

Advanced LLM Concepts for Developers

Deep Dive into AI-Assisted Development (Optional)

Emmanuel BRUNO
Université de Toulon, LIS UMR CNRS 7020
emmanuel.bruno@univ-tln.fr

2026-10-14

Abstract Course lectures and practices for JavaScript full-stack web development with AI-assisted workflows.

Table of contents

1	Part 1: Understanding LLMs	2	2.4	Role Prompting - Give the AI a	
1.1	What Is a Language Model?	2		Persona	6
1.2	What Are Tokens?	2	2.4.a	Useful Roles for Development .	6
1.2.a	Why Tokens Matter	3	2.5	System Prompts vs User Prompts ...	6
1.3	The Transformer Architecture : Core		2.5.a	Using System Prompts (When	
	Mechanism: Self-Attention	3		Available)	6
1.3.a	Why This Matters for Code ...	3	3	Part 3: Advanced Techniques	6
1.4	Training Process - Stage 1: Pre-		3.1	RAG: Retrieval-Augmented	
	training (Unsupervised)	3		Generation	6
1.4.a	Training Process - Stage 2: Fine-		3.1.a	The Solution: RAG	6
	tuning (Supervised)	3	3.1.b	Tools That Use RAG	7
1.4.b	Training Process - Stage 3:		3.2	Embeddings and Semantic Search ...	7
	Alignment (RLHF)	4	3.2.a	Why This Matters	7
1.5	The Current LLM Landscape (2025) -		3.2.b	Practical Application	7
	Major Models Comparison	4	3.3	Temperature: Controlling	
1.5.a	Specialized Code Models	4		Randomness	7
2	Part 2: Advanced Prompt Engineering ...	5	3.3.a	When to Use Each	8
2.1	Chain-of-Thought (CoT) Reasoning .	5	3.4	Top-p (Nucleus Sampling)	8
2.1.a	The Solution: Chain-of-		3.4.a	Temperature vs Top-p	8
	Thought	5	4	Part 4: Production-Grade AI-Assisted	
2.2	Few-Shot Learning	5		Development	8
2.3	Prompt Chaining : Breaking Complex		4.1	Code Review with AI	8
	Tasks into Steps	5	4.1.a	Systematic Review Process ...	8
			4.2	Refactoring Strategies	9

4.2.a	Pattern: Incremental Refactoring	9
4.3	Testing with AI	9
4.3.a	Generate Comprehensive Test Suites	9
4.3.b	Example: Testing a Search Function	9
4.4	Documentation with AI	10
4.4.a	Auto-Generate JSDoc Comments	10
4.5	Debugging with AI	10
4.5.a	The Debugging Conversation Pattern	10
4.5.b	Debugging Example	11
5	Part 5: Ethics, Limitations, and Best Practices	11
5.1	Understanding LLM Limitations ...	11
5.1.a	What LLMs Can't Do	11
5.2	Agents & human oversight	12
5.3	Common Failure Modes	12
5.3.a	Hallucinations	12
5.3.b	Outdated Information	12
5.3.c	Overconfidence	13
5.4	Ethical Considerations	13
5.4.a	Copyright and Licensing	13
5.4.b	Academic Integrity	13
5.4.c	Bias in AI Outputs	13
5.5	Privacy and Security	13
5.5.a	Don't Share Sensitive Information	13
5.5.b	Data Retention Policies	13
5.6	Building Good AI-Assisted Development Habits	14
5.6.a	The 80/20 Rule	14
5.6.b	Continuous Learning	14
5.7	Future Trends (2025 and Beyond) ..	14
5.7.a	Emerging Capabilities	14

i Optional Advanced Material

This document provides **in-depth technical knowledge** about how LLMs work and advanced techniques for AI-assisted development.

Prerequisites: Complete the LLM Quick Start session and Practice 1-2 first.

When to read: When you're comfortable with basic AI-assisted development and want to understand the underlying technology and advanced patterns.

1 Part 1: Understanding LLMs

1.1 What Is a Language Model?

At its core, an LLM is a **probabilistic sequence model** that predicts the next token:

$$P(\text{token}_n \mid \text{token}_1, \text{token}_2, \dots, \text{token}_{\{n-1\}})$$

Example in code generation:

```
// Input: "function add(a, b) { return
a + "
// Most likely next token: "b"
// Less likely: ";", "1", "0"
// Very unlikely: "elephant"
```

The model assigns probabilities to all possible next tokens and samples from this distribution.

