IISc Centre for Continuing Education

Agentic AI Coding

From Specification to Deployment

Course Reader

Prof. Deepak Subramani
Department of Computational and Data Sciences
Indian Institute of Science, Bengaluru

Cohort 2 · Feb–Apr 2027

Preview: Preface, Contents and Chapter 1

Draft v0.9 Preview | October 2026

Prof. Deepak Subramani, IISc. Preview for prospective Cohort 2 participants. Draft version, subject to revision.

Preface

This reader accompanies *Agentic AI Coding: From Specification to Deployment*, an eight-week applied course run by IISc's Centre for Continuing Education: seven weekends of live online classes followed by an optional in-person campus visit at IISc Bengaluru (hybrid for those who cannot travel). No previous experience with artificial intelligence or machine learning is required. Our central task is to learn how to specify what an application should do, ask a coding agent to implement it, and verify the result. Delegating code development makes precise requirements and informed review especially important. By the end of the course, we aim to build and evaluate an application in our own domain of interest.

The reader is organised in four parts.

- **Part I – Foundations** (Chapters 1–2) introduces AI, models, and agents. We distinguish rule-based from data-driven systems and predictive from generative tasks. We then explain language and multimodal models, and how a system uses a model to select actions, inspect their results, and check progress towards a goal.
- **Part II – Agentic Systems and Specification-Driven Development** (Chapters 3–4) explains the environment in which a coding agent works and the decisions we control: its instructions, available information, sequence of actions, and coordination with other agents. We develop specification-driven development (SDD) through examples involving a trip planner, a support bot, a review process, a computer-aided design assistant, and a café ordering agent. These examples show how to turn an idea into requirements and evidence of correctness.
- **Part III – Building, Testing and Deploying** (Chapters 5–7) follows the Musafir Travels application. We develop four connected specifications, review an implementation plan, test the resulting application, and work through cloud deployment and recovery. We then specify and add an AI-assisted trip-entry feature. The build uses Claude Code initially and Codex for the AI feature, showing how reviewed specifications support continuity across coding harnesses.
- **Part IV – Evaluation and System Design** (Chapters 8–9) develops evaluation of the running AI system through Musafir traces, representative cases, expert review and repeatable checks. We connect the resulting findings back to specifications and retain review of the development process. We use coding agents to compare designs, define component responsibilities, and plan for failures. We also distinguish using an agent to develop software from choosing to put an agent inside the running application.

How to use this reader. The banner at the start of each chapter identifies the relevant course weekend or weekends. Attempt each CHECKPOINT before consulting the chapter's Checkpoint Answers section. Use the End-of-Chapter Exercises to apply the ideas to our worked examples or to your own project.

If we are completely new to the subject, some terms may remain unclear on the first reading. We can mark an unfamiliar term, move past it for now, and continue reading to grasp the overall

Prof. Deepak Subramani, IISc. Preview for prospective Cohort 2 participants. Draft version, subject to revision.

concept. A major goal of professional education is to become comfortable with uncertainty during an initial reading, while learning how to resolve it. Understanding develops progressively: we return to difficult passages, consult the glossary and examples, ask questions, and revise our understanding over successive readings. We need not understand every term before beginning to see how the ideas fit together.

Appendix A provides the setup checklist. Complete the initial editor and coding-agent checks before Weekend 1; arrange deployment accounts before the deployment work. Appendix B provides a reusable specification template. Appendix C explains web applications and cloud deployment for readers new to those topics. The Glossary provides definitions of key terms.

The *Trip Planner Lab Guide* and the companion guides for the Support Bot, CAD Copilot, Review Loop, and Brews and Bytes provide additional practical examples. The Trip Planner in that guide is a separate teaching example from the recorded Musafir Travels build in Chapters 5–7; their features and deployment choices need not match. Use each guide with its own specifications. Section 9.12 suggests further directions.

Contents

Preface iii

I Foundations: AI and Agents 1

1 What Is Artificial Intelligence? 3
1.1 A Working Definition . . . 3
1.2 A Brief History of AI: Booms and Winters . . . 4
1.3 The AI "Snake Oil" Problem . . . 6
1.4 From Rules to Learning: The Machine Learning Mental Model . . . 6
1.5 Predictive AI versus Generative AI . . . 7
1.6 Foundation Models . . . 9
1.7 Language Models, Large Language Models, and Large Multimodal Models . . . 9
1.8 From Chat to Action: Where Agentic AI Begins . . . 10
1.9 Checkpoint Answers . . . 11
1.10 End-of-Chapter Exercises . . . 11

2 What Is an Agent? 13
2.1 Chat Interface versus Agentic System . . . 13
2.2 The Perceive–Reason–Act–Observe–Verify Loop . . . 14
2.3 The Spectrum of Agency . . . 15
2.4 Augmented Model Interactions (Level 1) . . . 17
2.4.1 How a Tool Call Is Actually Emitted . . . 17
2.5 The Model Context Protocol (MCP) . . . 18
2.6 Agentic Workflows (Level 2) . . . 20
2.7 Single-Agent Patterns and Full Autonomy (Level 3) . . . 21
2.8 Multi-Agent Systems . . . 22
2.9 A Spectrum, Not a Switch . . . 24
2.10 Checkpoint Answers . . . 24
2.11 End-of-Chapter Exercises . . . 24

II Agentic Systems and Specification-Driven Development 27

3 Agentic Coding Harnesses 29
3.1 What Is a Harness? . . . 29
3.2 How Claude Code Works: The Agentic Loop . . . 30
3.3 The Four Control Surfaces of a Coding Agent . . . 30
3.4 Seeing the Four Surfaces at Work: One Project, Four Builds . . . 33
3.4.1 Build 1: Prompt Only . . . 33
3.4.2 Build 2: Adding Context . . . 33
3.4.3 Build 3: Adding the Loop . . . 34

Prof. Deepak Subramani, IISc. Preview for prospective Cohort 2 participants. Draft version, subject to revision.

