Master AI Agent for Autonomous PRD Generation

A Comprehensive Framework Based on Industry Research and Best Practices

Executive Summary

This document presents a **Master AI Agent Prompt System** designed to autonomously generate complete Product Requirements Documents (PRDs) for web applications across all industries without requiring user input beyond the initial product concept. The system is grounded in comprehensive research of modern web application frameworks, architectures, industry standards, and best practices gathered from academic literature, industry documentation, and real-world implementations as of November 2025.

The framework encompasses:

- **Frontend Technologies**: React, Next.js, Vue, Svelte, Angular
- **Backend Architectures**: Microservices, Serverless, Monolithic, Modular Monolith
- **Database Strategies**: SQL (PostgreSQL, MySQL), NoSQL (MongoDB, Redis)
- **Cloud Platforms**: AWS, Azure, GCP
- **Industry-Specific Patterns**: SaaS, E-commerce, FinTech, HealthTech, EdTech, Enterprise

Part 1: Research Foundation

1.1 Modern Frontend Architecture (2025)

Framework Landscape

Next.js 15.5 (React Meta-Framework)

- **Market Position**: Dominant in React ecosystem (200% growth in adoption)
- **Key Features**: App Router, React 19 support, Turbopack (production builds in beta), Server Components, SSR/SSG/ISR
- **Best For**: Full-stack React applications, SEO-critical sites, enterprise applications
- **Performance**: Solid with optimizations, larger bundle sizes
- **Deployment**: Optimized for Vercel, flexible cloud deployment

SvelteKit (Svelte Meta-Framework)

- **Market Position**: 150% growth, 50% faster than React-based frameworks
- **Key Features**: Compiler-based (compiles to vanilla JS), minimal runtime, SSR/SSG support

- **Best For**: Performance-critical applications, developer experience focus
- **Performance**: Small bundles (typically 50KB), extremely fast load times
- **Developer Experience**: Simple syntax, gentle learning curve

Remix (React Meta-Framework)

- **Market Position**: Growing in enterprise adoption (Shopify Admin uses Remix)
- **Key Features**: Web fundamentals focus, nested routing, progressive enhancement
- **Best For**: Complex interactive applications, admin dashboards, data-heavy apps
- **Performance**: ~30% faster for complex admin flows (Shopify case study)

Qwik

- **Market Position**: Emerging, 5x growth
- **Key Features**: Zero initial JavaScript payload, fine-grained lazy loading, resumability
- **Best For**: Performance-first applications
- **Performance**: 10x faster initial render, under 10KB for many apps

Angular

- **Market Position**: Mature enterprise framework
- **Key Features**: Full-featured framework, TypeScript-first, strong typing
- **Best For**: Large enterprise applications, teams familiar with TypeScript

Architecture Patterns

Component-Based Architecture

- Universal pattern across React, Angular, Vue, Svelte
- Promotes reusability, maintainability, testability
- Atomic design principles for design systems

Micro Frontend Architecture

- Divides frontend into isolated, independently deployable applications
- Enables distributed team development
- Technologies: Module Federation, Single-SPA, Webpack 5
- Best for: Large enterprise applications with multiple teams

Composable UX

- Extension of micro frontends with focus on modular user experiences
- Design systems as critical infrastructure
- Independent, reusable interface elements

1.2 Backend Architecture Patterns (2025)

Microservices Architecture

Core Principles:

- Service decomposition by business domain
- Independent deployment and scaling
- Technology diversity (polyglot)
- Decentralized data management
- API-first design

Communication Patterns:

- Synchronous: RESTful APIs, gRPC, tRPC
- Asynchronous: Message queues (RabbitMQ, Kafka), Event buses

Data Management:

- Database per service pattern
- Event sourcing for audit trails
- CQRS (Command Query Responsibility Segregation)
- Saga pattern for distributed transactions

Best For:

- Scalable, complex systems (e-commerce, SaaS platforms)
- Organizations with multiple development teams
- Applications requiring independent service scaling

Challenges:

- Operational complexity
- Distributed system debugging
- Network latency considerations
- Data consistency across services

Serverless Architecture

Core Principles:

- Function-as-a-Service (FaaS): AWS Lambda, Azure Functions, Google Cloud Functions
- Event-driven execution
- Auto-scaling and pay-per-use pricing
- No server management

Key Patterns:

- Event notification
- Event-carried state transfer
- Queue-based load leveling
- Backend-for-Frontend (BFF)

Best For:

- Sporadic/unpredictable traffic
- Event-driven workloads (file uploads, IoT data)
- MVPs and rapid prototyping
- Microservices backends

Challenges:

- Cold start latency (up to 5s for inactive functions)
- Vendor lock-in complexity
- Debugging complexity in distributed environments
- High costs at scale for high-volume applications

Modular Monolith Architecture

Core Principles:

- Single deployable unit with modular internal structure
- Domain-driven design boundaries
- Shared database with logical separation
- Clear module interfaces

Benefits:

- Simpler deployment than microservices
- Easier debugging and testing
- Lower operational complexity
- Can evolve to microservices later

Best For:

- Startups and small teams
- Applications not yet requiring microservices scale
- Teams transitioning from monoliths to microservices

Hybrid Architectures

Characteristics:

- Combines monolith core with microservices for specific domains
- Strangler pattern for gradual migration
- Best of both worlds approach

1.3 API Design Patterns

RESTful APIs

Characteristics:

- Resource-based URLs
- HTTP methods (GET, POST, PUT, DELETE)
- Stateless communication
- Language-agnostic

Best For:

- Public APIs
- CRUD operations
- Broad compatibility requirements
- Mobile app backends

Considerations:

- Potential for over-fetching/under-fetching
- Multiple round trips for related data
- Manual type safety implementation

GraphQL

Characteristics:

- Single endpoint with flexible queries
- Client specifies exact data needs
- Schema-based with strong typing
- Subscription support for real-time data

Best For:

- Dynamic UIs with complex data requirements
- Data aggregation across multiple sources
- Mobile applications (minimize bandwidth)

- Applications with frequent schema changes

Challenges:

- Query complexity management
- Caching complexity
- Requires GraphQL expertise
- Performance considerations for complex queries

tRPC

Characteristics:

- End-to-end type safety for TypeScript
- Procedure-based RPC style
- No code generation required
- Automatic API documentation

Best For:

- Full-stack TypeScript applications (Next.js, monorepos)
- Internal APIs
- Teams prioritizing type safety
- Rapid development environments

Limitations:

- TypeScript only
- Not suitable for public APIs or polyglot systems

gRPC

Characteristics:

- Protocol Buffers for serialization
- HTTP/2 based
- Strong typing and code generation
- Bi-directional streaming

Best For:

- Microservices communication
- High-performance internal APIs
- Polyglot environments
- Real-time streaming requirements

1.4 Database Selection Guide

SQL Databases

PostgreSQL

- **Strengths**: ACID compliance, advanced features (JSON support, full-text search), extensibility, strong community
- **Use Cases**: Complex queries, transactional systems, data integrity requirements, analytics
- **Scalability**: Vertical scaling, read replicas, partitioning
- **Performance**: Excellent for complex queries with joins

MySQL

- **Strengths**: Simplicity, speed for simple queries, wide adoption
- **Use Cases**: Web applications, read-heavy workloads, WordPress/PHP ecosystems
- **Scalability**: Good horizontal scaling support
- **Performance**: Fast for simple read operations

NoSQL Databases

MongoDB

- **Strengths**: Flexible schema, document-oriented, horizontal scaling
- **Use Cases**: Unstructured/semi-structured data, rapid prototyping, real-time analytics
- **Scalability**: Excellent horizontal scaling via sharding
- **Trade-offs**: Eventual consistency, less suited for complex transactions

Redis

- **Strengths**: In-memory performance, pub/sub messaging, caching
- **Use Cases**: Session storage, caching layer, real-time features, leaderboards
- **Scalability**: Clustering support, extremely fast reads/writes
- **Limitations**: Data persistence options vary, memory constraints

Hybrid Approaches

PostgreSQL with JSON

- Combines relational integrity with flexible document storage
- Best of both worlds for semi-structured data
- Maintains ACID guarantees

Multi-Database Strategy

- PostgreSQL for transactional data

- MongoDB for flexible content
- Redis for caching and sessions
- Choose database per domain/service in microservices

1.5 Cloud Platform Comparison (AWS vs Azure vs GCP)

Amazon Web Services (AWS)

Strengths:

- Broadest service catalog (200+ services)
- Mature ecosystem and tooling
- Global infrastructure (most regions and availability zones)
- Strong documentation and community

Key Services:

- **Compute**: EC2, Lambda, ECS, EKS, Fargate
- **Storage**: S3, EBS, EFS
- **Database**: RDS (Aurora, PostgreSQL, MySQL), DynamoDB, ElastiCache
- **Networking**: VPC, CloudFront, Route 53
- **DevOps**: CodePipeline, CodeBuild, CodeDeploy

Pricing: Complex discounting (Savings Plans, Reserved Instances up to 72%, Spot Instances up to 90%)

Best For: Broad service needs, microservices, startup to enterprise scale

Microsoft Azure

Strengths:

- Deep Microsoft ecosystem integration (Active Directory, Office 365, Windows Server)
- Hybrid cloud excellence (Azure Arc, Stack HCI)
- Enterprise focus with strong compliance
- Excellent for .NET applications

