AGENTS.md

Framework

Training Guide

Comprehensive AI & AGENTS.md Training

From LLM Fundamentals to Production-Ready AI Solutions

Generated: October 9, 2025

Open Standard

AI-Powered Development

Production-Ready

Table of Contents

1. Foundational Concepts: Understanding LLMs & AI Ager

Start here to build a solid foundation in AI fundamentals. Learn about Large Language Models (LLMs) and AI Agents before diving into AGENTS.md workflows.

The AI Landscape: LLMs, Clients, and Agents

The world of artificial intelligence is rapidly evolving, with Large Language Models (LLMs) at the core of many recent advancements. LLMs are advanced AI systems that can understand, process, and generate human-like text. They are built using deep learning techniques, specifically neural networks known as transformers, and are trained on vast amounts of text data.

What are Large Language Models (LLMs)?

At their heart, LLMs are statistical models that learn to predict the next word in a sequence based on the words that came before it. This seemingly simple task allows them to develop a sophisticated understanding of grammar, context, and even some degree of reasoning.

Key Characteristics & Capabilities:

1. **Scale and Capacity**
   LLMs are defined by their massive size, with billions or even trillions of parameters, which enables them to capture intricate patterns in language.

2. **Training Data**
   They are trained on huge, diverse datasets to learn about various topics, writing styles, and contexts.

3. **Versatile Capabilities**
   LLMs can perform a wide range of tasks, from language translation and text summarization to creative writing and code generation.

4. **Contextual Understanding**
   A key strength of LLMs is their ability to understand and leverage context to produce coherent and relevant responses.

Universal Truths: The Problem of Hallucinations

LLMs can generate incorrect, nonsensical, or biased information - a phenomenon known as "hallucination." Always verify critical information.

Example	Hallucination Type	Why It Happens
Inventing citations or research papers that don't exist	Factual Fabrication	Pattern matching from training data creates plausible but false references
Confidently stating incorrect dates or historical events	Temporal Confusion	Training data conflicts or knowledge cutoff limitations
Creating code that looks correct but contains subtle bugs	Logical Inconsistency	Statistical prediction without true semantic understanding

Context is Everything

The quality of an LLM's output is directly tied to the quality and clarity of the input. Well-designed prompts that are clear, specific, and provide examples drastically improve performance.

Poor Context	Rich Context	Outcome Difference
"Write a function"	"Write a TypeScript function that validates email addresses using regex, returns boolean"	Specific context yields production-ready code vs. ambiguous generic code
"Summarize this article"	"Summarize this article in 3 bullet points for a technical audience, focusing on implementation details"	Targeted summary vs. generic overview
"Help me debug this"	"This React component throws 'Cannot read property of undefined'. Here's the code, error stack, and what I've tried"	Precise diagnosis vs. generic troubleshooting suggestions

The Token Economy: The Core of LLM Value

Tokens are the fundamental unit that drives API costs, latency, and contextual fidelity. Understanding tokenization is critical to building cost-effective, production-grade LLM applications.

How Tokenization Works:

Modern LLMs use subword tokenization methods like Byte-Pair Encoding (BPE), which breaks words into familiar chunks (e.g., "jumping" !' "jump" + "ing"). This balances vocabulary size with semantic richr

- Handles unknown words: Decomposes "bitcoiners" or "crayzy" into known parts
- Smaller vocabulary: More efficient than storing every word form separately
- Preserves semantics: Unlike character-level tokenization which is too general

Token Budgeting & Pricing Models

Output tokens are typically 3× more expensive than input tokens due to autoregressive generation costs. Strategic input optimization is key to profitability.

Model Provider	Input (per 1M tokens)	Output (per 1M tokens)	Use Case
GPT-4o (OpenAI)	$5.00	$15.00	High quality, speed
Claude 3 Sonnet	$3.00	Included/Varies	Large-scale tasks
DeepSeek V3	$0.50–$1.50	Included/Varies	Budget-sensitive workflows

Token Efficiency: Practical Optimization Strategies

Context Window Management

- Recency window: Keep last N messages (e.g., most recent 5 turns)
- Importance-based: Retain messages with keywords, user questions, critical info
- Summarization: Compress old history into brief summary before threshold
- Hybrid: Recent messages (full) + older summary + initial system prompt

Output Length Constraints

- Set max_tokens to prevent runaway generation
- Add explicit constraints: "Respond in 2-3 sentences" or "Max 50 words"
- Use structured formats to naturally limit length (bullet lists, tables)
- Remember: Output tokens cost 3× more than input — aggressive constraint pays off

User-to-LLM Integration: RAG & Function Calling

Retrieval-Augmented Generation (RAG) and Function Calling extend LLM capabilities by connecting them to external data sources and tools.

