Cursor AI Workshop

Build Apps with AI Assistance

Welcome!

What You'll Learn Today

- Master Cursor's core AI features
- Build apps faster with AI assistance
- Debug and refactor efficiently
- Apply AI pair programming in real projects

Workshop Format

- 6 hands-on labs (10 min each)
- Live demonstrations
- Q&A throughout

Section 1: Introduction (20 min)

Getting Cursor

Installation

- Download from cursor.sh
- Available for macOS, Windows, and Linux
- 1-click install, ~500MB download
- Built on VS Code - familiar interface

First Launch

1. Open Cursor
2. Sign in (required for AI features)
3. Import VS Code settings (optional)
4. Ready to code!

Cursor Plans & Pricing

Free Plan

- **$0/month**
- 2,000 completions/month
- 50 slow premium requests
- GPT-4 access (limited)
- Perfect for trying Cursor

Pro Plan

- **$20/month**
- Unlimited completions
- 500 fast premium requests/month
- Unlimited slow premium requests

What You Need for This Workshop

Minimum:

- Free Cursor account ✅
- Internet connection ✅
- Basic coding knowledge ✅

Recommended:

- Pro plan (optional - free tier works fine for workshop)
- 3 hours of focused time
- Willingness to experiment

Note: Free tier is sufficient for today's workshop!

Privacy & Settings

Privacy Options

- **Privacy Mode** - Code not used for training
- **Disable Telemetry** - Opt out of usage data
- **Local-only Mode** - Work offline (limited features)

Recommended Settings

```
Settings → Cursor Settings → General
✅ Enable Tab completion
✅ Enable Cmd+K
✅ Enable Chat
```

For Teams/Enterprise

- Enforce privacy mode across team

Setup Verification

Before we start coding, verify:

1. ✅ Cursor is installed and running
2. ✅ You're signed in (see account icon)
3. ✅ Tab suggestions appear when typing
4. ✅ Cmd+K opens inline edit prompt
5. ✅ Cmd+L opens chat sidebar

Having issues?

- Check internet connection
- Restart Cursor
- Check Settings → Cursor Settings
- Ask for help now!

What is Cursor?

Cursor is an AI-powered code editor built on VS Code that helps you:

- **Write code faster** with intelligent autocomplete
- **Understand codebases** through AI chat
- **Fix bugs quickly** with context-aware suggestions
- **Refactor confidently** with AI assistance

Key Difference: Built from the ground up for AI-native development

Why Use Cursor?

Traditional Coding:

```
Idea → Google → StackOverflow → Copy → Modify → Debug
```

With Cursor:

```
Idea → Ask Cursor → Review → Refine → Done
```

Benefits:

- 50-70% faster development
- Fewer context switches
- Better code quality
- Learn while you build

Core Features Overview

1. **Tab Autocomplete** - Next-gen code completion
2. **Cmd+K** - Inline code generation/editing
3. **Chat (Cmd+L)** - AI pair programmer
4. **@ Symbols** - Add context to queries
5. **Composer** - Multi-file editing

Tab Autocomplete

Smart, context-aware suggestions as you type

```
// Start typing...
function calculateTotalPrice

// Cursor suggests:
function calculateTotalPrice(items, taxRate) {
  const subtotal = items.reduce((sum, item) => sum + item.price, 0);
  return subtotal * (1 + taxRate);
}
```

Pro Tips:

- Accept: `Tab`
- Reject: Keep typing
- Works across multiple lines
- Learns from your codebase

Cmd+K (Inline Edit)

Generate or modify code inline

How to use:

1. Select code (or place cursor)
2. Press `Cmd+K` (Mac) or `Ctrl+K` (Windows)
3. Describe what you want
4. Review and accept/reject

Examples:

- "Add error handling"
- "Convert to async/await"
- "Add TypeScript types"
- "Optimize this function"

Chat (Cmd+L)

Your AI pair programmer

Use Chat for:

- Understanding unfamiliar code
- Planning implementation approaches
- Asking "how to" questions
- Getting explanations
- Brainstorming solutions

Pro Tips:

- Be specific in questions
- Provide context with @ symbols
- Use for learning, not just coding

@ Symbols - Adding Context

Tell Cursor what to focus on:

- `@Files` - Reference specific files
- `@Code` - Reference functions/classes
- `@Docs` - Search documentation
- `@Web` - Search the web
- `@Codebase` - Search entire project
- `@Chat` - Reference previous chat

Example:

```
@Files utils.js — How does the validation function work?
```

Lab 1 Preview: Build Production REST API

You'll build:

- Complete Express API with 5 CRUD endpoints
- JWT authentication system
- Input validation middleware
- Error handling

Cursor Modes:

- **Chat** - Plan API architecture
- **Cmd+K** - Add routes and validation
- **Composer** - Create multi-file auth system

Goal: Master when to use each Cursor mode with real code!