Key Services:

- **Compute**: Virtual Machines, Azure Functions, App Service, AKS
- **Storage**: Blob Storage, Azure Files
- **Database**: Azure SQL Database, Cosmos DB, Azure Cache for Redis
- **Networking**: Virtual Network, Azure CDN, Front Door
- **DevOps**: Azure DevOps, Azure Pipelines

Pricing: Hybrid Benefit reduces costs for existing Microsoft licenses

Best For: Enterprise organizations with Microsoft investments, hybrid scenarios

Google Cloud Platform (GCP)

Strengths:

- Data analytics and AI/ML leadership (BigQuery, Vertex AI)
- Kubernetes-native (GKE originated from Google)
- Cutting-edge networking performance
- Automatic sustained-use discounts
- Per-second billing

Key Services:

- **Compute**: Compute Engine, Cloud Functions, Cloud Run, GKE
- **Storage**: Cloud Storage
- **Database**: Cloud SQL, Firestore, Bigtable, Memorystore
- **Networking**: VPC, Cloud CDN, Cloud Load Balancing
- **Data**: BigQuery, Dataflow, Pub/Sub
- **AI/ML**: Vertex AI, AutoML, TPUs

Pricing: Most transparent, automatic discounts (sustained-use up to 30%)

Best For: Data-heavy applications, AI/ML workloads, containerized applications, startups

1.6 Authentication & Authorization

OAuth 2.0

Core Concepts:

- Delegated authorization framework
- Access tokens for secure resource access
- Authorization server manages permissions
- Four grant types: Authorization Code, Implicit, Client Credentials, Resource Owner Password

Use Cases:

- Third-party application integration
- Social login (Google, Facebook, GitHub)
- API access delegation
- Mobile and SPA applications

Best Practices:

- Use Authorization Code flow with PKCE
- Implement proper token refresh mechanisms
- Secure token storage
- Use HTTPS exclusively

JSON Web Tokens (JWT)

Characteristics:

- Self-contained tokens (header, payload, signature)
- Stateless authentication
- Compact and URL-safe
- Cross-domain authentication support

Token Structure:

```
Header: {"alg": "HS256", "typ": "JWT"}
Payload: {"sub": "user_id", "name": "John Doe", "exp": 1735689600}
Signature: HMACSHA256(base64(header) + "." + base64(payload), secret)
```

Best Practices for Authorization:

- **Keep JWTs short-lived** (minutes, not hours)
- **Store identity only** (user_id, org_id, high-level role)
- **Never include detailed permissions** (use external authorization system)
- **Always use HTTPS**
- **Sign with strong algorithms** (RS256, ES256)
- **Verify every token**
- **Use refresh tokens** for extended sessions

Critical Anti-Pattern: Embedding permissions in JWT

- Permissions change frequently
- Token becomes stale immediately
- No centralized control
- Security and compliance risks

Recommended Pattern: JWT + Policy Engine

- JWT identifies user
- Authorization system (Permit.io, OPA, custom) determines permissions
- Real-time policy evaluation
- Centralized audit trail

1.7 Multi-Tenant SaaS Architecture

Database Patterns

Pattern 1: Shared Everything (Shared DB, Shared Schema)

```
CREATE TABLE customers (
  id INT PRIMARY KEY,
  tenant_id INT NOT NULL,
  name VARCHAR(255),
  email VARCHAR(255),
  INDEX(tenant_id)
);
```

Pros:

- Lowest cost per tenant
- Easiest to maintain
- Cross-tenant analytics simple

Cons:

- Data leakage risk without proper safeguards
- "Noisy neighbor" performance issues
- Limited per-tenant customization

Best For: Cost-sensitive SaaS, small businesses, early-stage startups

Pattern 2: Shared DB, Isolated Schemas

```
CREATE SCHEMA tenant_1;
CREATE SCHEMA tenant_2;

CREATE TABLE tenant_1.customers (...);
CREATE TABLE tenant_2.customers (...);
```

Pros:

- Stronger logical boundaries
- Database-level isolation
- Better security than shared schema

Cons:

- Schema migrations multiply by tenant count
- ORM complexity
- Cross-tenant queries more difficult

Best For: Mid-market SaaS, compliance requirements, moderate tenant counts

Pattern 3: Database-per-Tenant

```
CREATE DATABASE tenant_1;
CREATE DATABASE tenant_2;
```

Pros:

- Maximum tenant isolation
- Per-tenant performance tuning
- Easier compliance (data residency)
- No "noisy neighbor" issues

Cons:

- High operational complexity
- Migration overhead (run on every database)
- Higher infrastructure costs
- Connection pooling challenges

Best For: Enterprise SaaS, regulated industries (healthcare, finance), premium tiers

Hybrid Model

- Small/trial tenants: Shared schema
- Mid-tier customers: Isolated schemas
- Enterprise customers: Dedicated databases

Benefits: Flexibility, cost optimization, meets diverse customer needs

Tenant Identification Strategies

Subdomain-based: `tenant1.app.com`, `tenant2.app.com`
Path-based: `app.com/tenant1`, `app.com/tenant2`
Header-based: Custom HTTP header with tenant ID
Token-based: Tenant ID in JWT claims

Security Best Practices

1. **Row-Level Security (RLS)**: PostgreSQL, Supabase, Neon support
2. **Application-level filtering**: Always include tenant_id in WHERE clauses
3. **Tenant context validation**: Verify user belongs to tenant before operations
4. **Audit logging**: Track all cross-tenant access attempts
5. **Regular security audits**: Test for data leakage

1.8 E-commerce Architecture

Core Components

Frontend Layer:

- Product catalog UI
- Shopping cart
- Checkout flow
- User account management
- Order tracking
- Search and filtering

Backend Services:

- Product management service
- Inventory service
- Order management service
- Payment processing service
- Shipping/fulfillment service
- Customer service
- Notification service

Integration Layer:

- Payment gateways (Stripe, PayPal, Square)
- Shipping providers (USPS, FedEx, UPS)
- ERP systems
- CRM systems
- Marketing tools (Mailchimp, HubSpot)
- Analytics platforms

Data Layer:

- Product catalog database
- Customer database
- Order history database
- Inventory management database
- Analytics data warehouse

Architecture Patterns

Monolithic E-commerce (Three-Tier):

- **Presentation Layer**: Web/mobile frontend
- **Application Layer**: Business logic, APIs
- **Data Layer**: Database(s)

Pros: Simple, fast to market, cost-effective for SMBs
Cons: Scaling limitations, deployment complexity at scale

Microservices E-commerce:

- Independent services per domain (catalog, cart, checkout, inventory, etc.)
- Event-driven communication (order placed → inventory updated → notification sent)
- API Gateway for unified frontend access

Pros: Independent scaling, fault isolation, team autonomy
Cons: Operational complexity, distributed transactions, eventual consistency

Headless E-commerce:

- Backend (APIs) separated from frontend
- CMS for content management (Contentful, Sanity)
- Frontend framework of choice
- Enables omnichannel (web, mobile, IoT)

Benefits: Flexibility, faster frontend iterations, modern UX

Key Patterns

Event-Driven Architecture:

- Inventory updates
- Order status changes
- Price modifications
- Customer notifications
- Enables real-time responsiveness

Caching Strategy:

- Product information (Redis)
- Session data
- Search results
- CDN for static assets

Search Architecture:

- Elasticsearch or Algolia for product search
- Faceted filtering
- Auto-complete
- Personalized recommendations

1.9 CI/CD and DevOps Best Practices

CI/CD Pipeline Components

Source Control:

- Git (GitHub, GitLab, Bitbucket)
- Trunk-based development
- Feature branches with short lifespans
- Pull request workflows

Continuous Integration:

- Automated builds on every commit
- Unit test execution
- Integration test execution
- Code quality checks (linters, SonarQube)
- Security scanning (Snyk, Dependabot)

Continuous Deployment:

- Automated deployment to staging
- Automated or manual promotion to production
- Blue-green deployments
- Canary releases
- Rollback capabilities

Pipeline Tools:

- **GitHub Actions**: Integrated with GitHub, YAML-based workflows
- **GitLab CI/CD**: Built-in, strong Kubernetes integration
- **CircleCI**: Fast, parallelization support, Docker-native
- **Azure DevOps**: Microsoft ecosystem, enterprise features
- **Jenkins**: Self-hosted, highly customizable

Containerization

Docker:

- Consistent environments across dev/staging/production
- Dependency isolation
- Simplified deployment
- Registry: Docker Hub, AWS ECR, Google Artifact Registry

Kubernetes:

- Container orchestration at scale
- Automated scaling and self-healing
- Service discovery and load balancing
- Rolling updates and rollbacks
- **Managed Services**: AWS EKS, Azure AKS, Google GKE

Helm:

- Package manager for Kubernetes
- Templating for K8s manifests
- Versioned releases
- Simplifies complex deployments

Infrastructure as Code

Terraform:

- Multi-cloud support
- Declarative infrastructure definitions
- State management
- Modular and reusable

CloudFormation (AWS):

- Native AWS support
- JSON/YAML templates
- Stack management

Pulumi:

- Use familiar programming languages (TypeScript, Python, Go)
- Cloud-agnostic
- State management

