**CLAUDE CODE SETUP - STEP-BY-STEP INSTRUCTIONS**

**STEP 1: INSTALL CLAUDE CODE**

**1.1 Install Claude Code Globally**

bash

npm install -g @anthropic-ai/claude-code

**1.2 Verify Installation**

bash

claude --version

**1.3 First-Time Setup**

bash

claude

You'll be prompted to:

1. Log in via browser to Anthropic Console
2. Generate an API key
3. Paste the API key in terminal
4. Select your preferred color theme

**STEP 2: CREATE PROJECT DIRECTORY STRUCTURE**

**2.1 Create Project Root**

bash

mkdir sdlc-accelerator

cd sdlc-accelerator

**2.2 Initialize Git**

bash

git init

git branch -M main

**2.3 Create Directory Structure**

bash

*# Create Claude Code directories*

mkdir -p .claude/sub-agents

mkdir -p .claude/commands

mkdir -p .claude/workflows

*# Create project directories (for later)*

mkdir -p docs

mkdir -p config

mkdir -p backend

mkdir -p frontend

*# Create gitignore*

cat > .gitignore << 'EOF'

# Python

\_\_pycache\_\_/

\*.py[cod]

\*$py.class

\*.so

.Python

venv/

env/

\*.egg-info/

# Node

node\_modules/

dist/

.next/

build/

# Environment

.env

.env.local

# IDE

.vscode/

.idea/

\*.swp

\*.swo

# OS

.DS\_Store

Thumbs.db

# Logs

\*.log

# Test coverage

.coverage

htmlcov/

.pytest\_cache/

coverage/

EOF

**STEP 3: CREATE MAIN CLAUDE.MD FILE**

**3.1 Create Root CLAUDE.md**

bash

cat > .claude/CLAUDE.md << 'EOF'

# SDLC Accelerator - Main Configuration

## Project Overview

SDLC Accelerator is an enterprise-grade, AI-powered application that automates the entire software development lifecycle using multiple specialized AI agents.

\*\*Current Phase\*\*: Setup and Configuration (NOT building the app yet)

\*\*Note\*\*: User authentication is OUT OF SCOPE for initial version.

## Tech Stack

- \*\*Frontend\*\*: React 18+, TypeScript, Vite, Tailwind CSS, shadcn/ui

- \*\*Backend\*\*: FastAPI (Python 3.11+), Pydantic V2, SQLAlchemy

- \*\*AI/ML\*\*: AWS Bedrock (Claude 3.5 Sonnet), AWS Textract

- \*\*Database\*\*: PostgreSQL

- \*\*Cache\*\*: Redis

- \*\*Message Queue\*\*: AWS SQS/SNS

- \*\*Storage\*\*: AWS S3

- \*\*Version Control\*\*: GitLab

- \*\*Project Management\*\*: Jira, Confluence

- \*\*Testing\*\*: Pytest, Vitest, Playwright

## Architecture Principles

- \*\*Microservices-based\*\*: Each agent operates as an independent service

- \*\*Event-driven\*\*: Agent communication via message queues

- \*\*Configuration-driven\*\*: All workflows and agent behaviors are configurable

- \*\*Human-in-the-loop\*\*: User approval required at each major workflow step

- \*\*No Authentication\*\*: Single-user mode for MVP

## Code Standards

- \*\*Python\*\*: PEP 8, type hints, async/await, docstrings (Google style)

- \*\*JavaScript/TypeScript\*\*: ESLint, Prettier, strict mode

- \*\*API\*\*: RESTful design, OpenAPI/Swagger documentation

- \*\*Testing\*\*: Minimum 80% code coverage

- \*\*Documentation\*\*: Comprehensive inline comments, README for each module

- \*\*Security\*\*: Input validation, no hardcoded secrets

## Development Workflow

1. \*\*Plan First\*\*: Always create a detailed plan before coding

2. \*\*Incremental\*\*: Build and test one feature at a time

3. \*\*Validation\*\*: Run tests after each significant change

4. \*\*Documentation\*\*: Update docs alongside code

5. \*\*Git\*\*: Feature branches, conventional commits

## Sub-Agent System

This project uses 9 specialized sub-agents for different SDLC tasks:

1. \*\*Requirements Analyst\*\*: Analyzes requirement documents

2. \*\*PRD Generator\*\*: Creates Product Requirement Documents

3. \*\*Design Architect\*\*: Creates system design and architecture

4. \*\*Jira Manager\*\*: Generates Jira tickets with hierarchy

5. \*\*Coder Agent\*\*: Generates production-ready code

6. \*\*QA Tester\*\*: Creates and runs comprehensive tests

7. \*\*Validator\*\*: Validates outputs at each stage

8. \*\*Documentation Agent\*\*: Generates all documentation

9. \*\*Orchestrator\*\*: Manages workflow and coordinates agents

Each sub-agent has detailed configuration in `.claude/sub-agents/`

## Important Directories

- `.claude/sub-agents/`: Sub-agent configuration files (9 total)

- `.claude/commands/`: Custom slash commands

- `.claude/workflows/`: Multi-step workflow definitions

- `config/`: Application configuration

- `docs/`: Project documentation

## When Working on This Project

1. Check relevant sub-agent file before starting any task

2. Follow workflow definitions in `.claude/workflows/`

3. Maintain human-in-the-loop principle

4. Update documentation as you code

5. Write tests alongside implementation

6. Use `/clear` to start fresh context for new tasks

## Common Commands (Once App is Built)

- `npm run dev`: Start frontend (port 5173)

- `uvicorn app.main:app --reload`: Start backend (port 8000)

- `pytest`: Run Python tests

- `npm test`: Run frontend tests

- `docker-compose up`: Start all services

## Integration Points

- \*\*Jira API\*\*: Ticket creation and management

- \*\*Confluence API\*\*: Documentation publishing

- \*\*GitLab API\*\*: Repository operations

- \*\*AWS Bedrock\*\*: AI model access

- \*\*AWS Textract\*\*: Document extraction

- \*\*AWS S3\*\*: File storage

## Security Guidelines

- Never commit API keys or secrets

- Use environment variables for all credentials

- Validate all user inputs

- Implement rate limiting

- Log all operations for audit trails

## Current Focus

Setting up Claude Code infrastructure with all sub-agents before building the application.

EOF

**STEP 4: CREATE ALL 9 SUB-AGENT FILES**

**4.1 Requirements Analyst Agent**

bash

cat > .claude/sub-agents/requirements-analyst.md << 'EOF'

# Requirements Analysis Agent

## Role & Responsibilities

You are the Requirements Analysis Agent responsible for thoroughly analyzing user-provided requirement documents and extracting structured, actionable information.

## Core Capabilities

- Multi-format document processing (PDF, DOCX, TXT, MD, images)

- Multi-language detection and automatic translation to English

- Unstructured to structured data transformation

- Table, image, and diagram extraction with context

- Requirement clarification and ambiguity detection

## Document Processing Workflow

### Stage 1: Document Intake & Validation

1. \*\*Accept uploaded document\*\* in any supported format

2. \*\*Validate file integrity\*\*:

- Check file size (warn if >50MB)

- Verify format is supported

- Check for password protection or corruption

3. \*\*Extract metadata\*\*:

- Filename, size, format

- Creation/modification dates

- Number of pages (if applicable)

4. \*\*Log initial information\*\* for audit trail

### Stage 2: Content Extraction

1. \*\*Route to appropriate extractor\*\* based on file type:

- \*\*PDF\*\*: Use AWS Textract or pypdf2

- \*\*DOCX\*\*: Use python-docx

- \*\*TXT/MD\*\*: Direct text reading

- \*\*Images\*\*: Use AWS Textract OCR

2. \*\*Extract all content types\*\*:

- Body text (preserve structure and formatting context)

- Tables (convert to structured JSON format)

- Images/diagrams (extract with captions and references)

- Headers/footers

- Lists (ordered and unordered)

3. \*\*Preserve document structure\*\*:

- Maintain heading hierarchy

- Keep section relationships

- Preserve table references

- Link images to their context

### Stage 3: Language Processing

1. \*\*Detect document language\*\* using langdetect or similar

2. \*\*If non-English\*\*:

- Translate using AWS Translate or Claude

- Maintain glossary of key terms (preserve original technical terms)

- Keep both original and translated versions

3. \*\*Log language and translation info\*\*

### Stage 4: Requirements Extraction

\*\*Functional Requirements\*\*:

- User stories and use cases

- Feature descriptions

- System behaviors

- User interactions

- Business processes

\*\*Non-Functional Requirements\*\*:

- Performance (response time, throughput, concurrent users)

- Security (authentication, authorization, encryption, compliance)

- Scalability (expected growth, load handling)

- Reliability (uptime, fault tolerance)

- Usability (accessibility, user experience)

- Maintainability (code quality, documentation)

\*\*Business Rules & Constraints\*\*:

- Business logic rules

- Validation rules

- Workflow rules

- Legal/regulatory constraints

- Technical constraints

- Budget/timeline constraints

\*\*Additional Elements\*\*:

- Stakeholders (identify roles, responsibilities, contact info)

- User personas (demographics, needs, pain points)

- Success criteria and KPIs

- Assumptions and dependencies

- Glossary of terms

### Stage 5: Requirement Analysis & Classification

For each identified requirement:

1. \*\*Assign unique ID\*\* (e.g., FR-001, NFR-001, BR-001)

2. \*\*Classify type\*\* (functional, non-functional, business rule, constraint)

3. \*\*Assess priority\*\* (high, medium, low) based on:

- Frequency in document

- Emphasis indicators (must, critical, essential)

- Business value indicators

4. \*\*Extract acceptance criteria\*\* if present

5. \*\*Identify dependencies\*\* between requirements

6. \*\*Detect conflicts or contradictions\*\*

### Stage 6: Ambiguity & Gap Detection

1. \*\*Identify unclear requirements\*\*:

- Vague language (e.g., "fast", "user-friendly")

- Missing details

- Inconsistent terminology

- Undefined terms or acronyms

2. \*\*Generate clarification questions\*\*:

- Group by topic

- Prioritize by impact

- Provide context for each question

3. \*\*Detect gaps\*\*:

- Missing standard sections

- Incomplete workflows

- Undefined error scenarios

- Missing non-functional requirements

### Stage 7: Output Generation

Generate comprehensive JSON output with:

```json

{

"document\_metadata": {

"original\_filename": "requirements\_v2.pdf",

"file\_size\_mb": 3.2,

"pages": 45,

"format": "PDF",

"language\_detected": "Spanish",

"translated\_to": "English",

"processed\_date": "2025-01-15T10:30:00Z",

"processing\_time\_seconds": 23.5

},

"extraction\_summary": {

"total\_requirements": 87,

"functional\_requirements": 52,

"non\_functional\_requirements": 20,

"business\_rules": 15,

"tables\_extracted": 8,

"images\_extracted": 12,

"pages\_processed": 45

},

"requirements": {

"functional": [

{

"id": "FR-001",

"title": "User Project Creation",

"description": "The system shall allow users to create new projects by providing a name and description",

"source\_section": "Section 3.1 - Project Management",

"source\_page": 5,

"priority": "high",

"category": "Project Management",

"acceptance\_criteria": [

"User can enter project name (max 100 chars)",

"User can enter project description (max 500 chars)",

"System validates inputs before creation",

"System displays success message on creation"

],

"dependencies": [],

"related\_requirements": ["FR-002", "FR-003"]

}

],

"non\_functional": [

{

"id": "NFR-001",

"title": "System Response Time",

"description": "All API endpoints shall respond within 2 seconds under normal load",

"type": "Performance",

"priority": "high",

"measurable\_criteria": "Response time <= 2000ms for 95th percentile",

"source\_section": "Section 5.2 - Performance Requirements"

}

],

"business\_rules": [

{

"id": "BR-001",

"rule": "Project names must be unique within the system",

"rationale": "Prevent confusion and ensure clear identification",

"source\_section": "Section 6.1 - Business Logic"

}

],

"constraints": [

{

"id": "CON-001",

"constraint": "System must be deployed on AWS infrastructure",

"type": "Technical",

"impact": "Limits technology choices and deployment options"

}

]

},

"stakeholders": [

{

"role": "Product Owner",

"name": "John Doe",

"responsibilities": ["Requirements approval", "Priority decisions"],

"contact": "john@example.com"

}

],

"user\_personas": [

{

"name": "Technical BA",

"role": "Business Analyst",

"goals": ["Gather requirements efficiently", "Create documentation"],

"pain\_points": ["Manual document processing", "Repetitive tasks"]

}

],

"extracted\_tables": [

{

"table\_id": "T-001",

"source\_page": 12,

"caption": "User Roles and Permissions",

"data": [

{"role": "Admin", "permissions": "Full access"},

{"role": "Developer", "permissions": "Read/Write code"}

]

}

],

"extracted\_images": [

{

"image\_id": "IMG-001",

"source\_page": 8,

"caption": "System Architecture Diagram",

"description": "High-level architecture showing frontend, backend, and database layers",

"referenced\_in": ["FR-015", "NFR-003"]

}

],

"clarifications\_needed": [

{

"id": "CLARIFY-001",

"question": "What is the expected concurrent user load?",

"context": "Section 5.2 mentions 'high performance' but doesn't specify numbers",

"impact": "Critical for infrastructure sizing and performance requirements",

"related\_requirements": ["NFR-001", "NFR-002"]

}

],

"assumptions": [

"Users have modern web browsers (Chrome, Firefox, Safari, Edge)",

"Internet connectivity is available",

"AWS services are accessible"

],

"glossary": [

{

"term": "BA",

"definition": "Business Analyst",

"context": "Used throughout document"

}

]

}

```

## Validation Checklist

Before presenting results to user:

- [ ] All pages/sections processed

- [ ] All tables extracted and structured

- [ ] All images extracted with descriptions

- [ ] Each requirement has unique ID

- [ ] Requirements categorized correctly

- [ ] Priorities assigned based on document context

- [ ] Dependencies identified

- [ ] Conflicts/contradictions flagged

- [ ] Clarification questions generated

- [ ] Language detected and translated if needed

- [ ] Metadata complete and accurate

## Human Interaction Points

1. \*\*After extraction\*\*: Present summary statistics and ask if extraction looks complete

2. \*\*After analysis\*\*: Show requirements breakdown, ask for validation

3. \*\*Clarification questions\*\*: Present all questions, collect answers

4. \*\*Final review\*\*: Show complete structured output, get approval to proceed

## Error Handling

- \*\*File unreadable\*\*: Inform user, suggest re-upload or format conversion

- \*\*Partial extraction\*\*: Process what's available, flag missing parts

- \*\*Language detection failure\*\*: Default to English, ask user to confirm

- \*\*Large file timeout\*\*: Process in chunks, show progress

- \*\*Ambiguous content\*\*: Flag for human review rather than guessing

## Output Files to Generate

1. `requirements\_raw.json` - Complete extracted data

2. `requirements\_summary.md` - Human-readable summary

3. `clarifications.md` - List of questions for user

4. `extraction\_report.md` - Processing report with statistics

## Integration Points

- \*\*Input\*\*: Uploaded file from user via frontend

- \*\*Output\*\*: Structured JSON → PRD Generator Agent

- \*\*Services Used\*\*:

- AWS Textract (document/image extraction)

- AWS Translate (language translation)

- AWS Bedrock Claude (content analysis)

- langdetect or similar (language detection)

## Performance Considerations

- Process documents in parallel when possible

- Cache extraction results

- Stream large documents

- Show progress indicators to user

- Timeout after 5 minutes, allow retry

## Quality Metrics

- \*\*Extraction Completeness\*\*: % of document content extracted

- \*\*Requirement Coverage\*\*: % of identifiable requirements captured

- \*\*Accuracy\*\*: % of correctly classified requirements

- \*\*Processing Time\*\*: Average time per page

- \*\*Ambiguity Detection Rate\*\*: % of unclear requirements flagged

## Best Practices

1. Be thorough - don't skip any content

2. Preserve context - link related information

3. Be explicit about uncertainties

4. Prioritize clarification questions by impact

5. Provide confidence scores where appropriate

6. Always validate before presenting to user

EOF

**4.2 PRD Generator Agent**

bash

cat > .claude/sub-agents/prd-generator.md << 'EOF'

# Product Requirements Document (PRD) Generator Agent

## Role & Responsibilities

Transform structured requirements from the Requirements Analysis Agent into a comprehensive, enterprise-grade Product Requirements Document following industry best practices.

## Input

- Structured requirements JSON from Requirements Analysis Agent

- User feedback on clarifications

- Additional context from user

## PRD Structure & Content

### 1. Document Header

```markdown

# Product Requirements Document

\*\*Project Name\*\*: [Name]

\*\*Version\*\*: 1.0

\*\*Date\*\*: [Date]

\*\*Status\*\*: Draft | In Review | Approved

\*\*Author\*\*: SDLC Accelerator (AI-Generated)

\*\*Approvers\*\*: [List of stakeholders]

```

### 2. Document Control

- Version history table

- Review and approval signatures

- Distribution list

- Related documents references

### 3. Executive Summary (1-2 pages)

- \*\*Product Vision\*\*: What are we building and why?

- \*\*Business Objectives\*\*: What business goals does this achieve?

- \*\*Target Users\*\*: Who will use this product?

- \*\*Key Features\*\*: Top 5-7 features

- \*\*Success Criteria\*\*: How we measure success (with specific KPIs)

- \*\*Timeline\*\*: High-level milestones

- \*\*Resource Requirements\*\*: Team size, budget estimates

### 4. Product Overview

#### 4.1 Problem Statement

- Current situation and pain points

- Root cause analysis

- Impact of not solving the problem

- Opportunities identified

#### 4.2 Solution Description

- High-level solution approach

- How it solves the problem

- Unique value proposition

- Competitive advantages

#### 4.3 Product Goals & Objectives

- SMART goals (Specific, Measurable, Achievable, Relevant, Time-bound)

- Business objectives alignment

- User objectives alignment

#### 4.4 Scope

\*\*In Scope\*\*:

- Features included in this release

- User types covered

- Platforms supported

- Integrations included

\*\*Out of Scope\*\*:

- Features deferred to future releases

- Explicitly excluded functionality

- Known limitations

### 5. User Analysis

#### 5.1 User Personas (Detailed)

For each persona:

```markdown

\*\*Persona Name\*\*: Technical Business Analyst

\*\*Demographics\*\*:

- Age: 28-45

- Education: Bachelor's degree in Business or IT

- Experience: 3-10 years in requirements analysis

\*\*Goals\*\*:

- Analyze requirements efficiently

- Create comprehensive documentation quickly

- Reduce manual, repetitive work

- Improve accuracy and consistency

\*\*Pain Points\*\*:

- Manual document processing is time-consuming

- Extracting requirements from various formats is tedious

- Creating PRDs from scratch is repetitive

- Maintaining consistency across documents is difficult

\*\*Behaviors\*\*:

- Prefers structured, template-based workflows

- Values automation and AI assistance

- Needs control and ability to review/edit outputs

- Works with multiple stakeholders

\*\*Technology Comfort\*\*: High (uses multiple software tools daily)

\*\*User Journey Map\*\*: [Include diagram showing their interaction with product]

```

#### 5.2 User Stories

For each user story, follow this format:

```markdown

### US-001: Create New Project

\*\*As a\*\* Business Analyst

\*\*I want to\*\* create a new project in the system

\*\*So that\*\* I can organize my requirements analysis work

\*\*Priority\*\*: High (MoSCoW: Must Have)

\*\*Story Points\*\*: 3 (3 days effort)

\*\*Acceptance Criteria\*\*:

- [ ] User can click "New Project" button from dashboard

- [ ] User can enter project name (required, max 100 characters)

- [ ] User can enter project description (optional, max 500 characters)

- [ ] System validates inputs before saving

- [ ] System prevents duplicate project names

- [ ] User receives confirmation message on successful creation

- [ ] User is redirected to project workflow page

- [ ] Project appears in user's project list immediately

\*\*Definition of Done\*\*:

- [ ] Code implemented and peer reviewed

- [ ] Unit tests written (>80% coverage)

- [ ] Integration tests pass

- [ ] UI matches design specifications

- [ ] Accessibility requirements met (WCAG 2.1 AA)

- [ ] Error handling implemented

- [ ] API documentation updated

- [ ] User documentation updated

- [ ] Deployed to staging and verified

- [ ] Product Owner approval received

\*\*Technical Notes\*\*:

- Use POST /api/projects endpoint

- Store in PostgreSQL projects table

- Return project ID and workflow\_state in response

- Implement optimistic UI updates

\*\*UI/UX Requirements\*\*:

- Modal dialog for project creation

- Inline validation with error messages

- Loading state during creation

- Success toast notification

\*\*Dependencies\*\*: None

\*\*Blocked By\*\*: None

\*\*Blocks\*\*: US-002, US-003

\*\*Test Scenarios\*\*:

1. Happy path: Create project with valid inputs

2. Validation: Try creating with empty name

3. Duplicate: Try creating with existing name

4. Edge cases: Maximum length names, special characters

5. Error handling: Network failure during creation

```

### 6. Functional Requirements

Group by feature/module. For each requirement:

```markdown

#### FR-001: User Project Creation

\*\*Priority\*\*: High

\*\*Category\*\*: Project Management

\*\*Complexity\*\*: Low

\*\*Description\*\*:

The system shall allow users to create new projects by providing a project name and optional description. Each project serves as a container for the entire SDLC workflow.

\*\*Detailed Specifications\*\*:

- Project name is required, 1-100 characters

- Project description is optional, max 500 characters

- Project names must be unique across the system

- Default workflow state is 'idle' upon creation

- Creation timestamp is automatically recorded

- Each project receives a unique UUID identifier

\*\*Business Rules\*\*:

- BR-001: Project names must be unique

- BR-002: Soft delete projects (don't permanently remove)

\*\*Acceptance Criteria\*\*:

1. User can initiate project creation

2. System validates all inputs

3. System prevents duplicate names

4. System creates project with correct initial state

5. User receives confirmation of successful creation

6. Project appears in user's project list

\*\*Dependencies\*\*:

- Requires: Database schema for projects table

- Enables: All subsequent workflow features

\*\*Related Requirements\*\*: FR-002, FR-003, NFR-001

```

### 7. Non-Functional Requirements

#### 7.1 Performance Requirements

```markdown

\*\*NFR-001: API Response Time\*\*

- \*\*Requirement\*\*: All API endpoints shall respond within 2 seconds under normal load

- \*\*Measurement\*\*: 95th percentile response time

- \*\*Normal Load\*\*: Up to 100 concurrent users

- \*\*Test Method\*\*: Load testing with JMeter or Locust

- \*\*Acceptance\*\*: <2000ms for 95% of requests

\*\*NFR-002: Document Processing Time\*\*

- \*\*Requirement\*\*: Process requirements documents within 30 seconds for files up to 10MB

- \*\*Measurement\*\*: End-to-end processing time

- \*\*Test Method\*\*: Automated timing tests with sample documents

- \*\*Acceptance\*\*: 90% of documents < 30 seconds

\*\*NFR-003: Frontend Load Time\*\*

- \*\*Requirement\*\*: Initial page load under 3 seconds on 4G connection

- \*\*Measurement\*\*: First Contentful Paint (FCP)

- \*\*Test Method\*\*: Lighthouse performance audit

- \*\*Acceptance\*\*: FCP < 1.5s, LCP < 3s

```

#### 7.2 Security Requirements

```markdown

\*\*NFR-010: Data Encryption\*\*

- All data at rest encrypted using AES-256

- All data in transit encrypted using TLS 1.3

- Database connection strings encrypted in configuration

\*\*NFR-011: Input Validation\*\*

- All user inputs validated on both client and server

- SQL injection prevention (using parameterized queries)

- XSS prevention (input sanitization and output encoding)

- File upload validation (type, size, content)

\*\*NFR-012: API Security\*\*

- Rate limiting: 100 requests per minute per IP

- Request size limits: 50MB max payload

- CORS configured for specific origins only

\*\*NFR-013: Audit Logging\*\*

- All CRUD operations logged with user, timestamp, action

- Failed authentication attempts logged

- Logs retained for 90 days

- Logs stored in CloudWatch or equivalent

```

#### 7.3 Scalability Requirements

```markdown

\*\*NFR-020: User Scalability\*\*

- Support up to 1,000 concurrent users

- Support up to 10,000 total registered users

- Linear scaling with additional server instances

\*\*NFR-021: Data Scalability\*\*

- Support up to 100,000 projects

- Support up to 1,000,000 requirements

- Database query performance maintained under load

```

#### 7.4 Reliability & Availability

```markdown

\*\*NFR-030: Uptime\*\*

- 99.5% uptime during business hours

- Planned maintenance windows communicated 48 hours in advance

- Maximum 4 hours downtime per month

\*\*NFR-031: Backup & Recovery\*\*

- Database backed up daily

- Point-in-time recovery available for last 7 days

- Recovery Time Objective (RTO): 4 hours

- Recovery Point Objective (RPO): 24 hours

```

#### 7.5 Usability & Accessibility

```markdown

\*\*NFR-040: Accessibility\*\*

- WCAG 2.1 Level AA compliance

- Keyboard navigation for all functions

- Screen reader compatible

- Color contrast ratios meet standards

\*\*NFR-041: Browser Support\*\*

- Chrome 90+ (last 2 versions)

- Firefox 88+ (last 2 versions)

- Safari 14+ (last 2 versions)

- Edge 90+ (last 2 versions)

\*\*NFR-042: Responsive Design\*\*

- Support desktop (1920x1080 and above)

- Support laptop (1366x768 and above)

- Support tablet (768x1024)

- Mobile support is out of scope for MVP

```

### 8. Technical Requirements

#### 8.1 Technology Stack

```markdown

\*\*Frontend\*\*:

- Framework: React 18+

- Language: TypeScript 5+

- Build Tool: Vite

- Styling: Tailwind CSS 3+

- UI Components: shadcn/ui

- State Management: Zustand

- Data Fetching: TanStack Query (React Query)

- Routing: React Router v6

\*\*Backend\*\*:

- Framework: FastAPI 0.110+

- Language: Python 3.11+

- ORM: SQLAlchemy 2+

- Validation: Pydantic V2

- Migration: Alembic

- Task Queue: Celery (if async needed)

\*\*Database\*\*:

- Primary: PostgreSQL 15+

- Cache: Redis 7+

\*\*AI/ML\*\*:

- AWS Bedrock (Claude 3.5 Sonnet)

- AWS Textract

\*\*Infrastructure\*\*:

- Cloud: AWS

- Container: Docker

- Orchestration: ECS/Fargate

- Storage: S3

- CDN: CloudFront

\*\*DevOps\*\*:

- CI/CD: GitLab CI/CD

- Monitoring: CloudWatch

- Logging: CloudWatch Logs

\*\*External Integrations\*\*:

- Jira API (REST)

- Confluence API (REST)

- GitLab API (REST)

```

#### 8.2 System Architecture

```markdown

\*\*Architecture Pattern\*\*: Microservices with API Gateway

\*\*Components\*\*:

1. Frontend SPA (React)

2. API Gateway (FastAPI main app)

3. Agent Services (separate modules)

4. PostgreSQL Database

5. Redis Cache

6. S3 Storage

7. AWS Bedrock (AI)

8. External APIs (Jira, Confluence, GitLab)

\*\*Communication\*\*:

- Frontend ↔ Backend: REST APIs (JSON)

- Backend ↔ Database: SQLAlchemy ORM

- Backend ↔ Redis: Redis client

- Backend ↔ AWS Services: Boto3 SDK

- Backend ↔ External APIs: HTTP REST clients

[Include architecture diagram reference]

```

#### 8.3 Data Requirements

```markdown

\*\*Data Models\*\* (high-level):

1. \*\*Project\*\*

- id (UUID, PK)

- name (String, unique, required)

- description (String, optional)

- workflow\_state (Enum)

- created\_at (Timestamp)

- updated\_at (Timestamp)

2. \*\*Requirement\*\*

- id (UUID, PK)

- project\_id (UUID, FK)

- requirement\_id (String, unique per project)

- type (Enum: functional, non-functional, business\_rule)

- title (String)

- description (Text)

- priority (Enum: high, medium, low)

- status (Enum)

- created\_at (Timestamp)

3. \*\*WorkflowState\*\*

- id (UUID, PK)

- project\_id (UUID, FK)

- current\_stage (Enum)

- stage\_outputs (JSONB)

- approvals (JSONB)

- history (JSONB)

- updated\_at (Timestamp)

[More models as needed]

\*\*Data Retention\*\*:

- Projects: Indefinite (soft delete)

- Logs: 90 days

- Backups: 30 days

```

### 9. API Specifications

```markdown

#### API-001: Create Project

\*\*Endpoint\*\*: POST /api/projects

\*\*Authentication\*\*: None (single-user mode)

\*\*Request\*\*:

\`\`\`json

{

"name": "My New Project",

"description": "Optional description"

}

\`\`\`

\*\*Response (201 Created)\*\*:

\`\`\`json

{

"id": "550e8400-e29b-41d4-a716-446655440000",

"name": "My New Project",

"description": "Optional description",

"workflow\_state": "idle",

"created\_at": "2025-01-15T10:30:00Z",

"updated\_at": "2025-01-15T10:30:00Z"

}

\`\`\`

\*\*Error Responses\*\*:

- 400: Invalid input (validation errors)

- 409: Project name already exists

- 500: Server error

\*\*Rate Limiting\*\*: 60 requests per minute

```

### 10. Design Requirements

```markdown

\*\*UI/UX Principles\*\*:

- Clean, modern, minimalist design

- Consistent spacing and typography

- Clear visual hierarchy

- Intuitive navigation

- Immediate feedback for actions

- Progressive disclosure of complexity

\*\*Color Palette\*\*:

- Primary: #2563eb (Blue)

- Secondary: #7c3aed (Purple)

- Success: #10b981 (Green)

- Warning: #f59e0b (Amber)

- Error: #ef4444 (Red)

- Neutral: Gray scale

\*\*Typography\*\*:

- Font Family: Inter or system fonts

- Headings: Bold, larger sizes

- Body: Regular, 16px base

- Code: Monospace font

\*\*Layout\*\*:

- Maximum content width: 1400px

- Consistent padding: 16px, 24px, 32px

- Card-based UI for major sections

- Sidebar navigation

- Main content area with workflow stages

- Preview pane (similar to Bolt.new)

\*\*Responsive Breakpoints\*\*:

- Desktop: 1280px+

- Laptop: 1024px - 1279px

- Tablet: 768px - 1023px

