

# PRD Generation Prompt for AI Coding Agents

## Quick Start

Copy this prompt into ChatGPT/Claude, then describe your MVP feature where indicated.

You are an expert product manager creating a PRD specifically for AI coding agents to implement an MVP.

## Feature Request

[DESCRIBE YOUR MVP FEATURE HERE]  
Example: "Build a webhook receiver that accepts GitHub events, stores them in PostgreSQL, and provides a REST API to query event history with filtering by repo and event type."

## Context (Optional)

- Tech Stack: [e.g., Python/FastAPI, Node/Express, etc.]
- Database: [PostgreSQL, MongoDB, etc.]
- Constraints: [must integrate with X, must handle Y requests/sec]
- Similar Code: [existing patterns in repo to follow]

## Your Task

Generate a comprehensive PRD optimized for AI agents that includes ALL context needed for implementation, validation loops, and hour-based timeline for MVP delivery.

## PRD OUTPUT STRUCTURE

### [FEATURE NAME] - AI Agent Implementation PRD

#### 1. Executive Summary

**Goal:** [One sentence what we're building] **Success Metric:** [Single most important metric]  
**Implementation Time:** [Total hours estimate] **Complexity:** Low/Medium/High

#### 2. What We're Building

##### Core Functionality (MVP)

- Must Have:** [Absolute minimum for working feature]
  - ☐ [Specific capability 1]

- ☐ [Specific capability 2]
- ☐ [Basic error handling]

## User Flow

1. Input: [What triggers this feature]

2. Process: [What happens internally]

3. Output: [What user/system receives]

4. Error: [What happens on failure]

## Success Criteria

- ☐ [Measurable outcome 1 - e.g., "Accepts 100 requests/sec"]
- ☐ [Measurable outcome 2 - e.g., "Returns response in <200ms"]
- ☐ [Measurable outcome 3 - e.g., "Handles malformed input gracefully"]

## 3. Technical Specification

### Data Model

```
python

# Core entities and relationships
class Entity:
    id: UUID
    created_at: datetime
    [critical fields with types]

# Request/Response schemas
class RequestModel:
    [fields with validation rules]

class ResponseModel:
    [fields with examples]
```

### API Endpoints (if applicable)

```
yaml
```

POST /api/[resource]:

body: {field1: type, field2: type}

response: {status: 200, data: {...}}

errors: [400, 404, 500]

GET /api/[resource]/{id}:

response: {id: uuid, ...}

errors: [404, 500]

## Database Schema

sql

*-- Core tables needed*

```
CREATE TABLE [table_name] (  
  id UUID PRIMARY KEY,  
  [fields with types and constraints],  
  created_at TIMESTAMP DEFAULT NOW()  
);
```

*-- Indexes for performance*

```
CREATE INDEX idx_[name] ON [table]([field]);
```

## 4. Implementation Blueprint

### Task Breakdown (with time estimates)

yaml

## Phase 1 - Core Setup (2 hours):

### Task 1.1 - Database Setup (30 min):

- CREATE tables and indexes
- SETUP connection pooling
- TEST connection

### Task 1.2 - Base Models (30 min):

- CREATE Pydantic/TypeScript models
- ADD validation rules
- SETUP error types

### Task 1.3 - Project Structure (1 hour):

- SETUP folder structure
- CREATE base files
- CONFIGURE environment variables

## Phase 2 - Core Logic (3-4 hours):

### Task 2.1 - Business Logic (2 hours):

- IMPLEMENT main processing function
- ADD input validation
- HANDLE edge cases

### Task 2.2 - Data Layer (1 hour):

- CREATE repository functions
- IMPLEMENT CRUD operations
- ADD transaction handling

### Task 2.3 - API Layer (1 hour):

- CREATE endpoints
- ADD request/response handling
- IMPLEMENT error middleware

## Phase 3 - Testing & Validation (2 hours):

### Task 3.1 - Unit Tests (1 hour):

- TEST happy path
- TEST error cases
- TEST edge cases

### Task 3.2 - Integration Tests (30 min):

- TEST API endpoints
- TEST database operations

### Task 3.3 - Fix & Polish (30 min):

- FIX any failing tests
- ADD logging
- CLEANUP code

Total: 7-8 hours for working MVP

## Pseudocode for Core Logic

python

*# Main processing function with critical details*

async def process\_feature(input\_data: InputModel) -> OutputModel:

*# CRITICAL: Validate first, fail fast*

validated = validate\_input(input\_data) *# raises ValidationError*

*# PATTERN: Use connection pooling (never create connections per request)*

async with get\_db\_connection() as conn:

*# GOTCHA: Check for duplicates before insert*

existing = await check\_existing(conn, validated.unique\_field)

if existing:

return OutputModel(status="duplicate", data=existing)

*# TRANSACTION: Ensure atomicity*

async with conn.transaction():

*# Insert with RETURNING to get generated fields*

result = await conn.fetch\_one(  
 "INSERT INTO table (...) VALUES (...) RETURNING \*",  
 validated.dict()  
)

*# CRITICAL: External API calls need timeout*

if needs\_external\_call:

external\_result = await call\_api\_with\_timeout(  
 data=result,  
 timeout=5 *# seconds*  
)

await update\_with\_external(conn, result.id, external\_result)

*# PATTERN: Consistent response format*

return OutputModel(status="success", data=result)