Monitoring and Observability

Logging:

- Centralized logging (ELK stack, Splunk, Datadog)
- Structured logging (JSON format)
- Log aggregation and search

Metrics:

- Prometheus + Grafana
- CloudWatch (AWS), Azure Monitor, Google Cloud Monitoring
- Application performance monitoring (APM): New Relic, Datadog, AppDynamics

Tracing:

- Distributed tracing (Jaeger, Zipkin)
- OpenTelemetry for instrumentation
- Request flow visualization

Alerting:

- PagerDuty, Opsgenie for on-call management
- Threshold-based alerts
- Anomaly detection

Part 2: Industry-Specific Application Templates

2.1 SaaS Application Template

Core Requirements:

- Multi-tenancy architecture
- User authentication and authorization
- Role-based access control (RBAC)
- Subscription management
- Payment processing (Stripe, Paddle)
- Usage tracking and billing
- Admin dashboard
- Customer dashboard
- API for integrations
- Webhooks for events
- Email notifications

- Audit logging

Technology Stack Recommendation:

- **Frontend**: Next.js 15 with TypeScript, Tailwind CSS
- **Backend**: Node.js with Express or NestJS, TypeScript
- **Database**: PostgreSQL (multi-tenant shared DB with tenant isolation)
- **Caching**: Redis for session management and performance
- **Authentication**: Auth0, Clerk, or custom JWT implementation
- **Payment**: Stripe for billing and subscriptions
- **Email**: SendGrid or AWS SES
- **Hosting**: Vercel (frontend), AWS or GCP (backend)
- **Monitoring**: Datadog or New Relic

Architecture Pattern: Microservices or Modular Monolith with potential microservices extraction

2.2 E-commerce Platform Template

Core Requirements:

- Product catalog management
- Shopping cart
- Checkout and payment processing
- Order management
- Inventory tracking
- Customer accounts
- Shipping integration
- Search and filtering
- Product recommendations
- Admin panel
- Analytics dashboard
- Multi-currency support
- Tax calculation
- Discount and coupon system

Technology Stack Recommendation:

- **Frontend**: Next.js or React with Redux, TypeScript
- **Backend**: Node.js (Express/NestJS) or Python (Django/FastAPI)
- **Database**: PostgreSQL for transactions, MongoDB for product catalog (optional)
- **Search**: Elasticsearch or Algolia

- **Caching**: Redis
- **CDN**: CloudFront, Cloudflare
- **Payment**: Stripe, PayPal, Square
- **Hosting**: AWS, GCP, or Azure
- **Queue**: RabbitMQ or AWS SQS for order processing

Architecture Pattern: Microservices with Event-Driven Architecture

2.3 FinTech Application Template

Core Requirements:

- Account management
- Transaction processing
- Payment gateway integration
- KYC (Know Your Customer) verification
- AML (Anti-Money Laundering) compliance
- Transaction history and reporting
- Real-time notifications
- Security (2FA, encryption at rest and in transit)
- Audit trails
- Regulatory reporting
- Dashboard and analytics
- API for third-party integrations

Technology Stack Recommendation:

- **Frontend**: React or Angular with TypeScript
- **Backend**: Java (Spring Boot) or Node.js with TypeScript
- **Database**: PostgreSQL for transactional integrity, Redis for caching
- **Queue**: Kafka for event streaming
- **Security**: OAuth 2.0, JWT, encryption libraries (AWS KMS, HashiCorp Vault)
- **Compliance**: Custom compliance modules
- **Hosting**: AWS or Azure with compliance certifications (SOC 2, PCI DSS)
- **Monitoring**: Splunk, Datadog for security and transaction monitoring

Architecture Pattern: Microservices with strong focus on security and compliance

2.4 HealthTech Application Template

Core Requirements:

- Patient management
- Appointment scheduling
- Electronic Health Records (EHR)
- Telemedicine capabilities (video consultations)
- Prescription management
- Billing and insurance integration
- HIPAA compliance
- Secure messaging
- Lab results integration
- Mobile app support
- Provider dashboard
- Patient portal

Technology Stack Recommendation:

- **Frontend**: React or Vue with TypeScript
- **Backend**: Node.js or Python (Django for HIPAA compliance features)
- **Database**: PostgreSQL with encryption at rest
- **Video**: Twilio Video or Agora.io
- **File Storage**: AWS S3 with encryption
- **Authentication**: OAuth 2.0, SAML for provider integration
- **Compliance**: HIPAA-compliant cloud services (AWS HIPAA, Azure HIPAA)
- **Messaging**: Encrypted messaging (Twilio, SendBird)
- **Hosting**: AWS or Azure with BAA (Business Associate Agreement)

Architecture Pattern: Monolithic or Modular Monolith with strict security boundaries

2.5 EdTech Application Template

Core Requirements:

- Course management (LMS)
- Student enrollment and profiles
- Instructor accounts
- Video hosting and streaming
- Assignment submission and grading
- Discussion forums

- Progress tracking and analytics
- Certificates and badges
- Payment for courses (optional)
- Mobile app support
- Live classes (video conferencing)
- Content library

Technology Stack Recommendation:

- **Frontend**: React or Next.js with TypeScript
- **Backend**: Node.js or Python (Django)
- **Database**: PostgreSQL, MongoDB for flexible content
- **Video**: Vimeo, Wistia, or custom (AWS MediaConvert + CloudFront)
- **Live Classes**: Zoom API, Twilio Video, or WebRTC
- **File Storage**: AWS S3 or Google Cloud Storage
- **Payment**: Stripe or Razorpay
- **Hosting**: AWS, GCP, or Vercel + serverless backend
- **Analytics**: Mixpanel or Google Analytics

Architecture Pattern: Modular Monolith or Microservices for larger platforms

2.6 Enterprise CRM/ERP Template

Core Requirements:

- Customer/lead management
- Sales pipeline tracking
- Task and activity management
- Reporting and dashboards
- Email integration
- Calendar integration
- Document management
- Workflow automation
- Role-based access control
- API for integrations
- Mobile access
- Customizable fields and modules

Technology Stack Recommendation:

- **Frontend**: Angular or React with TypeScript

- **Backend**: Java (Spring Boot) or .NET Core
- **Database**: PostgreSQL or Microsoft SQL Server
- **Search**: Elasticsearch
- **Queue**: RabbitMQ or Kafka
- **Authentication**: SAML, OAuth 2.0, Active Directory integration
- **File Storage**: On-premise or cloud (S3, Azure Blob)
- **Hosting**: On-premise, Azure, or AWS
- **Integration**: REST APIs, SOAP for legacy systems

Architecture Pattern: Modular Monolith with potential microservices for specific modules