1.2 What Are Tokens?

Tokens are the **fundamental units** that LLMs process:

- **English text:** ~4 characters per token on average
- **Code:** More variable (keywords, operators, identifiers)
- **Special characters:** Often their own tokens

Examples:

```
"Hello, world!" → ["Hello", ",", " ", "world", "!"] (4 tokens)
"function    calculateTotal()" →
["function", " calculate", "Total",
"(", ")", "] (5 tokens)
```

1.2.a Why Tokens Matter

Context Windows are measured in tokens:

Model	Context Window	Approximate Pages
GPT-3.5	4,096 tokens	~3 pages
GPT-4	8,192 tokens	~6 pages
GPT-4 Turbo	128,000 tokens	~96 pages
Claude 3	200,000 tokens	~150 pages
Gemini 1.5 Pro	1,000,000 tokens	~750 pages

Practical impact: - Longer context = more information the model can “remember” - Token costs affect API pricing - Token limits affect how much code you can provide as context

1.3 The Transformer Architecture : Core Mechanism: Self-Attention

LLMs use **transformer architecture** with **self-attention mechanisms** that:

1. **Weigh relationships** between all tokens in the input
2. **Capture context** both before and after each token
3. **Learn patterns** at multiple scales (syntax, semantics, structure)

Simplified example:

Input: "The cat sat on the mat"

Self-attention learns:

- "cat" relates strongly to "sat" (subject-verb)
- "sat" relates to "mat" (verb-location)
- "on" connects "sat" and "mat" (preposition relationship)

1.3.a Why This Matters for Code

Self-attention allows the model to:

- **Understand variable scope** across multiple lines
- **Track function calls** and their arguments
- **Maintain consistency** in naming conventions
- **Recognize patterns** (e.g., “this pattern is a React hook”)

1.4 Training Process - Stage 1: Pre-training (Unsupervised)

The model learns from **massive datasets**:

- **Web pages:** Billions of documents
- **Books:** Fiction, non-fiction, technical
- **Code repositories:** GitHub, GitLab, etc.
- **Academic papers:** arXiv, PubMed, etc.

Objective: Predict the next token given previous context

Result: General understanding of language and code patterns

1.4.a Training Process - Stage 2: Fine-tuning (Supervised)

The model is trained on **curated datasets**:

- **Instruction-following:** “Write a function that...”, “Explain how...”
- **Code completion:** High-quality code examples

- **Q&A pairs:** Carefully crafted question-answer datasets

Objective: Make the model helpful for specific tasks

Result: Better instruction-following and task-specific performance

1.4.b Training Process - Stage 3: Alignment (RLHF)

Reinforcement Learning from Human Feedback:

1. **Generate multiple outputs** for the same prompt
2. **Human raters rank** outputs by quality
3. **Model learns** to prefer highly-ranked outputs
4. **Iterate** to improve helpfulness, harmlessness, honesty

Objective: Align model behavior with human preferences

Result: Safer, more helpful, more truthful outputs

1.5 The Current LLM Landscape (2025) - Major Models Comparison

Model	Provider	Strengths	Context	Best For
GPT-5	OpenAI	Enhanced reasoning, creativity	128K	Complex tasks, long context
GPT-4o	OpenAI	Improved performance,	128K	General tasks, long

Model	Provider	Strengths	Context	Best For
Claude 4	Anthropic	Improved understanding, safety	200K	Conversational AI, long context
Gemini 1.5 Pro	Google	Massive context, multi-modal	1M	Document analysis, video
Codex	OpenAI	Code-specialized	8K	Code generation, completion
Llama 3	Meta	Open-source, versatile	32K	Research, customization
Code Llama	Meta	Open-source, code-focused	16K	Local deployment, privacy

1.5.a Specialized Code Models

GitHub Copilot uses:

- OpenAI Codex (GPT-4 based)
- Fine-tuned on billions of lines of code
- Optimized for IDE integration

2 Part 2: Advanced Prompt Engineering

2.1 Chain-of-Thought (CoT) Reasoning

The Problem

LLMs sometimes “jump to conclusions” without showing reasoning steps.