3.4.4 Build 4: Adding the Graph . . . 35
3.5 Choosing and Configuring a Harness . . . 36
3.5.1 Setting Up a Project: `/init` and `CLAUDE.md` . . . 36
3.5.2 The Command Surface . . . 37
3.6 Checkpoint Answers . . . 38
3.7 End-of-Chapter Exercises . . . 39

4 Specification-Driven Development: A Deep Dive 41
4.1 Vibe Coding vs. Specification-Driven Development . . . 41
4.2 The Architect–Contractor Mental Model . . . 42
4.3 Anatomy of a Good Specification . . . 42
4.4 The SDD Loop in Practice . . . 43
4.5 Writing Correct Tests from a Spec . . . 44
4.6 Five Worked Examples: Spec to Deployment . . . 45
4.6.1 Example 1: The Trip Planner (a Full-Stack Web App) . . . 45
4.6.2 Example 2: The Support Bot (an AI-Native App) . . . 46
4.6.3 Example 3: The Review Loop (a Multi-Agent System) . . . 47
4.6.4 Example 4: The CAD Copilot (a Cross-Disciplinary Example) . . . 47
4.6.5 Example 5: Brews and Bytes (An Evaluation-First Example) . . . 48
4.7 Failure Modes: Spec Drift, Scope Creep, and Unreviewed Plans . . . 49
4.8 How to Learn What Got Built . . . 50
4.9 Why This Matters . . . 50
4.10 Checkpoint Answers . . . 51
4.11 End-of-Chapter Exercises . . . 51

III Building, Testing and Deploying 53

5 Developing a Full-Stack Application from Four Specs 55
5.1 The Essential Web-Application Model . . . 55
5.1.1 Following One User Action . . . 56
5.1.2 What This Means for a Specification . . . 56
5.2 Beginning with Four Documents . . . 56
5.3 Developing the Goal Specification . . . 57
5.4 Developing the Front-End Specification . . . 60
5.4.1 Understanding Choices Before Accepting Them . . . 60
5.4.2 Reconciling Mockups with Requirements . . . 62
5.4.3 Writing Observable Behaviour . . . 63
5.5 Developing the Back-End Specification . . . 63
5.5.1 Turning an Unfamiliar Term into a Decision . . . 64
5.5.2 Changing a Constraint Deliberately . . . 64
5.6 Developing the API Contract . . . 64
5.6.1 Product Decisions and Their Encoding . . . 64
5.6.2 Enough Precision for Both Sides to Agree . . . 65
5.7 Reviewing the Four Specifications Together . . . 65
5.7.1 One Decision, Four Changes: Signup . . . 66
5.7.2 A Review Students Can Repeat . . . 67
5.8 From Reviewed Specifications to Implementation . . . 68
5.8.1 Why Local Development Comes First . . . 69
5.8.2 Tracing Requirements into Implementation Choices . . . 69
5.8.3 Reading the “Done When” Conditions . . . 69
5.9 Verification Changes the Specification . . . 70

5.10 Starting Your Own Project . . . 71
5.11 Checkpoint Answers . . . 72
5.12 End-of-Chapter Exercises . . . 72

6 Testing, CI/CD and Cloud Deployment 75
6.1 What Changes When We Leave the Laptop? . . . 75
6.1.1 Reviewing a Phase's Completion Claim . . . 76
6.2 From Acceptance Criteria to Test Suites . . . 76
6.3 Checking the Agent's Tests . . . 77
6.3.1 Reading the Result . . . 77
6.4 Continuous Integration: Repeating the Checks . . . 78
6.5 Phase 4: Turning the Deployment Spec into Infrastructure . . . 78
6.5.1 Reading the Architecture . . . 79
6.5.2 Configuration, Plan, Apply, and State . . . 79
6.5.3 A Failed Apply Can Leave Real Resources Behind . . . 80
6.6 Phase 5: Automating Backend Deployment . . . 80
6.6.1 The Workflow Has to Prove Its Identity . . . 81
6.6.2 What a Green Pipeline Establishes . . . 82
6.7 Phase 6: Making the Browser Journey Work . . . 83
6.7.1 An HTTPS Frontend Cannot Use Our HTTP API . . . 83
6.7.2 API Address, CORS, and Configuration . . . 84
6.7.3 Verifying the Application, Not Only Its Health Endpoint . . . 84
6.8 Phase 7: Turning Data Preservation into a Requirement . . . 85
6.8.1 What Must Survive? . . . 85
6.8.2 Expressing Recovery as Acceptance Criteria . . . 86
6.8.3 Proving the Restore Exposed More Missing Requirements . . . 86
6.8.4 Know What Has Actually Finished . . . 87
6.9 What We Can Claim at the End . . . 87
6.9.1 Does This Application Need So Many Resources? . . . 88
6.10 Shipping Our Own Application . . . 88
6.11 Checkpoint Answers . . . 88
6.12 End-of-Chapter Exercises . . . 88