Part 3: Master AI Agent Prompt for PRD Generation

3.1 Core System Prompt

```
You are an expert Product Requirements Document (PRD) Generator AI Agent with deep knowle

## Your Capabilities:

1. **Industry Expertise**: You understand application requirements across SaaS, E-commerc

2. **Technology Stack Selection**: You can recommend optimal technology stacks based on:
   - Application complexity and scale requirements
   - Team size and expertise
   - Budget constraints
   - Performance requirements
   - Security and compliance needs
   - Time-to-market considerations

3. **Architecture Design**: You can design appropriate architectures:
   - Monolithic, Modular Monolith, Microservices, Serverless
   - Multi-tenant SaaS patterns
   - E-commerce architectures
   - Event-driven systems
   - Hybrid approaches

4. **Best Practices Application**: You apply 2025 industry best practices for:
   - Authentication and authorization (OAuth 2.0, JWT)
   - API design (REST, GraphQL, tRPC, gRPC)
   - Database design and selection
   - Cloud platform selection (AWS, Azure, GCP)
   - CI/CD and DevOps
   - Security and compliance
   - Performance optimization

5. **Autonomous Decision Making**: You make informed decisions without user input when:
   - User requirements are clear but technology choices are unspecified
   - Industry standards apply
   - Best practices dictate optimal solutions
```

```
   - Trade-offs exist and you select based on common priorities

## Input Processing:

When provided with a product concept, you will:

1. Identify the industry and application category
2. Extract core functional requirements
3. Infer non-functional requirements (security, scalability, performance)
4. Determine user personas and target audience
5. Assess scale and growth expectations
6. Identify compliance and regulatory requirements
7. Map to closest industry template
8. Customize and extend based on unique aspects

## Output Format:

Generate a comprehensive PRD with the following structure:

### 1. Executive Summary
- Product name and vision
- Target market and audience
- Core value proposition
- High-level goals and success metrics

### 2. Product Overview
- Problem statement
- Solution approach
- Key differentiators
- User personas (3-5 detailed personas)

### 3. Functional Requirements
Organized by user journey and feature areas:
- User authentication and authorization
- Core features (categorized by module/domain)
- Admin features
- Integration requirements
- API requirements

Each requirement should include:
- Requirement ID
- Description
- User story format
- Acceptance criteria
- Priority (P0-Critical, P1-High, P2-Medium, P3-Low)
- Dependencies

### 4. Non-Functional Requirements
- Performance (page load times, API response times)
- Scalability (concurrent users, data volume)
- Security (authentication, authorization, encryption, compliance)
- Reliability and availability (uptime SLA)
- Usability (accessibility standards)
- Compatibility (browsers, devices)

### 5. Technology Stack Recommendation
```

```
Provide detailed recommendations with rationale:

**Frontend**:
- Framework and justification
- State management approach
- UI component library
- Build tools and bundlers

**Backend**:
- Language and framework
- Architecture pattern (monolith, microservices, serverless)
- API design pattern
- Middleware and libraries

**Database**:
- Primary database(s) with justification
- Caching strategy
- Search solution (if applicable)
- Data modeling approach

**Infrastructure**:
- Cloud provider recommendation
- Deployment strategy
- CI/CD pipeline
- Monitoring and logging

**Third-Party Services**:
- Authentication provider
- Payment processing
- Email service
- Storage solution
- Other integrations

### 6. System Architecture
- High-level architecture diagram (described in text)
- Component interaction flows
- Data flow diagrams
- Deployment architecture

### 7. Data Models
- Core entities and relationships
- Key database schemas
- API data models

### 8. User Experience Flow
- Key user journeys (described step-by-step)
- Wireframe descriptions for critical screens
- Navigation structure

### 9. Security and Compliance
- Authentication and authorization approach
- Data encryption (at rest and in transit)
- Compliance requirements (GDPR, HIPAA, SOC 2, etc.)
- Security best practices to implement

### 10. Integration Requirements
```

```
- Third-party services to integrate
- API specifications for external integrations
- Webhook requirements
- Data synchronization needs

### 11. Testing Strategy
- Unit testing approach
- Integration testing
- End-to-end testing
- Performance testing
- Security testing

### 12. Deployment and Operations
- Deployment strategy (blue-green, canary, rolling)
- Monitoring and alerting
- Backup and disaster recovery
- Scaling strategy

### 13. Development Phases and Timeline
- Phase 1: MVP (3-4 months)
  - Core features list
  - Success criteria
- Phase 2: Growth (3-4 months)
  - Additional features
  - Optimization focus
- Phase 3: Scale (ongoing)
  - Advanced features
  - Enterprise requirements

### 14. Success Metrics and KPIs
- User acquisition metrics
- Engagement metrics
- Performance metrics
- Business metrics
- Technical health metrics

### 15. Risks and Mitigation
- Technical risks
- Business risks
- Mitigation strategies

### 16. Open Questions and Assumptions
- List assumptions made
- Questions requiring stakeholder input
- Areas requiring further research

## Decision-Making Framework:

When making autonomous decisions, follow this priority order:

1. **Security First**: Always prioritize security and compliance
2. **User Experience**: Optimize for end-user experience
3. **Scalability**: Design for growth (10x current expected scale)
4. **Maintainability**: Choose technologies and patterns that are maintainable
5. **Cost Efficiency**: Balance features with reasonable costs
6. **Time-to-Market**: Consider development speed for MVPs
```

```
7. **Team Capability**: Assume standard full-stack development team unless specified
8. **Industry Standards**: Follow established patterns for the industry

## Technology Selection Criteria:

**Choose Next.js when**:
- SEO is critical
- Server-side rendering needed
- Full-stack React application
- Vercel deployment preferred

**Choose SvelteKit when**:
- Performance is top priority
- Smaller bundle sizes desired
- Developer experience emphasis
- Simpler learning curve needed

**Choose Microservices when**:
- Large team (10+ developers)
- Need independent scaling of services
- Different technologies per service
- Mature DevOps practices in place

**Choose Modular Monolith when**:
- Small to medium team (2-10 developers)
- Simpler deployment preferred
- Can evolve to microservices later
- Development speed prioritized

**Choose PostgreSQL when**:
- Complex queries and joins needed
- Strong data consistency required
- ACID compliance critical
- Advanced features needed (JSON, full-text search)

**Choose MongoDB when**:
- Flexible schema required
- Document-oriented data model
- Rapid prototyping
- Horizontal scaling priority

**Choose AWS when**:
- Broadest service catalog needed
- Microservices architecture
- Mature tooling and ecosystem important

**Choose GCP when**:
- Data analytics and AI/ML focus
- Kubernetes-native approach
- Cost optimization priority
- Automatic discounts valued

**Choose Azure when**:
- Microsoft ecosystem integration
- Hybrid cloud requirements
- Enterprise organization with Microsoft investments
```

Research Integration:

You have access to comprehensive research on:

1. **Frontend Frameworks**: Next.js 15.5, SvelteKit, Remix, Qwik, Angular - with performa

2. **Backend Patterns**: Microservices, Serverless, Modular Monolith, Hybrid - with scala

3. **API Design**: REST, GraphQL, tRPC, gRPC - with selection criteria and best practices

4. **Database Technologies**: PostgreSQL, MySQL, MongoDB, Redis - with performance charac

5. **Cloud Platforms**: AWS, Azure, GCP - with detailed service comparisons, pricing mode

6. **Architecture Patterns**: Multi-tenant SaaS, E-commerce, Event-Driven, CQRS - with im

7. **Security**: OAuth 2.0, JWT best practices, authentication patterns, authorization sy

8. **DevOps**: CI/CD pipelines, Docker, Kubernetes, Infrastructure as Code, monitoring

Apply this research to make informed recommendations in your PRDs.

Example Input Formats You Should Handle:

Minimal Input:
"Create a task management SaaS application"

Moderate Input:
"Create a B2B SaaS platform for project management with team collaboration features, time

Detailed Input:
"Create an e-commerce platform for handmade goods with vendor management, payment process

For all input levels, generate a complete PRD with appropriate depth and detail.

Constraints and Limitations to Consider:

1. **Budget Assumptions**: Unless specified, assume mid-market budget (not enterprise, no
2. **Team Size**: Assume 2-5 developers unless specified
3. **Timeline**: Generate realistic 3-4 month MVP timelines
4. **Compliance**: Identify likely compliance needs based on industry
5. **Scale**: Design for 10x growth from initial requirements

Quality Standards:

Your PRDs must be:
- **Comprehensive**: Cover all aspects of product development
- **Actionable**: Provide clear guidance for development teams
- **Realistic**: Include feasible timelines and scope
- **Professional**: Use industry-standard terminology and formats
- **Detailed**: Include specific technical recommendations with rationale
- **Balanced**: Consider trade-offs and provide reasoning for decisions

Iterative Refinement:

```
After generating the initial PRD, offer to:
1. Expand on any specific section
2. Adjust technology choices based on feedback
3. Add industry-specific compliance requirements
4. Refine scope or priorities
5. Generate additional documentation (API specs, data models, etc.)

Begin generating the PRD immediately upon receiving a product concept, using your compreh
```

3.2 Industry-Specific Enhancement Prompts

For SaaS Applications

```
Enhance the PRD for SaaS applications by including:

1. **Multi-Tenancy Strategy**:
   - Recommend database pattern (shared schema, isolated schema, or database-per-tenant)
   - Justify choice based on customer segment, compliance needs, and scale
   - Detail tenant identification mechanism
   - Specify data isolation approach

2. **Subscription Management**:
   - Pricing tier structure
   - Feature gating by plan
   - Trial period handling
   - Upgrade/downgrade flows
   - Billing cycle management

3. **Usage Tracking**:
   - Metering approach for usage-based pricing
   - Quota enforcement
   - Usage reporting and visibility

4. **Customer Onboarding**:
   - Sign-up flow
   - Email verification
   - Onboarding wizard
   - Sample data/templates

5. **Admin Panel Features**:
   - Tenant management
   - User management per tenant
   - Billing and subscription management
   - Usage analytics
   - Support tools

6. **Integration Ecosystem**:
   - Webhook system for events
   - REST API for programmatic access
   - OAuth 2.0 for third-party integrations
   - API documentation (Swagger/OpenAPI)

7. **SaaS-Specific Metrics**:
   - Monthly Recurring Revenue (MRR)
```

```
   - Customer Acquisition Cost (CAC)
   - Lifetime Value (LTV)
   - Churn rate
   - Activation rate
   - Net Promoter Score (NPS)
```

For E-commerce Applications

```
Enhance the PRD for e-commerce applications by including:

1. **Product Catalog Management**:
   - Product information management
   - Category and taxonomy structure
   - Product variants (size, color, etc.)
   - Inventory tracking
   - Bulk product uploads

2. **Shopping Experience**:
   - Product search and filtering
   - Product recommendations
   - Wishlist functionality
   - Product reviews and ratings
   - Recently viewed products

3. **Cart and Checkout**:
   - Shopping cart (persistent and guest)
   - Checkout flow (single-page vs multi-step)
   - Guest checkout option
   - Address validation
   - Shipping method selection
   - Tax calculation
   - Discount codes and promotions

4. **Payment Processing**:
   - Payment gateway integration (Stripe, PayPal, etc.)
   - Multiple payment methods support
   - PCI DSS compliance
   - Fraud detection
   - Payment retry logic for failed transactions

5. **Order Management**:
   - Order status tracking
   - Order history
   - Cancellation and refunds
   - Return management
   - Order notifications (email, SMS)

6. **Shipping and Fulfillment**:
   - Shipping carrier integration
   - Shipping rate calculation
   - Order fulfillment workflow
   - Tracking number integration
   - Multi-warehouse support (if applicable)

7. **Customer Accounts**:
```

```
   - User registration and login
   - Profile management
   - Order history
   - Saved addresses
   - Saved payment methods
   - Wishlist

8. **Admin Dashboard**:
   - Product management
   - Order management
   - Customer management
   - Inventory management
   - Analytics and reporting
   - Discount and promotion management

9. **E-commerce Metrics**:
   - Conversion rate
   - Average order value (AOV)
   - Cart abandonment rate
   - Customer lifetime value
   - Product performance metrics
```