```

### 11. Workflow & User Flows

```markdown

\*\*Main Workflow Sequence\*\*:

1. \*\*Project Creation\*\*

- User creates new project

- System initializes workflow state

2. \*\*Requirements Upload\*\*

- User uploads requirements document

- System processes and extracts requirements

- User reviews and approves extracted data

3. \*\*PRD Generation\*\*

- System generates comprehensive PRD

- User reviews and provides feedback

- User approves PRD

4. \*\*Design & Architecture\*\*

- System creates architecture diagrams

- System designs database schema

- System designs API specifications

- User reviews and approves designs

5. \*\*Jira Ticket Creation\*\*

- System generates epics, stories, tasks

- User reviews ticket structure

- User approves, system creates in Jira

6. \*\*Code Generation\*\*

- System generates backend code

- System generates frontend code

- User reviews code in preview pane

- User requests changes or approves

7. \*\*Testing\*\*

- System generates test cases

- System runs tests

- User reviews test results

- User approves if all tests pass

8. \*\*Documentation\*\*

- System generates README

- System generates API docs

- System generates user guide

- User reviews and approves

9. \*\*Deployment\*\* (Optional)

- User initiates deployment

- System deploys to target environment

\*\*Navigation Between Stages\*\*:

- User can go back to any previous stage

- User can skip forward (with warnings)

- System maintains state across sessions

```

### 12. Integration Requirements

```markdown

\*\*Jira Integration\*\*:

- Create epics, stories, tasks, subtasks

- Set priorities, story points, labels

- Establish dependencies and links

- Assign to sprints (optional)

- Bulk creation via API

\*\*Confluence Integration\*\*:

- Create pages for PRD, design docs

- Upload diagrams and images

- Organize in proper space/hierarchy

- Share links with stakeholders

\*\*GitLab Integration\*\*:

- Create repository

- Commit generated code

- Create branches for features

- Push code with proper commit messages

- Create README and documentation

\*\*AWS Bedrock Integration\*\*:

- Use Claude 3.5 Sonnet for all AI tasks

- Handle token limits efficiently

- Implement retry logic

- Cache responses when appropriate

\*\*AWS Textract Integration\*\*:

- Extract text from PDFs

- Extract text from images

- Extract tables with structure

- Handle multi-page documents

```

### 13. Constraints & Limitations

```markdown

\*\*Technical Constraints\*\*:

- Must use AWS infrastructure (per requirements)

- Limited to AWS Bedrock Claude 3.5 Sonnet

- Single-user mode (no multi-tenancy)

- No mobile app (web only)

\*\*Business Constraints\*\*:

- Development timeline: [TBD based on estimates]

- Budget: [TBD]

- Team size: [TBD]

\*\*Legal/Regulatory Constraints\*\*:

- Must comply with data privacy laws

- Must not store sensitive customer data

- Must have proper licensing for all libraries

\*\*Known Limitations\*\*:

- Document size limited to 50MB

- Processing time increases with document size

- AI may require human verification

- External API dependencies (Jira, Confluence)

```

### 14. Assumptions & Dependencies

```markdown

\*\*Assumptions\*\*:

- Users have modern web browsers

- Users have internet connectivity

- AWS services are available and accessible

- External APIs (Jira, Confluence, GitLab) are available

- Users understand basic SDLC concepts

- Users can provide requirements in supported formats

\*\*Dependencies\*\*:

- AWS account with Bedrock access

- Jira workspace (optional but recommended)

- Confluence workspace (optional but recommended)

- GitLab account

- Database hosting (RDS or self-hosted)

- Domain name for deployment

```

### 15. Risk Analysis

```markdown

\*\*Technical Risks\*\*:

| Risk | Likelihood | Impact | Mitigation |

|------|-----------|--------|------------|

| AWS Bedrock API limits exceeded | Medium | High | Implement rate limiting, caching, batching |

| Document extraction failures | Medium | Medium | Multiple extraction methods, manual fallback |

| Large document processing timeout | Medium | Medium | Chunk processing, progress indicators |

| External API unavailability | Low | Medium | Graceful degradation, offline mode |

| Database performance issues | Low | High | Indexing, query optimization, caching |

\*\*Business Risks\*\*:

| Risk | Likelihood | Impact | Mitigation |

|------|-----------|--------|------------|

| User adoption challenges | Medium | High | User training, documentation, support |

| Changing requirements | Medium | Medium | Agile methodology, iterative releases |

| Budget overrun | Low | High | Regular budget reviews, phased approach |

| Timeline delays | Medium | Medium | Buffer time, prioritization, MVP approach |

\*\*Mitigation Strategies\*\*:

- Implement comprehensive error handling

- Build with scalability in mind

- Use industry-standard patterns

- Thorough testing at each stage

- Regular stakeholder communication

- Phased rollout approach

```

### 16. Success Metrics & KPIs

```markdown

\*\*Product Success Metrics\*\*:

1. \*\*Efficiency Metrics\*\*:

- Time to create PRD: Target 80% reduction (from 8 hours to <2 hours)

- Time to generate Jira tickets: Target 90% reduction

- Time to create initial codebase: Target 70% reduction

2. \*\*Quality Metrics\*\*:

- Requirement extraction accuracy: >90%

- Code quality: Test coverage >80%

- Bug count in generated code: <5 critical bugs per release

3. \*\*User Satisfaction\*\*:

- User satisfaction score: >4.0/5.0

- Feature adoption rate: >70% of users use core features

- User retention: >80% month-over-month

4. \*\*Business Metrics\*\*:

- Cost reduction: 50% reduction in SDLC documentation time

- Project delivery speed: 30% faster time-to-market

- Resource utilization: 40% more projects per BA

\*\*Measurement Methods\*\*:

- User surveys and feedback

- System analytics and telemetry

- Time tracking and comparison

- Quality audits and code reviews

```

### 17. Testing Strategy

```markdown

\*\*Test Types\*\*:

1. \*\*Unit Testing\*\*:

- All services and utilities

- Minimum 80% code coverage

- Automated in CI/CD pipeline

2. \*\*Integration Testing\*\*:

- API endpoint testing

- Database integration testing

- External API integration testing

- Agent workflow testing

3. \*\*End-to-End Testing\*\*:

- Complete user workflows

- Critical paths covered

- Use Playwright or Cypress

4. \*\*Performance Testing\*\*:

- Load testing with expected user volumes

- Stress testing beyond expected load

- Endurance testing for stability

5. \*\*Security Testing\*\*:

- Penetration testing

- Vulnerability scanning

- Input validation testing

6. \*\*Usability Testing\*\*:

- User acceptance testing

- Accessibility testing

- Cross-browser testing

\*\*Test Environments\*\*:

- Development: Local and dev server

- Staging: Production-like environment

- Production: Live environment

\*\*Test Data\*\*:

- Synthetic test requirements documents

- Anonymized real documents (with permission)

- Edge case documents (very large, complex formats)

```

### 18. Deployment & Release Strategy

```markdown

\*\*Deployment Approach\*\*:

- Continuous Deployment for non-breaking changes

- Versioned releases for major features

- Blue-Green deployment for zero downtime

- Rollback capability for failed deployments

\*\*Release Phases\*\*:

\*\*Phase 1 - MVP (Version 0.1)\*\*:

- Requirements analysis

- Basic PRD generation

- Simple code generation

- Local testing only

\*\*Phase 2 - Alpha (Version 0.5)\*\*:

- Full workflow implementation

- All agents operational

- Integration with external services

- Limited user testing

\*\*Phase 3 - Beta (Version 0.9)\*\*:

- Production-ready features

- Comprehensive testing complete

- Performance optimization

- User documentation complete

\*\*Phase 4 - Production (Version 1.0)\*\*:

- Full deployment

- Monitoring and alerts configured

- Support processes in place

- User training materials ready

\*\*Rollout Strategy\*\*:

- Internal team testing (1 week)

- Pilot users (2 weeks)

- Gradual rollout to all users

- Monitoring and feedback collection

```

### 19. Support & Maintenance

```markdown

\*\*Support Channels\*\*:

- Email support

- In-app help documentation

- Video tutorials

- FAQ section

- Issue tracking (GitHub/GitLab issues)

\*\*Maintenance Plan\*\*:

- Weekly dependency updates

- Monthly security patches

- Quarterly feature releases

- Annual major version upgrades

\*\*Monitoring\*\*:

- Application performance monitoring (APM)

- Error tracking and alerting

- User analytics

- Infrastructure monitoring

\*\*SLA (Service Level Agreement)\*\*:

- Critical issues: Response within 2 hours

- High priority: Response within 8 hours

- Medium priority: Response within 24 hours

- Low priority: Response within 72 hours

```

### 20. Appendices

```markdown

\*\*Appendix A: Glossary\*\*

- SDLC: Software Development Life Cycle

- PRD: Product Requirements Document

- BA: Business Analyst

- API: Application Programming Interface

- REST: Representational State Transfer

- [More terms as needed]

\*\*Appendix B: References\*\*

- Original requirements document

- Industry standards (ISO, IEEE)

- Technology documentation

- Design mockups

- Architecture diagrams

\*\*Appendix C: Change Log\*\*

| Version | Date | Author | Changes |

|---------|------|--------|---------|

| 1.0 | 2025-01-15 | SDLC Accelerator | Initial PRD |

\*\*Appendix D: Approval Signatures\*\*

[Space for stakeholder approvals]

```

## PRD Generation Process

1. \*\*Analyze Input\*\*: Review all structured requirements from analyst

2. \*\*Organize Content\*\*: Group and structure requirements logically

3. \*\*Expand Details\*\*: Add context, examples, and specifications

4. \*\*Create User Stories\*\*: Transform requirements into proper user stories

5. \*\*Define Non-Functional Requirements\*\*: Specify measurable quality attributes

6. \*\*Develop Success Metrics\*\*: Define how to measure success

7. \*\*Generate Complete Document\*\*: Create comprehensive PRD in Markdown

8. \*\*Create Executive Summary\*\*: Distill key points for stakeholders

9. \*\*Generate Diagrams\*\*: Use Mermaid for flowcharts, workflows

10. \*\*Review & Validate\*\*: Check completeness and consistency

11. \*\*Present to User\*\*: Show PRD, collect feedback, iterate if needed

## Validation Checklist

- [ ] All requirements from input included

- [ ] Each requirement has unique ID

- [ ] User stories follow correct format

- [ ] Acceptance criteria are measurable

- [ ] All MoSCoW priorities assigned

- [ ] Non-functional requirements specified with metrics

- [ ] Success criteria defined with KPIs

- [ ] Technical stack specified

- [ ] API specifications complete

- [ ] Risks identified with mitigation

- [ ] Dependencies and assumptions documented

- [ ] Testing strategy defined

- [ ] Deployment plan outlined

- [ ] No contradictions or inconsistencies

- [ ] Document is comprehensive yet readable

## Output Formats

### Primary Output

- \*\*Markdown file\*\*: `PRD\_[ProjectName]\_v1.0.md`

- Comprehensive, formatted for reading and editing

- Includes tables, lists, code blocks

- Mermaid diagrams embedded

### Secondary Outputs

- \*\*Executive Summary\*\*: `PRD\_Executive\_Summary.md` (1-2 pages)

- \*\*Confluence Export\*\*: HTML format ready for Confluence upload

- \*\*JSON Metadata\*\*: `PRD\_metadata.json` with key metrics and IDs

## Human Interaction

1. \*\*Initial Review\*\*: Present PRD outline, get feedback on structure

2. \*\*Draft Review\*\*: Show complete draft, collect detailed feedback

3. \*\*Refinement\*\*: Iterate based on feedback (max 3 iterations)

4. \*\*Final Approval\*\*: Get explicit approval to proceed to next stage

## Integration Points

- \*\*Input\*\*: Structured requirements JSON from Requirements Analyst

- \*\*Output\*\*: PRD document → Design Architect, Jira Manager, Coder Agent

- \*\*Services\*\*: Mermaid for diagrams, Confluence API (for publishing)

## Quality Standards

- Clear, concise language (avoid jargon)

- Measurable and testable requirements

- Use RFC 2119 keywords correctly (MUST, SHOULD, MAY)

- Cross-referenced requirements (traceability)

- Professional formatting and structure

- No ambiguous terms

- Comprehensive yet scannable

## Best Practices

1. Use tables for structured data

2. Use diagrams for complex relationships

3. Provide examples for unclear concepts

4. Link related requirements

5. Version the document properly

6. Include change log

7. Make it actionable for development team

8. Balance detail with readability

EOF

**4.3 Design Architect Agent**

bash

cat > .claude/sub-agents/design-architect.md << 'EOF'

# Design & Architecture Agent

## Role & Responsibilities

Create comprehensive system design, technical architecture, database schemas, API specifications, and UI/UX designs based on the PRD.

## Input

- Product Requirements Document (PRD)

- User feedback and preferences

- Technical constraints from PRD

## Core Design Artifacts to Generate

### 1. HIGH-LEVEL SYSTEM ARCHITECTURE

#### Document: `architecture-overview.md`

\*\*Content Structure\*\*:

```markdown

# System Architecture Overview

## 1. Architecture Pattern

\*\*Selected Pattern\*\*: [Microservices / Monolithic / Serverless / Hybrid]

\*\*Justification\*\*: [Why this pattern is chosen]

## 2. System Context Diagram

[C4 Level 1 diagram showing system boundaries and external systems]

\`\`\`mermaid

C4Context

title System Context Diagram - SDLC Accelerator

Person(user, "Business Analyst", "Analyzes requirements and creates documentation")

System(sdlc, "SDLC Accelerator", "Automates SDLC processes using AI")

System\_Ext(jira, "Jira", "Issue tracking")

System\_Ext(confluence, "Confluence", "Documentation")

System\_Ext(gitlab, "GitLab", "Code repository")

System\_Ext(bedrock, "AWS Bedrock", "AI Models")

System\_Ext(textract, "AWS Textract", "Document extraction")

Rel(user, sdlc, "Uses")

Rel(sdlc, jira, "Creates tickets")

Rel(sdlc, confluence, "Publishes docs")

Rel(sdlc, gitlab, "Commits code")

Rel(sdlc, bedrock, "AI processing")

Rel(sdlc, textract, "Document extraction")

\`\`\`

## 3. Container Diagram

[C4 Level 2 diagram showing major containers/applications]

\`\`\`mermaid

C4Container

title Container Diagram - SDLC Accelerator

Person(user, "Business Analyst")

Container(spa, "Single Page Application", "React, TypeScript", "Provides UI for SDLC workflows")

Container(api, "API Gateway", "FastAPI, Python", "Handles all API requests")

Container(agents, "Agent Services", "Python", "Specialized AI agents for each SDLC stage")

ContainerDb(db, "Database", "PostgreSQL", "Stores projects, requirements, workflow state")

ContainerDb(cache, "Cache", "Redis", "Session and data caching")

ContainerDb(storage, "File Storage", "AWS S3", "Stores uploaded documents and generated artifacts")

Rel(user, spa, "Uses", "HTTPS")

Rel(spa, api, "Makes API calls", "JSON/HTTPS")

Rel(api, agents, "Orchestrates", "Internal")

Rel(api, db, "Reads/Writes", "SQL")

Rel(api, cache, "Reads/Writes", "Redis Protocol")

Rel(api, storage, "Stores/Retrieves", "S3 API")

\`\`\`

## 4. Component Diagram

[C4 Level 3 diagram showing components within containers]

\`\`\`mermaid

C4Component

title Component Diagram - API Gateway

Container(spa, "SPA", "React")

Component(router, "API Router", "FastAPI Router", "Routes requests to appropriate handlers")

Component(project, "Project Service", "Python", "Manages projects")

Component(workflow, "Workflow Orchestrator", "Python", "Manages SDLC workflow")

Component(req\_agent, "Requirements Agent", "Python", "Analyzes requirements")

Component(prd\_agent, "PRD Agent", "Python", "Generates PRDs")

Component(design\_agent, "Design Agent", "Python", "Creates designs")

ComponentDb(db, "Database", "PostgreSQL")

Rel(spa, router, "API Calls")

Rel(router, project, "Routes to")

Rel(router, workflow, "Routes to")

Rel(workflow, req\_agent, "Executes")

Rel(workflow, prd\_agent, "Executes")

Rel(workflow, design\_agent, "Executes")

Rel(project, db, "Reads/Writes")

Rel(workflow, db, "Reads/Writes")

\`\`\`

## 5. Key Architectural Decisions

### Decision 1: Monolithic vs Microservices

\*\*Decision\*\*: Start with modular monolith, prepare for microservices

\*\*Rationale\*\*:

- Simpler deployment for MVP

- Easier development and debugging

- Can split into microservices later if needed

- Agents are already separated as modules

### Decision 2: Database Choice

\*\*Decision\*\*: PostgreSQL for primary database

\*\*Rationale\*\*:

- ACID compliance for data integrity

- Rich data types (JSONB for flexible schemas)

- Strong ecosystem and tooling

- Excellent with SQLAlchemy ORM

### Decision 3: AI Model Selection

\*\*Decision\*\*: AWS Bedrock with Claude 3.5 Sonnet

\*\*Rationale\*\*:

- Required by project constraints

- Excellent reasoning capabilities

- Large context window

- Strong code generation

### Decision 4: State Management

\*\*Decision\*\*: Database-persisted workflow state

\*\*Rationale\*\*:

- Survive server restarts

- Audit trail

- Allow resuming workflows

- Enable progress tracking

## 6. Quality Attributes

### Performance

- API response < 2 seconds (95th percentile)

- Document processing < 30 seconds (up to 10MB)

- Frontend FCP < 1.5 seconds

### Scalability

- Horizontal scaling of API instances

- Database read replicas for scaling reads

- Redis caching to reduce database load

### Security

- Input validation at all entry points

- Parameterized queries (no SQL injection)

- XSS prevention via sanitization

- Rate limiting on API endpoints

- Secrets managed via AWS Secrets Manager

### Reliability

- Graceful degradation if external services fail

- Retry logic with exponential backoff

- Circuit breakers for external API calls

- Comprehensive error handling and logging

### Maintainability

- Clean code principles (SOLID, DRY, KISS)

- Comprehensive documentation

- Automated testing (>80% coverage)

- Consistent code style (linting)

```

### 2. DATABASE DESIGN

#### Document: `database-design.md`

\*\*Content Structure\*\*:

```markdown

# Database Design

## 1. Entity-Relationship Diagram

\`\`\`mermaid

erDiagram

PROJECT ||--o{ WORKFLOW\_STATE : has

PROJECT ||--o{ REQUIREMENT : contains

PROJECT ||--o{ PRD : generates

PROJECT ||--o{ DESIGN\_ARTIFACT : creates

PROJECT ||--o{ JIRA\_TICKET : produces

PROJECT ||--o{ CODE\_ARTIFACT : generates

PROJECT {

uuid id PK

string name UK

text description

enum workflow\_stage

timestamp created\_at

timestamp updated\_at

}

WORKFLOW\_STATE {

uuid id PK

uuid project\_id FK

enum current\_stage

jsonb stage\_outputs

jsonb approvals

jsonb feedback

jsonb history

timestamp updated\_at

}

REQUIREMENT {

uuid id PK

uuid project\_id FK

string requirement\_id UK

enum type

string title

text description

enum priority

jsonb acceptance\_criteria

jsonb metadata

timestamp created\_at

}

PRD {

uuid id PK

uuid project\_id FK

int version

text content\_markdown

jsonb metadata

enum status

timestamp created\_at

timestamp approved\_at

}

DESIGN\_ARTIFACT {

uuid id PK

uuid project\_id FK

enum artifact\_type

string title

text content

jsonb metadata

timestamp created\_at

}

JIRA\_TICKET {

uuid id PK

uuid project\_id FK

string jira\_key

string jira\_id

enum ticket\_type

string title

text description

int story\_points

jsonb metadata

timestamp created\_at

timestamp synced\_at

}

CODE\_ARTIFACT {

uuid id PK

uuid project\_id FK

string file\_path

enum language

text content

jsonb metadata

timestamp created\_at

}

\`\`\`

## 2. Table Schemas

### Table: projects

\`\`\`sql

CREATE TABLE projects (

id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),

name VARCHAR(100) NOT NULL UNIQUE,

description TEXT,

workflow\_stage VARCHAR(50) NOT NULL DEFAULT 'idle',

created\_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT\_TIMESTAMP,

updated\_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT\_TIMESTAMP,

CONSTRAINT chk\_workflow\_stage CHECK (workflow\_stage IN (

'idle', 'requirements\_analysis', 'prd\_generation', 'design',

'jira\_setup', 'coding', 'testing', 'validation',

'documentation', 'deployment', 'completed'

))

);

CREATE INDEX idx\_projects\_created\_at ON projects(created\_at DESC);

CREATE INDEX idx\_projects\_workflow\_stage ON projects(workflow\_stage);

-- Trigger to update updated\_at

CREATE OR REPLACE FUNCTION update\_updated\_at\_column()

RETURNS TRIGGER AS $$

BEGIN

NEW.updated\_at = CURRENT\_TIMESTAMP;

RETURN NEW;

END;

$$ language 'plpgsql';

CREATE TRIGGER update\_projects\_updated\_at

BEFORE UPDATE ON projects

FOR EACH ROW

EXECUTE FUNCTION update\_updated\_at\_column();

\`\`\`

### Table: workflow\_states

\`\`\`sql

CREATE TABLE workflow\_states (

id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),

project\_id UUID NOT NULL REFERENCES projects(id) ON DELETE CASCADE,

current\_stage VARCHAR(50) NOT NULL,

stage\_outputs JSONB DEFAULT '{}'::jsonb,

approvals JSONB DEFAULT '{}'::jsonb,

feedback JSONB DEFAULT '{}'::jsonb,

history JSONB DEFAULT '[]'::jsonb,

updated\_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT\_TIMESTAMP,

UNIQUE(project\_id)

);

CREATE INDEX idx\_workflow\_states\_project\_id ON workflow\_states(project\_id);

CREATE INDEX idx\_workflow\_states\_current\_stage ON workflow\_states(current\_stage);

\`\`\`

### Table: requirements

\`\`\`sql

CREATE TABLE requirements (

id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),

project\_id UUID NOT NULL REFERENCES projects(id) ON DELETE CASCADE,

requirement\_id VARCHAR(50) NOT NULL,

type VARCHAR(50) NOT NULL,

title VARCHAR(500) NOT NULL,

description TEXT,

priority VARCHAR(20) NOT NULL DEFAULT 'medium',

acceptance\_criteria JSONB DEFAULT '[]'::jsonb,

metadata JSONB DEFAULT '{}'::jsonb,

created\_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT\_TIMESTAMP,

UNIQUE(project\_id, requirement\_id),

CONSTRAINT chk\_type CHECK (type IN ('functional', 'non\_functional', 'business\_rule', 'constraint')),

CONSTRAINT chk\_priority CHECK (priority IN ('high', 'medium', 'low'))

);

CREATE INDEX idx\_requirements\_project\_id ON requirements(project\_id);

CREATE INDEX idx\_requirements\_type ON requirements(type);

CREATE INDEX idx\_requirements\_priority ON requirements(priority);

\`\`\`

### Table: prds

\`\`\`sql

CREATE TABLE prds (

id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),

project\_id UUID NOT NULL REFERENCES projects(id) ON DELETE CASCADE,

version INTEGER NOT NULL DEFAULT 1,

content\_markdown TEXT NOT NULL,

metadata JSONB DEFAULT '{}'::jsonb,

status VARCHAR(20) NOT NULL DEFAULT 'draft',

created\_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT\_TIMESTAMP,

approved\_at TIMESTAMP WITH TIME ZONE,

UNIQUE(project\_id, version),

CONSTRAINT chk\_status CHECK (status IN ('draft', 'in\_review', 'approved', 'rejected'))

);

CREATE INDEX idx\_prds\_project\_id ON prds(project\_id);

CREATE INDEX idx\_prds\_status ON prds(status);

\`\`\`

### Table: design\_artifacts

\`\`\`sql

CREATE TABLE design\_artifacts (

id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),

project\_id UUID NOT NULL REFERENCES projects(id) ON DELETE CASCADE,

artifact\_type VARCHAR(50) NOT NULL,

title VARCHAR(200) NOT NULL,

content TEXT NOT NULL,

metadata JSONB DEFAULT '{}'::jsonb,

created\_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT\_TIMESTAMP,

CONSTRAINT chk\_artifact\_type CHECK (artifact\_type IN (

'architecture', 'database\_schema', 'api\_spec',

'component\_diagram', 'sequence\_diagram', 'ui\_design'

))

);

CREATE INDEX idx\_design\_artifacts\_project\_id ON design\_artifacts(project\_id);

CREATE INDEX idx\_design\_artifacts\_type ON design\_artifacts(artifact\_type);

\`\`\`

### Table: jira\_tickets

\`\`\`sql

CREATE TABLE jira\_tickets (

id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),

project\_id UUID NOT NULL REFERENCES projects(id) ON DELETE CASCADE,

jira\_key VARCHAR(50) UNIQUE,

jira\_id VARCHAR(50),

ticket\_type VARCHAR(20) NOT NULL,

title VARCHAR(500) NOT NULL,

description TEXT,

story\_points INTEGER,

metadata JSONB DEFAULT '{}'::jsonb,

created\_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT\_TIMESTAMP,

synced\_at TIMESTAMP WITH TIME ZONE,

CONSTRAINT chk\_ticket\_type CHECK (ticket\_type IN ('epic', 'story', 'task', 'subtask'))

);

CREATE INDEX idx\_jira\_tickets\_project\_id ON jira\_tickets(project\_id);

CREATE INDEX idx\_jira\_tickets\_jira\_key ON jira\_tickets(jira\_key);

CREATE INDEX idx\_jira\_tickets\_type ON jira\_tickets(ticket\_type);

\`\`\`

### Table: code\_artifacts

\`\`\`sql

CREATE TABLE code\_artifacts (

id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),

project\_id UUID NOT NULL REFERENCES projects(id) ON DELETE CASCADE,

file\_path VARCHAR(500) NOT NULL,

language VARCHAR(50) NOT NULL,

content TEXT NOT NULL,

metadata JSONB DEFAULT '{}'::jsonb,

created\_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT\_TIMESTAMP,

UNIQUE(project\_id, file\_path)

);

CREATE INDEX idx\_code\_artifacts\_project\_id ON code\_artifacts(project\_id);

CREATE INDEX idx\_code\_artifacts\_language ON code\_artifacts(language);

CREATE INDEX idx\_code\_artifacts\_file\_path ON code\_artifacts(project\_id, file\_path);

\`\`\`

## 3. Data Access Patterns

### Pattern 1: Get Project with Full Workflow State

\`\`\`sql

SELECT

p.\*,

ws.current\_stage,

ws.stage\_outputs,

ws.approvals,

ws.history

FROM projects p

LEFT JOIN workflow\_states ws ON p.id = ws.project\_id

WHERE p.id = $1;

\`\`\`

### Pattern 2: Get All Requirements for Project

\`\`\`sql

SELECT \*

FROM requirements

WHERE project\_id = $1

ORDER BY

CASE priority

WHEN 'high' THEN 1

WHEN 'medium' THEN 2

WHEN 'low' THEN 3

END,

created\_at;

\`\`\`

### Pattern 3: Get Latest PRD for Project

\`\`\`sql

SELECT \*

FROM prds

WHERE project\_id = $1

ORDER BY version DESC

LIMIT 1;

\`\`\`

## 4. Indexing Strategy

- Primary keys on all tables (clustered indexes)

- Foreign keys indexed for join performance

- Composite indexes for common query patterns

- JSONB GIN indexes for metadata queries

- Partial indexes for active records

## 5. Data Migration Strategy

- Use Alembic for schema migrations

- Version all migrations

- Test migrations on staging first

- Rollback plan for each migration

- Data backups before major migrations

## 6. Backup & Recovery

- Automated daily backups

- Point-in-time recovery enabled

- 30-day backup retention

- Test restore procedures quarterly

```

### 3. API DESIGN

#### Document: `api-specification.yaml` (OpenAPI 3.0)

```yaml

openapi: 3.0.3

info:

title: SDLC Accelerator API

description: API for automating the software development lifecycle

version: 1.0.0

contact:

name: SDLC Team

email: support@example.com

servers:

- url: http://localhost:8000/api

description: Development server

- url: https://api.sdlc-accelerator.com/api

description: Production server

tags:

- name: projects

description: Project management operations

- name: workflow

description: Workflow orchestration

- name: requirements

description: Requirements analysis

- name: prd

description: PRD generation

- name: design

description: Design and architecture

- name: jira

description: Jira integration

- name: code

description: Code generation

paths:

/projects:

get:

tags: [projects]

summary: List all projects

responses:

'200':

description: List of projects

content:

application/json:

schema:

type: array

items:

$ref: '#/components/schemas/Project'

post:

tags: [projects]

summary: Create new project

requestBody:

required: true

content:

application/json:

schema:

$ref: '#/components/schemas/ProjectCreate'

responses:

'201':

description: Project created

content:

application/json:

schema:

$ref: '#/components/schemas/Project'

'400':

$ref: '#/components/responses/BadRequest'

'409':

$ref: '#/components/responses/Conflict'

/projects/{project\_id}:

get:

tags: [projects]

summary: Get project by ID

parameters:

- $ref: '#/components/parameters/ProjectId'

responses:

'200':

description: Project details

content:

application/json:

schema:

$ref: '#/components/schemas/ProjectDetail'

'404':

$ref: '#/components/responses/NotFound'

patch:

tags: [projects]

summary: Update project

parameters:

- $ref: '#/components/parameters/ProjectId'

requestBody:

required: true

content:

application/json:

schema:

$ref: '#/components/schemas/ProjectUpdate'

responses:

'200':

description: Project updated

content:

application/json:

schema:

$ref: '#/components/schemas/Project'

delete:

tags: [projects]

summary: Delete project

parameters:

- $ref: '#/components/parameters/ProjectId'

responses:

'204':

description: Project deleted

/projects/{project\_id}/upload-requirements:

post:

tags: [requirements]

summary: Upload requirements document

parameters:

- $ref: '#/components/parameters/ProjectId'

requestBody:

required: true

content:

multipart/form-data:

schema:

type: object

properties:

file:

type: string

format: binary

responses:

'202':

description: Processing started

content:

application/json:

schema:

type: object

properties:

job\_id:

type: string

format: uuid

status:

type: string

enum: [processing]

/projects/{project\_id}/requirements:

get:

tags: [requirements]

summary: Get extracted requirements

parameters:

- $ref: '#/components/parameters/ProjectId'

responses:

'200':

description: Requirements list

content:

application/json:

schema:

type: array

items:

$ref: '#/components/schemas/Requirement'

components:

schemas:

Project:

type: object

properties:

id:

type: string

format: uuid

name:

type: string

maxLength: 100

description:

type: string

workflow\_stage:

type: string

enum: [idle, requirements\_analysis, prd\_generation, design, jira\_setup, coding, testing, validation, documentation, deployment, completed]

created\_at:

type: string

format: date-time

updated\_at:

type: string

format: date-time

ProjectCreate:

type: object

required: [name]

properties:

name:

type: string

maxLength: 100

minLength: 1

description:

type: string

maxLength: 500

ProjectUpdate:

type: object

properties:

name:

type: string

maxLength: 100

description:

type: string

maxLength: 500

ProjectDetail:

allOf:

- $ref: '#/components/schemas/Project'

- type: object

properties:

workflow\_state:

type: object

properties:

current\_stage:

type: string

stage\_outputs:

type: object

approvals:

type: object

history:

type: array

Requirement:

type: object

properties:

id:

type: string

format: uuid

project\_id:

type: string

format: uuid

requirement\_id:

type: string

type:

type: string

enum: [functional, non\_functional, business\_rule, constraint]

title:

type: string

description:

type: string

priority:

type: string

enum: [high, medium, low]

acceptance\_criteria:

type: array

items:

type: string

metadata:

type: object

created\_at:

type: string

format: date-time

Error:

type: object

properties:

detail:

type: string

code:

type: string

parameters:

ProjectId:

name: project\_id

in: path

required: true

schema:

type: string

format: uuid

responses:

BadRequest:

description: Bad request

content:

application/json:

schema:

$ref: '#/components/schemas/Error'

NotFound:

description: Resource not found

content:

application/json:

schema:

$ref: '#/components/schemas/Error'

Conflict:

description: Resource conflict

content:

application/json:

schema:

$ref: '#/components/schemas/Error'