Example:

```
Prompt: "Write a function to validate an email address"
Output: [generates regex without explanation]
```

2.1.a The Solution: Chain-of-Thought Prompt with CoT:

```
Write a function to validate an email address.
Think step-by-step:
1. What are the rules for valid email addresses?
2. What edge cases should we handle?
3. What's the best approach (regex, parsing, library)?
4. How do we test it?

Then provide the implementation with comments explaining each part.
```

Result: More thoughtful, better-documented code with reasoning

2.2 Few-Shot Learning

Provide 2-3 examples of the pattern you want:

Example: Generating test cases

```
Given this function, generate test cases:
```

Example 1:

```
Function: function add(a, b) { return a + b; }
```

Tests:

- add(2, 3) should return 5
- add(-1, 1) should return 0
- add(0, 0) should return 0

Example 2:

```
Function: function isEven(n) { return n % 2 === 0; }
```

Tests:

- isEven(2) should return true
- isEven(3) should return false
- isEven(0) should return true

Now generate tests for:

```
Function: function findMax(arr) { return Math.max(...arr); }
```

Result: The model follows the established pattern

2.3 Prompt Chaining : Breaking Complex Tasks into Steps

Instead of one massive prompt, chain multiple prompts:

Task: Create a full authentication system

Chain:

1. **Prompt 1:** “Design a database schema for user authentication with email/password”
2. **Prompt 2:** “Using this schema [paste from step 1], write SQL migration scripts”
3. **Prompt 3:** “Write Express routes for user registration and login using this schema [paste]”
4. **Prompt 4:** “Write frontend React components for login/register forms”
5. **Prompt 5:** “Integrate the frontend [paste] with backend [paste] using fetch API”

Benefits:

- Smaller, more focused outputs
- Easier to verify each step
- Can adjust based on intermediate results

2.4 Role Prompting - Give the AI a Persona

Basic Prompt:

```
"Explain React hooks"
```

Role-Enhanced Prompt:

```
"You are an experienced React developer teaching L3 university students. Explain React hooks in a way that builds on their existing JavaScript knowledge. Use practical examples and avoid jargon."
```

Result: More appropriate tone, level, and examples

2.4.a Useful Roles for Development

- **"You are a senior developer reviewing code..."** → Better quality checks
- **"You are a security expert auditing..."** → Security-focused analysis
- **"You are a tech writer documenting..."** → Clear, user-friendly docs
- **"You are a QA engineer testing..."** → Thorough edge case coverage

2.5 System Prompts vs User Prompts

System Prompt (set by the application): - Defines overall behavior and constraints - Usually hidden from the user - Examples: "You are GitHub Copilot, a coding assistant..."

User Prompt (your input): - Specific request or context - What you type in the chat or editor

2.5.a Using System Prompts (When Available)

Some tools let you set custom system prompts:

Example for a project:

System Prompt:

```
"You are a full-stack developer working on an e-commerce review aggregator."
```

```
The stack is: React, Node.js, Express, MySQL, Tailwind CSS."
```

Follow these conventions:

- Use async/await, not callbacks
- Use functional React components with hooks
- Follow REST API best practices
- Write accessible HTML with ARIA labels
- Use parameterized SQL queries to prevent injection"

User Prompt:

```
"Create a review card component"
```

Result: AI knows your project context and conventions automatically

3 Part 3: Advanced Techniques

3.1 RAG: Retrieval-Augmented Generation

LLMs have **knowledge cutoffs** and can't access:

- Your private codebase
- Recent updates
- Company-specific documentation
- Database contents

3.1.a The Solution: RAG

How it works:

1. **Index** your documents/code in a vector database
2. **Retrieve** relevant chunks based on your query
3. **Augment** the prompt with retrieved context
4. **Generate** response using this context

Example workflow:

User asks: "How do we handle authentication in our app?"

System:

1. Searches codebase for "authentication" files
2. Retrieves: auth.js, login.js, middleware/auth.js
3. Builds prompt: "Given this code: [paste files], answer: How do we handle authentication?"
4. LLM generates answer based on YOUR actual code

3.1.b Tools That Use RAG

- **GitHub Copilot Chat:** Can search your repo
- **Custom solutions:** LangChain, LlamaIndex

3.2 Embeddings and Semantic Search

What Are Embeddings?

Embeddings convert text/code into vectors (arrays of numbers) that capture **semantic meaning**:

```
"authentication" → [0.23, -0.45, 0.67, ...] (1536 dimensions)
"login system"   → [0.21, -0.43, 0.69, ...] (similar vector!)
"banana recipe"  → [0.89, 0.12, -0.34, ...] (very different vector)
```

3.2.a Why This Matters

You can find **semantically similar** code even with different wording:

Search query: "error handling" **Matches:** - "try-catch blocks" - "exception management" - "graceful failure recovery"

Even if exact words don't appear!

