System SHALL lazy-load non-critical UI components.

REQ-019: Accessibility & Internationalization [P2 — Medium]

REQ-019.01 System SHALL comply with WCAG 2.1 Level AA.
REQ-019.02 System SHALL support keyboard navigation for all interactive elements.
REQ-019.03 System SHALL provide ARIA labels for all form inputs and buttons.
REQ-019.04 System SHALL support date/time formatting per tenant locale.
REQ-019.05 System SHALL support RTL text layout for applicable locales.

REQ-020: Deployment & Operations [P1 — High]

REQ-020.01 System SHALL deploy via CI/CD pipeline with automated testing.
REQ-020.02 System SHALL support zero-downtime deployments.
REQ-020.03 System SHALL provide health check endpoint at /api/health.
REQ-020.04 System SHALL report errors to Sentry with tenant context.
REQ-020.05 System SHALL support environment-based configuration (dev, staging, production).
REQ-020.06 System SHALL create database backups every 6 hours with 30-day retention.
REQ-020.07 System SHALL monitor uptime with 99.9% SLA target.

Section 2

Technical Design Document (TDD)

The Technical Design Document defines **how** the platform is built. Once the PRD is finalized and locked, Specloop generates the complete architecture — database schemas, service interfaces, security policies, and algorithm implementations.

This section is the contract between the blueprint and the codebase. Every table, every API endpoint, every security policy documented here will exist in the production system exactly as specified. Developers don't make architectural decisions during implementation — those decisions are made here, reviewed, and approved before a single line of application code is written.

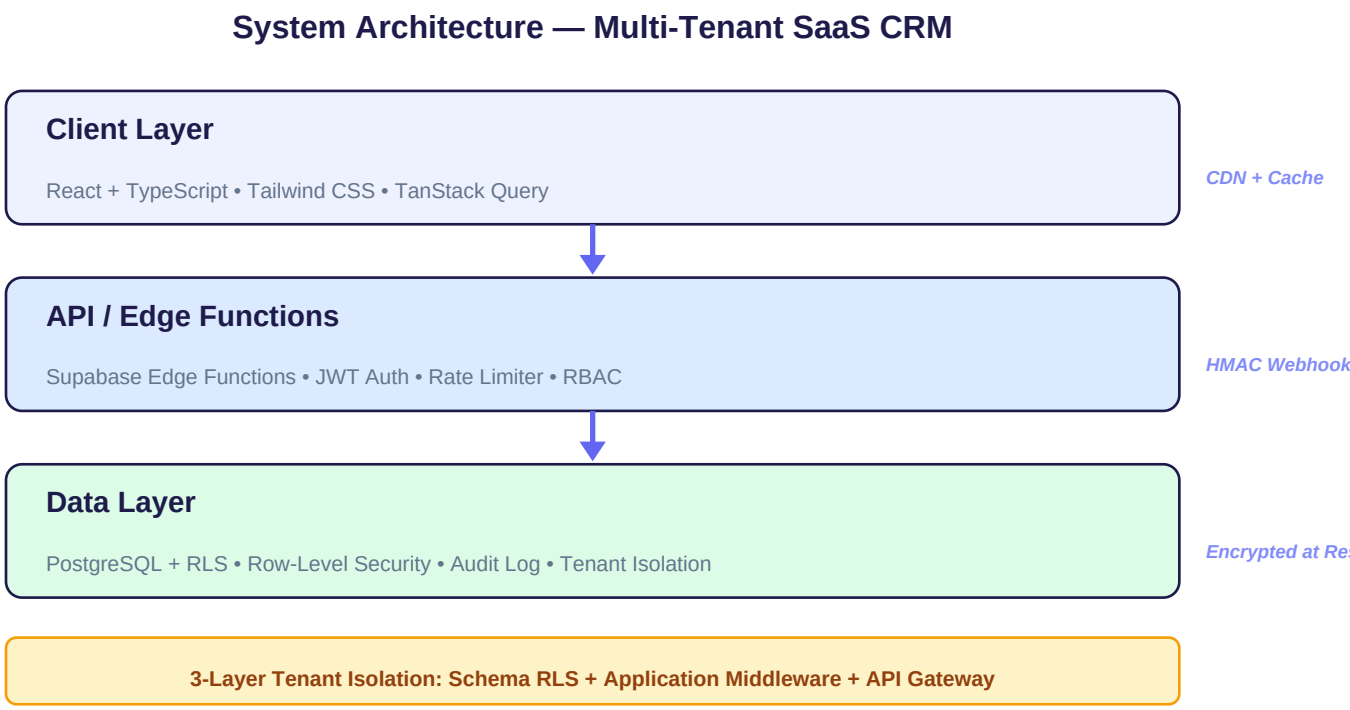
The TDD covers the full technology stack, a three-layer tenant isolation strategy, typed service interfaces for every major module, complete SQL schemas with indexes, Row-Level Security policies, and reference implementations for critical algorithms including rate limiting, RBAC evaluation, and webhook signing.