Section 2: AI-Powered Code Generation

From Idea to Code

Natural Language → Working Code

Traditional approach:

1. Search documentation
2. Find examples
3. Adapt to your needs
4. Fix errors
5. Test

With Cursor:

1. Describe what you want
2. Review generated code
3. Refine if needed

Effective Prompts

Bad Prompt:

```
"make a function"
```

Good Prompt:

```
"Create a function that validates email addresses using regex,
returns true/false, and includes test cases"
```

Better Prompt:

```
"Create an email validation function that:
- Accepts a string parameter
- Uses RFC 5322 compliant regex
- Returns boolean
- Handles null/undefined
- Include JSDoc comments
- Add 5 test cases"
```

Prompt Engineering Tips

Be Specific:

- Define inputs and outputs
- Specify edge cases
- Mention patterns/frameworks
- Include constraints

Iterate:

- Start broad, then refine
- Ask for modifications
- Build incrementally

Provide Examples:

- Show desired format

Generating Components

React Example:

Prompt:

```
Create a reusable Card component with:
- title, description, imageUrl props
- onClick handler
- hover effect with shadow
- responsive design
- TypeScript types
```

Cursor generates complete component with:

- PropTypes/TypeScript
- Styling
- Event handlers
- Best practices

Generating Functions

Backend Example:

Prompt:

```
Create an Express middleware function that:
- Validates JWT tokens
- Checks token expiration
- Extracts user ID
- Handles errors gracefully
- Returns 401 for invalid tokens
```

Result: Production-ready middleware with error handling!

Lab 2 Preview: Configure AI Standards & Automation

You'll create:

- `.cursorrules` - 100+ lines of team coding standards
- `AGENTS.md` - AI workflow automation guidelines
- Custom Hooks - Security validation
- Plan Mode workflows
- Background Agent tasks

Features:

- Auto-enforce team standards
- Guide autonomous AI agents
- Security audit automation
- Multi-step task planning

Section 3: Refactoring & Understanding Code

Understanding Legacy Code

Cursor helps you:

- Understand unfamiliar codebases
- Document undocumented code
- Identify patterns and anti-patterns
- Plan refactoring strategies

How:

1. Open file
2. Ask Chat: `@Files thisfile.js - Explain what this does`
3. Get detailed explanation

Code Explanation Examples

Ask Cursor:

- "What does this function do?"
- "Explain this algorithm step by step"
- "What are the potential bugs here?"
- "How can this be improved?"
- "What design pattern is this using?"

Cursor provides:

- Line-by-line breakdown
- Purpose and context
- Potential issues
- Improvement suggestions

Smart Refactoring

Common Refactoring Tasks:

1. **Extract function**: Select code → Cmd+K → "extract to function"
2. **Rename variables**: "rename variables to be more descriptive"
3. **Add types**: "add TypeScript types"
4. **Modernize**: "convert to ES6 syntax"
5. **Optimize**: "optimize for performance"

Cursor maintains:

- Code functionality
- Existing tests
- Style consistency

Before & After

Before:

```
function p(d) {
  var r = [];
  for(var i=0; i<d.length; i++) {
    if(d[i].a > 18) r.push(d[i]);
  }
  return r;
}
```

After (with Cursor):

```
function filterAdultUsers(users) {
  return users.filter(user => user.age > 18);
}
```

Lab 3 Preview: Refactor Legacy E-commerce Code

You'll fix:

- SQL injection vulnerabilities → Parameterized queries
- Plain text passwords → bcrypt hashing
- Callback hell → async/await
- Hardcoded credentials → Environment variables
- No error handling → Comprehensive try-catch

Use:

- `code -d before/ complete/` - See exact diffs
- **Chat** - Analyze security issues
- **Cmd+K** - Fix specific vulnerabilities
- **Composer** - Restructure architecture

Break (15 min)

Take a break! We'll resume with debugging.