For FinTech Applications

```
Enhance the PRD for FinTech applications by including:

1. **Regulatory Compliance**:
   - KYC (Know Your Customer) requirements
   - AML (Anti-Money Laundering) procedures
   - PCI DSS compliance (if handling card data)
   - SOC 2 certification path
   - Data residency requirements
   - Regulatory reporting

2. **Security Measures**:
   - Two-factor authentication (2FA)
   - Biometric authentication (mobile)
   - End-to-end encryption
   - Secure key management (AWS KMS, HashiCorp Vault)
   - Session management
   - Fraud detection and prevention
   - Transaction monitoring

3. **Account Management**:
   - Account creation and verification
   - Identity verification workflow
   - Account types and hierarchies
   - Account statements
   - Account closure procedures

4. **Transaction Processing**:
   - Real-time transaction processing
   - Transaction authorization
   - Transaction history
   - Transaction limits and controls
```

```
   - Dispute resolution
   - Transaction reversal and refunds

5. **Audit and Compliance**:
   - Comprehensive audit logs
   - Transaction trail
   - Compliance reporting
   - Data retention policies
   - Regulatory filing automation

6. **Risk Management**:
   - Credit risk assessment
   - Fraud scoring
   - Transaction monitoring rules
   - Suspicious activity reporting

7. **Integration Requirements**:
   - Banking API integrations (Plaid, Yodlee)
   - Payment processor integrations
   - Credit bureau integrations
   - Government database integrations for verification
```

For HealthTech Applications

```
Enhance the PRD for HealthTech applications by including:

1. **HIPAA Compliance**:
   - Protected Health Information (PHI) handling
   - Encryption requirements (at rest and in transit)
   - Access controls and audit logs
   - Business Associate Agreements (BAAs)
   - Data retention and disposal
   - Breach notification procedures

2. **Patient Management**:
   - Patient registration and demographics
   - Patient portal
   - Appointment scheduling
   - Appointment reminders
   - Patient history and records
   - Consent management

3. **Provider Features**:
   - Provider profiles and credentials
   - Provider directory
   - Schedule management
   - Clinical notes and documentation
   - Prescription management
   - Lab order and result management

4. **Telemedicine**:
   - Video consultation platform
   - Screen sharing
   - Chat functionality
   - Secure file sharing
```

```
   - E-prescription integration
   - Session recording (with consent)

5. **Electronic Health Records (EHR)**:
   - Medical history
   - Medication list
   - Allergies
   - Immunization records
   - Vital signs tracking
   - Lab results
   - Imaging reports

6. **Billing and Insurance**:
   - Insurance verification
   - Claims management
   - Billing and invoicing
   - Payment processing
   - Explanation of Benefits (EOB)

7. **Interoperability**:
   - HL7 FHIR support
   - Integration with existing EHR systems
   - Lab system integration
   - Pharmacy system integration

8. **Security and Privacy**:
   - Role-based access control (RBAC)
   - Attribute-based access control (ABAC)
   - Minimum necessary access principle
   - Patient consent management
   - Data anonymization for research
```

3.3 Usage Instructions

To use the Master AI Agent Prompt:

1. **Initialize the agent** with the Core System Prompt (Section 3.1)
2. **Provide product concept** - Can be as simple as:
   - "Create a CRM for real estate agents"
   - "Build a food delivery marketplace"
   - "Design a fitness tracking mobile app"
3. **Agent processes input** using:
   - Industry identification
   - Template matching
   - Research-based decision making
   - Best practice application
4. **Agent generates complete PRD** with:
   - All 16 sections completed

    - Technology stack recommendations
    - Architecture design
    - Implementation guidance
5. **Optional refinement** - Ask agent to:
    - Expand specific sections
    - Adjust technology choices
    - Add compliance requirements
    - Generate additional documentation

3.4 Example Interaction Flow

User Input:
"Create a B2B SaaS platform for inventory management targeting small warehouses"

Agent Processing:

- **Industry**: SaaS, Supply Chain/Logistics
- **Scale**: Small to medium businesses
- **Core Features**: Inventory tracking, order management, reporting
- **Architecture**: Modular monolith with potential microservices evolution
- **Multi-tenancy**: Shared database with tenant isolation
- **Tech Stack**: Next.js, Node.js, PostgreSQL, Redis

Agent Output:
Complete PRD with:

- Executive summary tailored to warehouse management
- User personas (warehouse manager, inventory clerk, business owner)
- Detailed functional requirements for inventory operations
- Technology stack with justification
- Security and data isolation approach
- Integration points (barcode scanners, shipping carriers)
- Mobile considerations for warehouse floor
- Reporting and analytics requirements
- Implementation phases and timeline

Part 4: PRD Template Structure

Section 1: Executive Summary

```
# [Product Name]

## Vision Statement
[One-paragraph description of what the product is and its ultimate goal]

## Target Market
- **Primary Market**: [e.g., Small to medium businesses in retail sector]
- **Geographic Focus**: [e.g., North America, Global]
- **Market Size**: [e.g., $X billion TAM]

## Core Value Proposition
[What unique value does this product provide? What problem does it solve better than alte

## High-Level Goals
1. [Goal 1 with measurable outcome]
2. [Goal 2 with measurable outcome]
3. [Goal 3 with measurable outcome]

## Success Metrics
- [Metric 1]: [Target value]
- [Metric 2]: [Target value]
- [Metric 3]: [Target value]
```

Section 2: Product Overview

```
## Problem Statement

### Current State
[Describe the current situation and pain points]

### Challenges
1. [Challenge 1]
2. [Challenge 2]
3. [Challenge 3]

### Impact
[Describe the cost/impact of not solving this problem]

## Solution Approach

### Our Solution
[Describe how the product solves the identified problems]

### Key Features (High-Level)
1. [Feature 1]
2. [Feature 3]
3. [Feature 3]

### Differentiators
```

```
1. [What makes this unique compared to alternatives?]
2. [Competitive advantage 2]
3. [Competitive advantage 3]

## User Personas

### Persona 1: [Name/Title]
- **Demographics**: [Age, location, industry]
- **Goals**: [What they want to achieve]
- **Pain Points**: [Current frustrations]
- **Technical Proficiency**: [Low/Medium/High]
- **Key Needs**: [What they need from the product]
- **Usage Context**: [When/where they'll use the product]

### Persona 2: [Name/Title]
[Repeat structure]

### Persona 3: [Name/Title]
[Repeat structure]
```

Section 3: Functional Requirements

```
## FR-AUTH: Authentication and Authorization

### FR-AUTH-001: User Registration (P0)
**User Story**: As a new user, I want to register for an account so that I can access the

**Requirements**:
- Email and password registration
- Email verification required
- Password strength requirements (min 8 chars, 1 uppercase, 1 number, 1 special char)
- Social login options (Google, GitHub)
- Terms of service and privacy policy acceptance

**Acceptance Criteria**:
- [ ] User can register with valid email and password
- [ ] Verification email sent within 1 minute
- [ ] Account not active until email verified
- [ ] Error messages for invalid inputs
- [ ] Social login redirects properly

**Dependencies**: None

---

### FR-AUTH-002: User Login (P0)
**User Story**: As a registered user, I want to log in securely so that I can access my a

**Requirements**:
- Email/password login
- Social login (Google, GitHub)
- "Remember me" option
- "Forgot password" flow
- Rate limiting (5 attempts per 15 minutes)
- Two-factor authentication (2FA) support
```

```
**Acceptance Criteria**:
- [ ] Successful login redirects to dashboard
- [ ] Invalid credentials show appropriate error
- [ ] 2FA prompt when enabled
- [ ] Session persists when "remember me" checked
- [ ] Account locked after 5 failed attempts

**Dependencies**: FR-AUTH-001

---

[Continue with all functional requirements organized by module]

## FR-CORE: Core Features

### FR-CORE-001: [Feature Name] (P0/P1/P2/P3)
[Follow same structure as auth requirements]

## FR-ADMIN: Admin Features

### FR-ADMIN-001: [Admin Feature] (P1)
[Follow same structure]

## FR-INTEGRATION: Integration Requirements

### FR-INT-001: [Integration Name] (P1/P2)
[Follow same structure]

## FR-API: API Requirements

### FR-API-001: [API Endpoint/Feature] (P1)
[Follow same structure]
```

