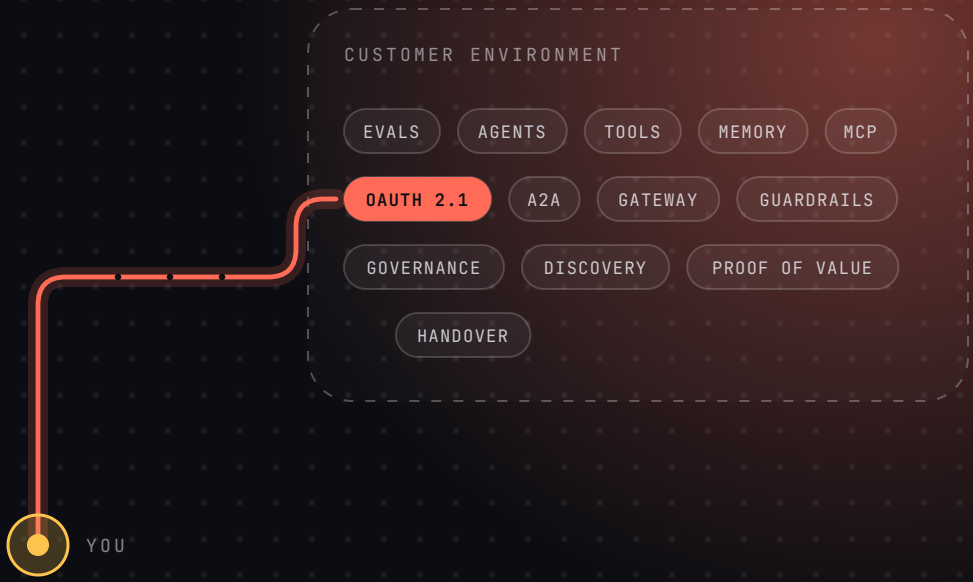
NB NEERAJ BHATYAL

VOL. 02 / 02



FROM SOFTWARE ENGINEERING TO AGENTIC AI SYSTEMS

The Forward Deployed Engineer

Volume 2 — Agents, MCP, and Engineering in the Field

V Building Agents
VI MCP and the Connected Enterprise
VII Enterprise AI Architecture
VIII The FDE in the Field

Neeraj Bhatyal

VOLUME 2 OF 2

The Forward Deployed Engineer

VOLUME 2

Free sample · full book at neerajbhatyal.com

The Forward Deployed Engineer
Volume 2 — Agents, MCP, and Engineering in the Field

Copyright © 2026 Neeraj Bhatyal. All rights reserved.

No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means — electronic, mechanical, photocopying, recording, or otherwise — without the prior written permission of the author, except for brief quotations in reviews and articles.

Published by Neeraj Bhatyal · neerajbhatyal.com

First edition, 2026.

Amazon Web Services, AWS, Microsoft, Azure, Google Cloud, Databricks, Snowflake, Kubernetes, Anthropic, Claude, OpenAI, LangGraph, and all other product and company names mentioned in this book are trademarks or registered trademarks of their respective owners. This book is independent and is not affiliated with, sponsored by, or endorsed by any of these companies.

Cloud services, product names, prices, and features change frequently. The information in this book was checked for accuracy at the time of writing, but readers should confirm details against current official documentation before relying on them. The author provides this book without warranty of any kind and accepts no liability for any loss or damage arising from its use.

The views expressed in this book are the author's own and do not represent the views of any current or former employer.

Acme, its employees, and all scenarios involving them are fictional. Any resemblance to real organizations or people is coincidental.

Contents at a Glance

PART V — BUILDING AGENTS

23 AI Application Architecture and Evaluation 3
24 Agent Fundamentals: Planning, Memory, and Tools 21
25 Agent Frameworks and SDKs 37
26 Agent Platforms on AWS, Azure, and Google Cloud 61
27 Agent Reliability, Observability, and Fleet Operations 79
28 Human Oversight, Bounded Autonomy, and Multi-Agent Systems 97

PART VI — MCP AND THE CONNECTED ENTERPRISE

29 The Model Context Protocol 117
30 Building MCP Servers and Clients 139
31 MCP Security, Identity, and Authorization 161
32 Integration Patterns: APIs, SaaS, Legacy Systems, and Events 183
33 Operating MCP at Enterprise Scale 207

PART VII — ENTERPRISE AI ARCHITECTURE

34 Enterprise AI Platforms and Control Planes 231
35 AI Security and Prompt Injection Defense 251
36 AI Governance, Risk, and Regulation 273

PART VIII — THE FDE IN THE FIELD

37 The FDE Operating Model and Customer Discovery 297
38 Working Inside Customer Environments 317
39 Prototypes, Vertical Slices, and Proof of Value 339
40 Delivery, Production Readiness, and Handover 359
41 Communication, Stakeholders, and Leadership Without Authority 381
42 Building Your FDE Career 399
43 Capstone: Acme From First Conversation to Production 417
Appendix A Agent and MCP Quick Reference 439
Appendix B Production Readiness and Review Checklists 449
Appendix C Templates for Field Engagements 455

Appendix D Further Reading 461
Glossary 465
About the Author 475
Index 477

Contents

PART V — BUILDING AGENTS

23 AI Application Architecture and Evaluation 3

The Model That Won the Demo 4
23.1 Common AI Application Patterns 5
23.2 Choosing the Simplest Pattern That Works 6
23.3 The Layers of a Production AI Application 7
23.4 Why Evaluation Is the Core Discipline 8
23.5 Building Evaluation Sets From Real Data 9
23.6 Metrics by Task Type 11
23.7 Graders: Code, Humans, and LLM Judges 12
23.8 Offline and Online Evaluation 15
23.9 Evaluation in CI as a Release Gate 16
23.10 Evaluation Tooling 17
Hands-On Exercise: Build an Evaluation Gate 18
Checklist: AI Application Architecture and Evaluation 18
Chapter Summary 19