Section 4: Debugging with AI

AI-Powered Debugging

Traditional debugging:

1. See error
2. Read stack trace
3. Google error message
4. Try random solutions
5. Repeat

With Cursor:

1. See error
2. Ask Cursor
3. Get explanation + fix
4. Apply solution

Debugging Workflow

1. Identify the error

```
Copy error message
```

2. Ask Cursor

```
Chat: "I'm getting this error: [paste error]
@Files problemFile.js - What's causing this?"
```

3. Get diagnosis

- Root cause explanation
- Why it's happening
- How to fix it

4. Apply fix

Common Debugging Scenarios

Runtime Errors:

```
"Why am I getting 'Cannot read property of undefined'?"
```

Logic Errors:

```
"This function returns wrong results for negative numbers"
```

Performance Issues:

```
"Why is this function slow with large arrays?"
```

Integration Issues:

```
"API call works in Postman but fails in my app"
```

Error Prevention

Use Cursor to:

- Add error handling before bugs occur
- Validate inputs
- Add defensive checks
- Implement logging

Prompt:

```
@Code myFunction - Add comprehensive error handling
including input validation and edge cases
```

Lab 4 Preview: Build Real-Time Notifications

You'll build:

- Complete WebSocket notification system with Socket.io
- Notification service + database schema
- Real-time frontend component
- Offline notification queue
- Event emitters throughout app

Composer Coordination:

- Creates 10+ new files
- Modifies 5+ existing files
- Integrates WebSocket + REST API + Frontend
- **Plan Mode** shows implementation steps

Section 5: Advanced Features

Composer Mode

Multi-file editing for complex changes

Use Composer when:

- Changing multiple related files
- Refactoring across codebase
- Building connected features
- Updating dependencies

How:

1. Open Composer (Cmd+Shift+I)
2. Describe multi-file change
3. Review diffs across files
4. Accept/reject changes

Context Management

Cursor uses context from:

- Current file
- Open files
- Recently edited files
- Codebase index

You can add context:

- `@Files` - Specific files
- `@Folders` - Entire directories
- `@Code` - Functions/classes
- `@Codebase` - Semantic search

Pro tip: More context = better results!

.cursorrules File

Customize AI behavior for your project

Create `.cursorrules` in project root:

```
# Project Rules
- Use TypeScript strict mode
- Follow Airbnb style guide
- Use functional components in React
- Prefer async/await over promises
- Add JSDoc comments to all functions
- Write unit tests for new functions
- NEVER hardcode secrets
- Always validate user inputs
```

Cursor follows these rules automatically!

Every AI interaction (Chat, Cmd+K, Composer) respects your rules.

AGENTS.md - Guide AI Workflows

Define how AI agents should work

Create `AGENTS.md` in project root:

```
### Workflow: Add New API Endpoint

Steps:
1. Create route file in /routes
2. Add validation middleware
3. Implement service logic
4. Add error handling
5. Create tests
6. Update API documentation

Files to Create:
- routes/feature.js
- services/feature.js
- tests/feature.test.js
```


Guides autonomous agents through complex multi-step tasks

Plan Mode

See AI's plan before execution

1. AI analyzes your request
2. Creates detailed, editable plan
3. You review and modify
4. AI executes approved plan
5. You review results step-by-step

Benefits:

- Transparency - see the approach
- Control - edit before execution
- Learning - understand AI's reasoning
- Quality - catch issues early

Background Agents

AI works while you work

Start a background agent:

```
Background Agent Task: Generate comprehensive test suite

Create unit tests for all services
Integration tests for all routes
E2E tests for user workflows
Target 80%+ coverage

Run as background agent.
```

Benefits:

- Parallel productivity
- Time-consuming tasks don't block you
- Check progress when ready

Hooks - Custom AI Control

Run custom scripts to control AI behavior

Create `.cursor/hooks/security-check.js`:

```
module.exports = {
  onBeforeEdit: (context) => {
    // Check for secrets before allowing edit
    if (containsSecrets(context.content)) {
      return {
        allow: false,
        message: "Detected secrets. Use env vars."
      };
    }
    return { allow: true };
  }
};
```

Benefits:

Codebase Indexing

Cursor indexes your codebase for:

- Semantic code search
- Context-aware suggestions
- Cross-file understanding
- Pattern recognition

How to use:

```
@Codebase how do we handle authentication?
```

Cursor searches entire project and finds:

- Auth functions
- Related middleware

Documentation Search

Search docs without leaving Cursor:

```
@Docs React hooks - how does useEffect cleanup work?
```

Cursor searches:

- Official documentation
- Common libraries
- Framework guides
- Best practices

Benefits:

- No context switching
- Answers in your chat

Custom Instructions

Settings → Cursor Settings → General

Add instructions that apply to all chats:

```
- I prefer verbose variable names
- Always add error handling
- Use TypeScript
- Follow TDD principles
- Explain your reasoning
```

Cursor remembers your preferences!