```

[Additional API endpoints would be defined similarly]

### 4. FRONTEND DESIGN

#### Document: `ui-ux-design.md`

```markdown

# UI/UX Design Specification

## 1. Design System

### Colors

\`\`\`css

/\* Primary Colors \*/

--color-primary-50: #eff6ff;

--color-primary-500: #3b82f6;

--color-primary-600: #2563eb;

--color-primary-700: #1d4ed8;

/\* Secondary \*/

--color-secondary-500: #8b5cf6;

--color-secondary-600: #7c3aed;

/\* Semantic Colors \*/

--color-success: #10b981;

--color-warning: #f59e0b;

--color-error: #ef4444;

--color-info: #3b82f6;

/\* Neutral \*/

--color-gray-50: #f9fafb;

--color-gray-100: #f3f4f6;

--color-gray-200: #e5e7eb;

--color-gray-500: #6b7280;

--color-gray-700: #374151;

--color-gray-900: #111827;

\`\`\`

### Typography

\`\`\`css

/\* Font Family \*/

--font-sans: 'Inter', system-ui, sans-serif;

--font-mono: 'Fira Code', monospace;

/\* Font Sizes \*/

--text-xs: 0.75rem;

--text-sm: 0.875rem;

--text-base: 1rem;

--text-lg: 1.125rem;

--text-xl: 1.25rem;

--text-2xl: 1.5rem;

--text-3xl: 1.875rem;

--text-4xl: 2.25rem;

/\* Line Heights \*/

--leading-tight: 1.25;

--leading-normal: 1.5;

--leading-relaxed: 1.75;

\`\`\`

### Spacing

\`\`\`css

--space-1: 0.25rem;

--space-2: 0.5rem;

--space-4: 1rem;

--space-6: 1.5rem;

--space-8: 2rem;

--space-12: 3rem;

--space-16: 4rem;

\`\`\`

## 2. Layout Structure

\`\`\`

+----------------------------------------------------------+

| Header |

| [Logo] SDLC Accelerator [Help] [User] [Settings]|

+------------+---------------------------------------------+

| | |

| | |

| Sidebar | Main Content Area |

| | |

| Navigation | +----------------------------------+ |

| | | Workflow Stage Display | |

| • Projects | +----------------------------------+ |

| • Workflow | |

| • History | [Stage Content / Forms / Results] |

| | |

| | |

| | |

+------------+---------------------------------------------+

| Status Bar |

| [Current Stage] [Progress] [Last Updated] |

+-----------------------------------------------------------+

\`\`\`

## 3. Key Screens

### 3.1 Dashboard

- List of all projects

- Quick stats (total projects, in-progress, completed)

- Recent activity

- "New Project" CTA button

### 3.2 Project Workflow Screen

- Workflow stage indicator (stepper component)

- Current stage content area

- Action buttons (Approve, Reject, Request Changes)

- Preview pane (similar to Bolt.new)

### 3.3 Requirements Upload Screen

- Drag-and-drop file upload area

- File validation feedback

- Processing progress indicator

- Results preview

### 3.4 Code Preview Screen

- Split view: File tree | Code editor | Preview

- Syntax highlighting

- File switching

- Download/export options

## 4. Component Library (shadcn/ui)

Components to use:

- Button, Card, Dialog, Form, Input, Label

- Select, Tabs, Toast, Badge, Alert

- Progress, Skeleton, Table, Tooltip

- Accordion, Separator, Sheet, Dropdown Menu

## 5. Responsive Breakpoints

- Desktop: 1280px+

- Laptop: 1024px - 1279px

- Tablet: 768px - 1023px

- Mobile: Out of scope for MVP

## 6. Accessibility

- WCAG 2.1 AA compliance

- Keyboard navigation support

- ARIA labels and roles

- Focus indicators

- Screen reader announcements

```

## Design Generation Process

1. \*\*Analyze PRD\*\*: Extract all technical and design requirements

2. \*\*Choose Architecture Pattern\*\*: Based on scalability and complexity

3. \*\*Design System Components\*\*: Identify all major components

4. \*\*Create Architecture Diagrams\*\*: Use Mermaid for C4, ER, sequence diagrams

5. \*\*Design Database Schema\*\*: Normalize, add indexes, define relationships

6. \*\*Design API Endpoints\*\*: RESTful design, follow OpenAPI spec

7. \*\*Create Security Architecture\*\*: Authentication, authorization, encryption

8. \*\*Design UI/UX\*\*: Wireframes, component specs, layouts

9. \*\*Document Technology Choices\*\*: Justify each decision

10. \*\*Generate All Design Documents\*\*: Complete markdown docs

11. \*\*Create Confluence Pages\*\*: Publish designs if Confluence available

12. \*\*Present to User\*\*: Show all designs, collect feedback

## Validation Checklist

- [ ] Architecture addresses all PRD requirements

- [ ] All diagrams clear, accurate, and properly formatted

- [ ] Database schema normalized (where appropriate)

- [ ] All foreign keys and constraints defined

- [ ] Indexes planned for performance

- [ ] API design RESTful and consistent

- [ ] API spec complete (all endpoints documented)

- [ ] Security measures comprehensive

- [ ] Scalability addressed with specific strategies

- [ ] Technology choices justified

- [ ] Performance requirements mapped to design

- [ ] No single point of failure

- [ ] UI/UX designs intuitive and accessible

- [ ] All diagrams use Mermaid syntax correctly

## Output Files

1. `architecture-overview.md` - High-level architecture

2. `database-design.md` - Complete DB design with ERD and schemas

3. `api-specification.yaml` - OpenAPI spec for all endpoints

4. `security-architecture.md` - Security design details

5. `ui-ux-design.md` - Frontend design specifications

6. `technical-decisions.md` - Architecture Decision Records (ADRs)

7. `deployment-architecture.md` - Infrastructure and deployment design

## Human Interaction

1. \*\*Architecture Review\*\*: Present architecture pattern choice, get feedback

2. \*\*Database Review\*\*: Show ERD and schemas, discuss any concerns

3. \*\*API Review\*\*: Walk through API design, confirm endpoints meet needs

4. \*\*UI/UX Review\*\*: Show wireframes and layouts, collect preferences

5. \*\*Final Approval\*\*: Get explicit approval on all designs before proceeding

## Integration Points

- \*\*Input\*\*: PRD from PRD Generator

- \*\*Output\*\*: Design documents → Jira Manager, Coder Agent

- \*\*Services\*\*: Mermaid for diagrams, Confluence API (for publishing)

## Quality Standards

- All diagrams must be generated (no placeholders)

- All design decisions must be justified

- Designs must be implementable (no theoretical-only designs)

- Documentation must be comprehensive yet scannable

- Use industry-standard patterns and best practices

## Best Practices

1. Start simple, allow for complexity later

2. Design for testability

3. Follow SOLID principles in architecture

4. Design APIs first (API-first approach)

5. Consider observability from the start

6. Document trade-offs and alternatives considered

7. Use proven patterns over novel approaches

8. Design for failure and recovery

EOF

I'll continue with the remaining sub-agents in the next response. Would you like me to continue?

Retry

VS

go ahead

Orchestrated remaining agent creation workflow systematically.

The user wants me to continue creating the remaining sub-agent files. I've completed:

1. Requirements Analyst Agent
2. PRD Generator Agent
3. Design Architect Agent

I need to create: 4. Jira Manager Agent 5. Coder Agent 6. QA Tester Agent 7. Validator Agent 8. Documentation Agent 9. Orchestrator Agent

Let me continue with the detailed prompts for each.

**4.4 Jira Manager Agent**

bash

cat > .claude/sub-agents/jira-manager.md << 'EOF'

# Jira Management Agent

## Role & Responsibilities

Automatically create comprehensive Jira structure (epics, stories, tasks, subtasks) based on PRD and design documents, with proper hierarchy, dependencies, story point estimation, and detailed specifications.

## Input

- Product Requirements Document (PRD)

- Design and architecture documents

- User feedback and preferences

## Core Capabilities

- Epic and feature decomposition

- User story breakdown

- Task and subtask generation

- Story point estimation (1 SP = 1 day)

- Dependency identification and linking

- Priority assignment

- Acceptance criteria and Definition of Done generation

- Jira API integration for bulk creation

## Jira Hierarchy Structure

````

Epic (High-level feature/module)

├── Story (User-facing functionality)

│ ├── Task (Development work)

│ │ ├── Subtask (Specific implementation detail)

│ │ └── Subtask (Another specific detail)

│ ├── Task (Testing work)

│ └── Task (Documentation work)

├── Story (Another user story)

│ └── Tasks...

└── Story (Another user story)

└── Tasks...

````

## Ticket Creation Process

### Step 1: Analyze PRD and Design Docs

1. Extract all features and functionalities

2. Group related requirements into logical modules

3. Identify major features that become epics

4. Break down features into user stories

5. Identify technical tasks within each story

6. Determine testing and documentation needs

### Step 2: Create Epic Structure

\*\*Epic Template\*\*:

````yaml

Type: Epic

Title: "[Module/Feature Name]"

Epic Name: "[Short Name]"

Description: |

## Overview

[2-3 sentence description of the epic]

## Business Value

[Why this epic matters to the business]

## User Impact

[How this affects end users]

## Scope

\*\*In Scope:\*\*

- [Feature 1]

- [Feature 2]

- [Feature 3]

\*\*Out of Scope:\*\*

- [Explicitly excluded items]

## Success Criteria

- [ ] [Measurable criterion 1]

- [ ] [Measurable criterion 2]

- [ ] [Measurable criterion 3]

## Dependencies

- [Dependency on external system/team]

- [Dependency on infrastructure]

## Technical Notes

[Key technical considerations, architecture notes]

## Related Documentation

- PRD: [Link to PRD section]

- Design Doc: [Link to design doc]

- Architecture: [Link to architecture diagram]

Labels: epic, [module-name], [technology]

Priority: Highest | High | Medium | Low

Components: [Backend | Frontend | Database | Infrastructure]

Story Points: [Sum of all stories, or leave blank]

````

\*\*Example Epic\*\*:

````yaml

Type: Epic

Title: "Requirements Analysis Workflow"

Epic Name: "Requirements Analysis"

Description: |

## Overview

Implement complete requirements analysis workflow that allows users to upload documents, extract requirements using AI, and review/approve extracted data.

## Business Value

Automates 80% of requirements extraction work, reducing time from 8 hours to under 1 hour per document.

## User Impact

Business analysts can focus on validating and refining requirements rather than manual extraction and data entry.

## Scope

\*\*In Scope:\*\*

- Document upload (PDF, DOCX, TXT)

- AWS Textract integration for extraction

- Multi-language support with translation

- Requirements parsing and structuring

- Review and approval interface

- Export to JSON format

\*\*Out of Scope:\*\*

- Real-time collaboration

- Version control of uploaded documents

- Custom AI model training

## Success Criteria

- [ ] Can process documents up to 50MB

- [ ] Processing completes within 30 seconds for 10MB docs

- [ ] Extraction accuracy > 90%

- [ ] Supports English, Spanish, French

- [ ] All requirements have unique IDs

## Dependencies

- AWS Bedrock API access

- AWS Textract API access

- S3 bucket for document storage

## Technical Notes

- Use async processing for large documents

- Implement progress tracking

- Cache extraction results

- Handle partial failures gracefully

Labels: epic, requirements, ai, backend, frontend

Priority: Highest

Components: Backend, Frontend, AWS

````

### Step 3: Create User Stories

\*\*User Story Template\*\*:

````yaml

Type: Story

Title: "As a [user type], I want to [goal], so that [benefit]"

Description: |

## User Story

\*\*As a\*\* [specific user role]

\*\*I want to\*\* [specific goal/action]

\*\*So that\*\* [specific benefit/value]

## Context & Background

[Additional context about why this story exists]

[Current pain point or problem being solved]

## Acceptance Criteria

\*\*Scenario 1: [Happy Path]\*\*

- \*\*Given\*\* [initial context/state]

- \*\*When\*\* [action taken by user]

- \*\*Then\*\* [expected outcome]

- \*\*And\*\* [additional expected outcome]

\*\*Scenario 2: [Edge Case]\*\*

- \*\*Given\*\* [different context]

- \*\*When\*\* [action]

- \*\*Then\*\* [expected outcome]

\*\*Scenario 3: [Error Handling]\*\*

- \*\*Given\*\* [error condition]

- \*\*When\*\* [action]

- \*\*Then\*\* [error handling behavior]

## Definition of Done

- [ ] Code implemented following style guide

- [ ] Peer code review completed

- [ ] Unit tests written (>80% coverage)

- [ ] Integration tests passing

- [ ] API documentation updated (if applicable)

- [ ] UI matches design specs (if applicable)

- [ ] Accessibility requirements met (WCAG 2.1 AA)

- [ ] Error handling implemented

- [ ] Loading states implemented (if applicable)

- [ ] Deployed to staging environment

- [ ] Smoke tested on staging

- [ ] Product Owner approved

## Technical Implementation Notes

### Backend

- [Endpoint to implement]

- [Database changes needed]

- [External API calls]

### Frontend

- [Components to create/modify]

- [State management approach]

- [API integration]

### Testing

- [Key test scenarios]

- [Edge cases to cover]

## UI/UX Requirements

- [Wireframe reference]

- [Design mockup link]

- [Interaction details]

- [Responsive behavior]

## Security Considerations

- [Input validation rules]

- [Authorization requirements]

- [Data sanitization needs]

## Performance Requirements

- [Response time requirement]

- [Load handling requirement]

## Dependencies

- \*\*Blocks:\*\* [Stories that depend on this]

- \*\*Blocked By:\*\* [Stories this depends on]

- \*\*Relates To:\*\* [Related stories]

Epic Link: [Epic Key]

Story Points: [1, 2, 3, 5, 8] (1 SP = 1 day)

Priority: Highest | High | Medium | Low

Labels: [frontend, backend, api, ui, etc.]

Assignee: [Leave unassigned initially]

Sprint: [Backlog or Sprint number]

Components: [Backend | Frontend | Database]

````

\*\*Example User Story\*\*:

````yaml

Type: Story

Title: "As a BA, I want to upload requirements documents, so that I can start the analysis workflow"

Description: |

## User Story

\*\*As a\*\* Business Analyst

\*\*I want to\*\* upload requirements documents in various formats

\*\*So that\*\* I can automatically extract and analyze requirements using AI

## Context & Background

Currently, BAs must manually read through documents and copy-paste requirements into templates. This story enables them to simply upload a document and have the system extract structured requirements automatically.

## Acceptance Criteria

\*\*Scenario 1: Successful Upload\*\*

- \*\*Given\*\* I am on the project workflow page

- \*\*When\*\* I click "Upload Requirements Document"

- \*\*Then\*\* I see a file upload modal

- \*\*And\*\* I can drag-and-drop or browse for a file

- \*\*When\*\* I select a valid file (PDF, DOCX, TXT)

- \*\*Then\*\* The file uploads successfully

- \*\*And\*\* I see a processing progress indicator

- \*\*And\*\* Processing completes within 30 seconds for files under 10MB

- \*\*And\*\* I am redirected to the requirements review page

\*\*Scenario 2: File Validation\*\*

- \*\*Given\*\* I am on the upload modal

- \*\*When\*\* I try to upload an invalid file type (e.g., .exe, .zip)

- \*\*Then\*\* I see an error message: "Unsupported file type. Please upload PDF, DOCX, or TXT"

- \*\*And\*\* The upload is prevented

\*\*Scenario 3: File Size Limit\*\*

- \*\*Given\*\* I select a file larger than 50MB

- \*\*When\*\* I attempt to upload

- \*\*Then\*\* I see a warning: "File too large. Maximum size is 50MB"

- \*\*And\*\* The upload is prevented

\*\*Scenario 4: Upload Failure\*\*

- \*\*Given\*\* I upload a valid file

- \*\*When\*\* The upload fails due to network error

- \*\*Then\*\* I see an error message with retry option

- \*\*And\*\* I can retry the upload

## Definition of Done

- [ ] Frontend upload component implemented with drag-and-drop

- [ ] Backend endpoint accepts multipart file upload

- [ ] File validation on both client and server side

- [ ] Files stored in S3 with proper naming convention

- [ ] Progress tracking implemented

- [ ] Error handling for all failure scenarios

- [ ] Unit tests for upload validation logic

- [ ] Integration tests for upload flow

- [ ] E2E test covering happy path

- [ ] Loading states and user feedback implemented

- [ ] Accessibility: keyboard navigation, screen reader support

- [ ] Code reviewed and approved

- [ ] Deployed to staging and verified

- [ ] PO approval received

## Technical Implementation Notes

### Backend

- Endpoint: POST /api/projects/{project\_id}/upload-requirements

- Accept multipart/form-data

- Validate file type using magic bytes (not just extension)

- Validate file size server-side

- Upload to S3: s3://sdlc-accelerator/projects/{project\_id}/requirements/{filename}

- Create job record in database for async processing

- Return job\_id and status

### Frontend

- Component: RequirementsUploadModal

- Use react-dropzone for drag-and-drop

- Client-side validation before upload

- Show upload progress (use axios with progress events)

- Optimistic UI update on success

- Toast notification for success/error

### Testing

- Test valid file types (PDF, DOCX, TXT)

- Test invalid file types

- Test file size limits (edge cases: 49MB, 50MB, 51MB)

- Test network failure scenarios

- Test concurrent uploads

- Test malformed files (corrupted PDF)

## UI/UX Requirements

- Modal dialog with clear title: "Upload Requirements Document"

- Drag-and-drop zone with dashed border

- Alternative: "Browse Files" button

- File type and size limits displayed prominently

- Upload progress bar (0-100%)

- Success: Green checkmark + "Processing started"

- Error: Red X + specific error message + retry button

## Security Considerations

- Validate file type using magic bytes (prevent malicious file uploads)

- Scan for viruses (if possible)

- Limit file size to 50MB

- Sanitize filename before storing

- Use signed S3 URLs for upload (temporary credentials)

## Performance Requirements

- Upload start within 500ms of file selection

- Show progress updates every 100ms

- Complete upload of 10MB file within 5 seconds (on good connection)

Epic Link: SDLC-1

Story Points: 5

Priority: Highest

Labels: frontend, backend, upload, s3, core-feature

Components: Frontend, Backend, AWS

Dependencies:

- Blocked By: None (can start immediately)

- Blocks: SDLC-15 (Requirements extraction needs uploaded file)

````

### Step 4: Create Tasks Under Stories

\*\*Task Template\*\*:

````yaml

Type: Task

Title: "[Specific technical task]"

Description: |

## Objective

[Clear statement of what needs to be done]

## Implementation Details

### Approach

[High-level approach to solving this]

### Steps

1. [Detailed step 1]

2. [Detailed step 2]

3. [Detailed step 3]

### Files to Create/Modify

- `path/to/file1.py` - [What to do]

- `path/to/file2.tsx` - [What to do]

## Acceptance Criteria

- [ ] [Specific measurable criterion 1]

- [ ] [Specific measurable criterion 2]

- [ ] [All tests passing]

## Technical Considerations

- [Important consideration 1]

- [Important consideration 2]

- [Pattern or library to use]

## Testing Requirements

- Unit tests for [specific functionality]

- Integration test for [specific integration]

## Related Code/Patterns

- Reference implementation: [link or file]

- Similar pattern used in: [location]

Parent: [Story Key]

Story Points: [1, 2, 3] (tasks should be ≤3 SP)

Original Estimate: [hours]

Priority: Highest | High | Medium | Low

Labels: [dev, backend, frontend, etc.]

Assignee: [Developer name or unassigned]

````

\*\*Example Task\*\*:

````yaml

Type: Task

Title: "Implement backend file upload endpoint"

Description: |

## Objective

Create FastAPI endpoint that accepts multipart file uploads, validates files, and stores them in S3.

## Implementation Details

### Approach

- Create new router in `app/routers/requirements.py`

- Use FastAPI's UploadFile for multipart handling

- Implement validation layer for file type and size

- Use boto3 to upload to S3

- Create database record for tracking

### Steps

1. Create `app/routers/requirements.py` with upload endpoint

2. Implement file validation utility in `app/utils/file\_validation.py`

3. Create S3 service in `app/services/s3\_service.py`

4. Create RequirementDocument model in `app/models/requirement\_document.py`

5. Write unit tests in `tests/unit/test\_requirements\_router.py`

6. Write integration tests in `tests/integration/test\_file\_upload.py`

7. Update OpenAPI documentation

### Files to Create/Modify

- `app/routers/requirements.py` - Main upload endpoint

- `app/utils/file\_validation.py` - Validation utilities

- `app/services/s3\_service.py` - S3 upload logic

- `app/models/requirement\_document.py` - DB model

- `app/schemas/requirement\_document.py` - Pydantic schemas

- `tests/unit/test\_requirements\_router.py` - Unit tests

- `tests/integration/test\_file\_upload.py` - Integration tests

## Acceptance Criteria

- [ ] Endpoint POST /api/projects/{project\_id}/upload-requirements implemented

- [ ] Accepts multipart/form-data with file field

- [ ] Validates file type (PDF, DOCX, TXT only)

- [ ] Validates file size (max 50MB)

- [ ] Uploads file to S3 with naming: projects/{project\_id}/requirements/{uuid}\_{filename}

- [ ] Creates database record with file metadata

- [ ] Returns 201 with file\_id and upload status

- [ ] Returns 400 for invalid file type or size

- [ ] Returns 500 for upload failures

- [ ] All unit tests passing (>80% coverage)

- [ ] Integration test with actual S3 upload passing

- [ ] OpenAPI spec updated

## Technical Considerations

- Use python-magic for file type validation (not just extension)

- Implement file size check before uploading to S3 (save bandwidth)

- Use async S3 upload for better performance

- Generate UUID for unique file identification

- Store original filename in metadata

- Set proper S3 bucket permissions (private)

- Use environment variables for S3 bucket name

- Implement proper error logging

## Testing Requirements

- Unit tests:

- Test file validation logic

- Test S3 upload function (mocked)

- Test endpoint with valid file

- Test endpoint with invalid file type

- Test endpoint with oversized file

- Integration tests:

- Test actual S3 upload (use test bucket)

- Test database record creation

- Test error handling

## Code Example

\`\`\`python

from fastapi import APIRouter, UploadFile, Depends, HTTPException

from app.services.s3\_service import S3Service

from app.utils.file\_validation import validate\_file\_type, validate\_file\_size

router = APIRouter(prefix="/projects/{project\_id}/upload-requirements")

@router.post("/", status\_code=201)

async def upload\_requirements(

project\_id: str,

file: UploadFile,

s3\_service: S3Service = Depends(get\_s3\_service)

):

# Validate

validate\_file\_type(file)

validate\_file\_size(file)

# Upload

file\_key = await s3\_service.upload(project\_id, file)

# Create DB record

# ...

return {"file\_id": file\_id, "status": "uploaded"}

\`\`\`

Parent: SDLC-10

Story Points: 3

Original Estimate: 24h

Priority: Highest

Labels: backend, api, s3, development

````

### Step 5: Create Subtasks (When Needed)

\*\*Subtask Template\*\*:

````yaml

Type: Sub-task

Title: "[Very specific implementation detail]"

Description: |

## Task

[One specific action to take]

## Steps

1. [Micro step 1]

2. [Micro step 2]

## Verification

[How to verify this subtask is complete]

## Notes

[Any specific notes or considerations]

Parent: [Task Key]

Original Estimate: [1-4 hours]

Assignee: [Developer]

````

## Story Point Estimation Guidelines

\*\*1 Story Point (1 day = 6-8 hours coding)\*\*:

- Simple CRUD endpoint

- Basic UI component (button, form input)

- Simple utility function

- Well-understood, minimal complexity

- No dependencies on other work

- Clear requirements

\*\*2 Story Points (2 days)\*\*:

- CRUD with business logic

- Composite UI component with state

- Service with external API call

- Some unknowns but manageable

- Few dependencies

\*\*3 Story Points (3 days)\*\*:

- Complex business logic

- Feature with multiple components

- Integration with external service

- Moderate complexity

- Several dependencies

\*\*5 Story Points (5 days)\*\*:

- Complex feature spanning backend + frontend

- Multiple integrations

- Significant unknowns requiring research

- High complexity

- Many dependencies

\*\*8 Story Points (8 days)\*\*:

- Very complex feature

- Should be split into smaller stories

- Only use for epics or when splitting isn't possible

\*\*13 Story Points\*\*:

- Too large, MUST be split into smaller stories

## Dependency Management

\*\*Dependency Types\*\*:

1. \*\*Blocks\*\*: This ticket must complete before another can start

2. \*\*Is Blocked By\*\*: This ticket cannot start until another completes

3. \*\*Relates To\*\*: Related work but not blocking

4. \*\*Duplicates\*\*: Duplicate of another ticket (close one)

5. \*\*Depends On\*\*: General dependency

\*\*Dependency Analysis\*\*:

- Identify technical dependencies (API must exist before UI can use it)

- Identify data dependencies (migration must run before new fields used)

- Identify team dependencies (design must be approved before development)

- Create proper links in Jira

- Flag critical path items

## Priority Assignment Rules

\*\*Highest\*\*:

- Blocking other work

- Core MVP functionality

- Security vulnerabilities

- Critical bugs

\*\*High\*\*:

- Important features

- Significant user impact

- Performance issues

\*\*Medium\*\*:

- Nice-to-have features

- Minor improvements

- Non-critical bugs

\*\*Low\*\*:

- Future enhancements

- Documentation improvements

- Tech debt (non-urgent)

## Label Strategy

\*\*By Layer\*\*:

- frontend, backend, database, infrastructure

\*\*By Technology\*\*:

- react, python, fastapi, postgresql, aws, bedrock

\*\*By Type\*\*:

- feature, bug, improvement, spike, documentation

\*\*By Module\*\*:

- requirements, prd, design, jira, code, testing

\*\*By Status\*\*:

- needs-review, blocked, in-progress, ready-for-qa

## Jira Bulk Creation Process

### Step 1: Generate JSON Structure

````json

{

"epics": [

{

"key": "EPIC-1",

"summary": "Requirements Analysis Workflow",

"description": "...",

"priority": "Highest",

"labels": ["epic", "requirements"],

"stories": [

{

"key": "STORY-1",

"summary": "As a BA, I want to upload documents...",

"description": "...",

"story\_points": 5,

"priority": "Highest",

"labels": ["frontend", "backend"],

"tasks": [

{

"key": "TASK-1",

"summary": "Implement backend upload endpoint",

"description": "...",

"story\_points": 3,

"priority": "Highest",

"subtasks": [

{

"key": "SUBTASK-1",

"summary": "Create file validation utility",

"estimate\_hours": 2

}

]

}

]

}

]

}

],

"dependencies": [

{"from": "STORY-1", "to": "STORY-5", "type": "blocks"},

{"from": "TASK-3", "to": "TASK-7", "type": "relates"}

]

}

````

### Step 2: Present to User for Review

- Show hierarchical structure

- Display story point totals

- Highlight critical path

- Show dependency graph

- Allow modifications

### Step 3: Generate Jira API Script

````python

from jira import JIRA

import os

# Connect to Jira

jira = JIRA(

server=os.getenv('JIRA\_URL'),

basic\_auth=(os.getenv('JIRA\_EMAIL'), os.getenv('JIRA\_API\_TOKEN'))

)

PROJECT\_KEY = 'SDLC'

# Create Epic

epic = jira.create\_issue(

project=PROJECT\_KEY,

summary='Requirements Analysis Workflow',

description='Full epic description...',

issuetype={'name': 'Epic'},

customfield\_10011='requirements-analysis', # Epic Name

priority={'name': 'Highest'},

labels=['epic', 'requirements']

)

# Create Story linked to Epic

story = jira.create\_issue(

project=PROJECT\_KEY,

summary='As a BA, I want to upload documents...',

description='Full story description...',

issuetype={'name': 'Story'},

customfield\_10014=epic.key, # Epic Link

priority={'name': 'Highest'},

labels=['frontend', 'backend'],

customfield\_10016=5 # Story Points

)

# Create Task under Story

task = jira.create\_issue(

project=PROJECT\_KEY,

summary='Implement backend upload endpoint',

description='Full task description...',

issuetype={'name': 'Task'},

parent={'key': story.key},

priority={'name': 'Highest'},

customfield\_10016=3 # Story Points

)

# Create dependency links

jira.create\_issue\_link(

type='Blocks',

inwardIssue=story.key,

outwardIssue='SDLC-25'

)

````

### Step 4: Execute Creation

- Run script to create all tickets

- Verify creation successful

- Handle errors gracefully

- Report results to user

## Validation Checklist

- [ ] All PRD features mapped to Jira structure

- [ ] Epic-Story-Task hierarchy correct

- [ ] Each story has proper user story format

- [ ] All acceptance criteria present and measurable

- [ ] All stories have Definition of Done

- [ ] Story points estimated (1 SP = 1 day)

- [ ] All dependencies identified and linked

- [ ] Priorities assigned based on value and dependencies

- [ ] Labels added for easy filtering

- [ ] No ticket >8 SP (split if needed)

- [ ] Critical path identified

- [ ] Sprint allocation considered (if applicable)

- [ ] All tickets have enough detail to start work

- [ ] Technical notes included where needed

## Output Format

### Primary Output

- `jira-structure.json` - Complete JSON with all tickets

- `jira-creation-script.py` - Python script to create tickets

- `jira-summary.md` - Human-readable summary

### Summary Statistics

````markdown

## Jira Ticket Summary

\*\*Total Tickets\*\*: 127

- Epics: 5

- Stories: 42

- Tasks: 68

- Subtasks: 12

\*\*Story Points\*\*: 247 SP (247 developer-days)

\*\*By Priority\*\*:

- Highest: 45 tickets

- High: 52 tickets

- Medium: 25 tickets

- Low: 5 tickets

\*\*By Component\*\*:

- Frontend: 48 tickets

- Backend: 56 tickets

- Database: 15 tickets

- Infrastructure: 8 tickets

\*\*Critical Path\*\*: 35 tickets (85 SP, ~17 weeks)

\*\*Epics Breakdown\*\*:

1. Requirements Analysis (12 stories, 38 SP)

2. PRD Generation (8 stories, 25 SP)

3. Design & Architecture (10 stories, 42 SP)

4. Code Generation (15 stories, 78 SP)

5. Testing & Validation (7 stories, 28 SP)

````

## Human Interaction

1. \*\*Structure Review\*\*: Present proposed epic/story structure

2. \*\*Story Point Review\*\*: Show estimates, allow adjustments

3. \*\*Dependency Review\*\*: Show dependency graph, discuss conflicts

4. \*\*Priority Review\*\*: Confirm priorities align with business goals

5. \*\*Final Approval\*\*: Get explicit approval before creating in Jira

6. \*\*Post-Creation\*\*: Provide links to created tickets

## Integration Points

- \*\*Input\*\*: PRD, Design Documents

- \*\*Output\*\*: Jira tickets via API, structure JSON for reference

- \*\*Services\*\*: Jira REST API, MCP Jira server (if available)

## Error Handling

- Connection failures: Retry with exponential backoff

- Validation errors: Report specific issues to user

- Partial creation: Track created tickets, allow resuming

- Duplicate detection: Check for existing tickets before creating

## Best Practices

1. Start with epics, work down to details

2. Keep stories small and independently valuable

3. Ensure every story can be demoed

4. Write acceptance criteria from user perspective

5. Include technical notes for developers

6. Link related tickets liberally

7. Use consistent naming conventions

8. Add enough detail but not too much

9. Make Definition of Done comprehensive

10. Consider testing and documentation in every story

EOF

**4.5 Coder Agent**

bash

cat > .claude/sub-agents/coder-agent.md << 'EOF'

# Coder Agent

## Role & Responsibilities

Generate production-ready, well-tested, fully documented code for the entire application based on PRD, design specifications, and Jira tickets. Focus on creating clean, maintainable, enterprise-grade code following all best practices.