Section 4: Non-Functional Requirements

```
## NFR-PERFORMANCE: Performance Requirements

### NFR-PERF-001: Page Load Time
- **Requirement**: All pages must load within 2 seconds on 4G connection
- **Measurement**: Lighthouse performance score &gt; 90
- **Critical Pages**: Homepage, Dashboard, Core workflows

### NFR-PERF-002: API Response Time
- **Requirement**: 95th percentile API response time &lt; 200ms
- **Measurement**: Datadog/New Relic monitoring
- **Critical APIs**: Authentication, Core data fetching

### NFR-PERF-003: Concurrent Users
- **Requirement**: Support 10,000 concurrent users without degradation
- **Measurement**: Load testing results
- **Scale Target**: Design for 100,000 concurrent users

## NFR-SECURITY: Security Requirements
```

```
### NFR-SEC-001: Data Encryption
- **At Rest**: AES-256 encryption for all sensitive data
- **In Transit**: TLS 1.3 for all communications
- **Key Management**: AWS KMS or equivalent

### NFR-SEC-002: Authentication Security
- **Password Storage**: bcrypt with salt (minimum 12 rounds)
- **Session Management**: JWT with 15-minute expiry, refresh token rotation
- **2FA**: TOTP-based (Google Authenticator compatible)

### NFR-SEC-003: Authorization
- **Model**: RBAC with fine-grained permissions
- **Enforcement**: Both API and UI level
- **Audit**: All permission changes logged

### NFR-SEC-004: Compliance
- **GDPR**: Data portability, right to be forgotten, consent management
- **SOC 2**: [If applicable] Annual audit, continuous monitoring
- **PCI DSS**: [If handling payments] Never store full card numbers

## NFR-SCALE: Scalability Requirements

### NFR-SCALE-001: Data Volume
- **Initial**: 100,000 records
- **Year 1**: 1 million records
- **Year 3**: 10 million records
- **Strategy**: Database sharding plan, archival strategy

### NFR-SCALE-002: User Growth
- **Month 1**: 1,000 users
- **Year 1**: 50,000 users
- **Year 3**: 500,000 users
- **Strategy**: Horizontal scaling, CDN, caching

## NFR-RELIABILITY: Reliability and Availability

### NFR-REL-001: Uptime SLA
- **Requirement**: 99.9% uptime (&lt; 45 minutes downtime/month)
- **Measurement**: Status page, synthetic monitoring
- **Maintenance Windows**: Announced 48 hours in advance

### NFR-REL-002: Disaster Recovery
- **RPO** (Recovery Point Objective): 1 hour
- **RTO** (Recovery Time Objective): 4 hours
- **Backup Strategy**: Hourly incremental, daily full, 30-day retention

## NFR-USABILITY: Usability Requirements

### NFR-USE-001: Accessibility
- **Standard**: WCAG 2.1 Level AA compliance
- **Screen Readers**: Compatible with JAWS, NVDA, VoiceOver
- **Keyboard Navigation**: All features accessible without mouse

### NFR-USE-002: Browser Support
- **Desktop**: Chrome (last 2 versions), Firefox (last 2), Safari (last 2), Edge (last 2)
- **Mobile**: iOS Safari (last 2 versions), Android Chrome (last 2)
```

```
- **Progressive Enhancement**: Core functionality works without JavaScript

## NFR-COMPATIBILITY: Compatibility Requirements

### NFR-COMPAT-001: Device Support
- **Desktop**: 1920x1080 and above
- **Tablet**: 768x1024 and above
- **Mobile**: 375x667 and above
- **Responsive**: Fluid layouts between breakpoints
```

Section 5: Technology Stack Recommendation

```
## Frontend Technology Stack

### Framework: [Selected Framework]
**Rationale**: [Why this framework was chosen based on requirements]

**Key Libraries**:
- **State Management**: [Redux, Zustand, Recoil, etc.]
- **UI Components**: [Material-UI, Ant Design, Chakra UI, Tailwind, etc.]
- **Forms**: [React Hook Form, Formik, etc.]
- **Data Fetching**: [React Query, SWR, RTK Query, etc.]
- **Routing**: [Next.js router, React Router, etc.]
- **Testing**: [Jest, React Testing Library, Playwright, Cypress]

**Build Tools**:
- **Bundler**: [Webpack, Vite, Turbopack]
- **Package Manager**: [npm, yarn, pnpm]
- **CSS Processing**: [PostCSS, Sass, Tailwind]

## Backend Technology Stack

### Language and Framework: [Selected Stack]
**Rationale**: [Why this was chosen]

### Architecture Pattern: [Monolith/Microservices/Serverless/Modular Monolith]
**Rationale**: [Justification based on scale, team size, complexity]

### API Design: [REST/GraphQL/tRPC/gRPC]
**Rationale**: [Why this API pattern was chosen]

**Key Libraries/Frameworks**:
- **Web Framework**: [Express, NestJS, FastAPI, Spring Boot, etc.]
- **ORM**: [Prisma, TypeORM, Sequelize, SQLAlchemy, etc.]
- **Validation**: [Zod, Joi, Yup, etc.]
- **Authentication**: [Passport, Auth0, custom JWT]
- **Testing**: [Jest, Mocha, pytest, JUnit]

## Database Strategy

### Primary Database: [PostgreSQL/MySQL/MongoDB/etc.]
**Rationale**: [Why this database was chosen]

**Schema Design Approach**:
- [Relational normalized design / Document-oriented / Hybrid]
```

```
- [Multi-tenant strategy if SaaS]

### Caching Layer: [Redis/Memcached]
**Use Cases**:
- Session storage
- Frequently accessed data
- Rate limiting
- Queue management

### Search Solution: [Elasticsearch/Algolia/None]
**Rationale**: [If applicable, why search solution is needed]

## Infrastructure and DevOps

### Cloud Provider: [AWS/Azure/GCP]
**Rationale**: [Why this provider was chosen]

**Key Services**:
- **Compute**: [EC2, Lambda, App Service, Cloud Run, etc.]
- **Storage**: [S3, Azure Blob, Cloud Storage]
- **Database**: [RDS, Azure SQL, Cloud SQL]
- **CDN**: [CloudFront, Azure CDN, Cloud CDN]
- **Monitoring**: [CloudWatch, Azure Monitor, Cloud Monitoring]

### Container Strategy: [Docker + Kubernetes/Docker Compose/Serverless]
**Rationale**: [Deployment strategy justification]

### CI/CD Pipeline: [GitHub Actions/GitLab CI/CircleCI/Azure DevOps]
**Pipeline Stages**:
1. Lint and format check
2. Unit tests
3. Integration tests
4. Build and containerize
5. Deploy to staging
6. E2E tests
7. Deploy to production

### Infrastructure as Code: [Terraform/CloudFormation/Pulumi]

### Monitoring and Logging
- **APM**: [Datadog, New Relic, AppDynamics]
- **Logging**: [ELK Stack, Splunk, CloudWatch Logs]
- **Error Tracking**: [Sentry, Rollbar]
- **Uptime Monitoring**: [Pingdom, UptimeRobot]

## Third-Party Services

### Authentication: [Auth0/Clerk/Firebase Auth/Custom]
**Features Needed**: OAuth 2.0, Social login, 2FA support

### Payment Processing: [Stripe/PayPal/Square] (if applicable)
**Features**: Subscriptions, one-time payments, webhooks

### Email Service: [SendGrid/AWS SES/Mailgun]
**Use Cases**: Transactional emails, notifications, marketing (optional)
```

```
### File Storage: [AWS S3/Azure Blob/Google Cloud Storage]
**Use Cases**: User uploads, document storage, backups

### Analytics: [Google Analytics/Mixpanel/Amplitude]
**Tracking**: User behavior, conversion funnels, retention

### Other Integrations
- [List any other third-party services]
```

Section 6: System Architecture

```
## High-Level Architecture

### Architecture Pattern: [Pattern Name]

**Description**:
[Describe the overall architecture - e.g., "Three-tier web application with React fronter

### Components:

1. **Frontend Layer**
   - React application served via CDN
   - Client-side routing
   - State management with Redux
   - API communication via REST

2. **API Gateway** (if applicable)
   - Request routing
   - Authentication verification
   - Rate limiting
   - Request/response transformation

3. **Application Layer**
   - [List microservices or application modules]
   - Business logic processing
   - Data validation
   - Authorization checks

4. **Data Layer**
   - PostgreSQL primary database
   - Redis caching layer
   - S3 for file storage
   - Elasticsearch for search (if applicable)

5. **Infrastructure Layer**
   - AWS ECS for container orchestration (or alternative)
   - Application Load Balancer
   - Auto-scaling groups
   - CloudWatch monitoring

### Data Flow

**User Request Flow**:
1. User accesses application via browser
2. CDN serves static assets (HTML, CSS, JS)
```

```
3. Client-side app makes API request
4. Request goes through API Gateway
5. Authentication middleware validates JWT
6. Request routed to appropriate service/controller
7. Business logic executed
8. Database queried (with caching check)
9. Response returned to client
10. UI updated

**Background Job Flow** (if applicable):
1. Event triggered (e.g., user signup)
2. Message published to queue
3. Worker process picks up message
4. Job executed (e.g., send welcome email)
5. Result logged
6. Queue message acknowledged

### Deployment Architecture

**Environments**:
- **Development**: Local development environment
- **Staging**: Production-like environment for testing
- **Production**: Live environment serving users

**Production Setup**:
- Multiple availability zones for high availability
- Auto-scaling based on CPU/memory metrics
- Database read replicas for scaling reads
- Regular automated backups
- CDN for global content delivery

### Security Architecture

- **Network Security**: VPC with private/public subnets
- **Access Control**: IAM roles and policies
- **Secrets Management**: AWS Secrets Manager or HashiCorp Vault
- **DDoS Protection**: AWS Shield or Cloudflare
- **Web Application Firewall**: AWS WAF or similar
```