Function Calling Use Cases:

- Fetch real-time weather/stock data
- Schedule appointments via calendar API
- Execute database queries
- Send emails or notifications

LLM Security: Vulnerabilities & Defense

LLMs face unique security challenges including prompt injection, data leakage, and adversarial attacks. Implement defense-in-depth strategies.

Input Validation

- Allowlist permitted characters/patterns
- Detect instruction-like phrases ("ignore", "forget")
- Length limits (prevent context overflow)

Output Monitoring

- Detect leaked system prompts
- Redact sensitive patterns (API keys, PII)
- Content moderation APIs

Best Practices for Building with LLMs and AI Agents

- Use clear, simple language in prompts
- Organize prompts into distinct, well-labeled sections
- Curate a small set of diverse, high-quality examples
- Guide the model's behavior rather than listing every edge case
- Tools should be well-documented and self-contained
- Start with a simple prototype using the most capable model
- Establish a performance baseline before optimizing
- Implement robust logging and monitoring for observability
- Track the agent's decision-making process to identify failure points

2. What is AGENTS.md?

AGENTS.md is an open standard that provides AI agents with structured, comprehensive context about your project. It acts as a single source of truth that eliminates ambiguity and ensures consistent AI behavior across all tools.

The Problem: Without AGENTS.md

- AI agents don't know your project structure
- AI agents don't know which commands to run
- AI agents don't know your code style preferences
- AI agents make incorrect assumptions about tech stack
- AI agents can't follow your security practices
- Developers waste time explaining the same things repeatedly

The Solution: With AGENTS.md

- AI agents know exactly how to build, test, and deploy
- AI agents follow your code style and security rules
- AI agents understand your project conventions
- Consistent AI behavior across GitHub Copilot, Cursor, Cline, etc.
- Onboarding new team members becomes faster

How It Works

1. Create AGENTS.md file in your project root
2. Document project structure, commands, and conventions
3. AI tools automatically read and follow the guidelines
4. Update AGENTS.md as your project evolves

What Goes in AGENTS.md?

- Project overview and tech stack
- Setup and build commands
- Code style rules (TypeScript, React, etc.)
- Testing instructions and coverage goals
- Security rules (validation, sanitization, auth)
- Deployment process and CI/CD workflows
- Database schema and migration commands
- API documentation and endpoints
- Environment variables and configuration

3. File Structure

A well-organized file structure is essential for maintainability and scalability. Here's the recommended structure for a full-stack TypeScript application with AGENTS.md integration.

```
project-root/
├── AGENTS.md                    # AI agent configuration
├── README.md                    # Human-readable documentation
├── package.json                 # Dependencies and scripts
├── tsconfig.json                # TypeScript configuration
├── vite.config.ts               # Build tool configuration
├── .env.example                 # Environment variable templa
├── src/
│   ├── App.tsx                  # Main application component
│   ├── main.tsx                 # Entry point
│   ├── components/              # React components
│   │   └── ui/                  # Reusable UI components
│   ├── services/                # API service layers
│   ├── types/                   # TypeScript type definitions
│   └── utils/                   # Utility functions
├── server/
│   ├── index.ts                 # Server entry point
│   ├── routes/                  # API routes
│   └── middleware/              # Express/Fastify middleware
├── prisma/
│   ├── schema.prisma            # Database schema
│   └── migrations/              # Database migrations
└── tests/
    ├── unit/                    # Unit tests
    └── integration/             # Integration tests
```

Key Organizational Principles

- Separation of Concerns: Keep frontend (src/), backend (server/), and database (prisma/) code separated
- Consistent Naming: Use kebab-case for files, PascalCase for components, camelCase for functions
- Colocation: Keep related files close together (e.g., Component.tsx, Component.test.tsx, Component.css)
- Index Exports: Use index.ts files to simplify imports and create clear public APIs
- Configuration Files: Keep all config files at the root for easy discovery

4. Tech Stack Decision Matrix

Choosing the right technology stack is crucial for project success. Here's a comprehensive decision matrix for common full-stack scenarios.

