Claude Code

Tutorial & Best Practices

AI-First Development for Modern Software Teams

https://github.com/emmanuelandre/claude-tutorial

Agenda

- Introduction to Claude Code
- AI-First Development Philosophy
- Getting Started
- Core Workflow
- Testing Strategy
- Best Practices
- Real-World Application

Introduction

What is Claude Code?

- AI-powered development tool for writing, debugging, and refactoring code
- Shifts paradigm from human writes → AI assists to AI writes → human validates

Core Benefits:

- 10-100x faster prototyping
- Consistent code standards
- Comprehensive testing
- Up-to-date documentation

Development Paradigm Shift

Traditional Development

✓ Human writes code

✓ AI assists occasionally

✓ Manual testing

✓ Documentation lags behind

✓ Inconsistent patterns

AI-First Development

• AI executes 100% of coding

• Human validates 100%

• Automated comprehensive tests

• Documentation generated with code

• Enforced consistency via CLAUDE.md

Getting Started

The CLAUDE.md File

- Your project's instruction manual for Claude Code
- Contains architecture, tech stack, and conventions

Essential Sections:

- Project overview and architecture
- Git workflow and commit format
- Code standards and testing requirements
- Project structure and common commands

- Updated throughout project lifecycle
- Ensures consistency across all sessions

Prerequisites & Setup

- Git 2.x or higher
- Code editor (VS Code, Cursor)
- Claude Code access (claude.ai/code)

Language-specific tools:

- Node.js (via nvm) for JavaScript/TypeScript
- Python 3.10+ with virtual environments
- Go 1.20+ for Go projects

- GitHub CLI (gh) recommended for PR management

AI-First Workflow

The 10-Step Development Process

- 1. Human writes detailed specification
- 2. AI designs database schema → Human reviews
- 3. AI implements repository layer → Human reviews
- 4. AI creates API endpoints → Human reviews
- 5. AI writes comprehensive API E2E tests → Human verifies
- 6. AI builds frontend components → Human reviews
- 7. AI creates UI E2E tests → Human verifies
- 8. AI updates documentation → Human reviews
- 9. Human conducts code review (security, performance, quality)
- 10. AI executes deployment → Human verifies

Clear Division of Responsibilities

AI Executes 100%

✓ Database schema design

✓ Repository implementation

✓ API endpoint creation

✓ E2E test writing

✓ Frontend component building

✓ Documentation updates

✓ Deployment execution

Human Validates 100%

• Write specifications

• Review schema design

• Approve implementations

• Verify tests pass

• Conduct code reviews

• Make architectural decisions

• Approve deployments

Testing Strategy

E2E Tests > Unit Tests

- Inverted testing pyramid: E2E tests are primary

Why E2E First?

- Test actual user flows
- Catch integration issues
- Verify the whole system
- Enable confident refactoring

- Component tests: Only for complex UI logic
- Unit tests: Optional - only for critical algorithms
- Measure test effectiveness, don't assume value

E2E Testing Best Practices

- Test user journeys, not implementation details
- Use stable data-test attributes, not CSS selectors

Coverage priorities:

- Happy path - must pass
- Critical failures - invalid inputs, permissions
- Edge cases - boundary conditions
- Error scenarios - network failures, timeouts

- Separate API and UI E2E tests
- Create reusable test commands and fixtures

Best Practices

Git Workflow Standards

Branch naming: /

- feature/, fix/, refactor/, docs/, test/, chore/

Conventional commits: (scope): subject

- feat, fix, refactor, docs, test, chore, perf

Hard rules - never break:

- Never commit directly to main
- Never merge without review
- Never commit code that fails tests
- Pre-commit checks MUST pass (lint, test, build)

Effective Prompt Engineering

- Be specific, not vague - include exact requirements
- Provide context - architecture, patterns, constraints
- Include examples - API contracts, expected behavior

Multi-step requests:

- Break complex features into clear steps
- Define validation criteria for each step
- Request tests and documentation explicitly

- Reference existing code patterns for consistency

Mandatory Quality Gates

Pre-Commit (Local):

- ■ Run linter (ESLint, golangci-lint, etc.)
- ■ Run all tests
- ■ Verify build succeeds
- ■ DO NOT commit if any check fails

Pre-Merge (CI):

- ■ All E2E tests pass
- ■ No linting errors
- ■ Build succeeds
- ■ Code review approved

• AI cannot merge PRs - humans only

Real-World Application

Keys to Success

1. CLAUDE.md is essential

- Keep it current and comprehensive
- Document all conventions and patterns

2. Clear handoffs between AI and human

- AI completes a step fully before handoff
- Human approves or requests changes

3. Systematic quality gates

- Tests must pass before proceeding
- Reviews must approve before merging

4. Iterative refinement

- Start working, refine as you go

- Commit small, atomic changes

Do's and Don'ts

✓ DO

✓ Write detailed specifications

✓ Prioritize E2E tests

✓ Review all AI output

✓ Use conventional commits

✓ Keep CLAUDE.md updated

✓ Run pre-commit checks

✓ Make small, focused PRs

✗ DON'T

• Make vague requests

• Skip testing

• Commit untested code

• Ignore CLAUDE.md

• Let AI merge PRs

• Commit directly to main

• Create large, unfocused PRs

Resources & Next Steps

Tutorial Repository:

- github.com/emmanuelandre/claude-tutorial

Official Documentation:

- Claude Code: claude.ai/code
- Claude Docs: docs.anthropic.com

Key Documentation Files:

- CLAUDE.md template for your projects
- Step-by-step feature workflow guide
- Testing strategy detailed guide
- Troubleshooting common issues

Thank You!

Questions?

github.com/emmanuelandre/claude-tutorial