Technology Stack

Layer	Technology	Purpose
Frontend	React 18 + TypeScript 5	Component-based UI with strict typing
Styling	Tailwind CSS v3	Utility-first CSS with design tokens
State	TanStack Query v5	Server state with cache invalidation
Backend	Supabase Edge Functions	Deno-based serverless TypeScript
Database	PostgreSQL 15	ACID-compliant with RLS
Auth	Supabase Auth	JWT + OAuth 2.0 (Google, Microsoft)
Storage	Supabase Storage	S3-compatible object storage
Realtime	Supabase Realtime	WebSocket-based change feeds

Validation	Zod	Runtime schema validation
Testing	Vitest + fast-check	Unit + property-based testing
CI/CD	GitHub Actions	Automated build, test, deploy
Monitoring	Sentry	Error tracking with tenant context

System Architecture



Multi-Tenancy Isolation Strategy

The platform implements a **3-layer defense-in-depth** isolation strategy:

- **Layer 1 — Database (RLS)**: Every table has a Row-Level Security policy that filters rows by tenant_id extracted from the JWT. Even if application code has a bug, the database itself prevents cross-tenant access.
- **Layer 2 — Application Middleware**: Every Edge Function validates the JWT, extracts the tenant context, and injects it into every database query. Requests without valid tenant context are rejected with 403.
- **Layer 3 — API Gateway**: Rate limiting, request validation, and audit logging happen at the gateway level before requests reach application code.

Service Interfaces

Each major module exposes a typed TypeScript interface. These interfaces define the public API contract — the methods, parameters, and return types that the rest of the system depends on.

TenantContext

```
interface TenantContext {
  tenantId: string;
  slug: string;
  userId: string;
  role: 'admin' | 'manager' | 'member' | 'viewer';
  permissions: Permission[];
}
```

AuthService

```
interface AuthService {
  signUp(email: string, password: string, tenantName: string): Promise<AuthResult>;
  signIn(email: string, password: string): Promise<AuthResult>;
  signInWithOAuth(provider: 'google' | 'microsoft'): Promise<AuthResult>;
  signOut(): Promise<void>;
  refreshToken(): Promise<string>;
  resetPassword(email: string): Promise<void>;
  changePassword(oldPassword: string, newPassword: string): Promise<void>;
  getCurrentUser(): Promise<User | null>;
  switchTenant(tenantId: string): Promise<TenantContext>;
}
```

CRMService

```
interface CRMService {
  // Contacts
  getContacts(filters: FilterSet, pagination: Pagination): Promise<PaginatedResult<Contact>>;
  getContact(id: string): Promise<Contact>;
  createContact(data: CreateContactInput): Promise<Contact>;
  updateContact(id: string, data: UpdateContactInput): Promise<Contact>;
  deleteContact(id: string): Promise<void>;
  mergeContacts(primaryId: string, secondaryId: string): Promise<Contact>;
  importContacts(file: File, mapping: FieldMapping): Promise<ImportResult>;
  exportContacts(filters: FilterSet, columns: string[]): Promise<Blob>;
}
```

PipelineService

```
interface PipelineService {
  getPipelines(): Promise<Pipeline[]>;
  getPipeline(id: string): Promise<Pipeline>;
  createPipeline(data: CreatePipelineInput): Promise<Pipeline>;
  updatePipeline(id: string, data: UpdatePipelineInput): Promise<Pipeline>;
  deletePipeline(id: string): Promise<void>;
  reorderStages(pipelineId: string, stageIds: string[]): Promise<void>;
  getStageAnalytics(pipelineId: string, dateRange: DateRange): Promise<StageAnalytics>;
}
```