Framework	Best For	Learning Curve	Ecosystem
React	Large apps, component reusability, team with React experience	Medium	Largest ecosystem
Vue	Rapid prototyping, progressive enhancement, simpler learning	Low	Growing ecosystem
Svelte	Performance-critical apps, smaller bundle sizes	Low	Smaller but quality
Angular	Enterprise apps, large teams, opinionated structure	High	Enterprise-focused

Backend Framework Comparison

Framework	Language	Performance	Use Case
Express.js	JavaScript/TypeScript	Good	Flexible, minimal, widely adopted
Fastify	JavaScript/TypeScript	Excellent	High performance, modern features
NestJS	TypeScript	Good	Enterprise, Angular-like structure
Django	Python	Good	Rapid development, batteries included
FastAPI	Python	Excellent	Modern Python, async, auto docs

Database Selection Guide

Database	Type	Best For	Scaling
PostgreSQL	Relational	Complex queries, data integrity, JSON support	Vertical + Horizontal
MySQL	Relational	Traditional web apps, read-heavy workloads	Vertical + Read replicas
MongoDB	Document	Flexible schemas, rapid iteration, JSON-like data	Horizontal sharding
Redis	Key-Value	Caching, sessions, real-time analytics	Horizontal clustering
SQLite	Relational	Embedded, prototyping, small apps	Single-instance

Detailed Technology Stack Analysis

Stack #1: Modern SPA (React + Vite + Fastify)

Decoupled architecture with separate frontend and backend

Components: React 18 + Vite | Fastify (Node.js) | PostgreSQL + Prisma

Use Cases: Admin dashboards, internal tools, data-intensive SPAs, real-time applications where SEO is not critical

Stack #2: Full-Stack Framework (Next.js)

Monolith architecture with server-side rendering

Components: Next.js 14+ (SSR/SSG) | PostgreSQL + Prisma

Use Cases: Marketing websites, e-commerce platforms, blogs, content-heavy sites requiring excellent SEO

Stack #3: Lightweight Performance (Svelte + Express)

Minimal bundle size and high performance

Components: Svelte (compiled) | Express (Node.js) | SQLite

Use Cases: Performance-critical apps, lightweight tools, embedded systems, IoT devices, offline-first applications

Stack #4: Enterprise Cloud (Next.js + Azure)

High-compliance cloud-native stack

Components: Next.js + Azure Functions | Azure SQL Database

Use Cases: Large-scale enterprise applications, government projects, healthcare/finance apps requiring compliance (HIPAA, SOC 2)

Stack #5: Rapid Prototyping (Vue + Hono + Drizzle)

Fast development and low-cost deployment

Components: Vue 3 | Hono (Edge Workers) | SQLite + Drizzle

Use Cases: MVPs and prototypes, hackathon projects, landing pages, startups validating ideas quickly

5. AI Examples & Best Practices

Real-world examples of effective AI integration patterns and prompt engineering techniques for software development.

Example 1: Effective Code Generation Prompts

'L Poor Prompt:

"Create a login form"

' Better Prompt:

"Create a React TypeScript login form component with email/password fields, validation using Zod, accessible ARIA labels, error handling, and a submit button that calls an async login API. Follow Material Design guidelines."

Example 2: AI-Assisted Code Review

Effective Prompt: "Review this React component for: (1) Security vulnerabilities (XSS, injection), (2) Performance issues (unnecessary re-renders, memory leaks), (3) Accessibility (WCAG 2.2 AA compliance), (4) TypeScript type safety, (5) Code style violations per our ESLint config."

Example 3: Automated Documentation Generation

Effective Prompt: "Generate comprehensive JSDoc comments for this API service class. Include: (1) Function purpose and behavior, (2) Parameter types and descriptions, (3) Return types and possible values, (4) Error conditions and exceptions, (5) Usage examples with TypeScript, (6) Performance considerations."