3.2.b Practical Application

When asking AI about your code:

1. Embedding model finds relevant files
2. Those files are added to prompt context
3. AI answers based on YOUR codebase, not generic knowledge

3.3 Temperature: Controlling Randomness

- Temperature is a parameter (0.0 to 1.0) that controls randomness in token sampling

Low temperature (0.0 - 0.3): Deterministic, focused

```
Prompt: "Complete: function add(a, b) {"
Temperature 0.1: "return a + b; }"
(always)
```

Medium temperature (0.5 - 0.7): Balanced

```
Temperature 0.5:
- "return a + b; }" (common)
- "return Number(a) + Number(b); }" (occasionally)
- "const sum = a + b; return sum; }" (rarely)
```

High temperature (0.8 - 1.0): Creative, diverse

```
Temperature 0.9:
- "return a + b; }"
```

```
- "return [a, b].reduce((x, y) => x + y);"
- "if (typeof a !== 'number') throw new Error()..."
- Etc. (very diverse outputs)
```

3.3.a When to Use Each

Temperature	Use Case	Example
0.0 - 0.2	Code completion, precise tasks	"Convert this SQL to a Sequelize query"
0.3 - 0.5	General coding, explanations	"Explain this algorithm"
0.6 - 0.8	Creative solutions, brainstorming	"Suggest 5 ways to improve this UI"
0.9 - 1.0	Highly creative tasks	"Generate unique product names"

Default for coding: Usually 0.2 - 0.4

3.4 Top-p (Nucleus Sampling)

How It Works : Instead of considering ALL possible tokens, consider only the **top p%** probability mass:

```
Top-p = 0.9 means:
- Rank all tokens by probability
- Consider only tokens that sum to 90% probability
- Sample from this subset
```

Result: Excludes very unlikely tokens while allowing some variety

3.4.a Temperature vs Top-p

Temperature: Changes the probability distribution shape **Top-p:** Truncates the distribution

Best practice: Adjust one OR the other, not both aggressively

4 Part 4: Production-Grade AI-Assisted Development

4.1 Code Review with AI

4.1.a Systematic Review Process

Step 1: Functionality Review

```
Prompt:
"Review this code for correctness and edge cases:
[paste code]"
```

```
Check for:
- Logic errors
- Edge cases (null, empty, invalid input)
- Off-by-one errors
- Race conditions (if async)"
```

Step 2: Security Review

```
Prompt:
"Security audit of this code:
[paste code]"
```

```
Check for:
- SQL injection vulnerabilities
- XSS vulnerabilities
- Authentication/authorization issues
- Sensitive data exposure
- Input validation gaps"
```

Step 3: Performance Review

```
Prompt:
"Analyze performance of this code:
[paste code]"
```

```
Identify:
- Big O complexity issues
- Unnecessary loops or operations
- Memory leaks (especially React/DOM)
- Database query optimization opportunities"
```

4.2 Refactoring Strategies

4.2.a Pattern: Incremental Refactoring

Don't ask for: "Refactor this entire file" **Do**

ask for: Specific, testable improvements

Example progression:

Step 1: Extract magic numbers

```
Prompt: "Replace magic numbers with
named constants"
Before: if (age > 18)
After:  const LEGAL_AGE = 18; if (age
> LEGAL_AGE)
```

Step 2: Extract functions

```
Prompt: "Extract this block into a
well-named function"
```

Step 3: Improve naming

```
Prompt: "Suggest more descriptive
variable names"
```

Step 4: Add error handling

```
Prompt: "Add proper error handling
with try-catch"
```

Each step is small, testable, and safe!

4.3 Testing with AI

4.3.a Generate Comprehensive Test Suites

Technique: Test-Driven Prompt Design

```
Prompt:
"I'm writing a function to [describe
functionality]."
```

1. First, generate a comprehensive list of test cases covering:
 - Happy path scenarios
 - Edge cases
 - Error conditions
 - Boundary values

2. Then write the test code using Jest

3. Finally, implement the function to pass all tests

```
Function      signature:      [provide
signature]"
```

Result: Tests are written BEFORE implementation (TDD approach)

4.3.b Example: Testing a Search Function

```
Prompt:
"Generate tests for a product search
function:"
```

```
function      searchProducts(products,
query) {
  // Returns products matching query
  in name or description
}
```