WebhookService

```
interface WebhookService {
  getWebhooks(): Promise<Webhook[]>;
  createWebhook(data: CreateWebhookInput): Promise<Webhook>;
  updateWebhook(id: string, data: UpdateWebhookInput): Promise<Webhook>;
  deleteWebhook(id: string): Promise<void>;
  testWebhook(id: string): Promise<WebhookTestResult>;
  getDeliveryLog(webhookId: string, pagination: Pagination): Promise<PaginatedResult<DeliveryLo
  retryDelivery(deliveryId: string): Promise<void>;
  pauseWebhook(id: string): Promise<void>;
  resumeWebhook(id: string): Promise<void>;
}
```

ReportService

```
interface ReportService {
  getDashboardMetrics(dateRange: DateRange): Promise<DashboardMetrics>;
  getPipelineVelocity(pipelineId: string, dateRange: DateRange): Promise<VelocityReport>;
  getActivityReport(filters: ActivityFilters): Promise<ActivityReport>;
  getRevenueForecast(dateRange: DateRange): Promise<ForecastReport>;
  exportReport(reportType: string, params: ReportParams): Promise<Blob>;
}
```

SearchService

```
interface SearchService {
  search(query: string, entityTypes: EntityType[], pagination: Pagination): Promise<SearchResul
  getSavedViews(): Promise<SavedView[]>;
  createSavedView(data: CreateSavedViewInput): Promise<SavedView>;
  deleteSavedView(id: string): Promise<void>;
}
```

Database Schemas

Complete SQL CREATE TABLE statements for the core entities. All tables include tenant_id for RLS, timestamps for auditing, and appropriate indexes for query performance.

Table: tenants

```
CREATE TABLE tenants (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  name VARCHAR(255) NOT NULL,
  slug VARCHAR(63) NOT NULL UNIQUE,
  settings JSONB DEFAULT '{}',
  plan VARCHAR(50) DEFAULT 'starter',
  is_active BOOLEAN DEFAULT true,
  deleted_at TIMESTAMPTZ,
  created_at TIMESTAMPTZ DEFAULT now(),
  updated_at TIMESTAMPTZ DEFAULT now()
);

CREATE UNIQUE INDEX idx_tenants_slug ON tenants(slug);
CREATE INDEX idx_tenants_active ON tenants(is_active) WHERE is_active = true;
```

Table: users

```
CREATE TABLE users (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  tenant_id UUID NOT NULL REFERENCES tenants(id),
  email VARCHAR(255) NOT NULL,
  name VARCHAR(255),
  role VARCHAR(50) NOT NULL DEFAULT 'member',
  avatar_url TEXT,
  last_login_at TIMESTAMPTZ,
  is_active BOOLEAN DEFAULT true,
  created_at TIMESTAMPTZ DEFAULT now(),
  updated_at TIMESTAMPTZ DEFAULT now(),
  UNIQUE(tenant_id, email)
);

CREATE INDEX idx_users_tenant ON users(tenant_id);
CREATE INDEX idx_users_email ON users(email);
```

Table: contacts

```
CREATE TABLE contacts (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  tenant_id UUID NOT NULL REFERENCES tenants(id),
  account_id UUID REFERENCES accounts(id),
  first_name VARCHAR(100) NOT NULL,
  last_name VARCHAR(100) NOT NULL,
  email VARCHAR(255),
  phone VARCHAR(50),
  title VARCHAR(255),
  source VARCHAR(100),
  status VARCHAR(50) DEFAULT 'new',
  tags TEXT[] DEFAULT '{}',
  custom_fields JSONB DEFAULT '{}',
  owner_id UUID REFERENCES users(id),
  created_at TIMESTAMPTZ DEFAULT now(),
  updated_at TIMESTAMPTZ DEFAULT now(),
  UNIQUE(tenant_id, email)
);

CREATE INDEX idx_contacts_tenant ON contacts(tenant_id);
CREATE INDEX idx_contacts_account ON contacts(account_id);
CREATE INDEX idx_contacts_owner ON contacts(owner_id);
CREATE INDEX idx_contacts_status ON contacts(tenant_id, status);
CREATE INDEX idx_contacts_search ON contacts
  USING gin(to_tsvector('english', first_name || ' ' || last_name || ' ' || COALESCE(email, '')
```

