Claude Code

Interactive Tutorial & Workshop

AI-First Development for Modern Software Teams

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

Workshop Agenda

- Part 1: Philosophy & Foundation
- Part 2: Getting Started Hands-On
- Part 3: Core Workflow (Interactive)
- Part 4: Testing Strategy (Live Demo)
- Part 5: Best Practices
- Part 6: Real-World Application

Part 1: Philosophy & Foundation

The AI-First Philosophy

- Traditional: Human writes code → AI assists
- AI-First: AI executes 100% → Human validates 100%

This means:

- AI handles ALL coding, testing, documentation
- Human handles ALL validation, review, decisions
- Clear handoff points at each step
- Systematic quality gates

Core Development Philosophy

API-First Development:

- Specifications drive implementation
- Database schema → API contracts → UI components
- E2E tests validate complete user journeys

Micro-Teams of 2:

- 2 humans + Claude Code = redundancy without overhead
- Each team owns end-to-end features
- Parallel development without bottlenecks

Zero External Dependencies:

- Be your own QA engineer
- Be your own DevOps engineer

- Own the entire vertical slice

Testing Philosophy

- E2E tests are MANDATORY (API + UI user journeys)
- Unit tests are MANDATORY (business logic, utilities, edge cases)
- Component tests are GOOD TO HAVE (test containers for microservices)

Test-First Approach:

- Define coverage targets before implementation
- Build testing infrastructure/framework first
- Track coverage from unit, component, and E2E tests
- Coverage thresholds vary by project (higher is better)
- Tests are your regression safety net

- Measure coverage to ensure quality and confidence

Part 2: Getting Started

What You Need

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

Language-specific tools:

- Go 1.21+ for backend
- Node.js 18+ (via nvm) for React
- Docker for deployment

- GitHub CLI (gh) for PR management

■ Hands-On: Create Your First Project

YOUR PROMPT (See prompts.md):

```
mkdir my-api-project
cd my-api-project
git init

# Now ask Claude:
Help me create a CLAUDE.md file for a Go API project with:
- PostgreSQL database
- JWT authentication
- RESTful endpoints
- Docker deployment

Include project structure, git workflow, and testing strategy.
```

What You Should See:

• Claude creates a comprehensive CLAUDE.md with architecture, commands, and conventions

■ Hands-On: Initialize Git Workflow

YOUR PROMPT (See prompts.md):

```
Create feature branch and set up git workflow:
- Branch naming convention: feature/initial-setup
- Conventional commits enabled
- Pre-commit hooks for linting and testing

Follow the git workflow section in CLAUDE.md
```

What You Should See:

• Git repository with proper branch structure and commit conventions

Part 3: AI-First Workflow

The 10-Step Process

- 1. Specification (Human writes detailed spec)
- 2. Database Schema (AI designs → Human reviews)
- 3. Repository Layer (AI implements → Human reviews)
- 4. API Endpoints (AI creates → Human reviews)
- 5. API E2E Tests (AI writes → Human verifies)
- 6. Frontend Components (AI builds → Human reviews)
- 7. UI E2E Tests (AI creates → Human verifies)
- 8. Documentation (AI updates → Human reviews)
- 9. Code Review (Human conducts)
- 10. Deployment (AI executes → Human verifies)

Hands-On: Step 1 - Write Specification

YOUR PROMPT (See prompts.md):

```
I need to implement user authentication for my API.

Requirements:
- Users register with email/password
- JWT tokens for authentication
- Password hashing with bcrypt
- Token refresh endpoint
- Logout (token invalidation)

Database:
- users table: id, email, password_hash, created_at, updated_at

API Endpoints:
POST /api/auth/register - Register new user
POST /api/auth/login - Login and get JWT
POST /api/auth/refresh - Refresh JWT token
POST /api/auth/logout - Invalidate token
```

```
Please create the database migration first.
```

What You Should See:

• Claude asks clarifying questions and confirms the specification

Expected: Database Migration (Go)