Include tests **for:**

- Case-insensitive matching
- Partial matches
- Empty query
- Empty product list

- Special characters
- Multiple word queries
- No matches found

Output:

```
describe('searchProducts', () => {
  const products = [
    { name: 'iPhone 13', description: 'Latest Apple phone' },
    { name: 'Samsung Galaxy', description: 'Android flagship' }
  ];

  test('finds products with case-insensitive match', () => {
    expect(searchProducts(products, 'iphone')).toHaveLength(1);
  });

  test('handles empty query by returning all products', () => {
    expect(searchProducts(products, '')).toEqual(products);
  });

  // ... more tests
});
```

4.4 Documentation with AI

4.4.a Auto-Generate JSDoc Comments

Prompt:

"Add comprehensive JSDoc comments to this function:
[paste function]

Include:

- Description of what the function does
- @param tags with types and descriptions

- @returns tag with type and description
- @throws tag for any errors
- @example showing usage"

Example output:

```
/**
 * Fetches product reviews from multiple sources and aggregates ratings
 *
 * @param {string} productId - Unique identifier for the product
 * @param {Array<string>} sources - Array of source names (e.g., ['amazon', 'bestbuy'])
 * @returns {Promise<Object>} Aggregated review data with average ratings
 * @throws {Error} If productId is invalid or sources array is empty
 *
 * @example
 * const reviews = await fetchReviews('prod-123', ['amazon', 'walmart']);
 * console.log(reviews.averageRating); // 4.2
 */
async function fetchReviews(productId, sources) {
  // Implementation...
}
```

4.5 Debugging with AI

4.5.a The Debugging Conversation Pattern

Step 1: Describe the problem clearly

"I'm trying to [expected behavior] but instead [actual behavior]."

Here's the relevant code:
[paste code]

Here's the error message:
[paste error]

What's causing this and how do I fix it?"

Step 2: Provide context if needed

"The function is called from [context].
It receives data in this format: [example]
I've verified that [what you've already checked]"

Step 3: Iterate based on suggestions

"I tried [suggestion] but now [new issue].
Here's the updated code: [paste]"

4.5.b Debugging Example

Prompt:
"My React component isn't re-rendering when state changes.

Code:

```
function ReviewList() {
  const [reviews, setReviews] =
    useState([]);

  async function fetchReviews() {
    const data = await fetch('/api/
reviews').then(r => r.json());
    reviews = data; // Bug is here!
  }

  return <div>{reviews.map(r =>
```

```
<ReviewCard key={r.id} review={r} /
>}</div>;
}
```

Error: Component doesn't update after
fetchReviews() is called.
What's wrong?"

AI Response:

"The issue is on line 6: you're
directly mutating the `reviews`
variable
instead of using the `setReviews`
setter function. React relies on
state
setters to trigger re-renders.

Fix:

```
- setReviews(data);
+ setReviews(data);
```

Explanation: React doesn't detect
direct assignments to state
variables..."

5 Part 5: Ethics, Limitations, and Best Practices

5.1 Understanding LLM Limita- tions

5.1.a What LLMs Can't Do

1. Access Real-Time Information

- Knowledge cutoff dates
- Can't browse current websites
- Can't check current package versions

2. Execute Code

- Can't test if generated code actually works
- Can't verify network requests
- Can't check database queries

3. Understand Business Context

- Doesn't know your company's requirements
- Can't prioritize features
- Can't make architectural decisions

4. **Guarantee Correctness**

- May generate syntactically correct but logically wrong code
- Can miss edge cases
- May suggest deprecated approaches

5.2 Agents & human oversight

Agents connect an LLM to external capabilities (retrieval, calculators, code execution, APIs). They can reduce hallucination and staleness and automate multi-step tasks — but they introduce new risks, so design guardrails up front.

- Core agent types
 - Retrieval (RAG) — return cited source chunks
 - Deterministic tools — calculators, test-runners for verification
 - Live APIs — controlled side-effecting calls (search, services)
 - Orchestrators — plan + execute multi-step workflows
- Safety patterns (minimum)
 - Principle of least privilege + full audit logs
 - Surface provenance and a confidence score with every answer
 - Cross-check critical facts with an independent retriever or API
 - Refuse or ask clarifying questions when confidence is low
 - Require human approval for side-effecting or high-risk actions
- Quick deployment checklist
 - Define scope & success metrics
 - Limit tool permissions and enable logging
 - Surface sources, confidence, and links for reviewers
 - Monitor for anomalies and add alerts

- Human sign-off for risky operations

Agents amplify productivity — treat them like instrumentation: design for auditability, testability, and clear human oversight.