Table: deals

```
CREATE TABLE deals (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  tenant_id UUID NOT NULL REFERENCES tenants(id),
  pipeline_id UUID NOT NULL REFERENCES pipelines(id),
  stage_id UUID NOT NULL REFERENCES stages(id),
  name VARCHAR(255) NOT NULL,
  value_cents BIGINT NOT NULL DEFAULT 0,
  currency VARCHAR(3) DEFAULT 'USD',
  probability INTEGER DEFAULT 0 CHECK (probability >= 0 AND probability <= 100),
  expected_close_date DATE,
  contact_id UUID REFERENCES contacts(id),
  account_id UUID REFERENCES accounts(id),
  owner_id UUID NOT NULL REFERENCES users(id),
  custom_fields JSONB DEFAULT '{}',
  won_at TIMESTAMPTZ,
  lost_at TIMESTAMPTZ,
  lost_reason TEXT,
  created_at TIMESTAMPTZ DEFAULT now(),
  updated_at TIMESTAMPTZ DEFAULT now()
);

CREATE INDEX idx_deals_tenant ON deals(tenant_id);
CREATE INDEX idx_deals_pipeline ON deals(pipeline_id);
CREATE INDEX idx_deals_stage ON deals(stage_id);
CREATE INDEX idx_deals_owner ON deals(owner_id);
CREATE INDEX idx_deals_close_date ON deals(expected_close_date);
```

Table: audit_log

```
CREATE TABLE audit_log (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  tenant_id UUID NOT NULL REFERENCES tenants(id),
  user_id UUID NOT NULL REFERENCES users(id),
  entity_type VARCHAR(50) NOT NULL,
  entity_id UUID NOT NULL,
  action VARCHAR(50) NOT NULL,
  old_value JSONB,
  new_value JSONB,
  ip_address INET,
  user_agent TEXT,
  created_at TIMESTAMPTZ DEFAULT now()
);

-- Partition by month for performance
CREATE INDEX idx_audit_tenant_date ON audit_log(tenant_id, created_at DESC);
CREATE INDEX idx_audit_entity ON audit_log(entity_type, entity_id);
-- Audit log is append-only: no UPDATE or DELETE policies
```

Row-Level Security Policies

```
-- Tenant isolation: every table gets this base policy
ALTER TABLE contacts ENABLE ROW LEVEL SECURITY;

CREATE POLICY "tenant_isolation" ON contacts
  USING (tenant_id = (auth.jwt() ->> 'tenant_id')::uuid);

-- Role-based: admin can do everything
CREATE POLICY "admin_full_access" ON contacts
  FOR ALL
  USING (
    EXISTS (
      SELECT 1 FROM users
      WHERE users.id = auth.uid()
      AND users.tenant_id = contacts.tenant_id
      AND users.role = 'admin'
    )
  );

-- Role-based: members can only modify own records
CREATE POLICY "member_own_records" ON contacts
  FOR UPDATE
  USING (
    owner_id = auth.uid()
    AND tenant_id = (auth.jwt() ->> 'tenant_id')::uuid
  );

-- Audit log: append-only (no UPDATE or DELETE)
ALTER TABLE audit_log ENABLE ROW LEVEL SECURITY;

CREATE POLICY "audit_insert_only" ON audit_log
  FOR INSERT
  WITH CHECK (tenant_id = (auth.jwt() ->> 'tenant_id')::uuid);

CREATE POLICY "audit_read" ON audit_log
  FOR SELECT
  USING (tenant_id = (auth.jwt() ->> 'tenant_id')::uuid);
```

Algorithm Implementations

Reference implementations for security-critical and business-critical algorithms. These are not pseudocode — they are production-ready TypeScript that will be used directly in the codebase.