7 Adding AI Features to Musafir Travels 91
7.1 What Are We Adding to Musafir? . . . 91
7.2 Feature One: Describe, Review and Create a Trip . . . 92
7.3 Develop the Specification with the Coding Agent . . . 93
7.3.1 Resolve Decisions One at a Time . . . 93
7.3.2 Turn Date Conventions into Examples . . . 94
7.3.3 Choose the Model and Its Integration . . . 94
7.3.4 Specify the Unhappy Paths Before Implementation . . . 95
7.4 How the Application Turns Words into a Trip . . . 96
7.4.1 Follow One Request Through the Backend . . . 96
7.4.2 Return Fields the Application Can Check . . . 97
7.4.3 Review and Confirmation Control Saving . . . 97
7.4.3.1 Restoring a Draft Without Replaying an Action . . . 98
7.4.4 Technical Detail: The SDK Call . . . 98
7.4.5 Prompt Rules and Code Checks Have Different Jobs . . . 99
7.5 Implement and Verify the First Feature . . . 99
7.5.1 Test the Integration and Evaluate the Model Separately . . . 100
7.6 Feature Two: Help Us Plan the Days . . . 101
7.7 Specify the Planner and Its Search Tools . . . 101

7.7.1 Resolve the Product Decisions . . . 101
7.7.2 Record the Agreed Behaviour . . . 102
7.7.3 Let the Coding Agent Propose the Tool Integration . . . 103
7.7.4 A Bounded Workflow with Model-Selected Searches . . . 103
7.7.5 Limits Are Separate Design Decisions . . . 104
7.7.6 Review the Implementation Plan and Refine the Interface . . . 105
7.8 Use the Working Planning Application . . . 106
7.9 What We Must Still Check . . . 109
7.10 Deploying the Extended Application . . . 109
7.11 From Building to Evaluating . . . 110
7.11.1 Extensions Still Requiring Their Own Specifications . . . 110
7.12 Checkpoint Answers . . . 110
7.13 End-of-Chapter Exercises . . . 111

IV Evaluation and System Design 113

8 Evaluating Agentic Systems 115
8.1 A Working Application Can Still Disappoint . . . 115
8.1.1 An Earlier Failure That Changed the Specification . . . 117
8.2 The Three Gulfs: Understanding, Specifying and Generalizing . . . 118
8.3 Observability: Follow One Request Through Musafir . . . 119
8.3.1 Specify Observability Before Implementing It . . . 119
8.3.2 Decide What We Capture and Retain . . . 120
8.3.3 Read the Trace as a Sequence of Decisions . . . 120
8.3.4 Open the Actual P1 Trace . . . 121
8.3.5 Contrast It with a Request That Actually Searched . . . 121
8.3.6 A Failure Can Expose Missing Evidence . . . 123
8.4 Sample the Behaviour We Need to Understand . . . 123
8.4.1 From a Combination to a Realistic Request . . . 124
8.4.2 Our Five Live Cases: Prompts, Expectations and Results . . . 124
8.4.3 Discovery Samples and Usage Estimates . . . 126
8.5 Open Coding: Describe What We Observe . . . 127
8.6 Axial Coding: Turn Notes into Useful Failure Categories . . . 128
8.7 Who Decides What Counts as Good? . . . 129
8.7.1 Use Disagreement to Improve the Specification . . . 129
8.7.2 The Expert's Role Continues After Labelling . . . 130
8.8 Turn Expectations into Evaluators . . . 130
8.8.1 Use Code for Rules Code Can Check . . . 130
8.8.2 Judge the Workflow as Well as Its Answer . . . 131
8.8.3 Keep Unknowns Visible . . . 131
8.9 Use a Model Judge After Defining the Judgement . . . 132
8.9.1 Calibrate Against Expert-Reviewed Cases . . . 132
8.9.2 Budget the Evaluation Separately . . . 132
8.10 Compare Changes and Preserve What Already Works . . . 133
8.10.1 Build a Golden Set from Reviewed Examples . . . 133
8.10.2 Apply the Unit-Testing Habit to AI Behaviour . . . 133
8.10.3 Use Phoenix as a Connected Review Workspace . . . 134
8.10.4 Control What Changes in a Comparison . . . 135
8.10.5 Define the Release Decision Before Looking at the Result . . . 135
8.11 Automating Evaluation Design and Improvement . . . 135
8.11.1 Review the Data Before Accepting an Automated Evaluator . . . 136

8.11.2 Hill Climbing as a Controlled Experiment . . . 136
8.11.3 Choose Model and Thinking Level Through Evaluation . . . 138
8.12 Retain Ownership of the Application and Its Evaluation . . . 138
8.13 Checkpoint Answers . . . 140
8.14 End-of-Chapter Exercises . . . 140

9 High-Level System Design with Agentic AI 143
9.1 What High-Level Design Must Explain . . . 143
9.1.1 Turn Adjectives into Conditions We Can Check . . . 144
9.2 Choosing How Much AI the Task Needs . . . 144
9.2.1 A Decision Flowchart for Choosing a Design . . . 145
9.2.2 The Minimal Viable Agent . . . 147
9.3 Applying the Framework to Different Applications . . . 147
9.3.1 A Course-Reader Assistant . . . 147
9.3.2 A Support Bot: Evidence, Private Data and Actions . . . 148
9.3.3 Document Review Against a Checklist . . . 149
9.3.4 Comparing the First Designs . . . 149
9.3.5 A Research Assistant: Make and Defend a Choice . . . 149
9.4 Using the Coding Agent as a Design Collaborator . . . 150
9.4.1 Record the Decision and Its Consequences . . . 150
9.5 Drawing Boundaries We Can Explain . . . 151
9.5.1 C4: Four Levels of Architectural Detail . . . 151
9.5.2 Contracts Include Failure Behaviour . . . 153
9.6 State, Tools, and Permission to Act . . . 153
9.6.1 A Small Tool Contract . . . 153
9.6.2 Treat Retrieved Text as Data . . . 154
9.7 Designing for Partial Failure . . . 154
9.7.1 Deadlines, Loops, and Honest Completion . . . 154
9.7.2 A Timeout Does Not Mean a Write Failed . . . 155
9.7.3 Queues and Background Work . . . 155
9.8 Latency, Cost, and Evidence for Complexity . . . 156
9.8.1 Observe Outcomes and the Request Path . . . 156
9.9 When to Bring in Multiple Agents . . . 157
9.10 Developing a Reviewable System Design . . . 157
9.11 Final Project Presentations . . . 158
9.12 Where to Go from Here . . . 159
9.13 Checkpoint Answers . . . 159
9.14 End-of-Chapter Exercises . . . 159

