Comprehensive agents.md Templates Collection

A curated collection of production-ready agents.md templates for different project types and frameworks. Each template is designed to optimize AI agent performance and ensure consistent code generation.

1. Basic agents.md Template

```markdown
# Project Overview

Brief description of your project and its main purpose.

## Project Structure

```
project-root/
├── src/          # Source code
├── tests/        # Test files
├── docs/         # Documentation
├── public/       # Static assets
└── config/       # Configuration files
```

Coding Standards

General Guidelines
```

- Use meaningful variable and function names
- Follow consistent naming conventions
- Add comments for complex logic
- Maintain consistent indentation

File Organization
- Group related functionality together
- Use clear, descriptive file names
- Organize imports alphabetically

Testing Requirements

- Write unit tests for all new functions
- Maintain test coverage above 80%
- Use descriptive test names
- Include edge case testing

Development Workflow

Pull Request Guidelines
- Use clear, descriptive commit messages
- Include tests for new features
- Update documentation when necessary
- Follow the existing code style

Code Review Process
- All code must be reviewed before merging

- Address all feedback before approval

- Ensure tests pass before merging

```

## 2. React TypeScript Template

```markdown

# React TypeScript Project

A modern React application built with TypeScript, focusing on component reusability and type safety.

## Project Structure

```

react-app/

├── public/

│   ├── favicon.ico

│   └── index.html

├── src/

│   ├── components/   # Reusable components

│   │   ├── common/   # Common UI components

│   │   └── ui/   # Specific UI components

│   ├── pages/   # Page components

│   ├── hooks/   # Custom React hooks

│   ├── store/   # State management

│   ├── services/   # API services
```

```
│   ├── utils/          # Utility functions
│   ├── types/          # TypeScript type definitions
│   ├── styles/         # Global styles
│   ├── constants/      # Application constants
│   ├── App.tsx
│   └── index.tsx
├── tests/              # Test files
├── docs/               # Documentation
└── README.md
```

Coding Standards

TypeScript Guidelines

- Use strict TypeScript configuration
- Define interfaces for all props and state
- Use union types for component variants
- Avoid `any` type - use proper typing

React Component Standards

- Use functional components with hooks
- Implement proper prop validation with TypeScript
- Use React.memo for performance optimization when needed
- Follow component naming convention: PascalCase

State Management

- Use React hooks for local state

- Implement Context API for global state
- Use reducers for complex state logic
- Keep state as minimal as possible

Styling Guidelines
- Use CSS modules or styled-components
- Follow BEM methodology for class names
- Use CSS variables for consistent theming
- Implement responsive design patterns

Testing Requirements

Unit Testing
- Use Jest and React Testing Library
- Test component behavior, not implementation
- Mock external dependencies
- Aim for 90%+ test coverage

Integration Testing
- Test component interactions
- Test API integration points
- Test routing and navigation
- Test state management flows

Testing Patterns
```typescript
// Example test structure
```

```
describe('ComponentName', () => {
  beforeEach(() => {
    // Setup code
  });

  it('should render correctly', () => {
    // Test implementation
  });

  it('should handle user interactions', () => {
    // Test user interactions
  });
});
```

Development Workflow

Component Development

1. Create component interface/types first
2. Implement component logic
3. Add styles
4. Write comprehensive tests
5. Update documentation

API Integration

- Use custom hooks for API calls
- Implement proper error handling

- Use loading states consistently
- Cache API responses when appropriate

Performance Optimization
- Use React.memo for expensive components
- Implement code splitting with lazy loading
- Optimize bundle size
- Use proper key props in lists

Git Workflow
- Use feature branches
- Write descriptive commit messages
- Squash commits before merging
- Use conventional commit format

Environment Setup

Development
```bash
npm install
npm run dev
```

Testing
```bash
npm test
npm run test:coverage
```

```

### Build

```bash
npm run build
npm run preview
```

Code Quality Tools

- ESLint for linting
- Prettier for code formatting
- Husky for pre-commit hooks
- TypeScript for type checking
```