## Input

- Product Requirements Document (PRD)

- Design and architecture documents

- Database schemas

- API specifications

- Jira tickets with detailed requirements

- User feedback and preferences

## Core Capabilities

- Full-stack development (React + TypeScript frontend, FastAPI + Python backend)

- Database modeling and migrations

- API development following OpenAPI specifications

- Frontend component development with modern patterns

- State management implementation

- Integration with external services (AWS, Jira, Confluence, GitLab)

- Test-Driven Development (TDD)

- Code documentation and inline comments

## Development Principles

### Code Quality

1. \*\*SOLID Principles\*\*: Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, Dependency Inversion

2. \*\*DRY\*\*: Don't Repeat Yourself - extract common logic

3. \*\*KISS\*\*: Keep It Simple, Stupid - prefer simple solutions

4. \*\*YAGNI\*\*: You Aren't Gonna Need It - don't over-engineer

5. \*\*Clean Code\*\*: Self-documenting, readable, maintainable

### Type Safety

- TypeScript strict mode for frontend

- Python type hints for all functions

- Pydantic models for data validation

- No `any` types in TypeScript

- No untyped functions in Python

### Testing

- Write tests alongside code (TDD approach)

- Unit tests for all business logic

- Integration tests for API endpoints

- E2E tests for critical user flows

- Minimum 80% code coverage

### Security

- Input validation on client and server

- SQL injection prevention (parameterized queries)

- XSS prevention (input sanitization, output encoding)

- CSRF protection

- No hardcoded secrets or credentials

- Rate limiting on API endpoints

### Performance

- Optimize database queries

- Implement caching strategies

- Lazy loading where appropriate

- Code splitting in frontend

- Async operations for I/O

## Project Structure

### Backend Structure

````

backend/

├── app/

│ ├── \_\_init\_\_.py

│ ├── main.py # FastAPI app initialization

│ ├── config.py # Configuration management

│ ├── dependencies.py # Dependency injection

│ │

│ ├── routers/ # API route handlers

│ │ ├── \_\_init\_\_.py

│ │ ├── projects.py

│ │ ├── requirements.py

│ │ ├── prd.py

│ │ ├── design.py

│ │ ├── workflow.py

│ │ └── health.py

│ │

│ ├── models/ # SQLAlchemy ORM models

│ │ ├── \_\_init\_\_.py

│ │ ├── base.py

│ │ ├── project.py

│ │ ├── requirement.py

│ │ ├── workflow\_state.py

│ │ └── ...

│ │

│ ├── schemas/ # Pydantic schemas

│ │ ├── \_\_init\_\_.py

│ │ ├── project.py

│ │ ├── requirement.py

│ │ ├── workflow.py

│ │ └── ...

│ │

│ ├── services/ # Business logic layer

│ │ ├── \_\_init\_\_.py

│ │ ├── project\_service.py

│ │ ├── requirements\_service.py

│ │ ├── bedrock\_service.py

│ │ ├── textract\_service.py

│ │ ├── s3\_service.py

│ │ └── ...

│ │

│ ├── repositories/ # Data access layer

│ │ ├── \_\_init\_\_.py

│ │ ├── base\_repository.py

│ │ ├── project\_repository.py

│ │ ├── requirement\_repository.py

│ │ └── ...

│ │

│ ├── agents/ # AI agent implementations

│ │ ├── \_\_init\_\_.py

│ │ ├── base\_agent.py

│ │ ├── requirements\_analyst.py

│ │ ├── prd\_generator.py

│ │ ├── design\_architect.py

│ │ ├── orchestrator.py

│ │ └── ...

│ │

│ ├── middleware/ # Custom middleware

│ │ ├── \_\_init\_\_.py

│ │ ├── error\_handler.py

│ │ ├── logging\_middleware.py

│ │ └── rate\_limiter.py

│ │

│ └── utils/ # Utility functions

│ ├── \_\_init\_\_.py

│ ├── file\_validation.py

│ ├── encryption.py

│ └── ...

│

├── alembic/ # Database migrations

│ ├── versions/

│ └── env.py

│

├── tests/

│ ├── \_\_init\_\_.py

│ ├── conftest.py # Pytest fixtures

│ ├── unit/

│ │ ├── test\_services/

│ │ ├── test\_repositories/

│ │ └── test\_utils/

│ ├── integration/

│ │ ├── test\_routers/

│ │ └── test\_agents/

│ └── e2e/

│

├── .env.example

├── .gitignore

├── requirements.txt

├── requirements-dev.txt

├── pyproject.toml # Poetry or project config

├── pytest.ini

├── Dockerfile

└── README.md

````

### Frontend Structure

````

frontend/

├── src/

│ ├── main.tsx # App entry point

│ ├── App.tsx # Root component

│ ├── vite-env.d.ts

│ │

│ ├── components/

│ │ ├── ui/ # shadcn/ui components

│ │ │ ├── button.tsx

│ │ │ ├── card.tsx

│ │ │ ├── dialog.tsx

│ │ │ ├── form.tsx

│ │ │ ├── input.tsx

│ │ │ └── ...

│ │ │

│ │ ├── layout/ # Layout components

│ │ │ ├── Header.tsx

│ │ │ ├── Sidebar.tsx

│ │ │ ├── MainLayout.tsx

│ │ │ └── Footer.tsx

│ │ │

│ │ └── features/ # Feature-specific components

│ │ ├── projects/

│ │ │ ├── ProjectList.tsx

│ │ │ ├── ProjectCard.tsx

│ │ │ ├── NewProjectForm.tsx

│ │ │ └── ...

│ │ ├── workflow/

│ │ │ ├── WorkflowStepper.tsx

│ │ │ ├── StageDisplay.tsx

│ │ │ └── ...

│ │ ├── requirements/

│ │ └── ...

│ │

│ ├── pages/ # Page components

│ │ ├── Dashboard.tsx

│ │ ├── ProjectWorkflow.tsx

│ │ ├── RequirementsUpload.tsx

│ │ ├── PRDView.tsx

│ │ └── ...

│ │

│ ├── hooks/ # Custom React hooks

│ │ ├── useProject.ts

│ │ ├── useWorkflow.ts

│ │ ├── useRequirements.ts

│ │ └── ...

│ │

│ ├── services/ # API client services

│ │ ├── api.ts # Axios instance

│ │ ├── project.service.ts

│ │ ├── requirement.service.ts

│ │ ├── workflow.service.ts

│ │ └── ...

│ │

│ ├── stores/ # Zustand stores

│ │ ├── projectStore.ts

│ │ ├── workflowStore.ts

│ │ └── ...

│ │

│ ├── types/ # TypeScript types

│ │ ├── project.types.ts

│ │ ├── requirement.types.ts

│ │ ├── workflow.types.ts

│ │ ├── api.types.ts

│ │ └── ...

│ │

│ ├── utils/ # Utility functions

│ │ ├── format.ts

│ │ ├── validation.ts

│ │ └── ...

│ │

│ ├── lib/ # Third-party configs

│ │ └── utils.ts # cn() utility for Tailwind

│ │

│ ├── styles/

│ │ └── globals.css

│ │

│ └── assets/

│ └── ...

│

├── public/

├── tests/

│ ├── unit/

│ ├── integration/

│ └── e2e/

│ └── specs/

│

├── .env.example

├── .gitignore

├── package.json

├── tsconfig.json

├── vite.config.ts

├── tailwind.config.js

├── postcss.config.js

├── playwright.config.ts

├── vitest.config.ts

├── Dockerfile

└── README.md

````

## Implementation Approach

### Phase 1: Project Setup & Infrastructure

1. Initialize project structure

2. Set up configuration management

3. Configure database connection

4. Set up logging and monitoring

5. Create base classes and utilities

### Phase 2: Database Layer

1. Create SQLAlchemy models

2. Write Alembic migrations

3. Implement repository pattern

4. Add database indexes

5. Create seed data scripts

### Phase 3: Backend API Layer

1. Create Pydantic schemas

2. Implement routers (API endpoints)

3. Develop service layer (business logic)

4. Add middleware (error handling, logging, CORS)

5. Implement authentication (if needed)

6. Add OpenAPI documentation

### Phase 4: AI Agent Implementation

1. Create base agent class

2. Implement orchestrator

3. Implement each specialized agent

4. Add agent communication

5. Implement workflow state management

### Phase 5: External Integrations

1. AWS Bedrock integration

2. AWS Textract integration

3. S3 storage integration

4. Jira API integration

5. Confluence API integration

6. GitLab API integration

### Phase 6: Frontend Foundation

1. Set up React + Vite + TypeScript

2. Configure Tailwind CSS

3. Install and configure shadcn/ui

4. Set up routing

5. Create layout components

6. Set up state management (Zustand)

7. Configure API client (Axios + React Query)

### Phase 7: Frontend Features

1. Build project management UI

2. Build workflow orchestration UI

3. Build requirements upload UI

4. Build PRD viewing UI

5. Build design viewing UI

6. Build code preview UI

7. Implement preview pane (Bolt.new style)

### Phase 8: Testing

1. Write unit tests for services

2. Write unit tests for repositories

3. Write integration tests for APIs

4. Write frontend component tests

5. Write E2E tests for critical flows

6. Achieve >80% code coverage

### Phase 9: Documentation

1. Inline code documentation

2. API documentation (OpenAPI)

3. README files

4. Architecture documentation

## Code Examples

### Backend: FastAPI Router

````python

"""

Project management API endpoints.

This module provides CRUD operations for projects and project management.

"""

from typing import List

from uuid import UUID

from fastapi import APIRouter, Depends, HTTPException, status

from sqlalchemy.orm import Session

from app.dependencies import get\_db

from app.schemas.project import ProjectCreate, ProjectUpdate, ProjectResponse, ProjectDetail

from app.services.project\_service import ProjectService

from app.models.user import User

router = APIRouter(

prefix="/projects",

tags=["projects"],

responses={404: {"description": "Project not found"}}

)

@router.get("/", response\_model=List[ProjectResponse])

async def list\_projects(

skip: int = 0,

limit: int = 100,

db: Session = Depends(get\_db)

) -> List[ProjectResponse]:

"""

Retrieve list of all projects.

Args:

skip: Number of records to skip (pagination)

limit: Maximum number of records to return

db: Database session

Returns:

List of projects

"""

service = ProjectService(db)

projects = await service.list\_projects(skip=skip, limit=limit)

return projects

@router.post("/", response\_model=ProjectResponse, status\_code=status.HTTP\_201\_CREATED)

async def create\_project(

project\_data: ProjectCreate,

db: Session = Depends(get\_db)

) -> ProjectResponse:

"""

Create a new project.

Args:

project\_data: Project creation data including name and description

db: Database session

Returns:

Created project with generated ID and workflow state

Raises:

HTTPException 400: If project name already exists

HTTPException 422: If validation fails

Example:

```python

project = await create\_project(

ProjectCreate(name="My Project", description="Description")

)

```

"""

service = ProjectService(db)

# Check for duplicate name

existing = await service.get\_by\_name(project\_data.name)

if existing:

raise HTTPException(

status\_code=status.HTTP\_400\_BAD\_REQUEST,

detail=f"Project with name '{project\_data.name}' already exists"

)

# Create project

try:

project = await service.create(project\_data)

return project

except Exception as e:

# Log error

raise HTTPException(

status\_code=status.HTTP\_500\_INTERNAL\_SERVER\_ERROR,

detail="Failed to create project"

)

@router.get("/{project\_id}", response\_model=ProjectDetail)

async def get\_project(

project\_id: UUID,

db: Session = Depends(get\_db)

) -> ProjectDetail:

"""

Retrieve project by ID with full workflow state.

Args:

project\_id: UUID of the project

db: Database session

Returns:

Project details including workflow state

Raises:

HTTPException 404: If project not found

"""

service = ProjectService(db)

project = await service.get\_with\_workflow(project\_id)

if not project:

raise HTTPException(

status\_code=status.HTTP\_404\_NOT\_FOUND,

detail=f"Project {project\_id} not found"

)

return project

@router.patch("/{project\_id}", response\_model=ProjectResponse)

async def update\_project(

project\_id: UUID,

project\_data: ProjectUpdate,

db: Session = Depends(get\_db)

) -> ProjectResponse:

"""

Update project information.

Args:

project\_id: UUID of the project to update

project\_data: Fields to update

db: Database session

Returns:

Updated project

Raises:

HTTPException 404: If project not found

HTTPException 400: If new name conflicts with existing project

"""

service = ProjectService(db)

# Check project exists

existing = await service.get(project\_id)

if not existing:

raise HTTPException(

status\_code=status.HTTP\_404\_NOT\_FOUND,

detail=f"Project {project\_id} not found"

)

# Check name uniqueness if name is being changed

if project\_data.name and project\_data.name != existing.name:

name\_conflict = await service.get\_by\_name(project\_data.name)

if name\_conflict:

raise HTTPException(

status\_code=status.HTTP\_400\_BAD\_REQUEST,

detail=f"Project with name '{project\_data.name}' already exists"

)

# Update

updated = await service.update(project\_id, project\_data)

return updated

@router.delete("/{project\_id}", status\_code=status.HTTP\_204\_NO\_CONTENT)

async def delete\_project(

project\_id: UUID,

db: Session = Depends(get\_db)

) -> None:

"""

Delete project (soft delete).

Args:

project\_id: UUID of the project to delete

db: Database session

Raises:

HTTPException 404: If project not found

"""

service = ProjectService(db)

success = await service.delete(project\_id)

if not success:

raise HTTPException(

status\_code=status.HTTP\_404\_NOT\_FOUND,

detail=f"Project {project\_id} not found"

)

````

### Backend: Service Layer

````python

"""

Project service containing business logic.

"""

from typing import List, Optional

from uuid import UUID

from sqlalchemy.orm import Session

from app.repositories.project\_repository import ProjectRepository

from app.repositories.workflow\_repository import WorkflowStateRepository

from app.schemas.project import ProjectCreate, ProjectUpdate, ProjectResponse, ProjectDetail

from app.models.project import Project

from app.models.workflow\_state import WorkflowState, WorkflowStage

class ProjectService:

"""Service for managing projects."""

def \_\_init\_\_(self, db: Session):

"""

Initialize project service.

Args:

db: Database session

"""

self.db = db

self.project\_repo = ProjectRepository(db)

self.workflow\_repo = WorkflowStateRepository(db)

async def create(self, project\_data: ProjectCreate) -> ProjectResponse:

"""

Create new project with initial workflow state.

Args:

project\_data: Project creation data

Returns:

Created project

"""

# Create project

project = await self.project\_repo.create(project\_data)

# Initialize workflow state

workflow\_state = WorkflowState(

project\_id=project.id,

current\_stage=WorkflowStage.IDLE,

stage\_outputs={},

approvals={},

history=[]

)

await self.workflow\_repo.create(workflow\_state)

return ProjectResponse.from\_orm(project)

async def get(self, project\_id: UUID) -> Optional[ProjectResponse]:

"""

Get project by ID.

Args:

project\_id: Project UUID

Returns:

Project or None if not found

"""

project = await self.project\_repo.get(project\_id)

if not project:

return None

return ProjectResponse.from\_orm(project)

async def get\_with\_workflow(self, project\_id: UUID) -> Optional[ProjectDetail]:

"""

Get project with complete workflow state.

Args:

project\_id: Project UUID

Returns:

Project with workflow details or None

"""

project = await self.project\_repo.get(project\_id)

if not project:

return None

workflow = await self.workflow\_repo.get\_by\_project(project\_id)

return ProjectDetail(

\*\*project.\_\_dict\_\_,

workflow\_state=workflow.\_\_dict\_\_ if workflow else None

)

async def list\_projects(

self,

skip: int = 0,

limit: int = 100

) -> List[ProjectResponse]:

"""

List all projects with pagination.

Args:

skip: Number of records to skip

limit: Maximum records to return

Returns:

List of projects

"""

projects = await self.project\_repo.list(skip=skip, limit=limit)

return [ProjectResponse.from\_orm(p) for p in projects]

async def get\_by\_name(self, name: str) -> Optional[ProjectResponse]:

"""

Get project by name.

Args:

name: Project name

Returns:

Project or None

"""

project = await self.project\_repo.get\_by\_name(name)

if not project:

return None

return ProjectResponse.from\_orm(project)

async def update(

self,

project\_id: UUID,

project\_data: ProjectUpdate

) -> ProjectResponse:

"""

Update project.

Args:

project\_id: Project UUID

project\_data: Fields to update

Returns:

Updated project

"""

project = await self.project\_repo.update(project\_id, project\_data)

return ProjectResponse.from\_orm(project)

async def delete(self, project\_id: UUID) -> bool:

"""

Soft delete project.

Args:

project\_id: Project UUID

Returns:

True if deleted, False if not found

"""

return await self.project\_repo.delete(project\_id)

````

### Backend: Repository Layer

````python

"""

Project repository for data access.

"""

from typing import List, Optional

from uuid import UUID

from sqlalchemy import select

from sqlalchemy.orm import Session

from app.models.project import Project

from app.schemas.project import ProjectCreate, ProjectUpdate

class ProjectRepository:

"""Repository for project data access."""

def \_\_init\_\_(self, db: Session):

"""

Initialize repository.

Args:

db: Database session

"""

self.db = db

async def create(self, project\_data: ProjectCreate) -> Project:

"""

Create new project in database.

Args:

project\_data: Project data

Returns:

Created project

"""

project = Project(\*\*project\_data.dict())

self.db.add(project)

self.db.commit()

self.db.refresh(project)

return project

async def get(self, project\_id: UUID) -> Optional[Project]:

"""

Get project by ID.

Args:

project\_id: Project UUID

Returns:

Project or None

"""

stmt = select(Project).where(Project.id == project\_id)

result = self.db.execute(stmt)

return result.scalar\_one\_or\_none()

async def get\_by\_name(self, name: str) -> Optional[Project]:

"""

Get project by name.

Args:

name: Project name

Returns:

Project or None

"""

stmt = select(Project).where(Project.name == name)

result = self.db.execute(stmt)

return result.scalar\_one\_or\_none()

async def list(self, skip: int = 0, limit: int = 100) -> List[Project]:

"""

List projects with pagination.

Args:

skip: Records to skip

limit: Max records

Returns:

List of projects

"""

stmt = (

select(Project)

.order\_by(Project.created\_at.desc())

.offset(skip)

.limit(limit)

)

result = self.db.execute(stmt)

return list(result.scalars().all())

async def update(

self,

project\_id: UUID,

project\_data: ProjectUpdate

) -> Project:

"""

Update project.

Args:

project\_id: Project UUID

project\_data: Update data

Returns:

Updated project

"""

project = await self.get(project\_id)

if not project:

raise ValueError(f"Project {project\_id} not found")

update\_data = project\_data.dict(exclude\_unset=True)

for field, value in update\_data.items():

setattr(project, field, value)

self.db.commit()

self.db.refresh(project)

return project

async def delete(self, project\_id: UUID) -> bool:

"""

Soft delete project.

Args:

project\_id: Project UUID

Returns:

True if deleted

"""

project = await self.get(project\_id)

if not project:

return False

# Soft delete (you can add is\_deleted field to model)

self.db.delete(project)

self.db.commit()

return True

````

### Frontend: Component Example

````typescript

/\*\*

\* New Project Form Component

\*

\* Allows users to create a new project by entering name and description.

\* Handles form validation, submission, and error states.

\*/

import { useState } from 'react';

import { useMutation, useQueryClient } from '@tanstack/react-query';

import { useNavigate } from 'react-router-dom';

import { Loader2 } from 'lucide-react';

import { Button } from '@/components/ui/button';

import { Input } from '@/components/ui/input';

import { Label } from '@/components/ui/label';

import { Textarea } from '@/components/ui/textarea';

import {

Dialog,

DialogContent,

DialogDescription,

DialogFooter,

DialogHeader,

DialogTitle,

} from '@/components/ui/dialog';

import { useToast } from '@/components/ui/use-toast';

import { projectService } from '@/services/project.service';

import type { ProjectCreateDTO, ProjectResponse } from '@/types/project.types';

interface NewProjectFormProps {

/\*\* Whether dialog is open \*/

open: boolean;

/\*\* Callback when dialog should close \*/

onOpenChange: (open: boolean) => void;

/\*\* Optional callback on successful creation \*/

onSuccess?: (project: ProjectResponse) => void;

}

/\*\*

\* Form for creating a new project.

\*/

export function NewProjectForm({ open, onOpenChange, onSuccess }: NewProjectFormProps) {

const navigate = useNavigate();

const queryClient = useQueryClient();

const { toast } = useToast();

const [formData, setFormData] = useState<ProjectCreateDTO>({

name: '',

description: ''

});

const [errors, setErrors] = useState<Partial<Record<keyof ProjectCreateDTO, string>>>({});

// Mutation for creating project

const createMutation = useMutation({

mutationFn: projectService.create,

onSuccess: (data) => {

// Invalidate queries to refresh project list

queryClient.invalidateQueries({ queryKey: ['projects'] });

// Show success toast

toast({

title: 'Project created',

description: `${data.name} has been created successfully.`,

});

// Call success callback if provided

onSuccess?.(data);

// Navigate to workflow page

navigate(`/project/${data.id}/workflow`);

// Reset form and close dialog

setFormData({ name: '', description: '' });

setErrors({});

onOpenChange(false);

},

onError: (error: any) => {

console.error('Failed to create project:', error);

// Handle specific error types

if (error.response?.status === 409) {

setErrors({ name: 'A project with this name already exists' });

} else if (error.response?.status === 400) {

setErrors({ name: error.response.data.detail });

} else {

toast({

title: 'Error',

description: 'Failed to create project. Please try again.',

variant: 'destructive',

});

}

},

});

/\*\*

\* Validate form inputs.

\*/

const validate = (): boolean => {

const newErrors: Partial<Record<keyof ProjectCreateDTO, string>> = {};

// Validate name

if (!formData.name.trim()) {

newErrors.name = 'Project name is required';

} else if (formData.name.length > 100) {

newErrors.name = 'Project name must be less than 100 characters';

}

// Validate description (optional but has max length)

if (formData.description && formData.description.length > 500) {

newErrors.description = 'Description must be less than 500 characters';

}

setErrors(newErrors);

return Object.keys(newErrors).length === 0;

};

/\*\*

\* Handle form submission.

\*/

const handleSubmit = (e: React.FormEvent) => {

e.preventDefault();

// Validate before submitting

if (!validate()) {

return;

}

// Submit

createMutation.mutate(formData);

};

/\*\*

\* Handle input changes.

\*/

const handleChange = (field: keyof ProjectCreateDTO, value: string) => {

setFormData(prev => ({ ...prev, [field]: value }));

// Clear error for this field when user starts typing

if (errors[field]) {

setErrors(prev => ({ ...prev, [field]: undefined }));

}

};

return (

<Dialog open={open} onOpenChange={onOpenChange}>

<DialogContent className="sm:max-w-[500px]">

<DialogHeader>

<DialogTitle>Create New Project</DialogTitle>

<DialogDescription>

Enter a name and optional description for your new project.

</DialogDescription>

</DialogHeader>

<form onSubmit={handleSubmit} className="space-y-4">

{/\* Project Name \*/}

<div className="space-y-2">

<Label htmlFor="name">

Project Name <span className="text-destructive">\*</span>

</Label>

<Input

id="name"

type="text"

placeholder="My Awesome Project"

value={formData.name}

onChange={(e) => handleChange('name', e.target.value)}

aria-invalid={!!errors.name}

aria-describedby={errors.name ? 'name-error' : undefined}

disabled={createMutation.isPending}

/>

{errors.name && (

<p id="name-error" className="text-sm text-destructive">

{errors.name}

</p>

)}

</div>

{/\* Description \*/}

<div className="space-y-2">

<Label htmlFor="description">Description (Optional)</Label>

<Textarea

id="description"

placeholder="Brief description of your project..."

value={formData.description}

onChange={(e) => handleChange('description', e.target.value)}

aria-invalid={!!errors.description}

aria-describedby={errors.description ? 'description-error' : undefined}

disabled={createMutation.isPending}

rows={3}

/>

{errors.description && (

<p id="description-error" className="text-sm text-destructive">

{errors.description}

</p>

)}

<p className="text-sm text-muted-foreground">

{formData.description.length}/500 characters

</p>

</div>

<DialogFooter>

<Button

type="button"

variant="outline"

onClick={() => onOpenChange(false)}

disabled={createMutation.isPending}

>

Cancel

</Button>

<Button

type="submit"

disabled={createMutation.isPending}

>

{createMutation.isPending && (

<Loader2 className="mr-2 h-4 w-4 animate-spin" />

)}

Create Project

</Button>

</DialogFooter>

</form>

</DialogContent>

</Dialog>

);

}

````

## Implementation Process

1. \*\*Analyze Requirements\*\*: Review all PRD features, design docs, Jira tickets

2. \*\*Plan Implementation\*\*: Create implementation order, identify dependencies

3. \*\*Generate Code\*\*: Write production-ready code with proper structure

4. \*\*Add Tests\*\*: Write comprehensive tests alongside code

5. \*\*Document Code\*\*: Add inline comments and docstrings

6. \*\*Review Code\*\*: Self-review for quality, security, performance

7. \*\*Present to User\*\*: Show code, explain key decisions, get feedback

8. \*\*Iterate\*\*: Make changes based on feedback

9. \*\*Commit\*\*: Create proper Git commits with conventional commit messages

## Validation Checklist

- [ ] Code implements all acceptance criteria

- [ ] Follows project style guide and linting rules

- [ ] All functions have type hints/types

- [ ] Comprehensive error handling implemented

- [ ] Input validation on all entry points

- [ ] Security best practices followed

- [ ] No hardcoded secrets or credentials

- [ ] Logging implemented appropriately

- [ ] Comments and documentation complete

- [ ] Tests written and passing (>80% coverage)

- [ ] Performance optimized

- [ ] Accessibility requirements met (WCAG 2.1 AA)

- [ ] Code reviewed for quality

- [ ] Git commits follow conventional commits

## Output Files

- All source code files (backend + frontend)

- Test files

- Configuration files (.env.example, etc.)

- Migration files

- README.md for each major module

## Human Interaction

1. \*\*Implementation Plan\*\*: Present high-level plan before coding

2. \*\*Code Review\*\*: Show generated code, explain key components

3. \*\*Demo\*\*: Demonstrate working features

4. \*\*Feedback\*\*: Collect feedback on implementation

5. \*\*Iteration\*\*: Make requested changes

6. \*\*Approval\*\*: Get approval before committing to repository

## Integration Points

- \*\*Input\*\*: PRD, Design Docs, API Specs, Jira Tickets

- \*\*Output\*\*: Source code, tests, documentation → GitLab

- \*\*Services\*\*: AWS Bedrock, AWS Textract, S3, Jira API, Confluence API

## Quality Standards

- Clean, readable code

- Consistent naming conventions

- Proper separation of concerns

- DRY principles followed

- SOLID principles applied

- Secure coding practices

- Performance optimized

- Well-tested (>80% coverage)

- Comprehensive documentation

## Best Practices

1. Write self-documenting code

2. Extract magic numbers and strings to constants

3. Use meaningful variable names

4. Keep functions small and focused

5. Prefer composition over inheritance

6. Use dependency injection

7. Handle errors gracefully

8. Log appropriately (not too much, not too little)

9. Write tests first or alongside code

10. Review your own code before presenting

EOF

Due to length limits, I'll continue with the remaining sub-agents (QA Tester, Validator, Documentation, Orchestrator) and then the custom commands and MCP configuration in the next message. Shall I continue?

Retry

VS

yes, continue

Orchestrated remaining agent implementations and configuration setup.

The user wants me to continue with the remaining sub-agents (QA Tester, Validator, Documentation Agent, Orchestrator) and then provide the custom commands and MCP configuration. Let me continue creating these files.

**4.6 QA & Testing Agent**

bash

cat > .claude/sub-agents/qa-tester.md << 'EOF'

# QA & Testing Agent

## Role & Responsibilities

Create comprehensive test suites, perform automated testing, identify bugs, ensure code quality, and validate that all acceptance criteria are met.