A Setup Instructions 163
A.1 Hardware and Connectivity . . . 163
A.2 Accounts and When We Need Them . . . 163
A.3 Step 1: Install Visual Studio Code . . . 164
A.4 Step 2: Install Claude Code . . . 164
A.5 Step 3: Sanity-Check Your Harness . . . 166
A.6 Step 4: Let Claude Code Install the Rest of Your Toolchain . . . 167
A.7 What Happens Next . . . 167

B Specification Template 169
B.1 The Template . . . 169
B.2 Guidance for Each Field . . . 170
B.3 A Filled-In Example: An Illustrative Trip Planner Goal Spec . . . 171

C Optional: Web Applications and Cloud Deployment **173**
C.1 What Runs in the Browser? . . . 173
C.2 What Happens on the Server? . . . 174
C.2.1 Addresses, Requests, and Responses . . . 174
C.3 Where the Data Lives . . . 175
C.3.1 Changing a Database: SQLAlchemy and Alembic . . . 175
C.4 Packaging an Application with Docker . . . 176
C.5 Serving the Application to Users . . . 176
C.6 Terraform: Infrastructure as Code . . . 177
C.7 Kubernetes and Helm . . . 177
C.8 Development, Staging, and Production . . . 178
C.9 What Changes at Scale? . . . 179
C.9.1 Adding AI Features and Durable Agent Work . . . 179
C.10 Connecting the Vocabulary Back to Specifications . . . 180

Glossary **181**

Part I

Foundations: AI and Agents

Prof. Deepak Subramani, IISc. Preview for prospective Cohort 2 participants. Draft version, subject to revision.

1

What Is Artificial Intelligence?

Maps to Weekend 1 — From LLMs to Autonomous Agents (Feb 13–14, 2027)

AI systems appear in applications as different as medical imaging, fraud detection, and coding assistance. The label alone tells us little about how a system works or what it can reliably do. We begin by distinguishing its task, its underlying mechanism, and the decisions it can make without our intervention.

1.1 A Working Definition

We use **intelligence** here to refer to capabilities such as learning, reasoning, and applying knowledge to a task. These capabilities need not arise in the same way. A computer can apply rules supplied by people, learn patterns from data, or combine both approaches.

DEFINITION: Artificial Intelligence

Artificial Intelligence (AI) concerns computer systems that perform tasks associated with intelligence, such as perception, reasoning, learning, and planning. A system need not learn from data to qualify: applying explicitly supplied knowledge is also part of AI.

DEFINITION: ANI, AGI, and ASI: The Breadth Spectrum

Artificial Narrow Intelligence (ANI) addresses a specific task or a bounded range of tasks.
Artificial General Intelligence (AGI) refers to a proposed level of broad, transferable capability comparable to human intellectual capability. Definitions and thresholds differ.
Artificial Superintelligence (ASI) refers to capability exceeding human performance across a broad range of domains. These labels describe ambitions and categories, not a precise measurement scale. We assess a particular system through its demonstrated capabilities and limitations.

Term	Breadth	Performance claim
ANI	A bounded range of tasks	Assessed on those tasks
AGI	Broad transfer across intellectual tasks	Proposed human-comparable capability
ASI	Broad capability across domains	Proposed performance beyond humans

Table 1.1: ANI, AGI, and ASI distinguish breadth and level of capability. They do not form a calibrated scale for ranking current systems.

Prof. Deepak Subramani, IISc. Preview for prospective Cohort 2 participants. Draft version, subject to revision.

An AI system's task and its mechanism are different questions. We first distinguish two ways of producing its behavior.

DEFINITION: Rule-Based AI

In **rule-based AI**, people specify the rules used to reach a decision. An expert system can apply a rule such as "if these symptoms are present, investigate this condition." In the fixed rule-based systems considered here, learning from examples does not change those rules. **Symbolic AI** represents knowledge explicitly, for example through symbols and logical relationships; rule-based systems are one approach within it. Other systems combine explicit rules with learned components.

DEFINITION: Data-Driven AI

Machine Learning (ML) uses data or experience to adjust a model's internal **parameters**: numbers that influence its output. During **training**, an algorithm adjusts them to improve a specified objective. In supervised learning, that objective measures how well predictions match example answers. After training, using the model to produce outputs for new inputs is called **inference**.

The source of behavior does not determine its autonomy. A rule-based chess program can select moves without intervention, just as a learned one can. A coding system can combine a learned language model with hand-written rules that control tool access and execution. Chapter 3 examines that surrounding software.

1.2 A Brief History of AI: Booms and Winters

AI research has experienced periods of rapid investment and periods of reduced funding, often called **AI winters**. The milestones below provide context for evaluating capability claims. They do not imply that progress followed one cause or one uninterrupted sequence.