SQL

```sql
-- migrations/001_create_users.up.sql
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    email VARCHAR(255) UNIQUE NOT NULL,
    password_hash VARCHAR(255) NOT NULL,
    created_at TIMESTAMP DEFAULT NOW(),
    updated_at TIMESTAMP DEFAULT NOW()
);

CREATE INDEX idx_users_email ON users(email);
```

■ Hands-On: Step 2 - Review Schema

YOUR PROMPT (See prompts.md):

```
Review the migration file Claude created.

Check:
- Column types are appropriate
- Indexes are in place
- Constraints are correct

If approved, say: "Looks good, please proceed to repository layer"
If changes needed: "Change X to Y because..."

Run the migration:
psql -d mydb -f migrations/001_create_users.up.sql
```

What You Should See:

• Database table created successfully with proper indexes

Expected: Repository Layer (Go)

GO

```
// internal/repository/user.go
type UserRepository struct {
    db *sql.DB
}

func (r *UserRepository) Create(ctx context.Context, email, passwordHash string) (*User, error) {
    query := `INSERT INTO users (email, password_hash) VALUES ($1, $2) RETURNING id, email, created_at`
    var user User
    err := r.db.QueryRowContext(ctx, query, email, passwordHash).Scan(&user.ID;, &user.Email;, &user.CreatedAt;)
    return &user;, err
}

func (r *UserRepository) FindByEmail(ctx context.Context, email string) (*User, error) {
    query := `SELECT id, email, password_hash, created_at FROM users WHERE email = $1`
    var user User
    err := r.db.QueryRowContext(ctx, query, email).Scan(&user.ID;, &user.Email;, &user.PasswordHash;, &user.CreatedAt;)
    return &user;, err
}
```

■ Hands-On: Step 4 - Request API Implementation

YOUR PROMPT (See prompts.md):

```
The repository layer looks good.

Now implement the API handlers:
1. POST /api/auth/register
- Validate email format
- Check password strength (min 8 chars)
- Hash password with bcrypt
- Return JWT token

2. POST /api/auth/login
- Validate credentials
- Compare password hash
- Return JWT token

3. POST /api/auth/refresh
- Validate existing token
- Issue new token
```

```
Include error handling and proper HTTP status codes.
```

What You Should See:

- Claude creates handler files with validation and error handling

Expected: API Handler (Go)

GO

```
// internal/handlers/auth.go
func (h *AuthHandler) Register(w http.ResponseWriter, r *http.Request) {
    var req RegisterRequest
    if err := json.NewDecoder(r.Body).Decode(&req;); err != nil {
        respondError(w, http.StatusBadRequest, "Invalid request")
        return
    }

    // Validate
    if !isValidEmail(req.Email) {
        respondError(w, http.StatusBadRequest, "Invalid email")
        return
    }
    if len(req.Password) < 8 {
        respondError(w, http.StatusBadRequest, "Password too short")
        return
    }

    // Hash password
    hash, err := bcrypt.GenerateFromPassword([]byte(req.Password), bcrypt.DefaultCost)
    if err != nil {
```

```
        respondError(w, http.StatusInternalServerError, "Failed to hash password")
        return
    }

    // Create user
    user, err := h.userRepo.Create(r.Context(), req.Email, string(hash))
    if err != nil {
        respondError(w, http.StatusConflict, "Email already exists")
        return
    }

    // Generate JWT
    token, err := h.jwtService.Generate(user.ID)
    respondJSON(w, http.StatusCreated, map[string]string{"token": token})
}
```

■ Hands-On: Step 5 - Request API Tests

YOUR PROMPT (See prompts.md):

```
Create comprehensive E2E tests for the auth endpoints using Cypress.

Test cases:
1. Register - Happy path
2. Register - Duplicate email (409 error)
3. Register - Invalid email (400 error)
4. Register - Weak password (400 error)
5. Login - Valid credentials
6. Login - Invalid credentials (401 error)
7. Login - Non-existent user (401 error)
8. Refresh - Valid token
9. Refresh - Expired token (401 error)

Use Cypress API testing (cy.request)
```