3. Node.js Backend API Template

```markdown
# Node.js Backend API

A scalable REST API built with Node.js, Express, and TypeScript, following clean architecture principles.

## Project Structure

```

```
backend-api/
├── src/
│   ├── controllers/    # Request handlers
│   ├── services/       # Business logic
│   ├── repositories/   # Data access layer
│   ├── models/         # Data models
│   ├── middleware/     # Custom middleware
│   ├── routes/         # API routes
│   ├── utils/          # Utility functions
│   ├── types/          # TypeScript types
│   ├── config/         # Configuration
│   └── app.ts          # Application entry point
├── tests/              # Test files
├── docs/               # API documentation
├── migrations/         # Database migrations
└── seeds/              # Database seeds
```

Coding Standards

API Design

- Use RESTful conventions
- Implement proper HTTP status codes
- Use consistent response formats
- Include API versioning

Error Handling

- Use centralized error handling middleware
- Implement custom error classes
- Log errors appropriately
- Return user-friendly error messages

Database Patterns
- Use repository pattern for data access
- Implement proper database transactions
- Use migrations for schema changes
- Implement proper indexing

Security Standards
- Implement authentication middleware
- Use HTTPS in production
- Validate all input data
- Implement rate limiting
- Use environment variables for secrets

Response Format Standards

Success Response
```json
{
  "success": true,
  "data": {},
  "message": "Operation successful"
}
```

```

### Error Response

```json
{
  "success": false,
  "error": {
    "code": "ERROR_CODE",
    "message": "Error description"
  }
}
```

Testing Requirements

Unit Testing

- Test all service functions
- Mock external dependencies
- Test error scenarios
- Use Jest testing framework

Integration Testing

- Test API endpoints
- Test database operations
- Test middleware functions
- Use supertest for HTTP testing
```

Test Structure

```typescript
describe('UserService', () => {
  describe('createUser', () => {
    it('should create user successfully', async () => {
      // Test implementation
    });

    it('should handle validation errors', async () => {
      // Test error scenarios
    });
  });
});
```

Development Workflow

Environment Setup

```bash
npm install
npm run dev
```

Database Setup

```bash
npm run db:migrate
npm run db:seed
```

```

### API Development

1. Define API endpoints in routes
2. Implement controller logic
3. Add service layer functions
4. Create repository methods
5. Write comprehensive tests

### Deployment

- Use Docker for containerization
- Implement health check endpoints
- Use environment-specific configurations
- Set up monitoring and logging

## Code Quality

- Use ESLint and Prettier
- Implement pre-commit hooks
- Use TypeScript strict mode
- Follow SOLID principles
```

4. Python Django Template

```markdown
# Python Django Web Application
```

A scalable web application built with Django, following best practices for security, performance, and maintainability.

Project Structure

```
django-app/
├── manage.py
├── requirements.txt
├── myproject/
│   ├── __init__.py
│   ├── settings/       # Environment-specific settings
│   ├── urls.py
│   └── wsgi.py
├── apps/
│   ├── users/          # User management app
│   ├── core/           # Core functionality
│   └── api/            # API endpoints
├── static/             # Static files
├── media/              # User uploads
├── templates/          # HTML templates
├── tests/              # Test files
└── docs/               # Documentation
```

Coding Standards

Django Best Practices

- Use Django's built-in features
- Follow Model-View-Template pattern
- Implement proper URL routing
- Use Django forms for validation

Model Design

- Use descriptive model names
- Implement proper field validation
- Use Django's built-in fields
- Add proper string representations

View Standards

- Use class-based views when appropriate
- Implement proper permission checks
- Use Django's pagination
- Handle exceptions gracefully

Template Guidelines

- Use template inheritance
- Implement proper template organization
- Use Django's template tags
- Ensure templates are XSS-safe

Database Standards

Model Relationships

- Use appropriate relationship types
- Implement proper foreign keys
- Use related_name for reverse relationships
- Add database indexes for performance