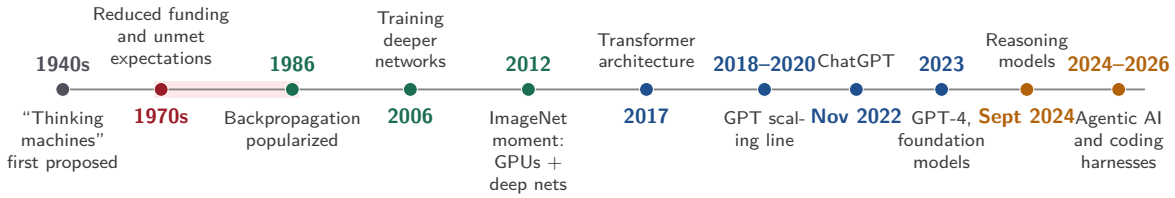


Figure 1.1: Selected AI milestones, 1940s–2026. The colors group early ideas, funding setbacks, neural-network developments, language models, and recent agentic coding tools.

- **1940s–1950s: early foundations.** Researchers develop mathematical models of neural computation and explore whether computers can perform tasks associated with thought.
- **1970s: funding setbacks.** Some ambitious AI projects fall short of expectations, and funding support reduced in parts of the field. This funding history is broader than the limitations of any one learning algorithm.[1]
- **1986: backpropagation becomes influential.** A **neural network** combines parameterized computations in layers. Backpropagation computes how each parameter contributes to training error, so an optimization algorithm can adjust it. Its use in learning internal representations helps renew interest in multilayer networks.[2]

[1] For a historical overview, see the Computer History Museum's AI and robotics timeline.
[2] Rumelhart, Hinton, and Williams, *Learning representations by back-propagating errors* (1986).

- **2006: training deeper networks.** Work on layer-by-layer training helps renew interest in networks with multiple intermediate layers. **Deep learning** refers to learning with such multilayer models, not merely a change of name.[3]
- **2012: image recognition.** A deep network achieves a major improvement on the ImageNet image-classification benchmark. A **benchmark** is a defined task and evaluation dataset used to compare systems. The work combines a large labeled dataset with effective training methods and GPUs, processors suited to many parallel numerical calculations.
- **2017: the Transformer.** A Transformer uses **attention** to combine information from relevant positions in a sequence. It allows substantial parallel computation during training. An autoregressive Transformer still generates a response one token at a time, using the tokens already available.[4]
- **2018–2020: scaling language models.** The GPT series explores larger models and training datasets. A **scaling law** describes an empirical relationship between quantities such as model size, training data, computation, and predictive loss. It does not guarantee improvement on every task.
- **November 2022: ChatGPT.** A conversational interface brings language-model interaction to a broad public audience.
- **2023: broader model use.** GPT-4 and other model families expand interest in using one broadly trained model for multiple tasks. We define foundation models below.
- **2024: reasoning models.** Models such as o1 draw attention to spending additional computation before returning an answer. The relevant question is how that computation changes performance and cost on a specified task.
- **2024–2026: agentic coding tools.** Products increasingly combine models with tools for inspecting files, changing code, and checking results. The surrounding **harness** manages these interactions. Its permission policy determines which actions require approval. This period marks a product development relevant to our course, not the origin of all research on agents.

EXAMPLE 1.1: A Limitation of a Simple Learning Model

A single-layer **perceptron** separates inputs using a linear boundary: a line in two dimensions, or its higher-dimensional counterpart. It cannot represent XOR. For binary inputs, XOR returns 1 when the inputs differ and 0 when they match: $(0,0) \mapsto 0$, $(0,1) \mapsto 1$, $(1,0) \mapsto 1$, and $(1,1) \mapsto 0$. No single straight line separates those two classes. Networks with nonlinear intermediate layers can represent this pattern. The limitation belongs to the model class, not to every early neural network.

IMPORTANT: Why This History Matters

The history gives us a reason to compare capability claims with evidence from the task we need to solve. A successful demonstration is a useful observation. Reliable performance across representative cases requires further evaluation.

[3] Hinton, Osindero, and Teh, A fast learning algorithm for deep belief nets (2006).
[4] Vaswani et al., Attention Is All You Need (2017).

1.3 The AI "Snake Oil" Problem

An "AI-powered" label is not evidence of effectiveness. Narayanan and Kapoor's *AI Snake Oil* examines the gap between AI claims and demonstrated capability.[5] We use the following three questions as this reader's framework for describing a system before evaluating it.

Q1. What task does the system perform? State the inputs, the required result, and the conditions in which it will be used. Human effort is not a reliable test of whether a task is easy to automate.

Q2. Where does its behavior come from? Identify explicit rules, learning from data, or a combination of both.

Q3. Which decisions does it make without intervention? Distinguish a fixed procedure from a system that selects its next step using intermediate results.

IMPORTANT: Read Critically

These questions describe a system; they do not establish that it works. We also ask what capability is claimed, what evidence supports it, and whether the evaluation represents the intended use. Rule-based systems can be AI.

A demonstration that succeeds once does not establish that an agent reliably meets a written specification. Chapter 4 develops a method for making that distinction checkable.

1.4 From Rules to Learning: The Machine Learning Mental Model

In a prediction problem, we collect inputs and estimate a quantity of interest. A machine-learning model represents a mapping from those inputs to an output. Training adjusts the mapping using examples or experience.

DEFINITION: Model

A **model** is a mathematical function f that maps an input to an output. In a parameterized model, adjustable numbers (called as parameters) determine the particular mapping. **Training**, or **fitting**, selects parameter values using a stated objective. Good performance on training examples alone does not establish performance on new inputs.