5.3 Common Failure Modes

5.3.a Hallucinations

What happens: AI confidently generates false information

Example:

```
User: "How do I use the React useProductReviews hook?"
```

```
AI: "Sure! Here's how:
import { useProductReviews } from
'react';
const { reviews, loading } =
useProductReviews(productId);"
```

```
Problem: No such built-in hook
exists!
```

Defense: - Verify function/library names in official docs - Test code before committing - Cross-reference with authoritative sources

5.3.b Outdated Information

What happens: AI suggests deprecated or old approaches

Example:

```
AI: "Use componentDidMount for API
calls in React"
```

```
Problem: Hooks (useEffect) are the
modern approach
```

Defense: - Check publication dates of techniques - Prefer official documentation - Ask "Is this the current best practice in 2025?"

5.3.c Overconfidence

What happens: AI presents uncertain information as fact

Example:

```
User: "What's the best database for this use case?"
AI: "PostgreSQL is definitely the best choice."
Problem: Many factors determine "best" choice
```

Defense: - Ask for trade-offs: "What are pros and cons of different options?" - Request alternatives: "What are 3 options and when to use each?" - Make your own decisions based on requirements

5.4 Ethical Considerations

5.4.a Copyright and Licensing

The Issue: LLMs are trained on copyrighted code

Your Responsibility: - Understand your code's provenance - Check if generated code includes copyrighted snippets - Add proper licenses to your projects - Don't claim AI-generated code as entirely your own work

Best Practice:

```
// README.md
This project uses AI-assisted development tools (GitHub Copilot).
Generated code has been reviewed, tested, and modified for our specific needs.
```

5.4.b Academic Integrity

For Students:

Allowed ☑: - Using AI for boilerplate code - Getting explanations of concepts - Debugging assistance - Code suggestions as learning tools

Not Allowed ✗: - Submitting AI-generated code without understanding it - Using AI for exams without permission - Claiming AI work as entirely your own - Bypassing learning objectives

Always: - Disclose AI usage if required - Demonstrate understanding when asked - Use AI to learn, not to avoid learning

5.4.c Bias in AI Outputs

The Issue: LLMs can reflect biases from training data

Examples: - Gender assumptions in user models - Cultural assumptions in UI design - Accessibility oversights - English-centric examples

Mitigation: - Review generated code for assumptions - Test with diverse users - Explicitly ask for inclusive design - Add bias checks to your prompts

5.5 Privacy and Security

5.5.a Don't Share Sensitive Information

Never include in prompts: - API keys, passwords, tokens - Customer data or PII - Proprietary algorithms - Confidential business logic - Production database credentials

Example of safe prompt:

```
☑ "Write a function to authenticate users with JWT tokens"
✗ "Write auth using this secret key: sk_live_abcl23xyz..."
```

5.5.b Data Retention Policies

Know your tools:

Tool	Data Retention	Training on Your Code
GitHub Copilot	Not used for training (opt-in only)	No
ChatGPT	Conversations may train models	Yes (opt-out available)
Free Plus	Can disable training	Your choice
Claude	Not used for training	No
On-premise models	Data stays local	You control

Best Practice:

- Use tools with clear privacy policies
- Opt out of training when possible
- Never paste production secrets

5.6 Building Good AI-Assisted Development Habits

5.6.a The 80/20 Rule

AI should handle: 80% of boilerplate, routine tasks
You should handle: 20% of critical thinking, architecture, business logic

Ideal workflow:

1. You design the solution architecture
2. AI generates boilerplate and structure
3. You review and modify generated code
4. AI helps debug issues
5. You write tests and verify

functionality

6. AI generates documentation

7. You review everything before committing

5.6.b Continuous Learning

Don't let AI prevent learning:

- **Read generated code** — Don't just copy-paste
- **Ask “why?”** — Understand the reasoning
- **Experiment** — Modify AI suggestions and see what breaks
- **Compare** — Try multiple approaches
- **Research** — Look up unfamiliar patterns

Remember: Today's junior developer who learns with AI becomes tomorrow's senior developer who uses AI effectively.

5.7 Future Trends (2025 and Beyond)

5.7.a Emerging Capabilities

1. Multimodal Models

- Input: Images, video, audio, code
- Output: Cross-modal responses
- Example: “Debug this screenshot of an error”

2. Autonomous Agents

- Multi-step task execution
- Self-correction loops
- Example: “Build and deploy this feature”

3. Specialized Code Models

- Fine-tuned for specific frameworks
- Organization-specific models
- Example: Models trained on your company's codebase

4. Real-Time Collaboration

- AI pair programming
- Contextual suggestions during coding
- Example: AI that understands your entire project