Prompt Engineering Best Practices

- Be Specific: Include exact requirements, constraints, and expected output format
- Provide Context: Reference your tech stack, coding standards, and project conventions
- Use Examples: Show input/output pairs for complex transformations
- Iterate Gradually: Start with a simple prompt and refine based on results
- Include Constraints: Specify what NOT to do (e.g., "Don't use deprecated APIs")
- Request Explanations: Ask for reasoning to verify the AI's understanding

- Version Prompts: Track successful prompts in your AGENTS.md for consistency

6. AI Patterns

Common architectural patterns for integrating AI capabilities into applications, from simple prompt-response to complex multi-agent systems.

Simple Completion

Single prompt, single response. Best for straightforward tasks like text generation, classification, or simple Q&A.

Use Case: Content generation, sentiment analysis, basic chatbots

Chain-of-Thought (CoT)

Prompts the model to break down complex reasoning into steps. Dramatically improves accuracy for multi-step problems.

Use Case: Mathematical reasoning, logical puzzles, debugging assistance

Retrieval-Augmented Generation (RAG)

Retrieves relevant context from a knowledge base before generating responses. Grounds outputs in factual data.

Use Case: Documentation Q&A, customer support, research assistance

Agent with Tools

LLM decides which external tools to call (APIs, databases, calculators) to accomplish tasks. Autonomous task execution.

Use Case: Data analysis, API integration, workflow automation

Multi-Agent Systems

Multiple specialized agents collaborate. Each agent has a specific role (researcher, coder, reviewer) and they work together.

Use Case: Complex software projects, research tasks, content pipelines

Reflection Loop

Agent generates output, self-critiques it, then refines. Iterative improvement until quality threshold met.

Use Case: Code optimization, essay writing, design iteration

7. MCP Servers

Model Context Protocol (MCP) is an open standard that enables seamless integration between AI applications and external data sources. MCP servers provide AI agents with access to tools, data, and capabilities beyond their training.

Why Use MCP?

- Standardized Protocol: One integration works across all MCP-compatible AI tools
- Secure Access: Controlled permissions and authentication for sensitive data
- Real-Time Data: Connect to live databases, APIs, and services
- Extensible: Build custom MCP servers for your specific needs
- Ecosystem: Growing library of pre-built MCP servers for common services

Popular MCP Servers

Filesystem MCP

Read/write files, navigate directory structures

GitHub MCP

Access repositories, issues, pull requests, code search

Database MCP

Query SQL databases, inspect schemas, run migrations

Web Search MCP

Perform web searches, scrape content, fetch URLs

Slack MCP

Send messages, read channels, manage workspace

Google Drive MCP

Access documents, spreadsheets, presentations

8. AI Solutions & Integrations

Modern AI development tools and platforms that integrate with the AGENTS.md framework for enhanced productivity.

AI Code Assistants

- GitHub Copilot: Inline code suggestions, chat interface, CLI integration
- Cursor: AI-first code editor with codebase understanding
- Cline: Autonomous coding agent for complex tasks
- Continue: Open-source AI assistant for VS Code

AI Platforms

- OpenAI: GPT-4, GPT-4o, DALL-E, Whisper APIs
- Anthropic: Claude 3 family (Opus, Sonnet, Haiku)
- Google AI: Gemini models, PaLM, Vertex AI
- Hugging Face: Open-source models and datasets

Development Tools

- LangChain: Framework for building LLM applications
- LlamaIndex: Data framework for LLM applications
- Vercel AI SDK: React hooks and utilities for AI
- Pinecone: Vector database for RAG systems

9. References & Resources

Essential resources, documentation, and communities for continuing your AI development journey.