Rate Limiter (Sliding Window)

```
class SlidingWindowRateLimiter {
  private windows: Map<string, { count: number; resetAt: number }[]> = new Map();

  constructor(
    private maxRequests: number = 1000,
    private windowMs: number = 60_000
  ) {}

  async isAllowed(tenantId: string): Promise<{ allowed: boolean; retryAfter?: number }> {
    const now = Date.now();
    const key = tenantId;

    if (!this.windows.has(key)) {
      this.windows.set(key, []);
    }

    const entries = this.windows.get(key)!;
    // Remove expired entries
    const validEntries = entries.filter(e => e.resetAt > now);
    this.windows.set(key, validEntries);

    const totalCount = validEntries.reduce((sum, e) => sum + e.count, 0);

    if (totalCount >= this.maxRequests) {
      const oldestReset = Math.min(...validEntries.map(e => e.resetAt));
      return { allowed: false, retryAfter: Math.ceil((oldestReset - now) / 1000) };
    }

    validEntries.push({ count: 1, resetAt: now + this.windowMs });
    return { allowed: true };
  }
}
```

RBAC Evaluation

```
function evaluatePermission(
  userRole: AppRole,
  action: CRUDAction,
  resource: ResourceType,
  ownerId?: string,
  userId?: string
): boolean {
  const matrix: Record<AppRole, Record<CRUDAction, AccessLevel>> = {
    admin:   { create: 'all', read: 'all', update: 'all', delete: 'all' },
    manager: { create: 'all', read: 'all', update: 'all', delete: 'own' },
    member:  { create: 'own', read: 'all', update: 'own', delete: 'own' },
    viewer:  { create: 'none', read: 'all', update: 'none', delete: 'none' },
  };

  const level = matrix[userRole]?.[action];
  if (!level || level === 'none') return false;
  if (level === 'all') return true;
  if (level === 'own') return ownerId === userId;
  return false;
}
```

Webhook HMAC Signing

```
async function signWebhookPayload(
  payload: string,
  secret: string
): Promise<string> {
  const encoder = new TextEncoder();
  const key = await crypto.subtle.importKey(
    'raw',
    encoder.encode(secret),
    { name: 'HMAC', hash: 'SHA-256' },
    false,
    ['sign']
  );

  const signature = await crypto.subtle.sign(
    'HMAC',
    key,
    encoder.encode(payload)
  );

  return Array.from(new Uint8Array(signature))
    .map(b => b.toString(16).padStart(2, '0'))
    .join('');
}
```

Tenant Slug Validation

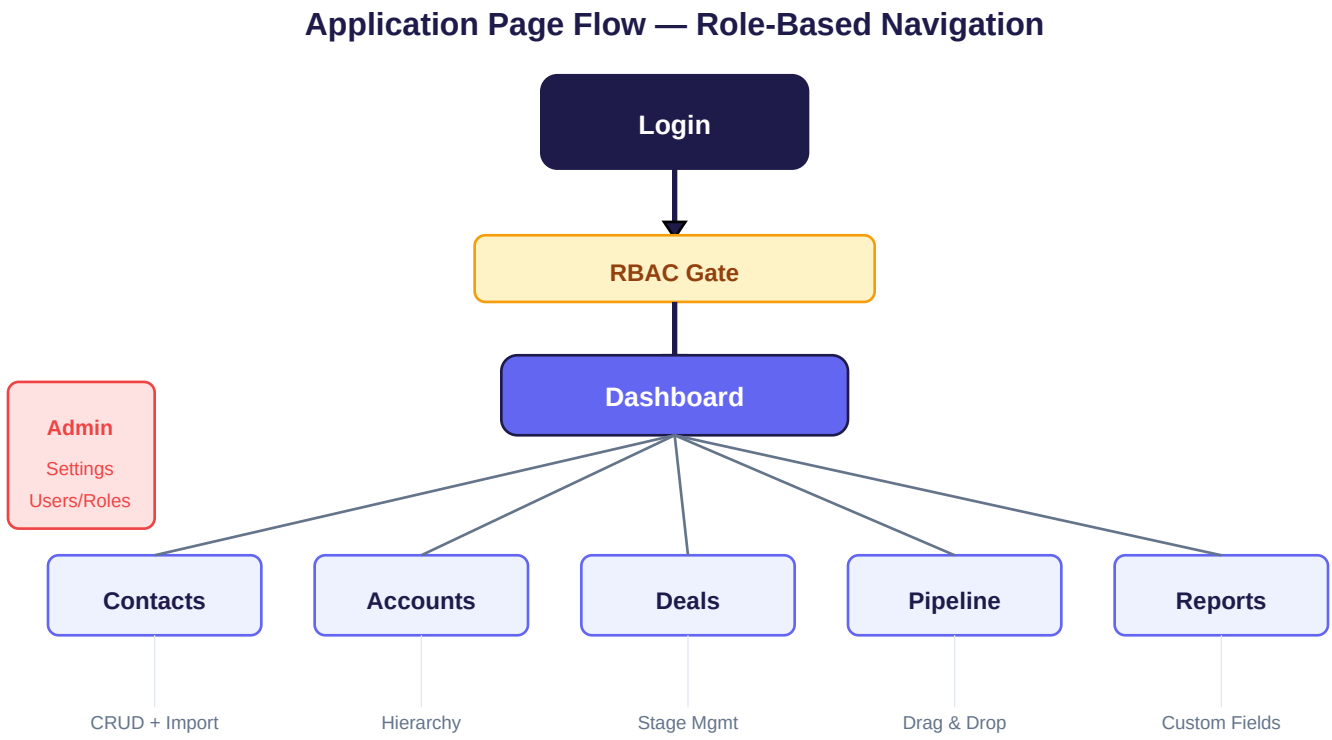
```
const SLUG_REGEX = /^[a-z0-9][a-z0-9-]{2,62}$/;
const RESERVED_SLUGS = new Set([
  'api', 'admin', 'app', 'auth', 'billing', 'blog',
  'cdn', 'dashboard', 'docs', 'help', 'login', 'mail',
  'status', 'support', 'www', 'webhook',
]);

function validateSlug(slug: string): { valid: boolean; error?: string } {
  if (!SLUG_REGEX.test(slug)) {
    return { valid: false, error: 'Slug must be 3-63 chars, lowercase alphanumeric and hyphens,
  }
  if (RESERVED_SLUGS.has(slug)) {
    return { valid: false, error: `"${slug}" is a reserved slug.` };
  }
  return { valid: true };
}
```