Section 7: Data Models

```
## Core Entities

### Entity: User
```

User {
id: UUID (PK)
email: String (unique, indexed)
password_hash: String
first_name: String
last_name: String
role: Enum(admin, user, manager)

tenant_id: UUID (FK) // if multi-tenant
is_active: Boolean
is_verified: Boolean
created_at: Timestamp
updated_at: Timestamp
last_login_at: Timestamp
}

```
**Relationships**:
- One-to-Many with Sessions
- Many-to-One with Tenant (if multi-tenant)
- One-to-Many with [Related Entity]

**Indexes**:
- email (unique)
- tenant_id, email (composite, for multi-tenant lookup)

---

### Entity: [Core Entity 2]
```

[Entity Name] {
// Define schema
}

```
**Relationships**:
[Define relationships]

**Indexes**:
[Define indexes]

---

[Repeat for all core entities]

## Database Schema Diagram (Described)

**Relationships Overview**:
- User → [Entity] (one-to-many)
- [Entity A] ↔ [Entity B] (many-to-many via junction table)
- [Other key relationships]

## API Data Models

### Request/Response Models

**User Registration Request**:
```json
{
  "email": "string (email format)",
```
```

```
  "password": "string (min 8 chars)",
  "first_name": "string",
  "last_name": "string",
  "terms_accepted": "boolean"
}
```

User Registration Response:

```
{
  "user": {
    "id": "uuid",
    "email": "string",
    "first_name": "string",
    "last_name": "string",
    "created_at": "ISO timestamp"
  },
  "message": "Verification email sent"
}
```

[Define key request/response models]

```
### Section 8: User Experience Flow

```markdown
## Key User Journeys

### Journey 1: New User Registration and Onboarding

**Steps**:
1. **Landing Page**
   - User arrives at homepage
   - Clear value proposition displayed
   - CTA buttons: "Sign Up" and "Log In"

2. **Registration Page**
   - Email and password fields
   - Social login buttons
   - Password strength indicator
   - Terms of service checkbox
   - Submit button

3. **Email Verification**
   - User receives verification email
   - Email contains verification link
   - User clicks link
   - Account activated message

4. **Onboarding Wizard** (if applicable)
   - Step 1: Profile completion
   - Step 2: Preference settings
   - Step 3: Quick tour or tutorial
   - Step 4: Optional sample data
```
```

```
5. **Dashboard Landing**
   - User arrives at main dashboard
   - Empty state with helpful prompts
   - Quick action buttons
   - Help resources prominently displayed

---

### Journey 2: [Core Feature Usage]

**Steps**:
[Describe step-by-step user flow]

**Wireframe Descriptions**:
- **Screen 1**: [Describe key elements and layout]
- **Screen 2**: [Describe key elements and layout]

---

[Continue for all critical user journeys]

## Navigation Structure

### Primary Navigation
- Dashboard (home icon)
- [Core Feature 1]
- [Core Feature 2]
- [Core Feature 3]
- Settings (gear icon)
- Help (question icon)

### Secondary Navigation (within modules)
[Describe sub-navigation structure]

### User Menu (top-right dropdown)
- Profile
- Settings
- Billing (if applicable)
- Help &amp;amp; Support
- Log Out
```

Sections 9-16

[Continue with similar detailed structure for remaining sections:

- Security and Compliance
- Integration Requirements
- Testing Strategy
- Deployment and Operations
- Development Phases and Timeline
- Success Metrics and KPIs

- Risks and Mitigation
- Open Questions and Assumptions]

Part 5: Application Templates by Industry

5.1 Complete Template: Project Management SaaS

[Full PRD example following the template structure]

5.2 Complete Template: E-commerce Marketplace

[Full PRD example]

5.3 Complete Template: FinTech Payment Platform

[Full PRD example]

[Additional templates as needed]

Part 6: Implementation Guide

6.1 From PRD to Development

Step 1: PRD Review and Approval

- Stakeholder review
- Technical feasibility validation
- Budget and timeline approval
- Prioritization finalization

Step 2: Technical Design Document (TDD)

- Detailed architecture diagrams
- API specifications (OpenAPI/Swagger)
- Database schema definitions
- Component interface definitions

Step 3: Development Environment Setup

- Repository creation and structure
- CI/CD pipeline configuration
- Development environment provisioning
- Third-party service account setup

Step 4: Sprint Planning

- Break down P0 requirements into user stories
- Estimate story points
- Plan first 2-3 sprints
- Assign work to team members

Step 5: Iterative Development

- 2-week sprint cycles
- Daily standups
- Code reviews
- Automated testing
- Sprint demos and retrospectives

6.2 Quality Assurance Checklist

Code Quality:

- [ ] Code follows style guide
- [ ] Unit test coverage > 80%
- [ ] No critical security vulnerabilities (Snyk/Dependabot)
- [ ] No critical performance issues (Lighthouse > 90)
- [ ] Accessibility audit passed (WCAG 2.1 AA)

Functional Testing:

- [ ] All P0 requirements tested
- [ ] All P1 requirements tested
- [ ] Edge cases covered
- [ ] Error handling tested
- [ ] Cross-browser testing completed

Non-Functional Testing:

- [ ] Load testing (expected concurrent users)
- [ ] Security testing (OWASP Top 10)
- [ ] Performance testing (page load, API response times)
- [ ] Disaster recovery tested

6.3 Deployment Checklist

Pre-Deployment:

- [ ] All tests passing
- [ ] Database migrations prepared

- [ ] Environment variables configured
- [ ] Third-party services configured
- [ ] Monitoring and alerting set up
- [ ] Rollback plan documented

Deployment:

- [ ] Database backup created
- [ ] Blue-green deployment or canary release
- [ ] Smoke tests executed
- [ ] Monitoring dashboard observed
- [ ] Status page updated

Post-Deployment:

- [ ] Production smoke tests passed
- [ ] User acceptance testing (UAT) completed
- [ ] Documentation updated
- [ ] Stakeholders notified
- [ ] Metrics baseline established

Conclusion

This Master AI Agent framework provides a comprehensive foundation for autonomously generating complete, production-ready Product Requirements Documents across all industries and application types. By leveraging extensive research on modern web development practices, cloud architectures, and industry-specific patterns, the system can make informed technical decisions and produce detailed, actionable documentation without requiring constant user input.

The framework is designed to be:

- **Comprehensive**: Covering all aspects of modern web application development
- **Research-Based**: Grounded in real-world practices and industry standards from 2025
- **Flexible**: Adaptable to any industry or application type
- **Autonomous**: Capable of making informed decisions based on best practices
- **Practical**: Providing actionable guidance that development teams can immediately use

By utilizing this system, product managers, technical leads, and development teams can accelerate the product definition phase while ensuring comprehensive coverage of all technical, functional, and business requirements.

References

[1] Next.js 15 Documentation. Vercel, 2025.

[2] "Comparative Review of Cloud Computing Platforms for Data Science Workflows." IEEE, 2022.

[3] "Multi-Tenant Architecture: A Comprehensive Framework for Building Scalable SaaS Applications." IJSRCSEIT, 2024.

[4] "The Serverless Computing Survey: A Technical Primer for Design Architecture." arXiv, 2022.

[5] "Backend to Business: Fullstack Architectures for Self-Serve RAG and LLM Workflows." JISEM, 2025.

[6] "Guide to creating modern web application architecture in 2025." Syndicode, 2025.

[7] "AWS vs Azure vs Google Cloud: The Ultimate 2025 Comparison Guide." Pilotcore, 2025.

[8] "How to Use JWTs for Authorization: Best Practices and Common Mistakes." Permit.io, 2025.

[9] "Microservices CI/CD pipeline on Kubernetes with Azure DevOps." Microsoft Learn, 2022.

[10] "Ecommerce Architecture: Types, Best Practices & Trends for 2025." Binmile, 2025.

[11] "Multi-Tenant Database Architecture Patterns Explained." Bytebase, 2025.

[12] "tRPC vs GraphQL vs REST: Choosing the right API design for modern web applications." SD Times, 2025.

[13] "SvelteKit vs Next.js." Better Stack, 2025.

[14] "Frontend Architecture Patterns You Need to Know in 2025." YouTube, 2025.

[15] "Understanding microservices architecture: Building scalable and resilient systems." WJAETS, 2025.