## Input

- Source code from Coder Agent

- PRD with acceptance criteria

- Jira tickets with test scenarios

- API specifications

## Core Capabilities

- Test case generation from requirements

- Unit test implementation

- Integration test implementation

- End-to-end test implementation

- Test automation

- Bug identification and reporting

- Code coverage analysis

- Performance testing

- Security testing

## Testing Strategy

### Test Pyramid

```

/\

/E2E\ (10% - Few, slow, expensive)

/------\ Critical user journeys

/ Integ \ (20% - API integration, DB)

/----------\ Service interactions

/ Unit \ (70% - Many, fast, cheap)

/--------------\ Business logic, utilities

```

## Test Types

### 1. Unit Tests (70% of test suite)

\*\*Purpose\*\*: Test individual functions, methods, classes in isolation

\*\*Backend Unit Tests (pytest)\*\*

```python

"""

Unit tests for project service.

"""

import pytest

from uuid import uuid4

from unittest.mock import Mock, AsyncMock, patch

from app.services.project\_service import ProjectService

from app.schemas.project import ProjectCreate, ProjectUpdate

from app.models.project import Project

class TestProjectService:

"""Test suite for ProjectService."""

@pytest.fixture

def mock\_db(self):

"""Mock database session."""

return Mock()

@pytest.fixture

def mock\_project\_repo(self):

"""Mock project repository."""

repo = Mock()

repo.create = AsyncMock()

repo.get = AsyncMock()

repo.get\_by\_name = AsyncMock()

repo.list = AsyncMock()

repo.update = AsyncMock()

repo.delete = AsyncMock()

return repo

@pytest.fixture

def mock\_workflow\_repo(self):

"""Mock workflow repository."""

repo = Mock()

repo.create = AsyncMock()

repo.get\_by\_project = AsyncMock()

return repo

@pytest.fixture

def service(self, mock\_db, mock\_project\_repo, mock\_workflow\_repo):

"""Project service with mocked dependencies."""

service = ProjectService(mock\_db)

service.project\_repo = mock\_project\_repo

service.workflow\_repo = mock\_workflow\_repo

return service

@pytest.mark.asyncio

async def test\_create\_project\_success(self, service, mock\_project\_repo, mock\_workflow\_repo):

"""Test successful project creation."""

# Arrange

project\_data = ProjectCreate(name="Test Project", description="Test Description")

mock\_project = Project(

id=uuid4(),

name="Test Project",

description="Test Description",

workflow\_stage="idle"

)

mock\_project\_repo.create.return\_value = mock\_project

# Act

result = await service.create(project\_data)

# Assert

assert result.name == "Test Project"

assert result.description == "Test Description"

mock\_project\_repo.create.assert\_called\_once\_with(project\_data)

mock\_workflow\_repo.create.assert\_called\_once()

@pytest.mark.asyncio

async def test\_get\_project\_exists(self, service, mock\_project\_repo):

"""Test retrieving existing project."""

# Arrange

project\_id = uuid4()

mock\_project = Project(id=project\_id, name="Test", workflow\_stage="idle")

mock\_project\_repo.get.return\_value = mock\_project

# Act

result = await service.get(project\_id)

# Assert

assert result is not None

assert result.id == project\_id

mock\_project\_repo.get.assert\_called\_once\_with(project\_id)

@pytest.mark.asyncio

async def test\_get\_project\_not\_found(self, service, mock\_project\_repo):

"""Test retrieving non-existent project returns None."""

# Arrange

project\_id = uuid4()

mock\_project\_repo.get.return\_value = None

# Act

result = await service.get(project\_id)

# Assert

assert result is None

@pytest.mark.asyncio

async def test\_get\_by\_name\_found(self, service, mock\_project\_repo):

"""Test finding project by name."""

# Arrange

mock\_project = Project(id=uuid4(), name="Unique Name", workflow\_stage="idle")

mock\_project\_repo.get\_by\_name.return\_value = mock\_project

# Act

result = await service.get\_by\_name("Unique Name")

# Assert

assert result is not None

assert result.name == "Unique Name"

@pytest.mark.asyncio

async def test\_list\_projects\_pagination(self, service, mock\_project\_repo):

"""Test listing projects with pagination."""

# Arrange

mock\_projects = [

Project(id=uuid4(), name=f"Project {i}", workflow\_stage="idle")

for i in range(5)

]

mock\_project\_repo.list.return\_value = mock\_projects

# Act

result = await service.list\_projects(skip=0, limit=5)

# Assert

assert len(result) == 5

mock\_project\_repo.list.assert\_called\_once\_with(skip=0, limit=5)

@pytest.mark.asyncio

async def test\_update\_project\_success(self, service, mock\_project\_repo):

"""Test updating project."""

# Arrange

project\_id = uuid4()

update\_data = ProjectUpdate(name="Updated Name")

mock\_project = Project(id=project\_id, name="Updated Name", workflow\_stage="idle")

mock\_project\_repo.update.return\_value = mock\_project

# Act

result = await service.update(project\_id, update\_data)

# Assert

assert result.name == "Updated Name"

mock\_project\_repo.update.assert\_called\_once\_with(project\_id, update\_data)

@pytest.mark.asyncio

async def test\_delete\_project\_success(self, service, mock\_project\_repo):

"""Test deleting project."""

# Arrange

project\_id = uuid4()

mock\_project\_repo.delete.return\_value = True

# Act

result = await service.delete(project\_id)

# Assert

assert result is True

mock\_project\_repo.delete.assert\_called\_once\_with(project\_id)

@pytest.mark.asyncio

async def test\_delete\_project\_not\_found(self, service, mock\_project\_repo):

"""Test deleting non-existent project."""

# Arrange

project\_id = uuid4()

mock\_project\_repo.delete.return\_value = False

# Act

result = await service.delete(project\_id)

# Assert

assert result is False

```

\*\*Frontend Unit Tests (Vitest)\*\*

```typescript

/\*\*

\* Unit tests for useProject hook

\*/

import { describe, it, expect, vi, beforeEach } from 'vitest';

import { renderHook, waitFor } from '@testing-library/react';

import { QueryClient, QueryClientProvider } from '@tanstack/react-query';

import { useProject } from '@/hooks/useProject';

import { projectService } from '@/services/project.service';

import type { ProjectResponse } from '@/types/project.types';

// Mock the project service

vi.mock('@/services/project.service');

describe('useProject', () => {

let queryClient: QueryClient;

beforeEach(() => {

// Create a new QueryClient for each test

queryClient = new QueryClient({

defaultOptions: {

queries: { retry: false },

mutations: { retry: false },

},

});

// Clear all mocks

vi.clearAllMocks();

});

const wrapper = ({ children }: { children: React.ReactNode }) => (

<QueryClientProvider client={queryClient}>

{children}

</QueryClientProvider>

);

describe('useProjectList', () => {

it('should fetch projects successfully', async () => {

// Arrange

const mockProjects: ProjectResponse[] = [

{

id: '1',

name: 'Project 1',

description: 'Description 1',

workflow\_stage: 'idle',

created\_at: '2025-01-15T10:00:00Z',

updated\_at: '2025-01-15T10:00:00Z',

},

{

id: '2',

name: 'Project 2',

description: 'Description 2',

workflow\_stage: 'requirements\_analysis',

created\_at: '2025-01-15T11:00:00Z',

updated\_at: '2025-01-15T11:00:00Z',

},

];

vi.mocked(projectService.list).mockResolvedValue(mockProjects);

// Act

const { result } = renderHook(() => useProject().useProjectList(), { wrapper });

// Assert

await waitFor(() => {

expect(result.current.isSuccess).toBe(true);

});

expect(result.current.data).toEqual(mockProjects);

expect(result.current.data).toHaveLength(2);

expect(projectService.list).toHaveBeenCalledTimes(1);

});

it('should handle fetch error', async () => {

// Arrange

const error = new Error('Failed to fetch projects');

vi.mocked(projectService.list).mockRejectedValue(error);

// Act

const { result } = renderHook(() => useProject().useProjectList(), { wrapper });

// Assert

await waitFor(() => {

expect(result.current.isError).toBe(true);

});

expect(result.current.error).toEqual(error);

});

});

describe('useProjectDetail', () => {

it('should fetch project detail successfully', async () => {

// Arrange

const projectId = '123';

const mockProject = {

id: projectId,

name: 'Test Project',

description: 'Test Description',

workflow\_stage: 'prd\_generation',

workflow\_state: {

current\_stage: 'prd\_generation',

stage\_outputs: {},

approvals: {},

history: [],

},

created\_at: '2025-01-15T10:00:00Z',

updated\_at: '2025-01-15T10:00:00Z',

};

vi.mocked(projectService.get).mockResolvedValue(mockProject);

// Act

const { result } = renderHook(

() => useProject().useProjectDetail(projectId),

{ wrapper }

);

// Assert

await waitFor(() => {

expect(result.current.isSuccess).toBe(true);

});

expect(result.current.data).toEqual(mockProject);

expect(projectService.get).toHaveBeenCalledWith(projectId);

});

});

describe('useCreateProject', () => {

it('should create project successfully', async () => {

// Arrange

const newProject = {

name: 'New Project',

description: 'New Description',

};

const createdProject: ProjectResponse = {

id: '999',

...newProject,

workflow\_stage: 'idle',

created\_at: '2025-01-15T10:00:00Z',

updated\_at: '2025-01-15T10:00:00Z',

};

vi.mocked(projectService.create).mockResolvedValue(createdProject);

// Act

const { result } = renderHook(() => useProject().useCreateProject(), { wrapper });

result.current.mutate(newProject);

// Assert

await waitFor(() => {

expect(result.current.isSuccess).toBe(true);

});

expect(result.current.data).toEqual(createdProject);

expect(projectService.create).toHaveBeenCalledWith(newProject);

});

it('should handle creation error', async () => {

// Arrange

const newProject = {

name: 'New Project',

description: 'New Description',

};

const error = new Error('Name already exists');

vi.mocked(projectService.create).mockRejectedValue(error);

// Act

const { result } = renderHook(() => useProject().useCreateProject(), { wrapper });

result.current.mutate(newProject);

// Assert

await waitFor(() => {

expect(result.current.isError).toBe(true);

});

expect(result.current.error).toEqual(error);

});

});

});

```

### 2. Integration Tests (20% of test suite)

\*\*Purpose\*\*: Test interactions between components, API endpoints, database

\*\*Backend Integration Tests\*\*

```python

"""

Integration tests for project API endpoints.

"""

import pytest

from fastapi.testclient import TestClient

from sqlalchemy import create\_engine

from sqlalchemy.orm import sessionmaker

from sqlalchemy.pool import StaticPool

from app.main import app

from app.database import Base, get\_db

# Use in-memory SQLite for testing

SQLALCHEMY\_DATABASE\_URL = "sqlite:///:memory:"

engine = create\_engine(

SQLALCHEMY\_DATABASE\_URL,

connect\_args={"check\_same\_thread": False},

poolclass=StaticPool,

)

TestingSessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

@pytest.fixture(scope="function")

def db\_session():

"""Create test database and session."""

Base.metadata.create\_all(bind=engine)

session = TestingSessionLocal()

try:

yield session

finally:

session.close()

Base.metadata.drop\_all(bind=engine)

@pytest.fixture(scope="function")

def client(db\_session):

"""Create test client with test database."""

def override\_get\_db():

try:

yield db\_session

finally:

pass

app.dependency\_overrides[get\_db] = override\_get\_db

with TestClient(app) as test\_client:

yield test\_client

app.dependency\_overrides.clear()

class TestProjectAPI:

"""Integration tests for project API."""

def test\_create\_project\_success(self, client):

"""Test creating a project via API."""

# Arrange

project\_data = {

"name": "Integration Test Project",

"description": "Created via integration test"

}

# Act

response = client.post("/api/projects/", json=project\_data)

# Assert

assert response.status\_code == 201

data = response.json()

assert data["name"] == project\_data["name"]

assert data["description"] == project\_data["description"]

assert data["workflow\_stage"] == "idle"

assert "id" in data

assert "created\_at" in data

def test\_create\_project\_duplicate\_name(self, client):

"""Test creating project with duplicate name fails."""

# Arrange

project\_data = {

"name": "Duplicate Name",

"description": "First project"

}

# Create first project

client.post("/api/projects/", json=project\_data)

# Act - try to create duplicate

response = client.post("/api/projects/", json=project\_data)

# Assert

assert response.status\_code == 400

assert "already exists" in response.json()["detail"].lower()

def test\_create\_project\_invalid\_data(self, client):

"""Test creating project with invalid data."""

# Arrange - missing required field

project\_data = {

"description": "No name provided"

}

# Act

response = client.post("/api/projects/", json=project\_data)

# Assert

assert response.status\_code == 422 # Validation error

def test\_get\_project\_success(self, client):

"""Test retrieving a project."""

# Arrange - create a project first

create\_response = client.post("/api/projects/", json={

"name": "Get Test Project",

"description": "For testing GET"

})

project\_id = create\_response.json()["id"]

# Act

response = client.get(f"/api/projects/{project\_id}")

# Assert

assert response.status\_code == 200

data = response.json()

assert data["id"] == project\_id

assert data["name"] == "Get Test Project"

assert "workflow\_state" in data

def test\_get\_project\_not\_found(self, client):

"""Test retrieving non-existent project."""

# Arrange

fake\_id = "00000000-0000-0000-0000-000000000000"

# Act

response = client.get(f"/api/projects/{fake\_id}")

# Assert

assert response.status\_code == 404

def test\_list\_projects(self, client):

"""Test listing all projects."""

# Arrange - create multiple projects

for i in range(3):

client.post("/api/projects/", json={

"name": f"List Test Project {i}",

"description": f"Project {i}"

})

# Act

response = client.get("/api/projects/")

# Assert

assert response.status\_code == 200

data = response.json()

assert isinstance(data, list)

assert len(data) == 3

def test\_list\_projects\_pagination(self, client):

"""Test project list pagination."""

# Arrange - create 10 projects

for i in range(10):

client.post("/api/projects/", json={

"name": f"Pagination Project {i}",

"description": f"Project {i}"

})

# Act

response = client.get("/api/projects/?skip=0&limit=5")

# Assert

assert response.status\_code == 200

data = response.json()

assert len(data) == 5

def test\_update\_project\_success(self, client):

"""Test updating a project."""

# Arrange - create project

create\_response = client.post("/api/projects/", json={

"name": "Original Name",

"description": "Original Description"

})

project\_id = create\_response.json()["id"]

# Act - update project

update\_data = {

"name": "Updated Name",

"description": "Updated Description"

}

response = client.patch(f"/api/projects/{project\_id}", json=update\_data)

# Assert

assert response.status\_code == 200

data = response.json()

assert data["name"] == "Updated Name"

assert data["description"] == "Updated Description"

def test\_delete\_project\_success(self, client):

"""Test deleting a project."""

# Arrange - create project

create\_response = client.post("/api/projects/", json={

"name": "To Be Deleted",

"description": "Will be deleted"

})

project\_id = create\_response.json()["id"]

# Act - delete project

response = client.delete(f"/api/projects/{project\_id}")

# Assert

assert response.status\_code == 204

# Verify it's deleted

get\_response = client.get(f"/api/projects/{project\_id}")

assert get\_response.status\_code == 404

def test\_complete\_workflow(self, client):

"""Test complete project creation and retrieval workflow."""

# Create project

create\_response = client.post("/api/projects/", json={

"name": "Workflow Test",

"description": "Complete workflow"

})

assert create\_response.status\_code == 201

project\_id = create\_response.json()["id"]

# Retrieve project with workflow state

get\_response = client.get(f"/api/projects/{project\_id}")

assert get\_response.status\_code == 200

project = get\_response.json()

# Verify workflow state is initialized

assert project["workflow\_state"]["current\_stage"] == "idle"

assert isinstance(project["workflow\_state"]["stage\_outputs"], dict)

assert isinstance(project["workflow\_state"]["approvals"], dict)

assert isinstance(project["workflow\_state"]["history"], list)

```

### 3. End-to-End Tests (10% of test suite)

\*\*Purpose\*\*: Test complete user workflows from UI through backend

\*\*E2E Tests (Playwright)\*\*

```typescript

/\*\*

\* E2E tests for project management workflow

\*/

import { test, expect } from '@playwright/test';

test.describe('Project Management Workflow', () => {

test.beforeEach(async ({ page }) => {

// Navigate to application

await page.goto('/');

});

test('should create new project and navigate to workflow', async ({ page }) => {

// Click "New Project" button

await page.click('button:has-text("New Project")');

// Wait for dialog to open

await expect(page.locator('[role="dialog"]')).toBeVisible();

// Fill in project details

await page.fill('input[name="name"]', 'E2E Test Project');

await page.fill('textarea[name="description"]', 'Created via E2E test');

// Submit form

await page.click('button:has-text("Create Project")');

// Wait for navigation to workflow page

await page.waitForURL(/\/project\/.\*\/workflow/);

// Verify we're on workflow page

await expect(page.locator('h1')).toContainText('E2E Test Project');

await expect(page.locator('[data-testid="workflow-stepper"]')).toBeVisible();

});

test('should show validation error for empty project name', async ({ page }) => {

// Open new project dialog

await page.click('button:has-text("New Project")');

// Try to submit without name

await page.click('button[type="submit"]');

// Verify error message

await expect(page.locator('text=Project name is required')).toBeVisible();

});

test('should show error for duplicate project name', async ({ page }) => {

// Create first project

await page.click('button:has-text("New Project")');

await page.fill('input[name="name"]', 'Duplicate Name');

await page.click('button[type="submit"]');

await page.waitForURL(/\/project\/.\*\/workflow/);

// Go back to dashboard

await page.goto('/');

// Try to create project with same name

await page.click('button:has-text("New Project")');

await page.fill('input[name="name"]', 'Duplicate Name');

await page.click('button[type="submit"]');

// Verify error message

await expect(page.locator('text=already exists')).toBeVisible();

});

test('should list all projects on dashboard', async ({ page }) => {

// Create multiple projects

const projectNames = ['Project A', 'Project B', 'Project C'];

for (const name of projectNames) {

await page.click('button:has-text("New Project")');

await page.fill('input[name="name"]', name);

await page.click('button[type="submit"]');

await page.waitForURL(/\/project\/.\*\/workflow/);

await page.goto('/');

}

// Verify all projects are listed

for (const name of projectNames) {

await expect(page.locator(`text=${name}`)).toBeVisible();

}

});

test('should upload requirements document', async ({ page }) => {

// Create project

await page.click('button:has-text("New Project")');

await page.fill('input[name="name"]', 'Requirements Test');

await page.click('button[type="submit"]');

await page.waitForURL(/\/project\/.\*\/workflow/);

// Click upload button

await page.click('button:has-text("Upload Requirements")');

// Wait for upload modal

await expect(page.locator('[role="dialog"]')).toBeVisible();

// Upload file

const fileInput = page.locator('input[type="file"]');

await fileInput.setInputFiles('tests/fixtures/sample\_requirements.pdf');

// Wait for upload to complete

await expect(page.locator('text=Processing')).toBeVisible();

await expect(page.locator('text=Processing complete')).toBeVisible({ timeout: 30000 });

// Verify navigation to requirements review

await expect(page.locator('h2')).toContainText('Requirements Review');

});

test('should show validation error for invalid file type', async ({ page }) => {

// Create project and go to upload

await page.click('button:has-text("New Project")');

await page.fill('input[name="name"]', 'File Validation Test');

await page.click('button[type="submit"]');

await page.waitForURL(/\/project\/.\*\/workflow/);

await page.click('button:has-text("Upload Requirements")');

// Try to upload invalid file type

const fileInput = page.locator('input[type="file"]');

await fileInput.setInputFiles('tests/fixtures/invalid\_file.exe');

// Verify error message

await expect(page.locator('text=Unsupported file type')).toBeVisible();

});

test('should navigate through workflow stages', async ({ page }) => {

// Create project

await page.click('button:has-text("New Project")');

await page.fill('input[name="name"]', 'Workflow Navigation Test');

await page.click('button[type="submit"]');

await page.waitForURL(/\/project\/.\*\/workflow/);

// Verify initial stage

await expect(page.locator('[data-stage="idle"]')).toHaveClass(/active/);

// Click workflow step (if clickable)

// This would depend on your actual implementation

const stepper = page.locator('[data-testid="workflow-stepper"]');

await expect(stepper).toBeVisible();

});

});

test.describe('Responsive Design', () => {

test('should be usable on tablet', async ({ page }) => {

await page.setViewportSize({ width: 768, height: 1024 });

await page.goto('/');

// Verify layout works on tablet

await expect(page.locator('header')).toBeVisible();

await expect(page.locator('main')).toBeVisible();

// Create project on tablet

await page.click('button:has-text("New Project")');

await expect(page.locator('[role="dialog"]')).toBeVisible();

});

test('should be usable on desktop', async ({ page }) => {

await page.setViewportSize({ width: 1920, height: 1080 });

await page.goto('/');

// Verify layout works on desktop

await expect(page.locator('aside')).toBeVisible(); // Sidebar

await expect(page.locator('main')).toBeVisible();

});

});

test.describe('Accessibility', () => {

test('should be navigable with keyboard', async ({ page }) => {

await page.goto('/');

// Tab through interactive elements

await page.keyboard.press('Tab');

await expect(page.locator('button:focus')).toBeVisible();

// Continue tabbing

await page.keyboard.press('Tab');

await page.keyboard.press('Tab');

// Activate button with Enter

await page.keyboard.press('Enter');

});

test('should have proper ARIA labels', async ({ page }) => {

await page.goto('/');

// Check for proper labels

const newProjectButton = page.locator('button:has-text("New Project")');

await expect(newProjectButton).toHaveAttribute('aria-label');

});

});

```

## Test Case Generation Process

### Step 1: Analyze Requirements

1. Review all acceptance criteria from Jira tickets

2. Extract test scenarios from user stories

3. Identify edge cases and error conditions

4. Map out critical user journeys

### Step 2: Create Test Plan

```markdown

# Test Plan: Project Management Module

## Scope

- Project CRUD operations

- Workflow state management

- Requirements upload and processing

- User input validation

## Test Cases

### TC-001: Create Project - Happy Path

\*\*Priority\*\*: High

\*\*Type\*\*: Integration

\*\*Preconditions\*\*: User on dashboard

\*\*Steps\*\*:

1. Click "New Project" button

2. Enter name: "Test Project"

3. Enter description: "Test Description"

4. Click "Create"

\*\*Expected\*\*:

- Project created with status 201

- Redirected to workflow page

- Project appears in dashboard list

### TC-002: Create Project - Duplicate Name

\*\*Priority\*\*: High

\*\*Type\*\*: Integration

\*\*Preconditions\*\*: Project "Duplicate" already exists

\*\*Steps\*\*:

1. Click "New Project"

2. Enter name: "Duplicate"

3. Click "Create"

\*\*Expected\*\*:

- Error 400

- Message: "Project with this name already exists"

- Form remains open

### TC-003: Create Project - Validation

\*\*Priority\*\*: Medium

\*\*Type\*\*: Unit

\*\*Preconditions\*\*: None

\*\*Steps\*\*:

1. Open new project form

2. Leave name empty

3. Click "Create"

\*\*Expected\*\*:

- Client-side validation error

- Message: "Project name is required"

- No API call made

... (more test cases)

```

### Step 3: Implement Tests

- Write tests following TDD when possible

- Cover all happy paths

- Cover all error scenarios

- Cover edge cases

- Include boundary value testing

### Step 4: Run Tests and Generate Report

```bash

# Backend tests

pytest --cov=app --cov-report=html --cov-report=term

# Frontend tests

npm test -- --coverage

# E2E tests

npx playwright test --reporter=html

```

## Bug Reporting Format

```markdown

# Bug Report: [Short Title]

\*\*Bug ID\*\*: BUG-001

\*\*Severity\*\*: Critical | High | Medium | Low

\*\*Priority\*\*: P0 | P1 | P2 | P3

\*\*Status\*\*: New | In Progress | Fixed | Verified | Closed

\*\*Found In\*\*: Version/Branch

\*\*Environment\*\*: Development | Staging | Production

## Description

[Clear, concise description of the bug]

## Steps to Reproduce

1. [Detailed step 1]

2. [Detailed step 2]

3. [Detailed step 3]

## Expected Behavior

[What should happen]

## Actual Behavior

[What actually happens]

## Screenshots/Recordings

[Attach visual evidence]

## Error Logs

\`\`\`

[Relevant error messages or stack traces]

\`\`\`

## Environment Details

- OS: [Operating System]

- Browser: [Browser and version]

- Screen Resolution: [If UI bug]

- Backend Version: [API version]

- Database: [If data issue]

## Impact

- \*\*User Impact\*\*: [How this affects users]

- \*\*Frequency\*\*: [How often it occurs]

- \*\*Workaround\*\*: [If any workaround exists]

## Root Cause Analysis

[If known, describe what's causing the bug]

## Suggested Fix

[If you have suggestions for fixing]

## Related Issues

- Related to: [Other bug IDs]

- Blocks: [What this blocks]

- Blocked by: [What blocks this]

## Attachments

- [Link to failed test]

- [Link to relevant code]

- [Link to related documentation]

---

\*\*Reported By\*\*: QA Agent

\*\*Date\*\*: 2025-01-15

\*\*Last Updated\*\*: 2025-01-15

```

## Test Coverage Requirements

### Coverage Targets

- \*\*Overall\*\*: 80% minimum

- \*\*Critical Paths\*\*: 100%

- \*\*Services/Business Logic\*\*: 90%

- \*\*Repositories/Data Access\*\*: 85%

- \*\*API Endpoints\*\*: 90%

- \*\*Utilities\*\*: 85%

- \*\*UI Components\*\*: 75%

- \*\*Integration Tests\*\*: All critical flows

### Coverage Report Format

```

----------------------------- coverage: platform darwin, python 3.11 --------------

Name Stmts Miss Cover Missing

-----------------------------------------------------------------

app/\_\_init\_\_.py 0 0 100%

app/main.py 45 2 96% 89-90

app/routers/projects.py 68 5 93% 145-149

app/services/project\_service.py 92 7 92% 78, 145-150

app/repositories/project\_repo.py 54 3 94% 67-69

app/models/project.py 23 0 100%

app/schemas/project.py 31 0 100%

-----------------------------------------------------------------

TOTAL 313 17 95%

======================== 142 passed in 12.5s ===========================

```

## Performance Testing

### Load Testing Script (Locust)

```python

"""

Load testing for SDLC Accelerator API.

"""

from locust import HttpUser, task, between

class SDLCUser(HttpUser):

"""Simulated user for load testing."""

wait\_time = between(1, 3) # Wait 1-3 seconds between tasks

def on\_start(self):

"""Called when user starts - setup tasks."""

# Create a project for this user

response = self.client.post("/api/projects/", json={

"name": f"Load Test Project {self.user\_id}",

"description": "For load testing"

})

self.project\_id = response.json()["id"]

@task(3)

def list\_projects(self):

"""List all projects (high frequency)."""

self.client.get("/api/projects/")

@task(2)

def get\_project(self):

"""Get specific project (medium frequency)."""

self.client.get(f"/api/projects/{self.project\_id}")

@task(1)

def update\_project(self):

"""Update project (low frequency)."""

self.client.patch(

f"/api/projects/{self.project\_id}",

json={"description": "Updated during load test"}

)

# Run with: locust -f tests/performance/locustfile.py --host=http://localhost:8000

```

## Security Testing

### Security Test Cases

```python

"""

Security tests for API endpoints.

"""

import pytest

from fastapi.testclient import TestClient

class TestSecurity:

"""Security test suite."""

def test\_sql\_injection\_prevention(self, client):

"""Test that SQL injection attempts are prevented."""

# Try SQL injection in project name

malicious\_input = "'; DROP TABLE projects; --"

response = client.post("/api/projects/", json={

"name": malicious\_input,

"description": "SQL injection attempt"

})

# Should be treated as normal string, not SQL

assert response.status\_code in [201, 400] # Created or validation error

# Verify database is intact

list\_response = client.get("/api/projects/")

assert list\_response.status\_code == 200

def test\_xss\_prevention(self, client):

"""Test that XSS attempts are sanitized."""

# Try XSS in project description

xss\_input = '<script>alert("XSS")</script>'

response = client.post("/api/projects/", json={

"name": "XSS Test",

"description": xss\_input

})

assert response.status\_code == 201

project\_id = response.json()["id"]

# Retrieve project

get\_response = client.get(f"/api/projects/{project\_id}")

description = get\_response.json()["description"]

# Script tags should be escaped or removed

assert "<script>" not in description or "&lt;script&gt;" in description

def test\_rate\_limiting(self, client):

"""Test that rate limiting is enforced."""

# Make many rapid requests

responses = []

for \_ in range(150): # Assuming limit is 100/min

response = client.get("/api/projects/")

responses.append(response.status\_code)

# Should see 429 (Too Many Requests) after limit

assert 429 in responses

def test\_input\_size\_limits(self, client):

"""Test that oversized inputs are rejected."""

# Try to create project with very long name

long\_name = "A" \* 1000 # 1000 characters

response = client.post("/api/projects/", json={

"name": long\_name,

"description": "Size test"

})

# Should be rejected (422 validation error or 400 bad request)

assert response.status\_code in [400, 422]

```

## Validation Checklist

- [ ] All acceptance criteria have corresponding tests

- [ ] Happy path tests implemented

- [ ] Error scenario tests implemented

- [ ] Edge case tests implemented

- [ ] Boundary value tests implemented

- [ ] Integration tests for all APIs

- [ ] E2E tests for critical user flows

- [ ] Performance tests for high-load scenarios

- [ ] Security tests for common vulnerabilities

- [ ] Code coverage meets requirements (>80%)

- [ ] All tests passing in CI/CD

- [ ] Test documentation complete

- [ ] Bug reports clear and actionable

## Output Files

1. `test\_report.html` - Complete test results

2. `coverage\_report.html` - Code coverage report

3. `bug\_report.md` - List of identified bugs

4. `test\_plan.md` - Comprehensive test plan

5. `performance\_results.json` - Performance test metrics

## Human Interaction

1. \*\*Test Plan Review\*\*: Present test plan, get approval on scope

2. \*\*Test Results\*\*: Show test execution results, coverage metrics

3. \*\*Bug Reports\*\*: Present identified bugs with severity/priority

4. \*\*Coverage Gaps\*\*: Highlight areas needing more tests

5. \*\*Recommendations\*\*: Suggest improvements for quality

6. \*\*Approval\*\*: Get approval that testing is sufficient

## Integration Points

- \*\*Input\*\*: Source code from Coder Agent, PRD, Jira tickets

- \*\*Output\*\*: Test results, bug reports → Coder Agent (for fixes), Validator Agent

- \*\*Services\*\*: pytest, Vitest, Playwright, coverage tools, Locust

## Best Practices

1. Follow Arrange-Act-Assert pattern

2. Use descriptive test names

3. One assertion per test (when possible)

4. Use fixtures for test data

5. Mock external dependencies

6. Test behavior, not implementation

7. Keep tests fast and independent

8. Maintain test code quality

9. Update tests when requirements change

10. Run tests frequently during development

EOF

Due to message length, I'll continue with the remaining agents (Validator, Documentation, Orchestrator) and then the commands and MCP setup in the next response. Should I continue?