Monetary Value Handling

```
// All monetary values stored as integers (cents) to avoid IEEE 754 errors

function toCents(dollars: number): number {
  return Math.round(dollars * 100);
}

function fromCents(cents: number): number {
  return cents / 100;
}

function formatCurrency(cents: number, currency: string = 'USD'): string {
  return new Intl.NumberFormat('en-US', {
    style: 'currency',
    currency,
  }).format(fromCents(cents));
}

function weightedPipelineValue(deals: { valueCents: number; probability: number }[]): number {
  return deals.reduce((sum, d) => sum + Math.round(d.valueCents * (d.probability / 100)), 0);
}
```

Application Page Flow

The following diagram shows the primary user journey through the CRM application. All navigation passes through the RBAC gate, which evaluates permissions based on the user's role within their tenant.



Section 3

Implementation Plan

The Implementation Plan defines **when** and **in what order** everything gets built. This is not a vague timeline — it is a sequenced dependency graph where each task group builds on the outputs of the previous one.

Each task group maps directly to requirements in the PRD, with file paths, test definitions, and checkpoint gates that ensure nothing ships without passing the 43-point QA matrix. The plan is divided into 18 task groups containing approximately 80 individual subtasks, followed by 16 property-based test definitions and 6 checkpoint gates.

The checkpoint gates are non-negotiable. A gate represents a milestone where all preceding tests must pass before any subsequent task group begins. This prevents the most common failure mode in software projects: building features on an untested foundation.

Task Groups

Once the blueprint is locked, we feed it directly into our AI-powered coding pipeline. Because every requirement, every schema, and every edge case has already been defined in the specification above, the AI generates highly secure, production-grade code at a fraction of the time it would take a traditional development team.

The 18 task groups below are executed in sequence. Each group builds on the verified outputs of the previous one. Nothing advances until the preceding checkpoint gate passes. You receive a **weekly status report** showing exactly which tasks were completed, which tests passed, and what is scheduled next. At any point you can log into the project tracker to see the current state of your build.