The same idea applies to a fitted straight line and to a language model with many parameters. Their model classes and training objectives differ, but both use parameters learned from data.

[5] Arvind Narayanan and Sayash Kapoor, *AI Snake Oil: What Artificial Intelligence Can Do, What It Can't, and How to Tell the Difference* (Princeton University Press, 2024).

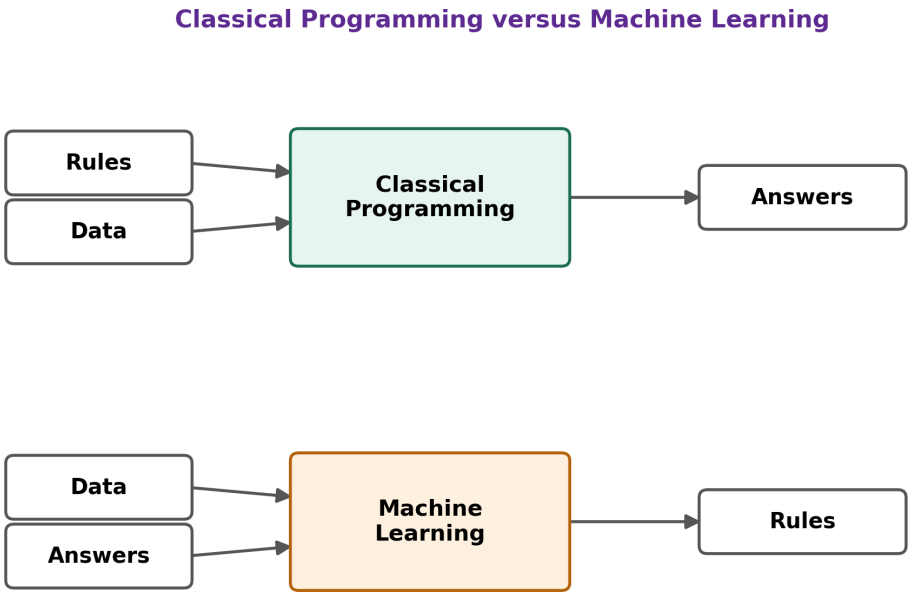


Figure 1.2: Classical programming (top): rules and data go in, and the program computes answers. Machine learning (bottom): data and answers (examples) go in, and training produces the rules — exactly the inversion the Celsius-to-Fahrenheit example below demonstrates.

EXAMPLE 1.2: Traditional Programming versus Machine Learning

Suppose we want to convert Celsius to Fahrenheit. In **traditional programming**, we supply the rule $F = 1.8C + 32$. The program uses input data but requires no training data. This is a conventional calculation; we need not label every fixed-rule program AI.
Alternatively, we can fit a line, $\hat{F} = wC + b$, to paired readings without supplying the conversion formula. **Least squares** chooses w and b to minimize the sum of squared prediction errors. Exact pairs at distinct Celsius values recover the conversion rule; noisy measurements generally give an approximation. The example illustrates learning a relationship within a chosen model class. It does not guarantee that fitting observations discovers the true mechanism.

CHECKPOINT 1.1

A spreadsheet macro sorts a column of numbers in ascending order when a button is clicked. Walk through Q1–Q3 above for this macro. Would you call it AI? Now repeat for a plugin that predicts next month's sales by fitting a trend line to two years of data. How do your answers change, and why?

Before the next definition, a **vector** means an ordered collection of numbers, and $\mathbb{R}^n$ denotes the set of vectors with n real-valued entries. A **modality** is a type of data, such as text or images. A **token** is an encoded unit of text, often a word fragment or punctuation mark. The model's **vocabulary** is its set of token identities. A probability distribution assigns probabilities to possible outcomes.

1.5 Predictive AI versus Generative AI

Machine learning systems are divided into two families by the shape of things they produce.

DEFINITION: Predictive AI vs. Generative AI, Formally

Let x denote an input of any data modality. A **Predictive AI** model computes $f(x) \in \mathbb{R}^n$, where n (the number of classes, or the number of continuous targets) is *fixed at training time*. A **Generative AI** model computes $G(x, z) \in \mathcal{S}$, where $\mathcal{S}$ is a space of sequences, images, or signals whose length or resolution is *not* fixed at training time, and z is a source of randomness that lets the same x produce different valid outputs on different calls.

A spam classifier or a house-price predictor is Predictive AI: the output shape (spam/not-spam; one price) is fixed before a single example is seen. A large language model is Generative AI: it produces a token sequence of whatever length the answer needs, one token at a time, each one sampled from a probability distribution over its vocabulary and fed back as input for the next token. This **autoregressive** mechanism is why two calls to the same chatbot with an identical prompt can produce two different answers: the randomness z in the definition above means that each generated token is a draw, not a deterministic lookup. Diffusion models, the dominant mechanism behind modern image and video generators, produce an output in the same variable-size way but by a different route: refining an entire image simultaneously across many denoising steps rather than emitting it one token at a time.