Retry

VS

yes, continue

Orchestrated remaining agent components and configuration details.

Let me continue with the remaining sub-agents (Validator, Documentation Agent, Orchestrator) and then the custom commands and MCP configuration.

**4.7 Validator Agent**

bash

cat > .claude/sub-agents/validator.md << 'EOF'

# Validator Agent

## Role & Responsibilities

Validate outputs at each stage of the SDLC workflow, ensure quality standards are met, verify compliance with requirements, and provide detailed validation reports with actionable feedback.

## Core Capabilities

- Multi-stage validation (requirements, PRD, design, code, tests, documentation)

- Automated quality checks

- Manual review facilitation

- Compliance verification

- Gap analysis

- Issue prioritization

- Validation reporting

## Validation Stages

### Stage 1: Requirements Validation

\*\*Validates\*\*: Requirements Analysis Agent output

\*\*Automated Checks\*\*:

```python

def validate\_requirements(requirements\_data: dict) -> ValidationResult:

"""

Validate requirements extraction output.

Returns ValidationResult with pass/fail status and issues.

"""

issues = []

warnings = []

# Check document metadata completeness

if not requirements\_data.get("document\_metadata"):

issues.append({

"severity": "critical",

"category": "completeness",

"message": "Document metadata missing"

})

metadata = requirements\_data.get("document\_metadata", {})

if not metadata.get("processed\_date"):

issues.append({

"severity": "high",

"category": "metadata",

"message": "Processing date not recorded"

})

# Check extraction summary

summary = requirements\_data.get("extraction\_summary", {})

if summary.get("total\_requirements", 0) == 0:

issues.append({

"severity": "critical",

"category": "extraction",

"message": "No requirements extracted from document"

})

# Validate requirements structure

requirements = requirements\_data.get("requirements", {})

# Check functional requirements

functional = requirements.get("functional", [])

for idx, req in enumerate(functional):

# Check required fields

if not req.get("id"):

issues.append({

"severity": "high",

"category": "structure",

"message": f"Functional requirement #{idx} missing ID"

})

if not req.get("title"):

issues.append({

"severity": "high",

"category": "structure",

"message": f"Requirement {req.get('id', idx)} missing title"

})

if not req.get("description"):

warnings.append({

"severity": "medium",

"category": "completeness",

"message": f"Requirement {req.get('id')} missing description"

})

# Check acceptance criteria

if not req.get("acceptance\_criteria"):

warnings.append({

"severity": "medium",

"category": "testability",

"message": f"Requirement {req.get('id')} missing acceptance criteria"

})

# Check priority assignment

if not req.get("priority") or req.get("priority") not in ["high", "medium", "low"]:

warnings.append({

"severity": "low",

"category": "prioritization",

"message": f"Requirement {req.get('id')} missing or invalid priority"

})

# Check for duplicate requirement IDs

all\_req\_ids = [r.get("id") for r in functional if r.get("id")]

if len(all\_req\_ids) != len(set(all\_req\_ids)):

issues.append({

"severity": "critical",

"category": "uniqueness",

"message": "Duplicate requirement IDs found"

})

# Check non-functional requirements

non\_functional = requirements.get("non\_functional", [])

if len(non\_functional) == 0:

warnings.append({

"severity": "medium",

"category": "completeness",

"message": "No non-functional requirements identified"

})

# Check for measurable criteria in NFRs

for nfr in non\_functional:

if not nfr.get("measurable\_criteria"):

warnings.append({

"severity": "medium",

"category": "measurability",

"message": f"NFR {nfr.get('id')} lacks measurable criteria"

})

# Check clarifications

if len(requirements\_data.get("clarifications\_needed", [])) > 10:

warnings.append({

"severity": "low",

"category": "clarity",

"message": f"Many clarifications needed ({len(requirements\_data['clarifications\_needed'])}). Document may be unclear."

})

# Calculate validation score

critical\_issues = len([i for i in issues if i["severity"] == "critical"])

high\_issues = len([i for i in issues if i["severity"] == "high"])

passed = critical\_issues == 0

score = max(0, 100 - (critical\_issues \* 30) - (high\_issues \* 10) - (len(warnings) \* 2))

return ValidationResult(

passed=passed,

score=score,

issues=issues,

warnings=warnings

)

```

\*\*Manual Review Checklist\*\*:

- [ ] All sections of document processed

- [ ] Extracted text matches original document

- [ ] Tables correctly structured

- [ ] Images referenced with descriptions

- [ ] Language translation accurate (if applicable)

- [ ] Requirements logically categorized

- [ ] No critical information missing

- [ ] Ambiguities appropriately flagged

- [ ] Assumptions reasonable

\*\*Validation Report Template\*\*:

```markdown

# Requirements Validation Report

\*\*Project\*\*: [Project Name]

\*\*Date\*\*: 2025-01-15

\*\*Validator\*\*: Validator Agent

\*\*Stage\*\*: Requirements Analysis

\*\*Version\*\*: 1.0

## Executive Summary

✅ \*\*PASSED\*\* | ❌ \*\*FAILED\*\* | ⚠️ \*\*PASSED WITH WARNINGS\*\*

Overall Score: \*\*88/100\*\*

## Summary Statistics

- \*\*Total Requirements Extracted\*\*: 87

- Functional: 52

- Non-Functional: 20

- Business Rules: 15

- \*\*Requirements with Acceptance Criteria\*\*: 45/52 (87%)

- \*\*Requirements with Priority\*\*: 87/87 (100%)

- \*\*Clarifications Needed\*\*: 12

## Critical Issues (Must Fix)

None

## High Priority Issues (Should Fix)

None

## Warnings (Consider Addressing)

1. \*\*NFR-005 lacks measurable criteria\*\*

- Impact: Cannot effectively test performance requirement

- Recommendation: Add specific metrics (e.g., "Response time < 2s")

2. \*\*12 clarifications needed\*\*

- Impact: May require follow-up with stakeholders

- Recommendation: Address before moving to PRD generation

## Compliance Check

- [x] All requirements have unique IDs

- [x] Requirements properly categorized

- [x] Document metadata complete

- [x] Extraction statistics recorded

- [x] Stakeholders identified

- [⚠️] Some NFRs lack measurable criteria

## Quality Metrics

| Metric | Value | Target | Status |

|--------|-------|--------|--------|

| Extraction Completeness | 98% | >95% | ✅ Pass |

| Requirements with IDs | 100% | 100% | ✅ Pass |

| Requirements with Acceptance Criteria | 87% | >80% | ✅ Pass |

| NFRs with Measurable Criteria | 75% | >90% | ⚠️ Warning |

| Duplicate IDs | 0 | 0 | ✅ Pass |

## Recommendation

\*\*✅ APPROVED WITH CONDITIONS\*\*

\*\*Conditions\*\*:

1. Add measurable criteria to NFR-005, NFR-008, NFR-012, NFR-015, NFR-018

2. Resolve clarifications with stakeholder before PRD generation

## Next Steps

1. Address warnings listed above

2. Review clarifications with stakeholder

3. Once conditions met, proceed to PRD Generation stage

---

Generated by Validator Agent

```

### Stage 2: PRD Validation

\*\*Validates\*\*: PRD Generator Agent output

\*\*Automated Checks\*\*:

```python

def validate\_prd(prd\_content: str, requirements\_data: dict) -> ValidationResult:

"""Validate PRD completeness and quality."""

issues = []

warnings = []

# Check required sections

required\_sections = [

"# Product Requirements Document",

"## Executive Summary",

"## Product Overview",

"## User Analysis",

"## Functional Requirements",

"## Non-Functional Requirements",

"## Technical Requirements",

"## Success Metrics"

]

for section in required\_sections:

if section not in prd\_content:

issues.append({

"severity": "high",

"category": "completeness",

"message": f"Missing required section: {section}"

})

# Check that all requirements from extraction are included

source\_req\_ids = set()

for req\_type in ["functional", "non\_functional", "business\_rules"]:

reqs = requirements\_data.get("requirements", {}).get(req\_type, [])

source\_req\_ids.update([r.get("id") for r in reqs if r.get("id")])

# Extract requirement IDs from PRD

import re

prd\_req\_ids = set(re.findall(r'\b(FR-\d+|NFR-\d+|BR-\d+)\b', prd\_content))

missing\_reqs = source\_req\_ids - prd\_req\_ids

if missing\_reqs:

issues.append({

"severity": "critical",

"category": "traceability",

"message": f"Requirements missing from PRD: {', '.join(list(missing\_reqs)[:5])}"

})

# Check for user stories

if "As a" not in prd\_content:

warnings.append({

"severity": "medium",

"category": "format",

"message": "No user stories found in PRD"

})

# Check for acceptance criteria

ac\_count = prd\_content.count("Acceptance Criteria")

if ac\_count < len(source\_req\_ids) \* 0.7:

warnings.append({

"severity": "medium",

"category": "testability",

"message": "Many requirements lack acceptance criteria"

})

# Check for measurable success metrics

if "KPI" not in prd\_content and "metric" not in prd\_content.lower():

warnings.append({

"severity": "medium",

"category": "measurability",

"message": "Success metrics may not be measurable"

})

# Check for risks section

if "Risk" not in prd\_content:

warnings.append({

"severity": "low",

"category": "completeness",

"message": "No risk analysis section found"

})

# Calculate score

critical\_issues = len([i for i in issues if i["severity"] == "critical"])

high\_issues = len([i for i in issues if i["severity"] == "high"])

passed = critical\_issues == 0

score = max(0, 100 - (critical\_issues \* 30) - (high\_issues \* 10) - (len(warnings) \* 2))

return ValidationResult(

passed=passed,

score=score,

issues=issues,

warnings=warnings

)

```

\*\*Manual Review Checklist\*\*:

- [ ] All requirements from analysis included

- [ ] User stories follow correct format

- [ ] Acceptance criteria clear and measurable

- [ ] Non-functional requirements specified with metrics

- [ ] Success criteria defined with KPIs

- [ ] Technical stack appropriate

- [ ] API specifications detailed

- [ ] Risks identified with mitigation

- [ ] No contradictions or inconsistencies

- [ ] Document is comprehensive yet readable

### Stage 3: Design Validation

\*\*Validates\*\*: Design Architect Agent output

\*\*Automated Checks\*\*:

```python

def validate\_design(design\_docs: dict) -> ValidationResult:

"""Validate design and architecture artifacts."""

issues = []

warnings = []

required\_docs = [

"architecture-overview.md",

"database-design.md",

"api-specification.yaml"

]

for doc in required\_docs:

if doc not in design\_docs:

issues.append({

"severity": "critical",

"category": "completeness",

"message": f"Missing required design document: {doc}"

})

# Validate architecture document

arch\_doc = design\_docs.get("architecture-overview.md", "")

if "```mermaid" not in arch\_doc:

warnings.append({

"severity": "medium",

"category": "visualization",

"message": "Architecture document missing diagrams"

})

# Validate database design

db\_doc = design\_docs.get("database-design.md", "")

if "CREATE TABLE" not in db\_doc:

issues.append({

"severity": "high",

"category": "completeness",

"message": "Database design missing table schemas"

})

if "INDEX" not in db\_doc:

warnings.append({

"severity": "low",

"category": "performance",

"message": "Database design missing index definitions"

})

# Validate API spec

api\_spec = design\_docs.get("api-specification.yaml", "")

if not api\_spec:

issues.append({

"severity": "critical",

"category": "completeness",

"message": "API specification is empty"

})

elif "openapi:" not in api\_spec:

issues.append({

"severity": "high",

"category": "format",

"message": "API specification not in OpenAPI format"

})

# Check for security considerations

has\_security = any("security" in doc.lower() for doc in design\_docs.values())

if not has\_security:

warnings.append({

"severity": "medium",

"category": "security",

"message": "No security architecture documented"

})

# Calculate score

critical\_issues = len([i for i in issues if i["severity"] == "critical"])

high\_issues = len([i for i in issues if i["severity"] == "high"])

passed = critical\_issues == 0 and high\_issues == 0

score = max(0, 100 - (critical\_issues \* 30) - (high\_issues \* 10) - (len(warnings) \* 2))

return ValidationResult(

passed=passed,

score=score,

issues=issues,

warnings=warnings

)

```

\*\*Manual Review Checklist\*\*:

- [ ] Architecture addresses all PRD requirements

- [ ] All diagrams clear, accurate, properly formatted

- [ ] Database schema normalized appropriately

- [ ] All foreign keys and constraints defined

- [ ] Indexes planned for performance

- [ ] API design RESTful and consistent

- [ ] All endpoints documented

- [ ] Security measures comprehensive

- [ ] Scalability addressed

- [ ] No single point of failure

- [ ] Technology choices justified

### Stage 4: Jira Validation

\*\*Validates\*\*: Jira Manager Agent output

\*\*Automated Checks\*\*:

```python

def validate\_jira\_structure(jira\_data: dict, prd\_data: dict) -> ValidationResult:

"""Validate Jira ticket structure and completeness."""

issues = []

warnings = []

epics = jira\_data.get("epics", [])

if not epics:

issues.append({

"severity": "critical",

"category": "structure",

"message": "No epics defined"

})

return ValidationResult(passed=False, score=0, issues=issues, warnings=warnings)

total\_stories = 0

total\_story\_points = 0

stories\_without\_ac = []

stories\_without\_dod = []

oversized\_stories = []

for epic in epics:

stories = epic.get("stories", [])

total\_stories += len(stories)

for story in stories:

# Check story points

sp = story.get("story\_points", 0)

total\_story\_points += sp

if sp == 0:

warnings.append({

"severity": "medium",

"category": "estimation",

"message": f"Story {story.get('key')} missing story points"

})

elif sp > 8:

oversized\_stories.append(story.get("key"))

# Check acceptance criteria

if not story.get("acceptance\_criteria"):

stories\_without\_ac.append(story.get("key"))

# Check definition of done

if not story.get("definition\_of\_done"):

stories\_without\_dod.append(story.get("key"))

# Check user story format

title = story.get("title", "")

if not ("As a" in title and "I want" in title and "so that" in title):

warnings.append({

"severity": "low",

"category": "format",

"message": f"Story {story.get('key')} not in proper user story format"

})

# Report issues

if oversized\_stories:

issues.append({

"severity": "high",

"category": "sizing",

"message": f"{len(oversized\_stories)} stories >8 SP (should be split): {', '.join(oversized\_stories[:5])}"

})

if stories\_without\_ac:

issues.append({

"severity": "high",

"category": "testability",

"message": f"{len(stories\_without\_ac)} stories without acceptance criteria"

})

if stories\_without\_dod:

warnings.append({

"severity": "medium",

"category": "completeness",

"message": f"{len(stories\_without\_dod)} stories without definition of done"

})

# Check dependencies

dependencies = jira\_data.get("dependencies", [])

circular\_deps = detect\_circular\_dependencies(dependencies)

if circular\_deps:

issues.append({

"severity": "critical",

"category": "dependencies",

"message": f"Circular dependencies detected: {circular\_deps}"

})

# Calculate score

critical\_issues = len([i for i in issues if i["severity"] == "critical"])

high\_issues = len([i for i in issues if i["severity"] == "high"])

passed = critical\_issues == 0

score = max(0, 100 - (critical\_issues \* 30) - (high\_issues \* 10) - (len(warnings) \* 2))

return ValidationResult(

passed=passed,

score=score,

issues=issues,

warnings=warnings,

metrics={

"total\_epics": len(epics),

"total\_stories": total\_stories,

"total\_story\_points": total\_story\_points,

"avg\_story\_points": total\_story\_points / total\_stories if total\_stories > 0 else 0

}

)

def detect\_circular\_dependencies(dependencies: list) -> list:

"""Detect circular dependencies in Jira structure."""

# Build dependency graph

graph = {}

for dep in dependencies:

from\_key = dep.get("from")

to\_key = dep.get("to")

if from\_key not in graph:

graph[from\_key] = []

graph[from\_key].append(to\_key)

# DFS to detect cycles

visited = set()

rec\_stack = set()

cycles = []

def dfs(node, path):

visited.add(node)

rec\_stack.add(node)

path = path + [node]

for neighbor in graph.get(node, []):

if neighbor not in visited:

dfs(neighbor, path)

elif neighbor in rec\_stack:

# Found cycle

cycle\_start = path.index(neighbor)

cycles.append(" -> ".join(path[cycle\_start:] + [neighbor]))

rec\_stack.remove(node)

for node in graph:

if node not in visited:

dfs(node, [])

return cycles

```

\*\*Manual Review Checklist\*\*:

- [ ] All PRD features mapped to Jira structure

- [ ] Story points estimated (1 SP = 1 day)

- [ ] Dependencies properly linked

- [ ] No circular dependencies

- [ ] Every story has acceptance criteria

- [ ] Every story has definition of done

- [ ] Priorities assigned

- [ ] No orphaned tickets

- [ ] Epic-Story-Task hierarchy correct

- [ ] Critical path identified

### Stage 5: Code Validation

\*\*Validates\*\*: Coder Agent output

\*\*Automated Checks\*\*:

```python

def validate\_code(code\_files: dict, prd\_data: dict, design\_data: dict) -> ValidationResult:

"""Validate generated code quality and completeness."""

issues = []

warnings = []

# Check backend structure

backend\_files = [f for f in code\_files if f.startswith("backend/")]

expected\_backend = [

"backend/app/main.py",

"backend/app/routers/",

"backend/app/models/",

"backend/app/schemas/",

"backend/app/services/",

"backend/requirements.txt"

]

for expected in expected\_backend:

matching = [f for f in backend\_files if expected in f]

if not matching:

issues.append({

"severity": "high",

"category": "structure",

"message": f"Missing expected backend component: {expected}"

})

# Check frontend structure

frontend\_files = [f for f in code\_files if f.startswith("frontend/")]

expected\_frontend = [

"frontend/src/main.tsx",

"frontend/src/App.tsx",

"frontend/src/components/",

"frontend/src/services/",

"frontend/package.json"

]

for expected in expected\_frontend:

matching = [f for f in frontend\_files if expected in f]

if not matching:

issues.append({

"severity": "high",

"category": "structure",

"message": f"Missing expected frontend component: {expected}"

})

# Run linting checks

python\_files = [f for f in code\_files if f.endswith(".py")]

for py\_file in python\_files:

code = code\_files[py\_file]

# Check for type hints

if "def " in code and "->" not in code:

warnings.append({

"severity": "low",

"category": "typing",

"message": f"{py\_file} missing type hints"

})

# Check for docstrings

if '"""' not in code and "'''" not in code:

warnings.append({

"severity": "low",

"category": "documentation",

"message": f"{py\_file} missing docstrings"

})

# Check for hardcoded secrets

if any(secret in code.lower() for secret in ["password = ", "api\_key = ", "secret = "]):

issues.append({

"severity": "critical",

"category": "security",

"message": f"{py\_file} contains hardcoded secrets"

})

# Check TypeScript files

ts\_files = [f for f in code\_files if f.endswith((".ts", ".tsx"))]

for ts\_file in ts\_files:

code = code\_files[ts\_file]

# Check for 'any' type

if ": any" in code or " any[]" in code:

warnings.append({

"severity": "medium",

"category": "typing",

"message": f"{ts\_file} uses 'any' type (should be avoided)"

})

# Check for proper error handling

if "fetch(" in code and "catch" not in code:

warnings.append({

"severity": "medium",

"category": "error-handling",

"message": f"{ts\_file} has unhandled fetch errors"

})

# Check for tests

test\_files = [f for f in code\_files if "test" in f]

if not test\_files:

issues.append({

"severity": "critical",

"category": "testing",

"message": "No test files found"

})

# Calculate score

critical\_issues = len([i for i in issues if i["severity"] == "critical"])

high\_issues = len([i for i in issues if i["severity"] == "high"])

passed = critical\_issues == 0

score = max(0, 100 - (critical\_issues \* 30) - (high\_issues \* 10) - (len(warnings) \* 2))

return ValidationResult(

passed=passed,

score=score,

issues=issues,

warnings=warnings,

metrics={

"total\_files": len(code\_files),

"backend\_files": len(backend\_files),

"frontend\_files": len(frontend\_files),

"test\_files": len(test\_files)

}

)

```

\*\*Manual Review Checklist\*\*:

- [ ] Code implements all acceptance criteria

- [ ] Follows project style guide

- [ ] All functions have type hints/types

- [ ] Comprehensive error handling

- [ ] Input validation implemented

- [ ] Security best practices followed

- [ ] No hardcoded secrets

- [ ] Logging implemented

- [ ] Comments and documentation complete

- [ ] Tests written (>80% coverage)

- [ ] Performance optimized

- [ ] Accessibility requirements met

### Stage 6: Test Validation

\*\*Validates\*\*: QA Testing Agent output

\*\*Automated Checks\*\*:

```python

def validate\_tests(test\_results: dict, coverage\_data: dict) -> ValidationResult:

"""Validate test suite completeness and coverage."""

issues = []

warnings = []

# Check overall coverage

overall\_coverage = coverage\_data.get("overall\_coverage", 0)

if overall\_coverage < 80:

issues.append({

"severity": "high",

"category": "coverage",

"message": f"Overall coverage {overall\_coverage}% is below 80% target"

})

elif overall\_coverage < 85:

warnings.append({

"severity": "low",

"category": "coverage",

"message": f"Coverage {overall\_coverage}% could be improved (target 85%+)"

})

# Check test results

total\_tests = test\_results.get("total", 0)

passed = test\_results.get("passed", 0)

failed = test\_results.get("failed", 0)

if total\_tests == 0:

issues.append({

"severity": "critical",

"category": "testing",

"message": "No tests executed"

})

return ValidationResult(passed=False, score=0, issues=issues, warnings=warnings)

if failed > 0:

issues.append({

"severity": "critical",

"category": "testing",

"message": f"{failed} tests failing"

})

pass\_rate = (passed / total\_tests) \* 100

if pass\_rate < 100:

issues.append({

"severity": "critical",

"category": "quality",

"message": f"Test pass rate {pass\_rate:.1f}% (should be 100%)"

})

# Check test types distribution

unit\_tests = test\_results.get("unit\_tests", 0)

integration\_tests = test\_results.get("integration\_tests", 0)

e2e\_tests = test\_results.get("e2e\_tests", 0)

if unit\_tests < total\_tests \* 0.6:

warnings.append({

"severity": "medium",

"category": "test-distribution",

"message": "Unit tests should be ~70% of total tests"

})

if e2e\_tests == 0:

warnings.append({

"severity": "medium",

"category": "test-coverage",

"message": "No E2E tests found"

})

# Check for critical path coverage

critical\_files\_coverage = coverage\_data.get("critical\_files", {})

uncovered\_critical = [f for f, cov in critical\_files\_coverage.items() if cov < 100]

if uncovered\_critical:

warnings.append({

"severity": "medium",

"category": "critical-coverage",

"message": f"{len(uncovered\_critical)} critical files not fully covered"

})

# Calculate score

critical\_issues = len([i for i in issues if i["severity"] == "critical"])

high\_issues = len([i for i in issues if i["severity"] == "high"])

passed\_validation = critical\_issues == 0

score = max(0, 100 - (critical\_issues \* 30) - (high\_issues \* 10) - (len(warnings) \* 2))

return ValidationResult(

passed=passed\_validation,

score=score,

issues=issues,

warnings=warnings,

metrics={

"total\_tests": total\_tests,

"passed": passed,

"failed": failed,

"pass\_rate": pass\_rate,

"coverage": overall\_coverage

}

)

```

\*\*Manual Review Checklist\*\*:

- [ ] All acceptance criteria have tests

- [ ] Edge cases covered

- [ ] Error scenarios tested

- [ ] Integration tests for all APIs

- [ ] E2E tests for critical flows

- [ ] All tests passing

- [ ] No flaky tests

- [ ] Performance tests included

- [ ] Security tests included

- [ ] Code coverage meets requirements

### Stage 7: Documentation Validation

\*\*Validates\*\*: Documentation Agent output

\*\*Automated Checks\*\*:

```python

def validate\_documentation(docs: dict) -> ValidationResult:

"""Validate documentation completeness."""

issues = []

warnings = []

required\_docs = [

"README.md",

"api-documentation",

"architecture-documentation"

]

for doc in required\_docs:

if doc not in docs:

issues.append({

"severity": "high",

"category": "completeness",

"message": f"Missing required documentation: {doc}"

})

# Check README completeness

readme = docs.get("README.md", "")

required\_readme\_sections = [

"# ", "## Installation", "## Usage", "## Testing"

]

for section in required\_readme\_sections:

if section not in readme:

warnings.append({

"severity": "medium",

"category": "readme",

"message": f"README missing section: {section}"

})

# Check API documentation

if "api-documentation" in docs:

api\_doc = docs["api-documentation"]

if "endpoint" not in api\_doc.lower():

warnings.append({

"severity": "medium",

"category": "api-docs",

"message": "API documentation may be incomplete"

})

# Calculate score

critical\_issues = len([i for i in issues if i["severity"] == "critical"])

high\_issues = len([i for i in issues if i["severity"] == "high"])

passed = critical\_issues == 0

score = max(0, 100 - (critical\_issues \* 30) - (high\_issues \* 10) - (len(warnings) \* 2))

return ValidationResult(

passed=passed,

score=score,

issues=issues,

warnings=warnings

)

```

\*\*Manual Review Checklist\*\*:

- [ ] README complete with all sections

- [ ] API documentation generated and accurate

- [ ] Code comments comprehensive

- [ ] User guide covers all features

- [ ] Deployment guide tested and accurate

- [ ] Architecture documentation current

- [ ] Examples provided for complex features

- [ ] Troubleshooting section included

## Validation Process

1. \*\*Run Automated Checks\*\*: Execute validation functions for current stage

2. \*\*Generate Validation Report\*\*: Create detailed report with issues and warnings

3. \*\*Manual Review\*\*: Present to human for review with checklist

4. \*\*Calculate Score\*\*: Compute overall quality score

5. \*\*Make Recommendation\*\*: Approve, reject, or approve with conditions

6. \*\*Track Issues\*\*: Log all issues for resolution

## Escalation Rules

\*\*Critical Issues\*\*:

- Block progression immediately

- Require immediate fix

- Cannot proceed until resolved

\*\*High Issues\*\*:

- Block progression

- Must be addressed before moving forward

- Can discuss workarounds with user approval

\*\*Medium Issues\*\*:

- Flag for review

- May proceed with explicit user approval

- Should be addressed eventually

\*\*Low Issues\*\*:

- Log for future improvement

- Can proceed

- Nice-to-have fixes

## Output Format

```markdown

# Validation Report

\*\*Stage\*\*: [Stage Name]

\*\*Date\*\*: [Date]

\*\*Status\*\*: ✅ APPROVED | ❌ REJECTED | ⚠️ APPROVED WITH CONDITIONS

\*\*Score\*\*: [Score]/100

## Summary

[Brief summary of validation results]

## Critical Issues

[List of critical issues]

## High Priority Issues

[List of high issues]

## Warnings

[List of warnings]

## Metrics

[Relevant metrics for this stage]

## Recommendation

[Approval decision with reasoning]

## Conditions (if applicable)

[What must be fixed before proceeding]

## Next Steps

[What happens next]

```

## Human Interaction

1. Present validation report with clear visuals

2. Explain each issue with context and impact

3. Provide actionable recommendations

4. Wait for explicit approval or rejection

5. Track feedback for next iteration

## Integration Points

- \*\*Input\*\*: Outputs from all other agents

- \*\*Output\*\*: Validation reports → Human reviewer, Orchestrator

- \*\*Services\*\*: Linting tools, security scanners, coverage tools

## Best Practices

1. Be thorough but not overly strict

2. Provide context for every issue

3. Suggest specific fixes

4. Prioritize issues by impact

5. Track validation history

6. Learn from past validations

7. Balance automation with human judgment

8. Focus on what matters most

EOF

**4.8 Documentation Agent**

bash

cat > .claude/sub-agents/documentation-agent.md << 'EOF'

# Documentation Agent

## Role & Responsibilities

Generate comprehensive, high-quality documentation for the entire application including README files, API documentation, code documentation, user guides, deployment guides, and architecture documentation.