[16] [17] [18] [19] [20] [21] [22] [23] [24] [25] [26] [27] [28] [29] [30] [31] [32] [33] [34] [35] [36] [37] [38] [39] [40] [41] [42] [43] [44] [45] [46] [47] [48] [49] [50] [51] [52] [53] [54] [55] [56] [57] [58] [59] [60] [61] [62] [63] [64] [65] [66] [67] [68] [69] [70] [71] [72] [73] [74] [75] [76] [77] [78] [79] [80] [81] [82] [83] [84] [85] [86] [87] [88] [89] [90] [91] [92] [93] [94] [95] [96] [97] [98] [99] [100] [101] [102] [103] [104] [105] [106] [107] [108] [109] [110] [111] [112]

⁂

1. https://www.ijisrt.com/stockvaluation-pro-a-professionalgrade-web-application-for-discounted-cash-flow-analysis-and-portfolio-tracking
2. https://isjem.com/download/smart-supermarket-management-system-using-mern-stack-and-spring-boot-backend/
3. https://jurnal.itscience.org/index.php/brilliance/article/view/5971
4. https://ulopenaccess.com/papers/ULETE_V02I01/ULETE20250201_010.pdf
5. https://psycholing-journal.com/index.php/journal/article/view/1582
6. https://theaspd.com/index.php/ijes/article/view/978
7. https://isjem.com/download/java-full-stack-development-for-robust-and-scalable-enterprise-architecture/
8. https://www.jisem-journal.com/index.php/journal/article/view/12321
9. https://ijsrem.com/download/estimation-of-carbon-footprints-in-coal-mines/
10. https://lorojournals.com/index.php/emsj/article/view/1672
11. https://arxiv.org/pdf/2501.18225.pdf
12. http://arxiv.org/pdf/2407.07428.pdf
13. https://arxiv.org/pdf/1803.08666.pdf
14. https://arxiv.org/pdf/0801.2618.pdf

15. http://arxiv.org/pdf/1612.03182.pdf
16. https://figshare.com/articles/journal_contribution/An_Approach_to_Developing_Multi-Tenancy_SaaS_Using_Metaprogramming/96680/1/files/101118.pdf
17. https://arxiv.org/pdf/1409.2156.pdf
18. https://arxiv.org/pdf/1901.11219.pdf
19. https://learn.microsoft.com/en-us/azure/azure-sql/database/saas-tenancy-app-design-patterns?view=azuresql
20. https://binmile.com/blog/ecommerce-architecture/
21. https://github.com/opulo-inc/prd-template
22. https://relevant.software/blog/multi-tenant-architecture/
23. https://www.unthinkable.co/blogs/building-e-commerce-architectures-that-scale/
24. https://zapier.com/agents/templates/prd-document-creator-7a1d9e
25. https://clerk.com/blog/how-to-design-multitenant-saas-architecture
26. https://www.nopcommerce.com/en/blog/ecommerce-website-architecture
27. https://copilot4devops.com/ai-product-requirements-document-generator/
28. https://www.bytebase.com/blog/multi-tenant-database-architecture-patterns-explained/
29. http://arxiv.org/pdf/2407.01535.pdf
30. https://carijournals.org/journals/index.php/IJCE/article/download/1821/2195
31. https://zenodo.org/record/4550449/files/MAP-EuroPlop2020bPaper.pdf
32. https://learn.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures
33. https://tymonglobal.com/blogs/next-js-vs-sveltekit-vs-qwik-best-framework-in-2025/
34. https://www.webstacks.com/blog/enterprise-tech-stack
35. https://syndicode.com/blog/web-application-architecture/
36. https://betterstack.com/community/guides/scaling-nodejs/sveltekit-vs-nextjs/
37. https://seclgroup.com/tips-to-choose-tech-stack-for-web-app-development/
38. https://www.youtube.com/watch?v=ixee55xm_d8
39. https://merge.rocks/blog/remix-vs-nextjs-2025-comparison
40. https://www.reddit.com/r/learnprogramming/comments/u0mhck/what_tech_stack_would_you_use_to_create_a_mainly/
41. https://blog.logrocket.com/guide-modern-frontend-architecture-patterns/
42. https://journalwjaets.com/node/1194
43. https://ieeexplore.ieee.org/document/11196324/
44. https://www.ijisrt.com/cloudbased-social-media-platforms-architectures-challenges-and-future-trends
45. https://link.springer.com/10.1007/978-3-032-04403-7_17
46. https://www.transdisciplinaryjournal.com/search?q=MFD-2025-1-009&search=search
47. https://ejournal.unsrat.ac.id/v3/index.php/informatika/article/view/50402
48. https://www.multidisciplinaryfrontiers.com/search?q=FMR-2025-1-099&search=search
49. https://dl.acm.org/doi/10.1145/3368235.3370269
50. https://www.semanticscholar.org/paper/7129b4250b977b44800c28006381e506b94014d7

51. https://ieeexplore.ieee.org/document/9540124/
52. https://wjaets.com/sites/default/files/WJAETS-2023-0226.pdf
53. https://arxiv.org/pdf/1908.10337.pdf
54. https://arxiv.org/pdf/2112.12921.pdf
55. https://arxiv.org/pdf/2401.11867.pdf
56. https://arxiv.org/pdf/2308.15281.pdf
57. https://arxiv.org/pdf/2305.13933.pdf
58. https://arxiv.org/pdf/2201.05825.pdf
59. http://arxiv.org/pdf/2112.01317v1.pdf
60. https://dev.to/shayan_saed/the-ultimate-guide-to-software-architecture-in-nextjs-from-monolith-to-microservices-i2c
61. https://sdtimes.com/graphql/trpc-vs-graphql-vs-rest-choosing-the-right-api-design-for-modern-web-applications/
62. https://dev.to/profilsoftware/database-comparison-sql-vs-nosql-mysql-vs-postgresql-vs-redis-vs-mongodb-2e0l
63. https://www.mindinventory.com/blog/software-architecture-patterns/
64. https://community.webshinetech.com/t/which-backend-api-architecture-do-you-prefer-for-2025-projects-restful-graphql-or-trpc/1543
65. https://xata.io/blog/sql-mysql-postgresql-nosql
66. https://insights.daffodilsw.com/blog/top-software-architecture-patterns
67. https://www.youtube.com/watch?v=veAb1fSp1Lk
68. https://www.datacamp.com/blog/postgresql-vs-mongodb
69. https://dzone.com/articles/post-monolith-architecture-2025
70. https://journal.uob.edu.bh:443/handle/123456789/5863
71. https://www.ijsrst.com/index.php/home/article/view/IJSRST2512368
72. http://www.emerald.com/jsit/article/27/2/197-213/1247269
73. https://ieeexplore.ieee.org/document/11108222/
74. https://journalwjaets.com/node/882
75. https://theamericanjournals.com/index.php/tajet/article/view/6676
76. https://ijsrem.com/download/leading-the-cloud-toward-sustainability-a-comprehensive-comparative-study-of-green-computing-in-aws-azure-and-gcp/
77. https://ijsrcseit.com/index.php/home/article/view/CSEIT25111668
78. https://ijarsct.co.in/Paper23373.pdf
79. https://ijhit.info/index.php/ijhit/article/view/89
80. https://arxiv.org/pdf/2412.03037.pdf
81. https://arxiv.org/pdf/2208.14515.pdf
82. https://turcomat.org/index.php/turkbilmat/article/download/13563/9729
83. https://arxiv.org/pdf/2412.06044.pdf
84. http://www.ijarcs.info/index.php/Ijarcs/article/download/5722/4778
85. http://www.mgijournal.com/Data/Issues_AdminPdf/200/COMPARATIVE STUDY OF CLOUD.pdf
86. https://www.ijfmr.com/papers/2022/6/1140.pdf

87. https://www.ijtsrd.com/papers/ijtsrd23170.pdf
88. https://learn.microsoft.com/en-us/answers/questions/2280200/which-one-is-better-azure-vs-aws-vs-gcp-and-why
89. https://blog.convisoappsec.com/en/secure-api-development-best-practices-oauth2-and-jwt/
90. https://learn.microsoft.com/en-us/azure/architecture/microservices/ci-cd-kubernetes
91. https://pilotcore.io/blog/aws-vs-azure-vs-google-cloud-comparison
92. https://dev.to/gervaisamoah/introduction-to-jwt-and-oauth-20-4bin
93. https://dev.to/arbythecoder/day-13-of-my-90-devops-project-setting-up-a-cicd-pipeline-with-docker-and-kubernetes-on-gitlab-52m
94. https://www.centizen.com/multicloud-strategies-comparison-2025/
95. https://www.permit.io/blog/how-to-use-jwts-for-authorization-best-practices-and-common-mistakes
96. https://k21academy.com/devops-job-bootcamp/end-to-end-ci-cd-pipeline-setup-for-kubernetes-with-circleci-a-beginners-guide/
97. https://northflank.com/blog/aws-vs-azure-vs-google-cloud
98. http://ieeexplore.ieee.org/document/4545607/
99. http://ieeexplore.ieee.org/document/5305420/
100. https://www.semanticscholar.org/paper/c56063706513cf5fbe629f43289eaed7b5024265
101. https://www.tandfonline.com/doi/full/10.1080/17517575.2010.492950
102. https://ijaem.net/issue_dcp/Multi Tenant yet Customizable Cloud Native SaaS Web Application leveraging AIML Architecture and Strategies.pdf
103. https://ijsrcseit.com/index.php/home/article/view/CSEIT241061151
104. https://aircconline.com/csit/papers/vol13/csit132423.pdf
105. https://ieeexplore.ieee.org/document/10092712/
106. https://ieeexplore.ieee.org/document/10019006/
107. https://carijournals.org/journals/index.php/IJCE/article/view/3010
108. http://thesai.org/Downloads/Volume5No11/Paper_23-A_Hybrid_Multi-Tenant_Database_Schema_for_Multi-Level_Quality_of_Service.pdf
109. http://engineer.sljol.info/articles/10.4038/engineer.v46i3.6782/galley/5280/download/
110. http://arxiv.org/pdf/2002.07582.pdf
111. http://arxiv.org/pdf/1409.1656.pdf
112. https://www.ccsenet.org/journal/index.php/cis/article/download/0/0/38044/38517