Official Documentation

- AGENTS.md Standard: https://github.com/ai-agents-team/agents.md
- OpenAI API Docs: https://platform.openai.com/docs
- Anthropic Claude Docs: https://docs.anthropic.com
- LangChain Docs: https://docs.langchain.com

Learning Resources

- Prompt Engineering Guide: https://promptingguide.ai
- Deep Learning AI Courses: https://deeplearning.ai
- Hugging Face Learn: https://huggingface.co/learn
- Fast.ai Practical Deep Learning: https://course.fast.ai

Communities

- r/LocalLLaMA (Reddit): Open-source LLM community
- AI Stack Exchange: Q&A for AI developers
- Discord: LangChain, Anthropic, OpenAI servers
- Twitter: Follow @karpathy, @sama, @danielgross

Tools & Utilities

- OpenAI Tokenizer: https://platform.openai.com/tokenizer
- Anthropic Workbench: https://console.anthropic.com
- Hugging Face Spaces: https://huggingface.co/spaces
- Weights & Biases: https://wandb.ai (experiment tracking)

10. WCAG 2.2 Accessibility Standards

Web Content Accessibility Guidelines (WCAG) 2.2 introduced new success criteria to improve accessibility for users with disabilities. All applications should strive for at least AA compliance.

Level A (Minimum)

- All non-text content has text alternatives (alt text for images)
- Captions provided for all prerecorded audio content
- Content can be presented in different ways without losing information
- Color is not used as the only visual means of conveying information
- All functionality available from keyboard
- Users can control time limits on content
- Content does not cause seizures (no flashing more than 3 times per second)

Level AA (Recommended)

- Captions for all live audio content
- Text contrast ratio of at least 4.5:1
- Text can be resized up to 200% without loss of functionality
- Images of text are avoided (use actual text when possible)
- Multiple ways to navigate to pages (search, sitemap, menu)
- Headings and labels are descriptive
- Focus visible on keyboard navigation
- WCAG 2.2 NEW: Focus Not Obscured (Minimum) - focus indicators never completely hidden
- WCAG 2.2 NEW: Dragging Movements - all drag operations have single-pointer alternatives
- WCAG 2.2 NEW: Target Size (Minimum) - touch targets minimum 24×24px
- WCAG 2.2 NEW: Accessible Authentication - no cognitive function tests for auth

WCAG 2.2 New Success Criteria (2023)

- 2.4.11 Focus Not Obscured (Minimum) - Level AA
- 2.4.12 Focus Not Obscured (Enhanced) - Level AAA
- 2.5.7 Dragging Movements - Level AA
- 2.5.8 Target Size (Minimum) - Level AA
- 3.2.6 Consistent Help - Level A
- 3.3.7 Redundant Entry - Level A
- 3.3.8 Accessible Authentication (Minimum) - Level AA
- 3.3.9 Accessible Authentication (Enhanced) - Level AAA

Key Takeaway

Accessibility is not optional. Build it into your development process from the start. Use automated testing tools (jest-axe, axe-core), conduct manual keyboard navigation tests, and involve users with disabilities in your testing process. AGENTS.md should include your accessibility standards to ensure AI-generated code meets compliance requirements.

11. Code Puppy - AI-Powered CLI Tool

Code Puppy is a primary AI-powered CLI tool for code generation, understanding, and multi-model orchestration. It supports the AGENTS.md standard and provides an interactive interface for working with multiple AI models simultaneously.

What is Code Puppy?

Purpose:	Primary CLI training tool for engineering teams
Type:	Interactive command-line interface
License:	MIT License (Open Source)

Installation

Requirements:

- Python 3.11 or higher
- pip (Python package manager)
- API keys for AI models (OpenAI, Gemini, Cerebras, Anthropic)

```
# Install via pip
pip install code-puppy

# Verify installation
code-puppy --version
```

Key Features

- Multi-Language Code Generation: Python, JavaScript, TypeScript, C++, Go, Rust, and more
- Interactive CLI Interface: Conversational mode for iterative development
- Round-Robin Model Distribution: Automatically distribute requests across multiple AI models
- Customizable Agent System: Define custom agents with Python or JSON
- AGENTS.md Support: Automatically reads and follows AGENTS.md instructions
- Code Quality Enforcement: Built-in quality principles enforced across all generated code

Usage Examples

Interactive Mode (Recommended for Training)

```
# Start interactive session
code-puppy --interactive

# In the interactive shell:
> write me a Python hello world program
> explain how async/await works in JavaScript
```

Direct Task Execution

```
# Execute a specific task
code-puppy "write me a C++ hello world program"

# Generate and execute in one command
code-puppy "create a Node.js Express server"
```