What You Should See:

- Cypress test file created with all test cases

Expected: E2E API Tests (Cypress)

JAVASCRIPT

```
// cypress/e2e/api/auth.cy.js
describe('API: Authentication', () => {
  it('registers new user successfully', () => {
    cy.request('POST', '/api/auth/register', {
      email: 'test@example.com',
      password: 'SecurePass123'
    })
    .then((response) => {
      expect(response.status).to.eq(201)
      expect(response.body).to.have.property('token')
    })
  })

  it('rejects duplicate email', () => {
    cy.request({
      method: 'POST',
      url: '/api/auth/register',
      body: { email: 'test@example.com', password: 'SecurePass123' },
      failOnStatusCode: false
    })
    .then((response) => {
```

```
      expect(response.status).to.eq(409)
    })
  })

  it('rejects weak password', () => {
    cy.request({
      method: 'POST',
      url: '/api/auth/register',
      body: { email: 'new@example.com', password: 'weak' },
      failOnStatusCode: false
    })
    .then((response) => {
      expect(response.status).to.eq(400)
    })
  })
})
```

Hands-On: Run API Tests

YOUR PROMPT (See prompts.md):

```
Run the API tests Claude created:

npx cypress run --spec "cypress/e2e/api/auth.cy.js"

Verify all tests pass.

If any fail, ask Claude:
"Test X is failing with error Y. Please investigate and fix.
```

What You Should See:

- All API tests pass (green checkmarks in terminal)

■ Hands-On: Step 6 - Request Frontend

YOUR PROMPT (See prompts.md):

```
Now let's build the UI for authentication.

Create React components:
1. LoginForm component
- Email and password inputs
- Form validation
- Submit calls /api/auth/login
- Stores JWT in localStorage

2. RegisterForm component
- Email, password, confirm password inputs
- Client-side validation
- Submit calls /api/auth/register
- Auto-login after registration

Use React Hook Form for form handling and Zustand for auth state.
```

What You Should See:

- React components created with forms and state management

Expected: Login Component (React)

JSX

```
// src/components/LoginForm.jsx
import { useForm } from 'react-hook-form'
import { useAuthStore } from '../stores/authStore'

export function LoginForm() {
  const { register, handleSubmit, formState: { errors } } = useForm()
  const login = useAuthStore(state => state.login)

  const onSubmit = async (data) => {
    try {
      const response = await fetch('/api/auth/login', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify(data)
      })

      if (!response.ok) throw new Error('Login failed')

      const { token } = await response.json()
      login(token) // Store in Zustand + localStorage
    } catch (err) {
```

```
      console.error(err)
    }
  }

  return (

              {...register('email', { required: true, pattern: /^[^@]+@[^@]+$/ })}
        placeholder="Email"
      />
      {errors.email && Valid email required}

              type="password"
        {...register('password', { required: true, minLength: 8 })}
        placeholder="Password"
      />
      {errors.password && Password must be 8+ characters}

      Login

  )
}
```

■ Hands-On: Step 7 - Request UI Tests

YOUR PROMPT (See prompts.md):

```
Create UI E2E tests for the authentication flow.

Test scenarios:
1. User can register new account
2. User can login with valid credentials
3. User cannot login with invalid password
4. Form validation works (weak password, invalid email)
5. User stays logged in after page refresh
6. User can logout

Use data-test attributes for stable selectors.
```

What You Should See:

• Cypress UI test file created with user journey tests

Expected: E2E UI Tests (Cypress)