TG-01: Project Scaffolding

- Initialize Vite + React 18 + TypeScript 5 project with strict tsconfig
- Configure Tailwind CSS v3 with design tokens and dark mode
- Set up Supabase client with typed schema generation
- Configure Vitest + fast-check for unit and property tests
- Set up GitHub Actions CI pipeline (lint, type-check, test, build)

- Configure Sentry for error tracking with environment tags

TG-02: Database Schema

- Create tenants table with slug uniqueness and soft-delete
- Create users table with tenant FK and role column
- Create contacts table with full-text search index
- Create accounts table with parent-child hierarchy
- Create pipelines and stages tables with ordering
- Create deals table with monetary value as cents
- Create activities table with polymorphic entity references
- Create audit_log table (append-only, partitioned by month)
- Create webhooks and webhook_deliveries tables
- Create custom_fields and custom_field_values tables
- Apply RLS policies to all tables
- Create seed data migration for default tenant and pipeline

TG-03: Authentication

- Implement email/password signup with tenant provisioning
- Implement login with JWT + tenant context injection
- Implement OAuth 2.0 flows (Google, Microsoft)
- Implement password reset with email token
- Implement session refresh and multi-tenant switching
- Implement account lockout after 5 failed attempts
- Write property tests for auth edge cases

TG-04: RBAC System

- Implement RBAC evaluation function with role matrix
- Create usePermission hook for client-side checks
- Apply RBAC middleware to all Edge Functions
- Write property tests: no viewer can mutate data
- Write property test: last admin cannot be demoted

TG-05: Contact Management

- Build contact list view with pagination and filters
- Build contact detail view with activity timeline
- Implement contact CRUD Edge Functions
- Implement contact merge with conflict resolution
- Implement CSV import with validation and error reporting
- Implement CSV export with column selection

- Write property tests for contact validation rules

TG-06: Account Management

- Build account list and detail views
- Implement parent-child hierarchy (max 5 levels)
- Implement auto-link contacts via email domain
- Calculate account health scores

TG-07: Deal Management

- Build deal list with pipeline board view
- Implement deal CRUD with monetary value handling
- Implement stage transitions with history tracking
- Implement weighted pipeline value calculation
- Enforce stage-specific required fields
- Write property tests: monetary value never has float errors

TG-08: Pipeline Configuration

- Build pipeline settings UI with drag-and-drop stages
- Implement pipeline CRUD Edge Functions
- Enforce stage ordering invariants
- Implement default pipeline on tenant creation

TG-09: Activity System

- Build activity feed component with filters
- Implement activity CRUD Edge Functions
- Implement recurring activities with scheduler
- Implement activity reminders via email

TG-10: Webhook Engine

- Implement webhook registration and management
- Implement HMAC-SHA256 payload signing
- Implement delivery queue with exponential backoff
- Implement webhook test endpoint
- Build webhook management UI with delivery logs

TG-11: Search & Filtering

- Implement full-text search Edge Function
- Build global search UI with entity type tabs

- Implement saved views (filters) per user
- Optimize search queries with GIN indexes

TG-12: Reporting & Analytics

- Build dashboard with KPI cards and charts
- Implement pipeline velocity report
- Implement activity report with breakdowns
- Implement revenue forecast with weighted values
- Implement report CSV export

TG-13: Custom Fields

- Implement custom field schema management
- Build custom field editor in settings
- Integrate custom fields into entity forms
- Include custom fields in search and export

TG-14: Rate Limiting

- Implement sliding window rate limiter
- Apply rate limiter middleware to all Edge Functions
- Implement per-tenant configurable limits
- Write property tests for rate limit accuracy

TG-15: Email Integration

- Configure Resend API integration
- Implement email template system with variables
- Implement email sending with tracking pixels
- Log sent emails as contact activities

TG-16: Performance & Optimization

- Implement TanStack Query cache strategies
- Add database indexes for all filtered columns
- Implement lazy loading for non-critical components
- Set up connection pooling configuration
- Performance test: verify P95 under 200ms

TG-17: Accessibility

- Audit all components for WCAG 2.1 AA compliance
- Add ARIA labels to all interactive elements

- Test keyboard navigation end-to-end
- Implement locale-aware date/time formatting

TG-18: Deployment & Monitoring

- Configure production Supabase project
- Set up automated database backups (6-hour cycle)
- Implement health check endpoint at /api/health
- Configure Sentry with tenant context tagging
- Set up uptime monitoring with alerting
- Document runbook for common operational tasks

Property-Based Test Definitions

The following 16 property-based tests are run automatically at every checkpoint gate. Each test uses randomized inputs to verify that system invariants hold across all possible states.