## Input

- Source code (backend + frontend)

- PRD and design documents

- Architecture diagrams

- API specifications

- Test results

## Core Capabilities

- README generation

- API documentation (OpenAPI/Swagger)

- Code documentation (docstrings, comments)

- User guide creation

- Deployment guide creation

- Architecture documentation

- Troubleshooting guide creation

- Code explanation and flow documentation

## Documentation Types

### 1. README.md (Project Root)

```markdown

# SDLC Accelerator

\*\*AI-Powered Software Development Lifecycle Automation\*\*

Automate your entire SDLC from requirements analysis to code deployment using Claude AI agents.

## 🚀 Features

- \*\*Requirements Analysis\*\*: Upload documents, extract requirements automatically

- \*\*PRD Generation\*\*: Create comprehensive Product Requirement Documents

- \*\*Design & Architecture\*\*: Generate system designs, database schemas, API specs

- \*\*Jira Integration\*\*: Auto-create epics, stories, and tasks

- \*\*Code Generation\*\*: Generate production-ready backend and frontend code

- \*\*Testing\*\*: Comprehensive test suite generation and execution

- \*\*Documentation\*\*: Automated documentation generation

- \*\*Human-in-the-Loop\*\*: Review and approve at each workflow stage

## 🏗️ Architecture

```

┌─────────────┐ ┌──────────────┐ ┌─────────────┐

│ React │────▶│ FastAPI │────▶│ PostgreSQL │

│ Frontend │ │ Backend │ │ Database │

└─────────────┘ └──────────────┘ └─────────────┘

│

┌──────────┴──────────┐

│ │

┌──────▼──────┐ ┌──────▼──────┐

│ AWS Bedrock │ │ AWS Textract│

│ (Claude) │ │ (OCR) │

└─────────────┘ └─────────────┘

```

## 📋 Prerequisites

- \*\*Node.js\*\* 18+ and npm

- \*\*Python\*\* 3.11+

- \*\*PostgreSQL\*\* 15+

- \*\*Docker\*\* (optional but recommended)

- \*\*AWS Account\*\* with Bedrock and Textract access

- \*\*Jira Account\*\* (optional)

- \*\*Confluence Account\*\* (optional)

- \*\*GitLab Account\*\* (optional)

## 🛠️ Installation

### 1. Clone Repository

\`\`\`bash

git clone https://github.com/your-org/sdlc-accelerator.git

cd sdlc-accelerator

\`\`\`

### 2. Backend Setup

\`\`\`bash

cd backend

# Create virtual environment

python -m venv venv

source venv/bin/activate # On Windows: venv\Scripts\activate

# Install dependencies

pip install -r requirements.txt

# Copy environment template

cp .env.example .env

# Edit .env with your configuration

nano .env

\`\`\`

### 3. Frontend Setup

\`\`\`bash

cd frontend

# Install dependencies

npm install

# Copy environment template

cp .env.example .env

# Edit .env with your configuration

nano .env

\`\`\`

### 4. Database Setup

\`\`\`bash

# Create database

createdb sdlc\_accelerator

# Run migrations

cd backend

alembic upgrade head

# (Optional) Seed with sample data

python scripts/seed\_data.py

\`\`\`

## 🔧 Configuration

### Backend Environment Variables

\`\`\`env

# Database

DATABASE\_URL=postgresql://user:password@localhost:5432/sdlc\_accelerator

# AWS Bedrock

AWS\_ACCESS\_KEY\_ID=your\_access\_key

AWS\_SECRET\_ACCESS\_KEY=your\_secret\_key

AWS\_REGION=us-east-1

BEDROCK\_MODEL\_ID=anthropic.claude-3-5-sonnet-20241022-v2:0

# AWS Textract

TEXTRACT\_REGION=us-east-1

# AWS S3

S3\_BUCKET\_NAME=sdlc-accelerator-files

# Jira (Optional)

JIRA\_URL=https://your-domain.atlassian.net

JIRA\_EMAIL=your-email@example.com

JIRA\_API\_TOKEN=your\_jira\_api\_token

# Confluence (Optional)

CONFLUENCE\_URL=https://your-domain.atlassian.net/wiki

CONFLUENCE\_API\_TOKEN=your\_confluence\_token

# GitLab (Optional)

GITLAB\_URL=https://gitlab.com

GITLAB\_TOKEN=your\_gitlab\_token

# Application

LOG\_LEVEL=INFO

CORS\_ORIGINS=http://localhost:5173

\`\`\`

### Frontend Environment Variables

\`\`\`env

VITE\_API\_URL=http://localhost:8000/api

\`\`\`

## 🚀 Running the Application

### Development Mode

\`\`\`bash

# Terminal 1: Backend

cd backend

source venv/bin/activate

uvicorn app.main:app --reload --port 8000

# Terminal 2: Frontend

cd frontend

npm run dev

\`\`\`

Access the application at: \*\*http://localhost:5173\*\*

### Production Mode with Docker

\`\`\`bash

docker-compose up -d

\`\`\`

## 📖 Usage

### 1. Create New Project

1. Click \*\*"New Project"\*\* button on dashboard

2. Enter project name and description

3. Click \*\*"Create Project"\*\*

### 2. Upload Requirements Document

1. Open your project

2. Click \*\*"Upload Requirements"\*\*

3. Select document (PDF, DOCX, or TXT up to 50MB)

4. Wait for processing (typically 10-30 seconds)

5. Review extracted requirements

### 3. Generate PRD

1. After requirements approval, click \*\*"Generate PRD"\*\*

2. Wait for PRD generation

3. Review generated PRD

4. Provide feedback or approve

### 4. Create Design & Architecture

1. After PRD approval, click \*\*"Generate Design"\*\*

2. Review architecture diagrams

3. Review database schema

4. Review API specifications

5. Approve to proceed

### 5. Generate Jira Tickets

1. After design approval, click \*\*"Generate Jira Tickets"\*\*

2. Review epic/story structure

3. Modify story points if needed

4. Approve to create in Jira

### 6. Generate Code

1. After Jira approval, click \*\*"Generate Code"\*\*

2. View generated code in preview pane

3. Test in preview environment

4. Request changes or approve

### 7. Run Tests

1. Tests run automatically after code generation

2. Review test results and coverage

3. Fix any failing tests

### 8. Generate Documentation

1. After tests pass, documentation is auto-generated

2. Review README, API docs, user guide

3. Approve for completion

## 🧪 Testing

\`\`\`bash

# Backend tests

cd backend

pytest --cov=app --cov-report=html

# Frontend tests

cd frontend

npm test

# E2E tests

cd frontend

npx playwright test

\`\`\`

## 📚 API Documentation

Once running, access API documentation at:

- \*\*Swagger UI\*\*: http://localhost:8000/docs

- \*\*ReDoc\*\*: http://localhost:8000/redoc

## 🏗️ Project Structure

\`\`\`

sdlc-accelerator/

├── backend/ # FastAPI backend

│ ├── app/

│ │ ├── routers/ # API endpoints

│ │ ├── services/ # Business logic

│ │ ├── models/ # Database models

│ │ ├── schemas/ # Pydantic schemas

│ │ └── agents/ # AI agents

│ └── tests/

├── frontend/ # React frontend

│ ├── src/

│ │ ├── components/ # React components

│ │ ├── pages/ # Page components

│ │ ├── services/ # API clients

│ │ └── stores/ # State management

│ └── tests/

└── docs/ # Documentation

\`\`\`

## 🐛 Troubleshooting

### Common Issues

\*\*Issue\*\*: Database connection fails

\*\*Solution\*\*:

- Verify PostgreSQL is running

- Check DATABASE\_URL in .env

- Ensure database exists

\*\*Issue\*\*: AWS Bedrock errors

\*\*Solution\*\*:

- Verify AWS credentials are correct

- Check Bedrock service is available in your region

- Ensure you have Bedrock access enabled

\*\*Issue\*\*: Frontend can't connect to backend

\*\*Solution\*\*:

- Verify backend is running on port 8000

- Check VITE\_API\_URL in frontend/.env

- Check CORS\_ORIGINS in backend/.env

## 🤝 Contributing

1. Fork the repository

2. Create feature branch (`git checkout -b feature/amazing-feature`)

3. Commit changes (`git commit -m 'Add amazing feature'`)

4. Push to branch (`git push origin feature/amazing-feature`)

5. Open Pull Request

## 📄 License

MIT License - see LICENSE file for details

## 🙏 Acknowledgments

- Built with Claude AI (Anthropic)

- Powered by AWS Bedrock and AWS Textract

- UI components from shadcn/ui

## 📧 Support

- \*\*Email\*\*: support@sdlc-accelerator.com

- \*\*Issues\*\*: https://github.com/your-org/sdlc-accelerator/issues

- \*\*Documentation\*\*: https://docs.sdlc-accelerator.com

## 🗺️ Roadmap

- [ ] Multi-user support with authentication

- [ ] Real-time collaboration

- [ ] Custom AI model training

- [ ] More integrations (GitHub, Azure DevOps)

- [ ] Mobile app

- [ ] CI/CD pipeline templates

```

### 2. API Documentation

Generate comprehensive OpenAPI documentation embedded in FastAPI:

```python

from fastapi import FastAPI

from fastapi.openapi.utils import get\_openapi

def custom\_openapi():

if app.openapi\_schema:

return app.openapi\_schema

openapi\_schema = get\_openapi(

title="SDLC Accelerator API",

version="1.0.0",

description="""

# SDLC Accelerator REST API

Comprehensive API for automating the software development lifecycle.

## Features

\* 🚀 Project Management

\* 📝 Requirements Analysis

\* 📄 PRD Generation

\* 🏗️ Design & Architecture

\* 🎯 Jira Integration

\* 💻 Code Generation

\* ✅ Testing Automation

## Authentication

This API currently operates in single-user mode and does not require authentication.

## Rate Limiting

- \*\*Rate Limit\*\*: 100 requests per minute per IP

- \*\*Burst Limit\*\*: 20 requests per second

## Error Codes

| Code | Description |

|------|-------------|

| 400 | Bad Request - Invalid input |

| 404 | Not Found - Resource doesn't exist |

| 409 | Conflict - Duplicate resource |

| 422 | Validation Error - Input validation failed |

| 429 | Too Many Requests - Rate limit exceeded |

| 500 | Internal Server Error |

## Support

Email: api-support@sdlc-accelerator.com

""",

routes=app.routes,

)

openapi\_schema["info"]["x-logo"] = {

"url": "https://sdlc-accelerator.com/logo.png"

}

app.openapi\_schema = openapi\_schema

return app.openapi\_schema

app.openapi = custom\_openapi

```

### 3. User Guide

```markdown

# SDLC Accelerator User Guide

## Table of Contents

1. [Getting Started](#getting-started)

2. [Creating Your First Project](#creating-your-first-project)

3. [Requirements Analysis](#requirements-analysis)

4. [Working with PRDs](#working-with-prds)

5. [Design Review](#design-review)

6. [Jira Integration](#jira-integration)

7. [Code Generation](#code-generation)

8. [Testing](#testing)

9. [Deployment](#deployment)

10. [Tips & Best Practices](#tips--best-practices)

## Getting Started

Welcome to SDLC Accelerator! This guide will help you automate your software development lifecycle.

### Dashboard Overview

When you first log in, you'll see the dashboard with:

- \*\*Projects List\*\*: All your projects

- \*\*New Project Button\*\*: Create a new project

- \*\*Search\*\*: Find projects quickly

- \*\*Filters\*\*: Filter by workflow stage

### Navigation

- \*\*Sidebar\*\*: Quick access to all features

- \*\*Breadcrumbs\*\*: Shows where you are

- \*\*Workflow Stepper\*\*: See current stage

## Creating Your First Project

1. Click \*\*"New Project"\*\* button

2. Fill in the form:

- \*\*Name\*\*: Give your project a descriptive name (required)

- \*\*Description\*\*: Brief description of the project (optional)

3. Click \*\*"Create Project"\*\*

You'll be redirected to the project workflow page.

## Requirements Analysis

### Uploading Documents

1. Click \*\*"Upload Requirements Document"\*\*

2. Choose your file:

- Supported formats: PDF, DOCX, TXT

- Maximum size: 50MB

3. Drag-and-drop or click to browse

4. Click \*\*"Upload"\*\*

### Processing

The system will:

- Extract text from your document

- Identify requirements

- Categorize them (functional, non-functional, etc.)

- Detect language and translate if needed

- Extract tables and images

This typically takes 10-30 seconds.

### Reviewing Results

Once processing completes:

1. Review extracted requirements

2. Check for any missing items

3. Answer clarification questions

4. Click \*\*"Approve"\*\* or \*\*"Request Changes"\*\*

\*\*Tips\*\*:

- Review carefully - this forms the foundation

- Answer all clarification questions

- Flag any incorrectly extracted items

## Working with PRDs

### PRD Generation

After approving requirements:

1. Click \*\*"Generate PRD"\*\*

2. Wait for generation (30-60 seconds)

3. Review the generated document

### What's in the PRD?

- Executive Summary

- Product Overview

- User Stories

- Functional Requirements

- Non-Functional Requirements

- Technical Specifications

- Success Metrics

- Risk Analysis

### Editing and Feedback

You can:

- Request specific changes

- Add missing sections

- Clarify ambiguous items

- Adjust priorities

### Approval

Once satisfied, click \*\*"Approve PRD"\*\* to proceed.

## Design Review

### Generated Design Artifacts

The system generates:

1. \*\*Architecture Diagrams\*\*: System components and interactions

2. \*\*Database Schema\*\*: Tables, relationships, indexes

3. \*\*API Specifications\*\*: Complete OpenAPI spec

4. \*\*UI/UX Designs\*\*: Wireframes and layouts

### Reviewing Designs

For each artifact:

1. Check it addresses requirements

2. Verify technical decisions make sense

3. Ensure scalability is considered

4. Confirm security measures are adequate

### Providing Feedback

Click on any design element to:

- Add comments

- Request changes

- Ask questions

- Suggest alternatives

## Jira Integration

### Connecting Jira

If not already connected:

1. Go to Settings

2. Click "Connect Jira"

3. Enter your Jira URL and API token

4. Test connection

### Generated Structure

The system creates:

- \*\*Epics\*\*: Major features

- \*\*Stories\*\*: User-facing functionality

- \*\*Tasks\*\*: Implementation work

- \*\*Subtasks\*\*: Specific details

### Customizing Tickets

Before creation, you can:

- Adjust story points

- Change priorities

- Modify descriptions

- Add custom fields

### Creating in Jira

1. Review the structure

2. Make any adjustments

3. Click \*\*"Create in Jira"\*\*

4. Wait for creation (may take 1-2 minutes)

## Code Generation

### Generated Code

The system generates:

- Complete backend (FastAPI)

- Complete frontend (React)

- Database migrations

- Tests

- Configuration files

### Preview Pane

The preview pane shows:

- \*\*File Tree\*\*: All generated files

- \*\*Code Editor\*\*: View and edit code

- \*\*Preview\*\*: See the running application

### Testing Generated Code

1. Click \*\*"Run Preview"\*\*

2. Interact with the application

3. Test all features

4. Check for any issues

### Requesting Changes

If you find issues:

1. Describe the problem

2. Specify desired behavior

3. Click \*\*"Request Changes"\*\*

The AI will regenerate the affected code.

## Testing

### Test Execution

Tests run automatically:

- Unit tests

- Integration tests

- E2E tests

### Viewing Results

See:

- Total tests run

- Pass/fail status

- Code coverage percentage

- Failed test details

### Failed Tests

If tests fail:

1. Review error messages

2. Check what's expected vs. actual

3. Request fixes from AI

4. Re-run tests

## Deployment

### Deployment Options

Choose your deployment:

- Docker containers

- AWS (ECS/Fargate)

- Manual deployment

### Before Deploying

Ensure:

- All tests passing

- Configuration complete

- Environment variables set

- Database migrations ready

### Deployment Process

1. Click \*\*"Deploy"\*\*

2. Select environment (staging/production)

3. Confirm deployment

4. Monitor progress

5. Verify deployment successful

## Tips & Best Practices

### Requirements Documents

✅ \*\*DO\*\*:

- Provide clear, detailed requirements

- Include examples where possible

- Specify acceptance criteria

- Include non-functional requirements

❌ \*\*DON'T\*\*:

- Use vague language ("fast", "user-friendly")

- Omit important details

- Mix multiple requirements together

- Forget about error scenarios

### PRD Review

✅ \*\*DO\*\*:

- Review thoroughly before approving

- Check all requirements are included

- Verify priorities make sense

- Ensure success metrics are measurable

❌ \*\*DON'T\*\*:

- Rush through approval

- Skip reading non-functional requirements

- Ignore warnings from validator

- Approve contradictory requirements

### Design Review

✅ \*\*DO\*\*:

- Think about scalability

- Consider security implications

- Verify database design is normalized

- Check API design is RESTful

❌ \*\*DON'T\*\*:

- Ignore architecture decisions

- Skip reviewing technical specs

- Overlook security architecture

- Accept designs you don't understand

### Code Review

✅ \*\*DO\*\*:

- Test all generated features

- Check error handling

- Verify input validation

- Review test coverage

❌ \*\*DON'T\*\*:

- Assume generated code is perfect

- Skip testing edge cases

- Ignore security warnings

- Deploy without reviewing

## Troubleshooting

### Document Upload Fails

\*\*Problem\*\*: File upload errors

\*\*Solutions\*\*:

- Check file size (< 50MB)

- Verify file format (PDF, DOCX, TXT)

- Try a different browser

- Check internet connection

### Processing Takes Too Long

\*\*Problem\*\*: Requirements processing timeout

\*\*Solutions\*\*:

- Large documents take longer

- Check document isn't corrupted

- Simplify complex formatting

- Split very large documents

### Jira Creation Fails

\*\*Problem\*\*: Can't create tickets in Jira

\*\*Solutions\*\*:

- Verify Jira credentials

- Check API token permissions

- Ensure project key is valid

- Try reconnecting Jira

### Code Preview Won't Run

\*\*Problem\*\*: Preview pane shows errors

\*\*Solutions\*\*:

- Check browser console for errors

- Clear browser cache

- Verify all dependencies loaded

- Try refreshing page

## Getting Help

- \*\*Email\*\*: support@sdlc-accelerator.com

- \*\*Documentation\*\*: https://docs.sdlc-accelerator.com

- \*\*Community\*\*: https://community.sdlc-accelerator.com

- \*\*GitHub Issues\*\*: https://github.com/your-org/sdlc-accelerator/issues

```

### 4. Deployment Guide

```markdown

# Deployment Guide

## AWS Deployment

### Prerequisites

- AWS Account with administrative access

- AWS CLI configured

- Docker installed locally

- Domain name (optional but recommended)

### Step 1: Create AWS Resources

#### 1.1 Create RDS PostgreSQL Instance

\`\`\`bash

aws rds create-db-instance \

--db-instance-identifier sdlc-accelerator-db \

--db-instance-class db.t3.medium \

--engine postgres \

--engine-version 15.4 \

--master-username admin \

--master-user-password [SECURE-PASSWORD] \

--allocated-storage 20 \

--vpc-security-group-ids sg-xxxxxxxxx \

--db-subnet-group-name my-db-subnet-group \

--backup-retention-period 7 \

--preferred-backup-window "03:00-04:00" \

--preferred-maintenance-window "Mon:04:00-Mon:05:00" \

--multi-az

\`\`\`

#### 1.2 Create S3 Bucket

\`\`\`bash

aws s3 mb s3://sdlc-accelerator-files --region us-east-1

# Enable versioning

aws s3api put-bucket-versioning \

--bucket sdlc-accelerator-files \

--versioning-configuration Status=Enabled

# Set lifecycle policy

aws s3api put-bucket-lifecycle-configuration \

--bucket sdlc-accelerator-files \

--lifecycle-configuration file://s3-lifecycle.json

\`\`\`

#### 1.3 Create ECR Repositories

\`\`\`bash

# Backend repository

aws ecr create-repository \

--repository-name sdlc-accelerator-backend \

--region us-east-1

# Frontend repository

aws ecr create-repository \

--repository-name sdlc-accelerator-frontend \

--region us-east-1

\`\`\`

### Step 2: Build and Push Images

#### 2.1 Build Backend

\`\`\`bash

cd backend

# Build image

docker build -t sdlc-accelerator-backend:latest .

# Tag for ECR

docker tag sdlc-accelerator-backend:latest \

[ACCOUNT-ID].dkr.ecr.us-east-1.amazonaws.com/sdlc-accelerator-backend:latest

# Login to ECR

aws ecr get-login-password --region us-east-1 | \

docker login --username AWS --password-stdin \

[ACCOUNT-ID].dkr.ecr.us-east-1.amazonaws.com

# Push to ECR

docker push [ACCOUNT-ID].dkr.ecr.us-east-1.amazonaws.com/sdlc-accelerator-backend:latest

\`\`\`

#### 2.2 Build Frontend

\`\`\`bash

cd frontend

# Build image

docker build -t sdlc-accelerator-frontend:latest .

# Tag and push (same as backend)

docker tag sdlc-accelerator-frontend:latest \

[ACCOUNT-ID].dkr.ecr.us-east-1.amazonaws.com/sdlc-accelerator-frontend:latest

docker push [ACCOUNT-ID].dkr.ecr.us-east-1.amazonaws.com/sdlc-accelerator-frontend:latest

\`\`\`

[Continue with ECS setup, CloudFront configuration, monitoring setup, etc.]

```

## Documentation Generation Process

1. \*\*Analyze Codebase\*\*: Read all source files, understand structure

2. \*\*Extract Information\*\*: APIs, components, services, configurations

3. \*\*Generate README\*\*: Comprehensive project README

4. \*\*Generate API Docs\*\*: Complete OpenAPI documentation

5. \*\*Generate User Guide\*\*: Step-by-step usage instructions

6. \*\*Generate Deployment Guide\*\*: Infrastructure and deployment steps

7. \*\*Add Code Comments\*\*: Inline documentation where missing

8. \*\*Create Architecture Docs\*\*: High-level system documentation

9. \*\*Generate Troubleshooting Guide\*\*: Common issues and solutions

10. \*\*Review and Polish\*\*: Ensure consistency and completeness

11. \*\*Present to User\*\*: Show documentation for review

12. \*\*Commit to Repository\*\*: Add documentation files to Git

## Validation Checklist

- [ ] README complete with all standard sections

- [ ] Installation instructions tested and accurate

- [ ] API documentation generated (OpenAPI/Swagger)

- [ ] All API endpoints documented with examples

- [ ] Code comments comprehensive (functions, classes, complex logic)

- [ ] User guide covers all major features

- [ ] User guide includes screenshots (if applicable)

- [ ] Deployment guide includes all environments

- [ ] Troubleshooting section addresses common issues

- [ ] Architecture diagrams included and current

- [ ] Examples provided for complex features

- [ ] No broken links or references

- [ ] Consistent formatting and style

- [ ] No placeholder text (all sections complete)

## Output Files

1. `README.md` - Main project README

2. `docs/USER\_GUIDE.md` - Complete user guide

3. `docs/API\_DOCUMENTATION.md` - API reference

4. `docs/DEPLOYMENT\_GUIDE.md` - Deployment instructions

5. `docs/ARCHITECTURE.md` - Architecture documentation

6. `docs/TROUBLESHOOTING.md` - Troubleshooting guide

7. `docs/CONTRIBUTING.md` - Contribution guidelines

8. Inline code documentation (docstrings, comments)

## Human Interaction

1. \*\*Outline Review\*\*: Present documentation structure for approval

2. \*\*Draft Review\*\*: Show generated docs for feedback

3. \*\*Iteration\*\*: Make requested changes

4. \*\*Final Approval\*\*: Get explicit approval before committing

5. \*\*Publishing\*\*: Optionally publish to Confluence

## Integration Points

- \*\*Input\*\*: Source code, PRD, Design docs, Architecture

- \*\*Output\*\*: Documentation files → GitLab, Confluence (if enabled)

- \*\*Services\*\*: OpenAPI generator, Mermaid, Confluence API

## Best Practices

1. Write for your audience (technical vs. non-technical)

2. Use clear, concise language

3. Include visual aids (diagrams, screenshots)

4. Provide examples for complex concepts

5. Keep documentation up-to-date

6. Use consistent formatting

7. Make it searchable (good headings, TOC)

8. Test all instructions

9. Include troubleshooting tips

10. Link related documentation

EOF

Now let me create the Orchestrator agent and then move to custom commands and MCP configuration. Continuing...

Retry

VS

ok

Orchestrated setup completion for Claude Code configuration.

Let me continue with the Orchestrator agent, then the custom commands, and finally the MCP configuration to complete the Claude Code setup.

**4.9 Orchestrator Agent**

bash

cat > .claude/sub-agents/orchestrator.md << 'EOF'

# Orchestrator Agent

## Role & Responsibilities

Manage the entire SDLC workflow, coordinate between all sub-agents, enforce human-in-the-loop at each stage, maintain workflow state, handle errors, and provide progress tracking.

## Core Capabilities

- Workflow stage management

- Sub-agent coordination and orchestration

- State persistence and recovery

- Human interaction management

- Progress tracking and reporting

- Error handling and recovery

- Stage validation enforcement

- User feedback integration

## Workflow Stages

```python

from enum import Enum

class WorkflowStage(Enum):

"""All possible workflow stages."""

IDLE = "idle"

REQUIREMENTS\_ANALYSIS = "requirements\_analysis"

PRD\_GENERATION = "prd\_generation"

DESIGN = "design"

JIRA\_SETUP = "jira\_setup"

CODING = "coding"

TESTING = "testing"

VALIDATION = "validation"

DOCUMENTATION = "documentation"

DEPLOYMENT = "deployment"

COMPLETED = "completed"

ERROR = "error"

```

## Workflow State Management

```python

from dataclasses import dataclass, field

from typing import Dict, Any, List, Optional

from datetime import datetime

from uuid import UUID

@dataclass

class WorkflowState:

"""

Complete workflow state for a project.

This state is persisted to the database and allows

resuming workflows across sessions.

"""

project\_id: UUID

current\_stage: WorkflowStage = WorkflowStage.IDLE

stage\_outputs: Dict[str, Any] = field(default\_factory=dict)

approvals: Dict[str, bool] = field(default\_factory=dict)

feedback: Dict[str, str] = field(default\_factory=dict)

history: List[Dict[str, Any]] = field(default\_factory=list)

created\_at: datetime = field(default\_factory=datetime.now)

updated\_at: datetime = field(default\_factory=datetime.now)

error\_count: int = 0

last\_error: Optional[str] = None

def advance\_stage(self, next\_stage: WorkflowStage) -> None:

"""

Advance to the next workflow stage.

Args:

next\_stage: The stage to advance to

"""

self.history.append({

"from\_stage": self.current\_stage.value,

"to\_stage": next\_stage.value,

"timestamp": datetime.now().isoformat(),

"action": "advance"

})

self.current\_stage = next\_stage

self.updated\_at = datetime.now()

def rollback\_stage(self, target\_stage: WorkflowStage) -> None:

"""

Rollback to a previous stage.

Args:

target\_stage: The stage to rollback to

"""

self.history.append({

"from\_stage": self.current\_stage.value,

"to\_stage": target\_stage.value,

"timestamp": datetime.now().isoformat(),

"action": "rollback"

})

self.current\_stage = target\_stage

self.updated\_at = datetime.now()

def store\_output(self, stage: str, output: Any) -> None:

"""

Store output from a completed stage.

Args:

stage: Stage name

output: Output data from the stage

"""

self.stage\_outputs[stage] = output

self.updated\_at = datetime.now()

def record\_approval(self, stage: str, approved: bool, feedback: str = "") -> None:

"""

Record human approval decision for a stage.

Args:

stage: Stage name

approved: Whether stage was approved

feedback: Optional feedback text

"""

self.approvals[stage] = approved

if feedback:

self.feedback[stage] = feedback

self.updated\_at = datetime.now()

def record\_error(self, error\_message: str) -> None:

"""

Record an error occurrence.

Args:

error\_message: Description of the error

"""

self.error\_count += 1

self.last\_error = error\_message

self.history.append({

"stage": self.current\_stage.value,

"timestamp": datetime.now().isoformat(),

"action": "error",

"message": error\_message

})

self.updated\_at = datetime.now()

def get\_progress\_percentage(self) -> float:

"""

Calculate overall workflow progress.

Returns:

Progress percentage (0-100)

"""

all\_stages = list(WorkflowStage)

# Exclude IDLE, ERROR

workflow\_stages = [s for s in all\_stages if s not in [WorkflowStage.IDLE, WorkflowStage.ERROR]]

try:

current\_index = workflow\_stages.index(self.current\_stage)

return (current\_index / len(workflow\_stages)) \* 100

except ValueError:

# Current stage not in workflow stages

if self.current\_stage == WorkflowStage.COMPLETED:

return 100.0

return 0.0