JAVASCRIPT

```
// cypress/e2e/ui/auth.cy.js
describe('UI: Authentication Flow', () => {
  beforeEach(() => {
    cy.visit('/login')
  })

  it('registers and logs in new user', () => {
    // Register
    cy.get('[data-test="register-link"]').click()
    cy.get('[data-test="email-input"]').type('newuser@example.com')
    cy.get('[data-test="password-input"]').type('SecurePass123')
    cy.get('[data-test="submit-button"]').click()

    // Should redirect to dashboard
    cy.url().should('include', '/dashboard')
    cy.get('[data-test="user-email"]').should('contain', 'newuser@example.com')
  })

  it('shows validation errors for weak password', () => {
    cy.get('[data-test="password-input"]').type('weak')
    cy.get('[data-test="submit-button"]').click()
```

```
    cy.get('[data-test="error-message"]').should('contain', '8+ characters')
  })

  it('persists login after refresh', () => {
    // Login first
    cy.get('[data-test="email-input"]').type('test@example.com')
    cy.get('[data-test="password-input"]').type('SecurePass123')
    cy.get('[data-test="submit-button"]').click()

    // Refresh page
    cy.reload()

    // Still logged in
    cy.url().should('include', '/dashboard')
  })
})
```

Part 4: Testing Strategy

E2E First Testing

- 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

■ Hands-On: Practice: Write E2E Test

YOUR PROMPT (See prompts.md):

```
Write an E2E test for a password reset flow.

Feature requirements:
- User enters email on /forgot-password page
- System sends reset link to email
- User clicks link, enters new password
- User can login with new password

Create BOTH API and UI E2E tests for this flow.

Ask Claude to implement the feature + tests following the 10-step process.
```

What You Should See:

• Complete password reset feature with passing E2E tests

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

Part 5: 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 commit directly to main
- Never merge without review
- Pre-commit checks MUST pass (lint, test, build)

■ Hands-On: Practice: Proper Git Workflow

YOUR PROMPT (See prompts.md):

```
You've implemented the auth feature. Now create a proper PR.

Steps:
1. Review your changes: git status, git diff
2. Run pre-commit checks (lint, tests, build)
3. Commit with conventional format
4. Push branch
5. Create PR with description

Ask Claude:
"Please help me commit the auth feature and create a PR.
Run all pre-commit checks first.
Use conventional commit format.
Include a detailed PR description.
```

What You Should See:

- PR created with passing CI checks and proper commit messages

Effective Prompt Engineering

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

Multi-step requests:

- Break complex features into clear steps
- Define validation criteria
- Request tests explicitly

- Reference existing patterns for consistency

■ Hands-On: Practice: Better Prompts

YOUR PROMPT (See prompts.md):

```
Compare these two prompts:

■ BAD:
"Add search to the users page"

■ GOOD:
"Add search functionality to users page:
- Search input in page header
- Filter by email and name (case-insensitive)
- Debounce 300ms
- Update URL query params
- Clear button
- Show 'No results' when empty

Follow our existing search pattern from products page.
Include E2E test for search + clear + no results."

Practice: Write a detailed prompt for adding pagination.
```

What You Should See:

• You create a comprehensive, specific prompt with clear requirements

Part 6: 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

Common Mistakes to Avoid

- ■ Vague prompts → Be specific with examples
- ■ Skipping tests → Always write E2E tests first
- ■ Committing untested code → Run checks locally
- ■ Ignoring CLAUDE.md → Keep it updated
- ■ Large unfocused PRs → Make small, atomic changes
- ■ Review all AI output before committing
- ■ Use conventional commits
- ■ Run pre-commit checks

■ Hands-On: Final Exercise: Complete Feature

YOUR PROMPT (See prompts.md):

```
Build a complete feature end-to-end:

Feature: User Profile Management
- View profile page showing user info
- Edit profile (name, email, avatar)
- Password change form
- Profile picture upload

Follow the full 10-step process:
1. Write specification (you)
2. Database schema (Claude → you review)
3. Repository layer (Claude → you review)
4. API endpoints (Claude → you review)
5. API E2E tests (Claude → you verify)
6. React components (Claude → you review)
7. UI E2E tests (Claude → you verify)
8. Documentation (Claude → you review)
9. Code review (you conduct)
```

```
10. Create PR (Claude → you merge)

Time limit: 30 minutes
```

What You Should See:

• Complete profile feature with passing tests and PR ready for review

Scaling to Large Projects

The Challenge of Large Projects

When projects grow:

- 100+ tasks across multiple modules
- Complex dependencies between features
- Weeks or months of development
- Context loss between sessions
- Need systematic progress tracking

The Solution:

- Phased development with living documentation
- Centralized project planning structure
- Continuous progress monitoring

Project Planning Structure

Directory Layout:

- project/planning/ - Master plan and progress
- project/specs/ - Detailed feature specifications
- project/sessions/ - Session summaries
- project/development/ - Quick reference guides

Key Files:

- devplan.md - Master plan with all phases
- devprogress.md - Living progress tracker
- database.md - Complete schema documentation
- Session notes - What happened each session

The Master Plan (devplan.md)

Contains:

- 10 phases with clear objectives
- Dependency mapping (what depends on what)
- Vertical slice workflow per feature
- All tasks with checkboxes
- Estimated timelines

Workflow per Feature:

- DB → Backend API → API Tests → UI → UI Tests → Docs
- Complete each layer before moving forward
- Never skip the testing phases

Progress Tracker (devprogress.md)

Update After Every Session:

■ Mark completed tasks with [x]

■ Update phase percentages

■ Track current sprint goals

■ Document blockers and decisions

■ Calculate overall progress

Quick Stats Table:

■ Phase | Status | Progress | Backend | UI | Tests

■ ■ Not Started | ■ In Progress | ■ Complete

■ Visual overview of project health

■ Hands-On: EXERCISE: Multi-Phase Task Manager (Phase 0)

YOUR PROMPT (See prompts.md):

```
Goal: Build a Task Management System across multiple phases

Phase 0: Foundation (We'll do this)
- User auth (Google OAuth + JWT)
- Organizations and teams
- Basic permissions

Phase 1: Projects (Demo only - out of scope)
- Project CRUD
- Project members
- Roles and permissions

Phase 2-3: Not implemented (Show planning only)

Step 1: Create Planning Structure
mkdir task-manager && cd task-manager
```

```
git init
mkdir -p project/{planning,specs,sessions,development}

Step 2: Ask Claude to create devplan.md
"Create project/planning/devplan.md for a Task Management System.

Break into 4 phases:
- Phase 0: Foundation (Auth, Users, Orgs)
- Phase 1: Projects (CRUD, Members)
- Phase 2: Tasks (CRUD, Comments, Attachments)
- Phase 3: Dashboard (Analytics)

For each phase list: DB tables, repos, API endpoints, UI pages, tests.
Use vertical slice workflow. Include checkboxes."

Expected: Complete devplan.md with ~80-100 tasks
```

What You Should See:

• devplan.md created with 4 phases and dependency mapping

■ Hands-On: EXERCISE: Create Progress Tracker

YOUR PROMPT (See prompts.md):

```
Step 3: Create devprogress.md

"Create project/planning/devprogress.md based on devplan.md.

Add:
- Quick Stats table (4 phases with status indicators)
- Current Sprint section (Phase 0 - Foundation)
- Sprint Goals (5-7 goals for Phase 0)
- All Phase 0 tasks with checkboxes
- Set Phase 0 to '■ In Progress' with 0% initially
- Mark other phases as '■ Not Started'"

Expected: devprogress.md showing Phase 0 ready to start

Step 4: Create database.md

"Create project/planning/database.md with complete schema.
```

```
Tables: organizations, users, user_groups, permissions,
projects, project_members, tasks, comments, attachments

For each: columns, types, PKs, FKs, indexes, constraints"

Expected: Full database schema for all phases
```

What You Should See:

• devprogress.md and database.md created

■ Hands-On: EXERCISE: Implement Phase 0 Database

YOUR PROMPT (See prompts.md):

```
Step 5: Database Migrations

"I'm in api/ directory. Initialize Go project and create migrations.

- Create go.mod for 'task-manager-api'
- Create: cmd/api, internal/{handlers,middleware,models,repository}, migrations/
- Create migrations for Phase 0 tables:
001_create_organizations.up.sql
002_create_users.up.sql
003_create_user_groups.up.sql
004_create_permissions.up.sql

Include seed data:
- Default organization
- Admin user group
- Permissions: users:*, projects:*, tasks:*"
```

```
Expected: Go project initialized, 4 migrations created

Step 6: Update Progress

"Update project/planning/devprogress.md:
- Mark database setup tasks as [x]
- Update Phase 0 percentage
- Update 'Completed This Session'"

Expected: Progress tracker shows ~20-30% Phase 0 complete
```

What You Should See:

• Migrations created and progress updated

Hands-On: EXERCISE: Implement Auth System

YOUR PROMPT (See prompts.md):

```
Step 7: Google OAuth + JWT

"Implement Google OAuth + JWT authentication:

1. models/user.go - User struct
2. repository/user_repository.go - GetByID, GetByEmail, Create, Update
3. handlers/auth_handler.go - GoogleLogin, GoogleCallback, GetMe
4. middleware/auth.go - JWT validation
5. cmd/api/main.go - Routes

Dependencies: gorilla/mux, golang-jwt/jwt/v5, lib/pq, oauth2/google
Use repository pattern - handlers never touch DB directly."

Expected: Auth system with OAuth and JWT

Step 8: Update Progress Again

"Update devprogress.md:
```

```
- Mark auth, repository, API tasks as [x]
- Update Phase 0 percentage
- Add to 'Completed This Session'"

Expected: Progress shows ~60-70% Phase 0 complete
```

What You Should See:

• Auth implemented and progress updated

Phase 1+ (Out of Workshop Scope)

Instructor Demo:

■ How to transition from Phase 0 to Phase 1

■ Mark Phase 0 as ■ Complete (100%)

■ Start Phase 1 as ■ In Progress

■ Create migrations for projects tables

■ Create session note documenting Phase 0

Phases 2-3 in devplan:

■ Shows how project continues beyond workshop

■ Demonstrates long-term planning

■ Each phase builds on previous

■ Progress tracking keeps you on track

Best Practices for Large Projects

Update Progress Religiously:

- After every session - mark completed tasks
- Calculate percentages accurately
- Document blockers immediately
- Create session notes with decisions

Keep Plans vs Reality Aligned:

- devplan.md = original blueprint (stable)
- devprogress.md = current reality (dynamic)
- Adjust plan when reality diverges significantly

Use Phase-Based Branches:

- phase-0-foundation, phase-1-projects, etc.

- Complete entire phase before merging
- Easier to track and review large changes

When to Use This Approach

Small Projects (<20 tasks):

■■ Don't need planning structure

■■ Simple CLAUDE.md is enough

Medium Projects (20-50 tasks):

■■ Create devplan.md and devprogress.md

■■ Update progress after sessions

Large Projects (50+ tasks):

■■ Full planning structure

■■ Daily progress updates

■■ Session notes after every session

■■ Detailed specs for complex features

Summary & Action Plan

What We Learned

- ✓ AI-First philosophy: AI executes, Human validates
- ✓ 10-step systematic development process
- ✓ E2E tests as primary testing strategy
- ✓ Proper git workflow and conventional commits
- ✓ Effective prompt engineering techniques
- ✓ CLAUDE.md as project instruction manual
- ✓ Scaling to large projects with phased planning
- ✓ Progress tracking with devplan.md and devprogress.md
- ✓ Quality gates at every step

Your Action Plan

Next Steps:

- 1. Create CLAUDE.md for your project
- 2. Set up git workflow (branches, conventional commits)
- 3. Start with one feature using 10-step process
- 4. Write E2E tests for everything
- 5. Review and iterate

Resources:

- Tutorial: github.com/emmanuelandre/claude-tutorial
- Claude Code: claude.ai/code
- Prompts: See prompts.md file

Thank You!

Questions?

github.com/emmanuelandre/claude-tutorial