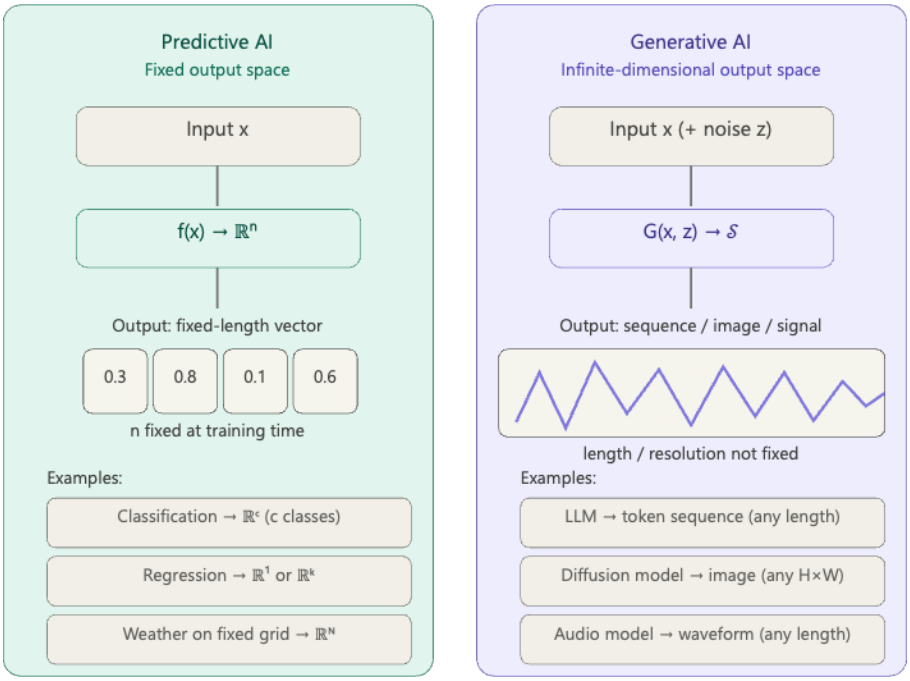


Figure 1.3: Predictive AI produces a fixed-length output vector whose dimension n is fixed at training time. Generative AI produces an output — a sequence, an image, a signal — whose length or resolution is *not* fixed at training time.

CHECKPOINT 1.2

If a commercial chatbot used *greedy* decoding (always emitting the single most likely next token, with no randomness) instead of sampling, would two calls with the identical prompt still be guaranteed to produce the same answer? What does this imply about why "the model said something different this time" is not, by itself, evidence of a bug?

1.6 Foundation Models

A broadly trained model can be reused for multiple subsequent tasks. This reuse can support prediction and generation. It is a separate property from the output-dimensionality distinction used above.

DEFINITION: Foundation Model

A **foundation model** is trained on broad data and designed to support adaptation to multiple tasks. A *downstream task* is a later application of that model, such as answering questions or classifying documents.[a]

[a] Bommasani et al., On the Opportunities and Risks of Foundation Models (2021).

Pre-training learns general patterns from broad data. **Post-training** further shapes behavior for intended uses, for example following instructions. Its objectives and computational scale depend on the model. We do not need the training algorithms to use these models, but we do need to distinguish changing their parameters from supplying information in a request.

We can adapt how an application uses a model in several ways. These approaches can be combined; their cost depends on the model, data, infrastructure, and volume of use.

- **Prompt engineering**: supplying instructions, examples, and relevant information in the model's input without changing its parameters.
- **Retrieval-Augmented Generation (RAG)**: the application retrieves relevant information from an external source and includes it in the context used to generate an answer. The source can be updated without retraining the model.
- **Fine-tuning**: further training that updates some or all model parameters using task-specific data. It changes the model itself rather than only the information supplied for one answer.

The support-bot example in Chapter 4 uses retrieval because its information source changes. Project specifications also provide context for a coding agent. Reading a specification is not fine-tuning: it does not update the model's parameters.

CHECKPOINT 1.3

A hospital needs an assistant to answer authorized questions using this week's patient records. The records must not be used to train the model. How can retrieval or directly supplied context meet the information requirement? Why is fine-tuning unsuitable, and what privacy questions remain?

1.7 Language Models, Large Language Models, and Large Multimodal Models

We next distinguish a model's language capabilities from the kinds of input and output it supports.

DEFINITION: Language Model

A **language model** assigns probabilities to language sequences. A next-token language model estimates the next token given the preceding tokens. A phone keyboard's next-word suggestions illustrate a related use. The term does not impose a minimum model size.

DEFINITION: Large Language Model (LLM)

A **Large Language Model (LLM)** is a language model trained at substantial scale, using many parameters and a broad dataset. Such models can support tasks including summarization and code generation. Scale matters, but data, training methods, and post-training also influence capability.

DEFINITION: Large Multimodal Model (LMM)

A **Large Multimodal Model (LMM)** processes more than one data type, such as text and images. Some multimodal systems extend language models; others use different architectures. Supported inputs and outputs must be checked for the particular model. A multimodal model used only for text still retains its other capabilities.

Concept	What it describes	Example use
Language model	Probabilities over language	Suggest a next word from preceding text
Large language model	Language modeling at scale	Summarize a document supplied as text
Multimodal model	More than one data type	Read an image of a sketch and generate code

Table 1.2: Language modeling, scale, and multimodality describe related capabilities. They are not a strict containment hierarchy.

EXAMPLE 1.3: Telling the Three Apart

A phone keyboard suggesting "you" after "how are" illustrates language prediction. Summarizing a report supplied as text uses a language model's text capabilities. Interpreting a photograph of an app sketch and producing webpage code additionally requires image understanding. Classify the capabilities needed by each task rather than inferring the whole model architecture from one interaction.

In our coding work, image-capable models can inspect a screenshot alongside source code. We still verify whether the generated interface matches the requested behavior and appearance.

CHECKPOINT 1.4

Your phone's keyboard suggests the next word as you type. Separately, you show Claude a photograph of a whiteboard sketch and ask it to write matching React code. Classify each system as a language model, an LLM, or an LMM, and justify each answer in one sentence.