```

## Orchestrator Implementation

```python

from typing import Dict, Any, Optional

import logging

logger = logging.getLogger(\_\_name\_\_)

class WorkflowOrchestrator:

"""

Main orchestrator for SDLC workflow.

Coordinates all sub-agents and manages the workflow lifecycle.

"""

def \_\_init\_\_(self, db\_session):

"""

Initialize orchestrator with dependencies.

Args:

db\_session: Database session for state persistence

"""

self.db = db\_session

self.state: Optional[WorkflowState] = None

# Initialize all sub-agents

self.requirements\_agent = RequirementsAnalystAgent()

self.prd\_agent = PRDGeneratorAgent()

self.design\_agent = DesignArchitectAgent()

self.jira\_agent = JiraManagerAgent()

self.coder\_agent = CoderAgent()

self.qa\_agent = QATestingAgent()

self.validator\_agent = ValidatorAgent()

self.documentation\_agent = DocumentationAgent()

async def start\_workflow(

self,

project\_id: UUID,

initial\_input: Dict[str, Any]

) -> Dict[str, Any]:

"""

Start a new SDLC workflow for a project.

Args:

project\_id: UUID of the project

initial\_input: Initial input data (e.g., uploaded file)

Returns:

Workflow status and next steps

"""

# Initialize workflow state

self.state = WorkflowState(project\_id=project\_id)

await self.\_persist\_state()

logger.info(f"Starting workflow for project {project\_id}")

# Begin with requirements analysis

return await self.\_advance\_to\_stage(

WorkflowStage.REQUIREMENTS\_ANALYSIS,

initial\_input

)

async def resume\_workflow(self, project\_id: UUID) -> Dict[str, Any]:

"""

Resume an existing workflow.

Args:

project\_id: UUID of the project

Returns:

Current workflow status

"""

# Load existing state

self.state = await self.\_load\_state(project\_id)

if not self.state:

raise ValueError(f"No workflow found for project {project\_id}")

logger.info(f"Resuming workflow for project {project\_id} at stage {self.state.current\_stage}")

return self.\_get\_current\_status()

async def execute\_current\_stage(

self,

input\_data: Optional[Dict[str, Any]] = None

) -> Dict[str, Any]:

"""

Execute the current workflow stage.

Args:

input\_data: Optional input data for the stage

Returns:

Stage execution results

"""

if not self.state:

raise ValueError("No active workflow")

stage = self.state.current\_stage

# Route to appropriate stage handler

stage\_handlers = {

WorkflowStage.REQUIREMENTS\_ANALYSIS: self.\_execute\_requirements\_analysis,

WorkflowStage.PRD\_GENERATION: self.\_execute\_prd\_generation,

WorkflowStage.DESIGN: self.\_execute\_design,

WorkflowStage.JIRA\_SETUP: self.\_execute\_jira\_setup,

WorkflowStage.CODING: self.\_execute\_coding,

WorkflowStage.TESTING: self.\_execute\_testing,

WorkflowStage.VALIDATION: self.\_execute\_validation,

WorkflowStage.DOCUMENTATION: self.\_execute\_documentation,

}

handler = stage\_handlers.get(stage)

if not handler:

raise ValueError(f"No handler for stage {stage}")

try:

# Execute stage

result = await handler(input\_data)

return result

except Exception as e:

# Handle error

logger.error(f"Error executing stage {stage}: {str(e)}")

self.state.record\_error(str(e))

self.state.current\_stage = WorkflowStage.ERROR

await self.\_persist\_state()

raise

async def handle\_approval(

self,

approved: bool,

feedback: str = ""

) -> Dict[str, Any]:

"""

Handle human approval decision for current stage.

Args:

approved: Whether the stage output is approved

feedback: Optional feedback text

Returns:

Next steps based on approval decision

"""

if not self.state:

raise ValueError("No active workflow")

stage = self.state.current\_stage

logger.info(f"Received approval for {stage}: {approved}")

# Record approval

self.state.record\_approval(stage.value, approved, feedback)

await self.\_persist\_state()

if approved:

# Advance to next stage

next\_stage = self.\_get\_next\_stage(stage)

if next\_stage:

return await self.\_advance\_to\_stage(next\_stage)

else:

# Workflow complete

self.state.current\_stage = WorkflowStage.COMPLETED

await self.\_persist\_state()

return {

"status": "completed",

"message": "Workflow completed successfully!"

}

else:

# Rejected - retry current stage with feedback

logger.info(f"Stage {stage} rejected. Feedback: {feedback}")

return {

"status": "retry\_needed",

"stage": stage.value,

"feedback": feedback,

"message": "Please provide corrections and retry"

}

async def navigate\_to\_stage(self, target\_stage: WorkflowStage) -> Dict[str, Any]:

"""

Navigate to a specific stage (go back or forward).

Args:

target\_stage: The stage to navigate to

Returns:

Navigation result

"""

if not self.state:

raise ValueError("No active workflow")

current = self.state.current\_stage

logger.info(f"Navigating from {current} to {target\_stage}")

# Warn if going backward

if self.\_stage\_index(target\_stage) < self.\_stage\_index(current):

logger.warning(f"Going backward from {current} to {target\_stage}")

# Update state

self.state.rollback\_stage(target\_stage)

await self.\_persist\_state()

return {

"status": "navigated",

"current\_stage": target\_stage.value,

"message": f"Navigated to {target\_stage.value}"

}

# Private methods

async def \_execute\_requirements\_analysis(

self,

input\_data: Dict[str, Any]

) -> Dict[str, Any]:

"""Execute requirements analysis stage."""

logger.info("Executing requirements analysis")

# Execute requirements analyst agent

result = await self.requirements\_agent.analyze(input\_data)

# Validate result

validation = await self.validator\_agent.validate\_requirements(result)

# Store output

self.state.store\_output("requirements\_analysis", {

"requirements": result,

"validation": validation

})

await self.\_persist\_state()

# Present to user

return {

"status": "awaiting\_approval",

"stage": "requirements\_analysis",

"output": result,

"validation": validation,

"message": "Requirements analysis complete. Please review and approve."

}

async def \_execute\_prd\_generation(

self,

input\_data: Optional[Dict[str, Any]] = None

) -> Dict[str, Any]:

"""Execute PRD generation stage."""

logger.info("Executing PRD generation")

# Get requirements from previous stage

requirements = self.state.stage\_outputs.get("requirements\_analysis", {}).get("requirements")

if not requirements:

raise ValueError("Requirements not found from previous stage")

# Execute PRD generator agent

result = await self.prd\_agent.generate(requirements)

# Validate result

validation = await self.validator\_agent.validate\_prd(result, requirements)

# Store output

self.state.store\_output("prd\_generation", {

"prd": result,

"validation": validation

})

await self.\_persist\_state()

return {

"status": "awaiting\_approval",

"stage": "prd\_generation",

"output": result,

"validation": validation,

"message": "PRD generated. Please review and approve."

}

async def \_execute\_design(

self,

input\_data: Optional[Dict[str, Any]] = None

) -> Dict[str, Any]:

"""Execute design and architecture stage."""

logger.info("Executing design and architecture")

# Get PRD from previous stage

prd = self.state.stage\_outputs.get("prd\_generation", {}).get("prd")

if not prd:

raise ValueError("PRD not found from previous stage")

# Execute design architect agent

result = await self.design\_agent.design(prd)

# Validate result

validation = await self.validator\_agent.validate\_design(result)

# Store output

self.state.store\_output("design", {

"design\_artifacts": result,

"validation": validation

})

await self.\_persist\_state()

return {

"status": "awaiting\_approval",

"stage": "design",

"output": result,

"validation": validation,

"message": "Design and architecture complete. Please review and approve."

}

async def \_execute\_jira\_setup(

self,

input\_data: Optional[Dict[str, Any]] = None

) -> Dict[str, Any]:

"""Execute Jira ticket creation stage."""

logger.info("Executing Jira setup")

# Get PRD and design from previous stages

prd = self.state.stage\_outputs.get("prd\_generation", {}).get("prd")

design = self.state.stage\_outputs.get("design", {}).get("design\_artifacts")

# Execute Jira manager agent

result = await self.jira\_agent.create\_structure(prd, design)

# Validate result

validation = await self.validator\_agent.validate\_jira\_structure(result, prd)

# Store output

self.state.store\_output("jira\_setup", {

"jira\_structure": result,

"validation": validation

})

await self.\_persist\_state()

return {

"status": "awaiting\_approval",

"stage": "jira\_setup",

"output": result,

"validation": validation,

"message": "Jira structure created. Please review before creating tickets."

}

async def \_execute\_coding(

self,

input\_data: Optional[Dict[str, Any]] = None

) -> Dict[str, Any]:

"""Execute code generation stage."""

logger.info("Executing code generation")

# Get all previous outputs

prd = self.state.stage\_outputs.get("prd\_generation", {}).get("prd")

design = self.state.stage\_outputs.get("design", {}).get("design\_artifacts")

jira = self.state.stage\_outputs.get("jira\_setup", {}).get("jira\_structure")

# Execute coder agent

result = await self.coder\_agent.generate\_code(prd, design, jira)

# Validate result

validation = await self.validator\_agent.validate\_code(result, prd, design)

# Store output

self.state.store\_output("coding", {

"code\_artifacts": result,

"validation": validation

})

await self.\_persist\_state()

return {

"status": "awaiting\_approval",

"stage": "coding",

"output": result,

"validation": validation,

"message": "Code generated. Please review in preview pane."

}

async def \_execute\_testing(

self,

input\_data: Optional[Dict[str, Any]] = None

) -> Dict[str, Any]:

"""Execute testing stage."""

logger.info("Executing testing")

# Get code from previous stage

code = self.state.stage\_outputs.get("coding", {}).get("code\_artifacts")

prd = self.state.stage\_outputs.get("prd\_generation", {}).get("prd")

# Execute QA agent

result = await self.qa\_agent.generate\_and\_run\_tests(code, prd)

# Validate result

validation = await self.validator\_agent.validate\_tests(result)

# Store output

self.state.store\_output("testing", {

"test\_results": result,

"validation": validation

})

await self.\_persist\_state()

return {

"status": "awaiting\_approval",

"stage": "testing",

"output": result,

"validation": validation,

"message": "Testing complete. Please review test results."

}

async def \_execute\_validation(

self,

input\_data: Optional[Dict[str, Any]] = None

) -> Dict[str, Any]:

"""Execute final validation stage."""

logger.info("Executing final validation")

# Get all outputs

all\_outputs = self.state.stage\_outputs

# Execute comprehensive validation

result = await self.validator\_agent.validate\_complete\_workflow(all\_outputs)

# Store output

self.state.store\_output("validation", {

"final\_validation": result

})

await self.\_persist\_state()

return {

"status": "awaiting\_approval",

"stage": "validation",

"output": result,

"message": "Final validation complete. Please review."

}

async def \_execute\_documentation(

self,

input\_data: Optional[Dict[str, Any]] = None

) -> Dict[str, Any]:

"""Execute documentation generation stage."""

logger.info("Executing documentation generation")

# Get all outputs

code = self.state.stage\_outputs.get("coding", {}).get("code\_artifacts")

prd = self.state.stage\_outputs.get("prd\_generation", {}).get("prd")

design = self.state.stage\_outputs.get("design", {}).get("design\_artifacts")

# Execute documentation agent

result = await self.documentation\_agent.generate\_all\_docs(code, prd, design)

# Validate result

validation = await self.validator\_agent.validate\_documentation(result)

# Store output

self.state.store\_output("documentation", {

"documentation": result,

"validation": validation

})

await self.\_persist\_state()

return {

"status": "awaiting\_approval",

"stage": "documentation",

"output": result,

"validation": validation,

"message": "Documentation generated. Please review."

}

async def \_advance\_to\_stage(

self,

next\_stage: WorkflowStage,

input\_data: Optional[Dict[str, Any]] = None

) -> Dict[str, Any]:

"""Advance workflow to next stage."""

self.state.advance\_stage(next\_stage)

await self.\_persist\_state()

# Execute the new stage

return await self.execute\_current\_stage(input\_data)

def \_get\_next\_stage(self, current: WorkflowStage) -> Optional[WorkflowStage]:

"""Get the next stage in the workflow."""

stage\_order = [

WorkflowStage.IDLE,

WorkflowStage.REQUIREMENTS\_ANALYSIS,

WorkflowStage.PRD\_GENERATION,

WorkflowStage.DESIGN,

WorkflowStage.JIRA\_SETUP,

WorkflowStage.CODING,

WorkflowStage.TESTING,

WorkflowStage.VALIDATION,

WorkflowStage.DOCUMENTATION,

WorkflowStage.DEPLOYMENT,

WorkflowStage.COMPLETED

]

try:

current\_index = stage\_order.index(current)

if current\_index < len(stage\_order) - 1:

return stage\_order[current\_index + 1]

except ValueError:

pass

return None

def \_stage\_index(self, stage: WorkflowStage) -> int:

"""Get index of stage in workflow order."""

stage\_order = [

WorkflowStage.IDLE,

WorkflowStage.REQUIREMENTS\_ANALYSIS,

WorkflowStage.PRD\_GENERATION,

WorkflowStage.DESIGN,

WorkflowStage.JIRA\_SETUP,

WorkflowStage.CODING,

WorkflowStage.TESTING,

WorkflowStage.VALIDATION,

WorkflowStage.DOCUMENTATION,

WorkflowStage.DEPLOYMENT,

WorkflowStage.COMPLETED

]

try:

return stage\_order.index(stage)

except ValueError:

return -1

def \_get\_current\_status(self) -> Dict[str, Any]:

"""Get current workflow status."""

if not self.state:

return {"status": "no\_workflow"}

return {

"status": "active",

"current\_stage": self.state.current\_stage.value,

"progress": self.state.get\_progress\_percentage(),

"completed\_stages": [

stage for stage, approved in self.state.approvals.items()

if approved

],

"last\_updated": self.state.updated\_at.isoformat()

}

async def \_persist\_state(self) -> None:

"""Persist workflow state to database."""

# Save to database

# Implementation depends on your database setup

pass

async def \_load\_state(self, project\_id: UUID) -> Optional[WorkflowState]:

"""Load workflow state from database."""

# Load from database

# Implementation depends on your database setup

pass

```

## Error Handling

```python

class WorkflowError(Exception):

"""Base exception for workflow errors."""

pass

class StageExecutionError(WorkflowError):

"""Error during stage execution."""

pass

class ValidationError(WorkflowError):

"""Error during validation."""

pass

class StateError(WorkflowError):

"""Error with workflow state."""

pass

async def handle\_workflow\_error(

error: Exception,

orchestrator: WorkflowOrchestrator

) -> Dict[str, Any]:

"""

Handle workflow errors with recovery options.

Args:

error: The exception that occurred

orchestrator: The orchestrator instance

Returns:

Error handling result with recovery options

"""

logger.error(f"Workflow error: {str(error)}", exc\_info=True)

# Record error in state

if orchestrator.state:

orchestrator.state.record\_error(str(error))

await orchestrator.\_persist\_state()

# Determine recovery options

recovery\_options = [

{

"action": "retry",

"label": "Retry Current Stage",

"description": "Attempt to execute the current stage again"

},

{

"action": "rollback",

"label": "Go Back to Previous Stage",

"description": "Return to the previous stage and try again"

},

{

"action": "skip",

"label": "Skip This Stage",

"description": "Skip to next stage (may cause issues)"

},

{

"action": "abort",

"label": "Abort Workflow",

"description": "Stop the workflow (can resume later)"

}

]

return {

"status": "error",

"error\_message": str(error),

"error\_type": type(error).\_\_name\_\_,

"current\_stage": orchestrator.state.current\_stage.value if orchestrator.state else None,

"recovery\_options": recovery\_options

}

```

## Progress Tracking

```python

def get\_detailed\_progress(state: WorkflowState) -> Dict[str, Any]:

"""

Get detailed progress information.

Args:

state: Current workflow state

Returns:

Detailed progress report

"""

all\_stages = [

WorkflowStage.REQUIREMENTS\_ANALYSIS,

WorkflowStage.PRD\_GENERATION,

WorkflowStage.DESIGN,

WorkflowStage.JIRA\_SETUP,

WorkflowStage.CODING,

WorkflowStage.TESTING,

WorkflowStage.VALIDATION,

WorkflowStage.DOCUMENTATION

]

stage\_status = {}

for stage in all\_stages:

stage\_name = stage.value

if stage\_name in state.approvals:

status = "completed" if state.approvals[stage\_name] else "rejected"

elif stage == state.current\_stage:

status = "in\_progress"

elif state.\_stage\_index(stage) < state.\_stage\_index(state.current\_stage):

status = "completed"

else:

status = "pending"

stage\_status[stage\_name] = {

"status": status,

"has\_output": stage\_name in state.stage\_outputs,

"has\_feedback": stage\_name in state.feedback

}

return {

"overall\_progress": state.get\_progress\_percentage(),

"current\_stage": state.current\_stage.value,

"stages": stage\_status,

"total\_stages": len(all\_stages),

"completed\_stages": len([s for s, info in stage\_status.items() if info["status"] == "completed"]),

"error\_count": state.error\_count,

"last\_error": state.last\_error

}

```

## Validation Checklist

- [ ] All stages properly defined

- [ ] State persistence working correctly

- [ ] Error handling comprehensive

- [ ] Recovery options available

- [ ] Progress tracking accurate

- [ ] Human approval enforced at each stage

- [ ] User can navigate to any previous stage

- [ ] Workflow can be resumed after interruption

- [ ] All sub-agents properly integrated

- [ ] Validation enforced before stage advancement

## Human Interaction

This agent IS the main interface for human interaction:

- Presents outputs from all sub-agents

- Collects approvals and feedback

- Allows workflow navigation

- Provides progress updates

- Handles error recovery decisions

- Enables stage re-execution

## Integration Points

- \*\*Coordinates\*\*: All 8 sub-agents

- \*\*Manages\*\*: Complete workflow lifecycle

- \*\*Persists\*\*: Workflow state to database

- \*\*Output\*\*: Orchestrated SDLC workflow results

## Best Practices

1. Always persist state after each action

2. Provide clear status updates to user

3. Handle errors gracefully with recovery options

4. Allow flexible navigation through stages

5. Enforce validation at each stage

6. Track all decisions and changes in history

7. Provide detailed progress information

8. Enable workflow resumption

9. Log all important events

10. Make rollback safe and predictable

EOF

**STEP 5: CREATE CUSTOM COMMANDS**

**5.1 Create /start-project Command**

bash

cat > .claude/commands/start-project.md << 'EOF'

# Start Project Command

Initialize a new SDLC Accelerator project and begin the workflow.

## Execution Steps

1. \*\*Verify Setup\*\*

- Check that we're in the sdlc-accelerator directory

- Verify all sub-agent files exist

- Confirm CLAUDE.md is configured

2. \*\*Create Project Structure\*\* (if first time)

- Ensure backend/ and frontend/ directories exist

- Create necessary subdirectories

- Initialize git if not already initialized

3. \*\*Prompt User for Project Details\*\*

- Ask for project name

- Ask for project description (optional)

- Confirm details before proceeding

4. \*\*Initialize Project\*\*

- Create project entry in planning

- Initialize workflow state as IDLE

- Set up project-specific documentation folder

5. \*\*Guide User to Upload Requirements\*\*

- Explain the next step (requirements document upload)

- Show supported file formats (PDF, DOCX, TXT)

- Explain what will happen next

6. \*\*Set Context\*\*

- Load Requirements Analyst sub-agent context

- Prepare for document processing

- Ready to begin workflow

## Usage

Simply type `/start-project` in Claude Code to begin.

## Example Interaction

```

User: /start-project

Claude: Let's start a new SDLC Accelerator project!

Project Name: [User enters name]

Description (optional): [User enters description]

✅ Project "My New App" initialized!

Next Steps:

1. Prepare your requirements document (PDF, DOCX, or TXT)

2. Use /upload-requirements to begin analysis

3. I'll extract and structure all requirements automatically

Ready to upload requirements? Type /upload-requirements when ready.

```

EOF

**5.2 Create /review-stage Command**

bash

cat > .claude/commands/review-stage.md << 'EOF'

# Review Stage Command

Present current workflow stage output for human review and approval.

## Execution Steps

1. \*\*Identify Current Stage\*\*

- Check workflow state

- Determine which stage we're in

- Load relevant sub-agent context

2. \*\*Gather Stage Output\*\*

- Retrieve output from current stage

- Load validation results

- Prepare summary statistics

3. \*\*Present Output\*\*

- Format output for readability

- Highlight key points

- Show validation report

- Display any warnings or issues

4. \*\*Run Validation\*\*

- Execute validator agent checks

- Generate validation report

- Calculate quality score

5. \*\*Request Approval\*\*

- Ask: Approve, Reject, or Request Changes?

- If rejected: Collect specific feedback

- If approved: Prepare to advance to next stage

6. \*\*Handle Decision\*\*

- If approved: Advance to next stage

- If rejected: Retry with feedback incorporated

- If changes requested: Make adjustments and re-present

## Usage

Type `/review-stage` to review and approve current stage.

## Example Interaction

```

User: /review-stage

Claude:

# Requirements Analysis - Review

## Summary

✅ Processed: requirements\_v1.pdf (45 pages)

✅ Extracted: 87 requirements

- Functional: 52

- Non-Functional: 20

- Business Rules: 15

## Validation Report

\*\*Score\*\*: 88/100 ✅ PASSED

⚠️ Warnings:

- 5 NFRs missing measurable criteria

- 12 clarifications needed

## Next Steps

Do you approve these requirements to proceed to PRD generation?

1. ✅ Approve - Continue to PRD Generation

2. ❌ Reject - Provide feedback and retry

3. 🔧 Request Changes - Specify what to modify

Your choice:

```

EOF

**5.3 Create /navigate-workflow Command**

bash

cat > .claude/commands/navigate-workflow.md << 'EOF'

# Navigate Workflow Command

Allow user to jump to any stage in the workflow.

## Execution Steps

1. \*\*Display Workflow Status\*\*

- Show all stages with their status

- Highlight current stage

- Mark completed stages

- Show pending stages

2. \*\*Show Navigation Options\*\*

- List all accessible stages

- Indicate which stages can be navigated to

- Warn about going backward

3. \*\*Handle Selection\*\*

- Validate selected stage is valid

- Warn if going backward (potential data loss)

- Confirm navigation decision

4. \*\*Navigate\*\*

- Update workflow state

- Load selected stage context

- Present stage information

5. \*\*Resume Stage\*\*

- If stage incomplete: Resume execution

- If stage complete: Allow re-execution or continue

## Usage

Type `/navigate-workflow` to jump to any workflow stage.

## Example Interaction

```

User: /navigate-workflow

Claude:

# Workflow Navigation

\*\*Current Project\*\*: My Application

\*\*Current Stage\*\*: JIRA\_SETUP (Stage 4 of 8)

\*\*Progress\*\*: 50%

## Workflow Stages

1. ✅ Requirements Analysis [COMPLETED]

2. ✅ PRD Generation [COMPLETED]

3. ✅ Design & Architecture [COMPLETED]

4. 🔵 Jira Setup [CURRENT]

5. ⏳ Code Generation [PENDING]

6. ⏳ Testing [PENDING]

7. ⏳ Validation [PENDING]

8. ⏳ Documentation [PENDING]

Where would you like to go?

Enter stage number or name:

```

EOF

**5.4 Create /generate-report Command**

bash

cat > .claude/commands/generate-report.md << 'EOF'

# Generate Report Command

Create comprehensive project status report.

## Report Sections

1. \*\*Project Overview\*\*

- Project name and description

- Creation date

- Current status

2. \*\*Workflow Progress\*\*

- Current stage

- Progress percentage

- Completed stages with timestamps

- Pending stages

3. \*\*Stage Summaries\*\*

- For each completed stage:

- Output summary

- Validation results

- Human feedback (if any)

- Key metrics

4. \*\*Quality Metrics\*\*

- Validation scores per stage

- Overall quality score

- Test coverage (if testing complete)

- Code quality metrics (if coding complete)

5. \*\*Issues & Warnings\*\*

- Critical issues (if any)

- Warnings across all stages

- Recommendations

6. \*\*Next Steps\*\*

- What needs to be done next

- Estimated time remaining

- Blockers (if any)

7. \*\*Artifacts Generated\*\*

- List of all generated artifacts

- Links to documents

- Code repository information

## Usage

Type `/generate-report` to create status report.

## Output Format

Report is generated in Markdown format and can be:

- Saved to file

- Copied to clipboard

- Exported to PDF

- Published to Confluence

## Example Output

```markdown

# SDLC Accelerator Project Report

\*\*Project\*\*: Customer Portal Application

\*\*Generated\*\*: 2025-01-15 14:30:00

\*\*Status\*\*: In Progress

## Executive Summary

Project is 50% complete with 4 of 8 stages finished.

All stages passed validation with scores >85%.

## Progress Overview

- \*\*Current Stage\*\*: Jira Setup

- \*\*Progress\*\*: 50% (4/8 stages)

- \*\*Started\*\*: 2025-01-15 09:00:00

- \*\*Time Elapsed\*\*: 5.5 hours

## Completed Stages

### 1. Requirements Analysis ✅

- \*\*Completed\*\*: 2025-01-15 10:30:00

- \*\*Validation Score\*\*: 88/100

- \*\*Requirements Extracted\*\*: 87

- \*\*Key Outputs\*\*: requirements\_structured.json

...

```

EOF

**5.5 Create /help Command**

bash

cat > .claude/commands/help.md << 'EOF'

# Help Command

Display help information about Claude Code setup and available commands.

## Topics

1. \*\*Getting Started\*\*

- How to use Claude Code for SDLC Accelerator

- Project initialization

- Workflow overview

2. \*\*Available Commands\*\*

- List all custom commands

- Explain each command's purpose

- Show usage examples

3. \*\*Sub-Agents\*\*

- List all sub-agents

- Explain what each agent does

- Show which agent handles which stage

4. \*\*Workflow Stages\*\*

- Explain each stage

- Show typical duration

- List outputs for each stage

5. \*\*Troubleshooting\*\*

- Common issues and solutions

- How to report bugs

- Where to get more help

## Usage

Type `/help` or `/help [topic]` for information.

## Example

```

User: /help

Claude:

# SDLC Accelerator - Claude Code Help

## Available Commands

- `/start-project` - Initialize new project

- `/review-stage` - Review and approve current stage

- `/navigate-workflow` - Jump to any workflow stage

- `/generate-report` - Create project status report

- `/help` - Show this help

## Sub-Agents

1. Requirements Analyst - Extracts and structures requirements

2. PRD Generator - Creates Product Requirements Documents

3. Design Architect - Generates system designs

4. Jira Manager - Creates Jira ticket structure

5. Coder - Generates production code

6. QA Tester - Creates and runs tests

7. Validator - Validates outputs at each stage

8. Documentation - Generates documentation

9. Orchestrator - Manages the workflow

## Need More Help?

- Type `/help [command-name]` for command details

- Check `.claude/sub-agents/` for agent documentation

- Review `.claude/CLAUDE.md` for project overview

What would you like help with?

```

EOF

**STEP 6: CREATE MCP CONFIGURATION**

**6.1 Create .mcp.json**

bash

cat > .mcp.json << 'EOF'

{

"mcpServers": {

"jira": {

"command": "npx",

"args": ["-y", "@modelcontextprotocol/server-jira"],

"env": {

"JIRA\_URL": "${JIRA\_URL}",

"JIRA\_EMAIL": "${JIRA\_EMAIL}",

"JIRA\_API\_TOKEN": "${JIRA\_API\_TOKEN}"

}

},

"confluence": {

"command": "npx",

"args": ["-y", "@modelcontextprotocol/server-confluence"],

"env": {

"CONFLUENCE\_URL": "${CONFLUENCE\_URL}",

"CONFLUENCE\_EMAIL": "${CONFLUENCE\_EMAIL}",

"CONFLUENCE\_API\_TOKEN": "${CONFLUENCE\_API\_TOKEN}"

}

},

"gitlab": {

"command": "npx",

"args": ["-y", "@modelcontextprotocol/server-gitlab"],

"env": {

"GITLAB\_URL": "https://gitlab.com",

"GITLAB\_TOKEN": "${GITLAB\_TOKEN}"

}

},

"filesystem": {

"command": "npx",

"args": ["-y", "@modelcontextprotocol/server-filesystem"],

"args\_extra": ["${PWD}"]

}

}

}

EOF

**6.2 Create .env.example for MCP**

bash

cat > .env.mcp.example << 'EOF'

# MCP Server Configuration

# Copy this to .env.mcp and fill in your actual values

# Jira Configuration

JIRA\_URL=https://your-domain.atlassian.net

JIRA\_EMAIL=your-email@example.com

JIRA\_API\_TOKEN=your\_jira\_api\_token

# Confluence Configuration

CONFLUENCE\_URL=https://your-domain.atlassian.net/wiki

CONFLUENCE\_EMAIL=your-email@example.com

CONFLUENCE\_API\_TOKEN=your\_confluence\_api\_token

# GitLab Configuration

GITLAB\_URL=https://gitlab.com

GITLAB\_TOKEN=your\_gitlab\_personal\_access\_token

# AWS Configuration (for Bedrock and Textract)

AWS\_ACCESS\_KEY\_ID=your\_aws\_access\_key

AWS\_SECRET\_ACCESS\_KEY=your\_aws\_secret\_key

AWS\_REGION=us-east-1

EOF

**COMPLETE SETUP VERIFICATION CHECKLIST**

bash

cat > SETUP\_CHECKLIST.md << 'EOF'

# Claude Code Setup Verification Checklist

## ✅ Phase 1: Project Structure

- [ ] Created `sdlc-accelerator/` root directory

- [ ] Initialized git repository

- [ ] Created `.claude/` directory

- [ ] Created `.claude/sub-agents/` directory

- [ ] Created `.claude/commands/` directory

- [ ] Created `.claude/workflows/` directory

- [ ] Created `docs/`, `config/`, `backend/`, `frontend/` directories

- [ ] Created `.gitignore` file

## ✅ Phase 2: Main Configuration

- [ ] Created `.claude/CLAUDE.md` with project overview

- [ ] Configuration includes tech stack

- [ ] Configuration includes architecture principles

- [ ] Configuration includes code standards

- [ ] Configuration includes sub-agent system overview

## ✅ Phase 3: Sub-Agent Files (9 total)

- [ ] Created `requirements-analyst.md` with detailed prompts

- [ ] Created `prd-generator.md` with detailed prompts

- [ ] Created `design-architect.md` with detailed prompts

- [ ] Created `jira-manager.md` with detailed prompts

- [ ] Created `coder-agent.md` with detailed prompts

- [ ] Created `qa-tester.md` with detailed prompts

- [ ] Created `validator.md` with detailed prompts

- [ ] Created `documentation-agent.md` with detailed prompts

- [ ] Created `orchestrator.md` with detailed prompts

## ✅ Phase 4: MCP Configuration

- [ ] Created `.mcp.json` with server configurations

- [ ] Created `.env.mcp.example` template

- [ ] Configured Jira MCP server (if using Jira)

- [ ] Configured Confluence MCP server (if using Confluence)

- [ ] Configured GitLab MCP server

- [ ] Configured filesystem MCP server

## ✅ Phase 5: Custom Commands (5 total)

- [ ] Created `/start-project` command

- [ ] Created `/review-stage` command

- [ ] Created `/navigate-workflow` command

- [ ] Created `/generate-report` command

- [ ] Created `/help` command

## ✅ Phase 6: Verification

- [ ] All files created successfully

- [ ] No syntax errors in markdown files

- [ ] All file paths correct

- [ ] Claude Code installed globally (`claude --version`)

- [ ] Can start Claude Code in project directory (`claude`)

- [ ] Claude recognizes CLAUDE.md file

- [ ] Custom commands appear in Claude Code

## ✅ Phase 7: Test Basic Functionality

- [ ] Type `/help` - command works

- [ ] Check sub-agents are recognized

- [ ] Verify MCP servers can connect (if configured)

- [ ] Test /start-project flow

## 📝 Notes

- Do NOT start building the SDLC Accelerator application yet

- This setup prepares Claude Code infrastructure only

- After verification, we'll begin application development

- Keep this checklist for reference

## ❓ Having Issues?

1. Verify you're in the correct directory

2. Check all file paths match exactly

3. Ensure no typos in filenames

4. Restart Claude Code if needed

5. Check Claude Code documentation: https://docs.claude.com/en/docs/claude-code

EOF

```

---

*## \*\*FINAL SETUP SUMMARY\*\**

You now have a \*\*complete Claude Code setup\*\* with:

*### ✅ \*\*9 Sub-Agents with Detailed Prompts\*\*:*

1. Requirements Analyst

2. PRD Generator

3. Design Architect

4. Jira Manager

5. Coder Agent

6. QA Tester

7. Validator

8. Documentation Agent

9. Orchestrator

*### ✅ \*\*5 Custom Commands\*\*:*

1. `/start-project`

2. `/review-stage`

3. `/navigate-workflow`

4. `/generate-report`

5. `/help`

*### ✅ \*\*MCP Integrations\*\* (configured):*

- Jira

- Confluence

- GitLab

- Filesystem

*### ✅ \*\*Complete Project Structure\*\*:*

```

sdlc-accelerator/

├── .claude/

│ ├── CLAUDE.md *# Main config*

│ ├── sub-agents/ *# 9 agent files*

│ │ ├── requirements-analyst.md

│ │ ├── prd-generator.md

│ │ ├── design-architect.md

│ │ ├── jira-manager.md

│ │ ├── coder-agent.md

│ │ ├── qa-tester.md

│ │ ├── validator.md

│ │ ├── documentation-agent.md

│ │ └── orchestrator.md

│ └── commands/ *# 5 command files*

│ ├── start-project.md

│ ├── review-stage.md

│ ├── navigate-workflow.md

│ ├── generate-report.md

│ └── help.md

├── .mcp.json *# MCP server config*

├── .env.mcp.example *# MCP env template*

├── .gitignore

├── SETUP\_CHECKLIST.md

├── docs/

├── config/

├── backend/

└── frontend/

**NEXT STEPS**

1. **Verify your setup** using the checklist
2. **Test Claude Code** by running claude in your project directory
3. **Try /help** command to verify everything works
4. Once verified, **we'll start building the SDLC Accelerator application** using these sub-agents

**Ready to verify your setup?** Let me know once you've created all these files, and we'll test that everything works before starting the actual application build!