## 5. Context & References

### Required Documentation

yaml

#### MUST\_READ:

- **url**: [Framework docs - specific to routing/middleware]  
**section**: [Exact section needed]  
**why**: [Prevents common mistake X]
- **url**: [Database driver docs]  
**section**: [Connection pooling]  
**critical**: [Max connections = CPU cores \* 2]
- **url**: [External API docs if needed]  
**section**: [Rate limits, auth]  
**gotcha**: [Returns 429 after X requests/sec]

## Codebase Patterns to Follow

yaml

#### COPY\_PATTERNS\_FROM:

- **file**: src/existing\_feature.py  
**why**: [Same error handling pattern]
- **file**: src/db/repository.py  
**why**: [Database query patterns]
- **file**: tests/test\_similar.py  
**why**: [Test structure and mocking approach]

## Known Gotchas & Solutions

python

*# CRITICAL GOTCHAS:*

- # 1. Framework: FastAPI requires async functions for routes*
- # 2. Database: PostgreSQL arrays need special handling: use ANY() not IN*
- # 3. Testing: Must use pytest-asyncio for async tests*
- # 4. Deployment: Environment variables must be set before import*
- # 5. Performance: Never use SELECT \* in production queries*
- # 6. Security: Always parameterize queries, never string concatenation*

## 6. Validation & Testing

### Level 1: Syntax Check (Run First)

bash

```
# Python
ruff check . --fix
mypy .

# Node/TypeScript
npm run lint
npm run type-check

# Fix any errors before proceeding
```

## Level 2: Unit Tests

```
python

# Required test cases
def test_happy_path():
    """Basic functionality works"""
    result = process_feature(valid_input)
    assert result.status == "success"

def test_invalid_input():
    """Validation rejects bad input"""
    with pytest.raises(ValidationError):
        process_feature(invalid_input)

def test_duplicate_handling():
    """Handles duplicates gracefully"""
    # Insert once
    process_feature(input_data)
    # Insert again
    result = process_feature(input_data)
    assert result.status == "duplicate"

def test_external_api_timeout():
    """Handles external timeouts"""
    with mock.patch('external_api.call', side_effect=TimeoutError):
        result = process_feature(valid_input)
        assert result.status == "error"
        assert "timeout" in result.message
```

## Level 3: Integration Test

```
bash
```

*# Start service*

`python -m src.main --dev` *# or: npm run dev*

*# Test endpoints*

```
curl -X POST http://localhost:8000/api/feature \  
-H "Content-Type: application/json" \  
-d '{"test": "data"}'
```

*# Expected: {"status": "success", "data": {...}}*

*# If error: Check logs for stack trace*

## Success Checklist

- ☐ All tests pass: `pytest tests/ -v`
- ☐ No lint errors: `ruff check .`
- ☐ No type errors: `mypy .`
- ☐ API responds correctly to curl test
- ☐ Handles malformed input without crashing
- ☐ Database operations are atomic
- ☐ External calls have timeouts
- ☐ Errors are logged but don't leak internals

## 7. File Structure for MVP

bash



```

project/
├── src/
│   ├── models/
│   │   ├── __init__.py
│   │   ├── database.py  # SQLAlchemy/Prisma models
│   │   └── schemas.py   # Pydantic/Zod schemas
│   ├── api/
│   │   ├── __init__.py
│   │   └── routes.py    # API endpoints
│   ├── core/
│   │   ├── __init__.py
│   │   ├── config.py   # Environment variables
│   │   └── database.py # Connection management
│   ├── services/
│   │   ├── __init__.py
│   │   └── feature.py  # Business logic
│   └── main.py         # App entry point
├── tests/
│   ├── unit/
│   │   └── test_feature.py
│   └── integration/
│       └── test_api.py
├── .env.example        # Required environment variables
├── requirements.txt    # or package.json
└── README.md          # Setup instructions

```

## 8. Environment Variables

```

bash

# .env.example
DATABASE_URL=postgresql://user:pass@localhost/dbname
API_KEY=your_external_api_key_here
LOG_LEVEL=INFO
MAX_CONNECTIONS=10
REQUEST_TIMEOUT=30
RATE_LIMIT=100 # requests per second

```

## 9. Common Pitfalls to Avoid

- ❌ DON'T create database connections per request (use pooling)
- ❌ DON'T forget to handle None/null values from database
- ❌ DON'T use synchronous code in async context
- ❌ DON'T trust external API responses (validate everything)

- ❌ DON'T log sensitive data (passwords, tokens)
- ❌ DON'T use string concatenation for SQL (use parameters)
- ❌ DON'T ignore failing tests (fix the code, not the test)
- ❌ DON'T skip error handling for "happy path only"

## 10. Definition of Done for MVP

The feature is complete when:

1. ✅ Core functionality works end-to-end
  2. ✅ All validation tests pass
  3. ✅ Error cases return appropriate messages
  4. ✅ Performance meets basic requirements (<500ms response)
  5. ✅ Code follows existing patterns in codebase
  6. ✅ Basic logging is in place
  7. ✅ README includes setup and run instructions
- 

## Additional Instructions for You (ChatGPT/Claude):

When generating the PRD:

1. **Be Specific:** Use concrete examples, not placeholders
2. **Include Critical Details:** Database indexes, connection pooling, timeout values
3. **Provide Working Code:** Pseudocode should be nearly executable
4. **Focus on MVP:** Skip nice-to-haves, focus on core functionality
5. **Time Accurately:** Break down into 15-30 minute implementable chunks
6. **Include Validation:** Every phase should have a test to verify it works
7. **Pattern Match:** Reference existing code patterns when possible
8. **Error First:** Design error handling before happy path

**Output:** Complete PRD in markdown, ready for an AI agent to implement in one session.