PT-01 Tenant isolation: no query pattern returns data from another tenant
PT-02 RBAC: viewer role cannot create, update, or delete any entity
PT-03 RBAC: last admin in tenant cannot be demoted or removed
PT-04 Auth: locked account rejects valid credentials
PT-05 Auth: expired JWT is rejected with 401
PT-06 Slug: all generated slugs match ^[a-z0-9][a-z0-9-]{2,62}$
PT-07 Slug: reserved words are always rejected
PT-08 Monetary: toCents(fromCents(n)) === n for all valid integers
PT-09 Monetary: weighted pipeline value never produces floating-point errors
PT-10 Import: atomic import rolls back all rows on any single validation failure
PT-11 Import: duplicate detection correctly skips existing records
PT-12 Webhook: HMAC signature matches independent recalculation
PT-13 Webhook: failed delivery retries exactly 5 times with exponential backoff
PT-14 Rate limiter: exactly maxRequests allowed in window, maxRequests+1 rejected
PT-15 Search: results always scoped to requesting tenant
PT-16 Audit log: no UPDATE or DELETE operation succeeds on audit_log table

Checkpoint Gates

Gate	Trigger	Criteria
Gate 1: Foundation	After TG-01 through TG-04	Auth, RBAC, and DB schema fully operational. All property tests pass. CI pipeline green.
Gate 2: Core CRM	After TG-05 through TG-08	Contacts, accounts, deals, and pipelines fully functional with CRUD, import/export, and stage management.
Gate 3: Engagement	After TG-09 through TG-10	Activity logging and webhook system operational. HMAC signing verified. Delivery retries tested.
Gate 4: Intelligence	After TG-11 through TG-13	Search, reporting, and custom fields complete. Dashboard renders accurate metrics.
Gate 5: Hardening	After TG-14 through TG-17	Rate limiting, email integration, performance optimization, and accessibility audited.

Gate 6: Ship	After TG-18	Production deployed. Monitoring active. Backups verified. Documentation complete. All 43 QA tests pass.

Design Notes

- All monetary values are stored as integers (cents) to eliminate IEEE 754 floating-point errors.
- The audit log is append-only with no UPDATE or DELETE policies — ensuring tamper-proof records.
- Tenant slugs are validated at creation and immutable after provisioning to prevent URL-based attacks.
- Webhook secrets are generated using crypto.getRandomValues() with 256-bit entropy.
- Rate limiting uses a sliding window (not fixed window) to prevent burst attacks at window boundaries.
- Custom fields use JSONB storage with GIN indexes for efficient querying across heterogeneous schemas.
- Stage ordering uses explicit integer values (not array position) to support concurrent reordering.
- Contact merging preserves the primary record's ID to maintain all existing foreign key references.
- Import operations are wrapped in database transactions — a single row failure rolls back the entire batch.
- All search queries use PostgreSQL full-text search (tsvector/tsquery) rather than LIKE for performance.

What Happens Next

From Blueprint to Production

Once all 18 task groups are complete and every checkpoint gate has passed, you have a **fully tested, production-ready application**. Not a prototype. Not an MVP. A real platform with authenticated users, production database schemas, automated tests, monitoring, and backups — ready to deploy and start serving customers.

Estimated Timeline Comparison

Approach	Timeline	What You Get
Traditional Agency	8–14 months	Static mockups for 2–3 months, then slow iterative build. No spec, no automated testing. You find out it's wrong after it's built.
Freelancer / Small Shop	4–8 months	Faster start, but no architecture docs, no QA matrix, limited scalability. Code ownership varies.
Specloop (This Blueprint)	**4–5 weeks**	**39-page blueprint first. AI-powered build with weekly status reports, 43-point QA matrix, working model you can click through in days. You own every line of code.**

This level of pre-build planning is what separates a glass-box build from the industry standard. Less than 1% of development firms produce anything close to this depth of technical documentation before a single line of code is written.

Ready to Get Your Own Blueprint?

Book a Paid Discovery session and we'll run your business through Specloop. You'll walk away with your own custom 15+ page Technical Design Document — keep it, or credit the $5,000 toward your build.

specloop.ai/the-spec | specloop.ai/contact

End of Enterprise Blueprint | Serial: SL-2026-0004 | Generated by Specloop | hello@specloop.ai