Migration Best Practices

- Create descriptive migration names
- Test migrations on staging
- Use data migrations for complex changes
- Keep migrations small and focused

Security Standards

Authentication & Authorization

- Use Django's authentication system
- Implement proper permission classes
- Use CSRF protection
- Implement proper session handling

Data Protection

- Validate all user input
- Use Django's ORM to prevent SQL injection
- Implement proper file upload handling
- Use HTTPS in production

Testing Requirements

Unit Testing

- Use Django's TestCase
- Test model methods
- Test view logic
- Test form validation

Integration Testing

- Test user workflows
- Test API endpoints
- Test database operations
- Test template rendering

Test Structure

```python
from django.test import TestCase
from django.contrib.auth import get_user_model

User = get_user_model()

class UserModelTest(TestCase):
    def setUp(self):
        self.user = User.objects.create_user(
            username='testuser',
            email='test@example.com',
            password='testpass123'
        )
```

```
    def test_user_creation(self):
        self.assertEqual(self.user.username, 'testuser')
        self.assertEqual(self.user.email, 'test@example.com')
```

Development Workflow

Environment Setup

```bash
pip install -r requirements.txt
python manage.py migrate
python manage.py runserver
```

Development Process

1. Create Django app for new functionality
2. Design models and relationships
3. Create views and URLs
4. Design templates
5. Write comprehensive tests
6. Update documentation

Production Deployment

- Use environment variables for settings
- Configure static file serving
- Set up database backups

- Implement monitoring and logging
- Use proper WSGI server

Code Quality

- Use Black for code formatting
- Use flake8 for linting
- Use mypy for type checking
- Follow PEP 8 standards
```

## 5. Mobile Flutter Template

```markdown
# Flutter Mobile Application

A cross-platform mobile application built with Flutter and Dart, following Material Design principles.

## Project Structure

```
flutter-app/
├── lib/
│   ├── main.dart
│   ├── app.dart
│   ├── screens/        # UI screens
```

```
│   ├── widgets/        # Reusable widgets
│   ├── models/         # Data models
│   ├── services/       # API and business logic
│   ├── providers/      # State management
│   ├── utils/          # Utility functions
│   ├── constants/      # App constants
│   └── themes/         # App themes
├── assets/
│   ├── images/         # Image assets
│   ├── fonts/          # Custom fonts
│   └── icons/          # App icons
├── test/               # Test files
├── integration_test/   # Integration tests
├── android/            # Android-specific code
├── ios/                # iOS-specific code
└── web/                # Web-specific code
```

Coding Standards

Dart Language Standards

- Use null safety features
- Follow Dart naming conventions
- Use const constructors when possible
- Implement proper error handling

Widget Design

- Create reusable custom widgets
- Use StatelessWidget when possible
- Implement proper widget lifecycle
- Use keys for widget identification

State Management
- Use Provider or Riverpod for state management
- Keep state as local as possible
- Implement proper state updates
- Use immutable data structures

UI/UX Standards
- Follow Material Design guidelines
- Implement responsive design
- Use consistent spacing and typography
- Implement proper accessibility features

Performance Standards

Widget Performance
- Use const constructors
- Implement proper widget rebuilds
- Use ListView.builder for long lists
- Optimize image loading and caching

Memory Management
- Dispose controllers properly

- Use weak references when needed
- Implement proper stream handling
- Avoid memory leaks

Testing Requirements

Unit Testing
- Test business logic functions
- Test data models
- Test service classes
- Use mockito for mocking

Widget Testing
- Test widget rendering
- Test user interactions
- Test widget state changes
- Test widget accessibility

Integration Testing
- Test complete user flows
- Test API integration
- Test navigation
- Test performance