Lab 5 Preview: Production-Ready Deployment

You'll accomplish:

- Comprehensive test suite (80%+ coverage)
- Complete CI/CD pipeline (GitHub Actions)
- Docker containerization
- Security hardening with custom Hooks
- Monitoring and logging (Winston, Sentry)
- API documentation (Swagger)
- Automated deployment with rollback

All Features Combined:

- **Hooks** - Security audit automation
- **Background Agent** - Test generation

Section 6: Best Practices

Do's and Don'ts

DO:

- Review all AI-generated code
- Provide specific context
- Iterate on prompts
- Ask "why" to learn
- Use version control

DON'T:

- Blindly accept suggestions
- Share sensitive code/data
- Ignore security implications
- Skip testing AI code

Security Considerations

Remember:

- AI suggestions may have vulnerabilities
- Review security-critical code carefully
- Don't expose API keys/secrets
- Validate inputs in generated code
- Test authentication/authorization

Ask Cursor:

```
"Review this code for security vulnerabilities"
```

Testing AI-Generated Code

Always:

1. Read the code
2. Understand the logic
3. Test edge cases
4. Verify assumptions
5. Run existing tests

Ask Cursor to:

- Generate test cases
- Identify edge cases
- Create mock data
- Write integration tests

Productivity Tips

Keyboard Shortcuts:

- `Cmd/Ctrl + K` - Inline edit
- `Cmd/Ctrl + L` - Open chat
- `Cmd/Ctrl + I` - Composer
- `Tab` - Accept suggestion
- `Esc` - Reject suggestion

Workflow:

1. Use Tab for simple completions
2. Use Cmd+K for modifications
3. Use Chat for questions/planning
4. Use Composer for multi-file changes

Learning with Cursor

Cursor is a learning tool:

- Ask "why" and "how"
- Request explanations
- Explore alternatives
- Understand, don't just copy

Example:

```
"Explain why this approach is better than using a for loop"
```

You'll learn while building!

When NOT to Use AI

Use human judgment for:

- Architecture decisions
- Business logic validation
- Security-critical code
- Performance-critical sections
- Complex algorithms (verify!)

AI is a tool, not a replacement for thinking

Recap & Next Steps

What You've Learned

Core Features:

- Tab autocomplete
- Cmd+K inline editing
- Chat for problem solving
- Context with @ symbols

Advanced Skills:

- Code generation
- Refactoring
- Debugging
- Multi-file editing

Your AI-Powered Workflow

```
1. Understand the problem
   ↓
2. Ask Cursor for approach
   ↓
3. Generate initial code
   ↓
4. Review and refine
   ↓
5. Test and debug
   ↓
6. Iterate
```

Continue Learning

Resources:

- Cursor Documentation
- Cursor Forum
- Tutorial Videos
- Practice daily!

Tips:

- Use Cursor for real projects
- Experiment with features
- Share learnings with team
- Stay updated on new features

Q&A

Questions?

Thank you for attending!

Stay in touch:

- Workshop materials: [GitHub link]
- Questions: [Email/Slack]
- More workshops: [Website]

Quick Reference

Keyboard Shortcuts

Action	Mac	Windows/Linux
Inline Edit	Cmd+K	Ctrl+K
Chat	Cmd+L	Ctrl+L
Composer	Cmd+I	Ctrl+I
Accept	Tab	Tab
Reject	Esc	Esc

@ Symbol Reference

Symbol	Purpose	Example
@Files	Reference files	@Files utils.js
@Code	Reference symbols	@Code myFunction
@Docs	Search docs	@Docs react hooks
@Web	Search web	@Web latest features
@Codebase	Search project	@Codebase authentication

Prompt Templates

Generate Function:

```
Create a function that [purpose]
- Input: [describe]
- Output: [describe]
- Handle: [edge cases]
- Include: [tests/docs]
```

Refactor:

```
Refactor this code to:
- [improvement 1]
- [improvement 2]
- Maintain existing functionality
```

Thank You!

Happy coding with Cursor! 🚀

Start building amazing things with AI assistance.