1.8 From Chat to Action: Where Agentic AI Begins

A text-only response describes an action. A tool-enabled system can ask surrounding software to carry it out. For example, a coding system can request a file, inspect a test result, and choose its next step. Both systems can present a chat interface; their ability to execute actions is the relevant difference.

DEFINITION: Agentic AI

In this course, an **AI Agent** is a system that uses a (large language) model to select steps toward a goal and uses surrounding software to execute permitted actions. **Agentic AI** refers to building and using systems with such capabilities. The amount of decision-making delegated to the model can vary.

An agentic system needs relevant task **state**: information about what has happened and what remains to be done. It also needs tools for obtaining information or acting. The surrounding software decides which tools are available and how execution is limited. Chapter 2 explains that loop and the choices it delegates to the model.

The next chapter combines models, supplied context, and tool execution into an agentic system. Its central question is what must surround a model before it can act toward a goal.

1.9 Checkpoint Answers

Try each checkpoint yourself before reading its answer here; the value of a checkpoint is in attempting it, not in reading the solution.

Checkpoint 1.1 (spreadsheet macro versus sales-trend plugin). The macro performs sorting using an explicitly programmed procedure. The plugin estimates sales using parameters fitted to past observations. These mechanisms differ, but neither description establishes accuracy or useful autonomy. We assess the forecast on appropriate unseen cases and inspect which decisions the surrounding application delegates.

Checkpoint 1.2 (greedy decoding and repeated prompts). Greedy decoding selects the highest-probability token at each step. It gives the same sequence when the model, complete input, tie-breaking rule, and computation are identical and deterministic. Repeating the visible prompt in a commercial chatbot does not establish all those conditions. Sampling is one possible cause of different replies, but changes in context, model version, or numerical execution can also matter. A difference in correctness needs investigation regardless of its cause.[6]

Checkpoint 1.3 (hospital assistant on this week's records). Retrieval can select current records for each authorized question. A small set of records could also be supplied directly in the prompt. Neither approach requires parameter updates during inference. Privacy still depends on access controls and the service's data-handling arrangements. Fine-tuning is unsuitable for keeping individual records current and conflicts with the stated prohibition on training on them.

Checkpoint 1.4 (phone keyboard versus Claude reading a sketch). The keyboard task uses language prediction. Interpreting a photograph and producing code requires multimodal capability because the input includes an image. Model size and other capabilities cannot be established solely from observing either task. React is a library for building a web user interface; understanding its code is separate from reading the image.

1.10 End-of-Chapter Exercises

Exercise 1.1. A company claims its "AI-powered" resume screener uses "advanced neural networks" to predict job success. On inspection, it turns out to check whether a resume contains a fixed list of keywords, with no training data involved anywhere. Using the three diagnostic questions of Section 1.3, is this system AI in the technical sense this chapter defines? Explain.

[6] For numerical reproducibility limits, see PyTorch's reproducibility guidance.

Exercise 1.2. Explain, in one or two sentences, why a classifier trained only on labeled cat and dog photos is Predictive AI, while a modern model that turns a text description into a brand-new photo is Generative AI. Use the formal distinction of Section 1.5.

Exercise 1.3. Name the AI winter described in Section 1.2, and explain in your own words what caused it.

Exercise 1.4. A startup fine-tunes a foundation model on last year's product-review dataset and ships it as "the assistant that knows everything about our products." Six months later the company launches five new products, and the assistant confidently gives wrong answers about them. Using Section 1.6's three adaptation mechanisms, explain what went wrong and how you would fix it.

Exercise 1.5. Is the large language model behind a modern coding assistant a Predictive AI system or a Generative AI system? Justify your answer using Section 1.5, and name one thing a pure text-only coding assistant could not do that Section 1.7's Large Multimodal Models can.

Answers

A1.1. The fixed keyword list does not support the advertised claim that the screener uses neural networks. It is an explicitly programmed mechanism. Whether we apply the broad AI label is less important than describing the mechanism accurately and testing the claimed ability to predict job success.

A1.2. The cat/dog classifier computes an output in a fixed-size space decided at training time, here, two categories, cat or dog, which is precisely Section 1.5's definition of Predictive AI. The text-to-image model instead produces an entire image whose resolution is not fixed at training time, and includes a random ingredient, so the same prompt run twice can generate two different valid images, which is precisely the definition of Generative AI.

A1.3. The chapter describes funding setbacks in the 1970s, when some ambitious projects failed to meet expectations. Do not reduce this history to one neural-network limitation. A single-layer perceptron cannot represent XOR, but networks with nonlinear intermediate layers can; this is a distinction between model classes.

A1.4. Further training does not automatically supply facts about products released afterwards. Retrieve information from an updated product source when answering, and check that responses are supported by that source. Retrieval improves access to current information; it does not by itself guarantee a correct answer.

A1.5. Generative: it produces a token sequence (code) whose length is not fixed in advance, generated one token at a time, matching Section 1.5's formal definition exactly. A pure text-only model could not directly take a screenshot of a broken interface, or a hand-drawn wireframe, as input; that visual information would first have to be translated into text by something else, since a text-only model has no way to process images, audio, or video on its own.

End of the Preview

The full reader — nine chapters, three appendices, a glossary and four companion project guides — is provided to enrolled participants.

Agentic AI Coding: From Specification to Deployment — Cohort 2

Live online, Saturdays and Sundays, 13 February – 28 March 2027
Optional in-person campus visit at IISc Bengaluru, Saturday 3 April 2027 (tentative)
Hybrid for those who cannot travel.

See the Cohort 2 course page for registration updates.
https://deepaksubramani.in/agentic-ai-coding/

Course page: https://deepaksubramani.in/agentic-ai-coding/