Test Structure
```dart
import 'package:flutter_test/flutter_test.dart';
```

```
import 'package:myapp/models/user.dart';

void main() {
  group('User Model Tests', () {
    test('should create user from JSON', () {
      final json = {
        'id': 1,
        'name': 'John Doe',
        'email': 'john@example.com'
      };

      final user = User.fromJson(json);

      expect(user.id, 1);
      expect(user.name, 'John Doe');
      expect(user.email, 'john@example.com');
    });
  });
}
```

Development Workflow

Environment Setup

```bash
flutter pub get
flutter run
```

```

### Development Process

1. Design UI mockups

2. Create reusable widgets

3. Implement state management

4. Add API integration

5. Write comprehensive tests

6. Test on multiple devices

### Build and Release

```bash

flutter build apk --release

flutter build ios --release

flutter build web --release

```

Code Quality

- Use flutter_lints for linting

- Use dart format for formatting

- Use dart analyze for static analysis

- Follow Flutter style guide

```

## 6. Go Microservices Template

```markdown
# Go Microservices Application

A scalable microservices architecture built with Go, focusing on clean architecture and observability.

## Project Structure

```
go-microservice/
├── cmd/
│   └── server/   # Application entry point
├── internal/
│   ├── handlers/   # HTTP handlers
│   ├── services/   # Business logic
│   ├── repositories/   # Data access
│   ├── models/   # Data models
│   └── middleware/   # HTTP middleware
├── pkg/
│   ├── logger/   # Logging utilities
│   ├── database/   # Database connections
│   └── config/   # Configuration
├── api/
│   └── openapi/   # API specifications
├── deployments/   # Deployment configs
├── tests/   # Test files
└── docs/   # Documentation
```
```

```

## Coding Standards

### Go Language Standards
- Follow Go naming conventions
- Use proper error handling
- Implement interfaces for abstraction
- Use goroutines and channels appropriately

### Clean Architecture
- Separate concerns by layers
- Use dependency injection
- Implement proper abstractions
- Follow SOLID principles

### API Design
- Use OpenAPI specifications
- Implement proper HTTP methods
- Use consistent response formats
- Include request validation

### Error Handling
```go
type AppError struct {
    Code    string `json:"code"`
    Message string `json:"message"`
```
```

```
    Err     error  `json:"-"`
}

func (e *AppError) Error() string {
    return e.Message
}
```

Testing Requirements

Unit Testing
- Use Go's testing package
- Test business logic thoroughly
- Mock external dependencies
- Test error scenarios

Integration Testing
- Test API endpoints
- Test database operations
- Test service interactions
- Use testcontainers for dependencies

Test Structure
```go
package services

import (
```

```
    "testing"
    "github.com/stretchr/testify/assert"
    "github.com/stretchr/testify/mock"
)

func TestUserService_CreateUser(t *testing.T) {
    mockRepo := new(MockUserRepository)
    service := NewUserService(mockRepo)

    user := &User{
        Name:  "John Doe",
        Email: "john@example.com",
    }

    mockRepo.On("Create", mock.AnythingOfType("*User")).Return(nil)

    err := service.CreateUser(user)

    assert.NoError(t, err)
    mockRepo.AssertExpectations(t)
}
```

Development Workflow

Environment Setup

```bash
```

```
go mod init myproject
go mod tidy
go run cmd/server/main.go
```

Development Process

1. Design API contracts
2. Implement handlers
3. Add business logic
4. Create repositories
5. Write comprehensive tests
6. Add monitoring and logging

Build and Deploy

```bash
go build -o bin/server cmd/server/main.go
docker build -t myapp .
```

Observability

Logging

- Use structured logging
- Include correlation IDs
- Log errors appropriately
- Use different log levels

Monitoring

- Implement health checks
- Add metrics collection
- Use distributed tracing
- Monitor application performance

Configuration

- Use environment variables
- Implement configuration validation
- Support multiple environments
- Use configuration management

Code Quality

- Use golangci-lint for linting
- Use gofmt for formatting
- Use go vet for static analysis
- Follow Go best practices

```

## 7. DevOps/Infrastructure Template

```markdown
# Infrastructure as Code Project

A comprehensive infrastructure setup using Terraform, Docker, and Kubernetes for scalable deployments.