Interactive Commands

`/agent`
Check current agent or switch to a different agent configuration

`/truncate <N>`
Manage message history by truncating to last N messages

`/mcp`
Manage Model Context Protocol (MCP) servers for extended capabilities

`/exit or /quit`
Exit the interactive session

Configuration

Environment Variables (API Keys):

```
# OpenAI (GPT-4, GPT-3.5)
export OPENAI_API_KEY=your_openai_key

# Google Gemini
export GEMINI_API_KEY=your_gemini_key

# Anthropic Claude
export ANTHROPIC_API_KEY=your_anthropic_key
```

Supported AI Models

OpenAI
GPT-4, GPT-3.5 Turbo, Custom models

Google
Gemini Pro, Gemini Flash, Gemini models

Anthropic

Claude 3 Opus, Claude 3 Sonnet, Claude 3 Haiku

Cerebras

Fast inference, Custom models

Round-Robin

Auto-distribute, Load balancing, Cost optimization

Getting Started Guide

1. Install Code Puppy: Follow installation instructions above for your OS
2. Set up API Keys: Export your AI model API keys as environment variables
3. Start Interactive Mode: Run code-puppy --interactive to begin
4. Try Example Tasks: Start with simple tasks like "write hello world in Python"
5. Explore Advanced Features: Use custom agents, multi-model distribution, and AGENTS.md integration

12. Project Documentation - AGENTS.md & README.md

This section showcases the actual AGENTS.md and README.md files from this project, demonstrating real-world implementation of the AGENTS.md standard for AI agent collaboration and human-readable documentation.

AGENTS.md - AI Agent Instructions

File Location: ~/Projects/agents-md-demo/AGENTS.md

Version: 1.0.0 | Last Updated: 2025-10-07 | Framework: AGENTS.md Standard

Project Overview

- Modern full-stack application with enterprise-grade standards
- Tech Stack: React 18 + TypeScript + Vite + Fastify + PostgreSQL
- 80%+ test coverage requirement
- WCAG 2.2 AA accessibility compliance
- Supported by 20+ AI tools (Copilot, Cursor, Claude Code, etc.)

Code Style Guidelines

- TypeScript strict mode enabled (no any types)
- Functional components with hooks (no class components)
- All user inputs validated with Zod schemas
- No mock data in code (use database seed files)
- Environment variables for all configuration

Security Rules

- No secrets in code (environment variables only)
- Input validation with Zod for all endpoints
- Prisma ORM prevents SQL injection
- JWT authentication with expiration
- Password hashing with bcrypt (10+ rounds)

Testing Requirements

- 80%+ code coverage mandatory
- Unit tests for business logic
- Component tests with React Testing Library
- Integration tests for API endpoints
- Accessibility tests with jest-axe

README.md - Human Documentation

File Location: ~/Projects/agents-md-demo/README.md

Project: AGENTS.md Framework Demo | License: MIT | WCAG 2.2 AA Compliant

Features

- Beautiful UI - Modern, responsive design with Tailwind CSS
- Accessible - WCAG 2.2 Level AA compliant
- Secure - Input validation, authentication, security best practices
- Fast - Optimized build with Vite, fast API with Fastify
- Tested - Comprehensive test coverage with Vitest
- Well-Documented - AGENTS.md for AI agents, README for humans

Quick Start Commands

- git clone <repo-url> && cd agents-md-demo
- npm install
- cp .env.example .env (configure environment)
- npm run db:migrate && npm run db:seed
- npm run dev:all (starts frontend on :5175, backend on :5176)

Available Scripts

- npm run dev:all - Start both frontend and backend
- npm run build - Build for production
- npm test - Run all tests with coverage
- npm run lint - Check code quality
- npm run db:studio - Open database GUI

Performance Targets

- First Contentful Paint (FCP): < 1.8s
- Largest Contentful Paint (LCP): < 2.5s
- Bundle size: < 250KB gzipped
- `Lighthouse score: "e90 (all categories)`
- `Test coverage: "e80%`

Key Takeaway

The AGENTS.md file provides structured instructions for AI agents (code style, testing, security), while README.md offers human-readable documentation (setup, features, commands). Together, they ensure consistent behavior across 20+ AI tools and enable effective team collaboration. Both files are living documents that evolve with the project.