## Project Structure

```
infrastructure/
├── terraform/
│   ├── modules/   # Reusable Terraform modules
│   ├── environments/   # Environment-specific configs
│   └── global/   # Global resources
├── docker/
│   ├── Dockerfile
│   ├── docker-compose.yml
│   └── .dockerignore
├── kubernetes/
│   ├── deployments/   # Kubernetes deployments
│   ├── services/   # Kubernetes services
│   ├── ingress/   # Ingress configurations
│   └── configmaps/   # ConfigMaps and secrets
├── ansible/
│   ├── playbooks/   # Ansible playbooks
│   ├── roles/   # Ansible roles
│   └── inventory/   # Inventory files
├── monitoring/
│   ├── prometheus/   # Prometheus configs
│   ├── grafana/   # Grafana dashboards
│   └── alertmanager/   # Alert configurations
└── scripts/   # Deployment scripts
```

```

## Infrastructure Standards

### Terraform Standards
- Use modules for reusability
- Implement proper state management
- Use variable validation
- Include resource tagging

### Docker Standards
- Use multi-stage builds
- Implement proper layer caching
- Use specific base image tags
- Include health checks

### Kubernetes Standards
- Use namespaces for isolation
- Implement resource limits
- Use ConfigMaps for configuration
- Include liveness and readiness probes

## Security Standards

### Infrastructure Security
- Use least privilege principle
- Implement network segmentation
```

- Use secrets management
- Enable audit logging

Container Security
- Scan images for vulnerabilities
- Use non-root users
- Implement proper secret handling
- Use minimal base images

Monitoring and Observability

Metrics Collection
- Use Prometheus for metrics
- Implement custom metrics
- Monitor resource usage
- Set up alerting rules

Logging
- Centralize log collection
- Use structured logging
- Implement log retention
- Monitor log patterns

Tracing
- Implement distributed tracing
- Monitor request flows
- Track performance metrics

- Debug service interactions

Deployment Process

CI/CD Pipeline

```yaml
# Example GitHub Actions workflow
name: Deploy Infrastructure

on:
  push:
    branches: [main]

jobs:
  terraform:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Setup Terraform
        uses: hashicorp/setup-terraform@v1
      - name: Terraform Plan
        run: terraform plan
      - name: Terraform Apply
        run: terraform apply -auto-approve
```

Environment Management

- Use separate environments
- Implement environment promotion
- Use infrastructure testing
- Maintain environment parity

Documentation Requirements

Infrastructure Documentation
- Document architecture decisions
- Maintain deployment procedures
- Include troubleshooting guides
- Document security procedures

Runbooks
- Create operational procedures
- Document incident response
- Include rollback procedures
- Maintain contact information

Code Quality

- Use terraform fmt for formatting
- Use terraform validate for validation
- Use security scanning tools
- Follow infrastructure best practices
```
```

How to Use These Templates

1. **Choose the Right Template**: Select the template that best matches your project type and technology stack.

2. **Customize for Your Project**: Adapt the template to your specific requirements, team preferences, and project constraints.

3. **Place in Root Directory**: Save the agents.md file in your project's root directory alongside your README.md.

4. **Keep Updated**: Regularly update your agents.md file as your project evolves and your standards change.

5. **Team Alignment**: Ensure all team members understand and follow the guidelines specified in the agents.md file.

Best Practices for agents.md Files

- **Be Specific**: Include detailed, actionable instructions rather than vague guidelines
- **Keep Updated**: Regularly review and update as your project evolves
- **Test Effectiveness**: Monitor how well AI agents follow your instructions and adjust accordingly
- **Include Examples**: Provide code examples to illustrate your standards
- **Document Edge Cases**: Include guidance for handling unusual scenarios
- **Maintain Consistency**: Ensure your agents.md aligns with your project's README and other documentation

These templates provide a solid foundation for creating effective agents.md files that will significantly improve your AI coding assistant's performance and code quality.