24 Agent Fundamentals: Planning, Memory, and Tools 21

Fourteen Searches for One Pump 22
24.1 What an Agent Is 23
24.2 The Agent Loop 24
24.3 Planning Patterns 25
24.4 Designing Tools for Models 27
24.5 Memory 29
24.6 State and Checkpoints 30
24.7 Stopping Conditions, Budgets, and Loop Limits 32
24.8 When Not to Use an Agent 33
24.9 Fixing the First Triage Agent 33
Hands-On Exercise: Build and Bound a Small Agent 35
Checklist: Agent Fundamentals 35
Chapter Summary 36

25 Agent Frameworks and SDKs 37

The Loop That Became a Framework 38
25.1 What a Framework Actually Gives You 39
25.2 Writing the Loop Yourself on a Provider SDK 40
25.3 The Framework Landscape in Mid-2026 42
25.4 Choosing a Framework 46
25.5 The Triage Agent as a LangGraph Graph 47
25.6 Tools: Structured Definitions and Validation 49

25.7 State, Persistence, and Streaming 51
25.8 Testing Agents 52
25.9 Observability Hooks and OpenTelemetry 54
25.10 Portability, Lock-in, and Over-abstraction 54
25.11 Acme's Decision Record 56
Hands-On Exercise: One Agent, Two Ways 58
Checklist: Choosing and Using an Agent Framework 59
Chapter Summary 59

26 Agent Platforms on AWS, Azure, and Google Cloud 61
The Agent That Ran on a Laptop 62
26.1 Why Managed Agent Platforms Exist 63
26.2 The Building Blocks of an Agent Platform 63
26.3 Amazon Bedrock AgentCore 65
26.4 Microsoft Foundry Agent Service 67
26.5 Google Vertex AI Agent Engine, ADK, and Agent Builder 69
26.6 Comparing the Three Platforms 70
26.7 Self-Hosting Agents on Kubernetes and Containers 71
26.8 Choosing a Platform 72
26.9 Designing for Portability 74
26.10 Acme: One Agent Estate, Two Clouds 75
Hands-On Exercise: Run One Agent on Two Platforms 77
Checklist: Choosing and Using an Agent Platform 77
Chapter Summary 78

27 Agent Reliability, Observability, and Fleet Operations 79
Nobody's Contract Had Expired 80
27.1 How Agents Fail 81
27.2 Containing Failure in a Single Step 82
27.3 Containing Failure Across a Run 84
27.4 Fallbacks and Safe Defaults 85
27.5 Tracing Agent Runs With OpenTelemetry 86
27.6 The Metrics That Matter 87
27.7 Online Evaluation and Sampling for Review 88
27.8 The LLMOps Lifecycle: Versioning, Rollout, and Rollback 89
27.9 Operating a Fleet of Agents 91
27.10 Acme: From Incident to Fleet Operations 93
Hands-On Exercise: Instrument, Break, and Recover an Agent 94
Checklist: Agent Reliability and Operations 94
Chapter Summary 95

28 Human Oversight, Bounded Autonomy, and Multi-Agent Systems 97
Sixty Approvals Before Lunch 98
28.1 The Autonomy Ladder 99
28.2 Risk-Based Action Boundaries 100
28.3 Designing Approval Steps 101
28.4 Kill Switches, Pause, and Moving on the Ladder 103

28.5 When Multiple Agents Help — and When They Hurt 104
28.6 Multi-Agent Patterns 105
28.7 Agent2Agent (A2A) and MCP 106
28.8 Trust Between Agents 108
28.9 Acme: Credits Within Limits 109
28.10 Acme: The Day Two Agents Disagreed 109
Hands-On Exercise: Bounded Autonomy and a Small Supervisor 111
Checklist: Oversight and Multi-Agent Design 112
Chapter Summary 113

PART VI — MCP AND THE CONNECTED ENTERPRISE

29 The Model Context Protocol 117
Fourteen Ways to Fetch a Case 118
29.1 The N×M Integration Problem 119
29.2 Hosts, Clients, and Servers 120
29.3 The Message Layer: JSON-RPC 2.0 121
29.4 Lifecycle and Capability Negotiation 122
29.5 Server Primitives: Tools, Resources, and Prompts 124
29.6 Client Features: Sampling, Roots, and Elicitation 127
29.7 Transports: stdio and Streamable HTTP 129
29.8 Tool Annotations: Hints, Not Guarantees 130
29.9 Versions and Governance of the Specification 131
29.10 What MCP Is Not — and How It Compares 133
29.11 Acme's First Two Servers 135
Hands-On Exercise: Watch the Protocol on the Wire 136
Checklist: Adopting MCP 137
Chapter Summary 137

30 Building MCP Servers and Clients 139
Forty-Three Tools and a Confused Agent 140
30.1 Designing for a Model, Not a Program 141
30.2 Granularity and Naming 142
30.3 Descriptions, Schemas, and Structured Output 143
30.4 Pagination, Large Results, and Errors 145
30.5 Implementing a Server With the Python SDK 146
30.6 Inside a Production Server 149
30.7 Local and Remote Servers 150
30.8 Testing: Inspector, Unit Tests, and Evaluations 151
30.9 Building the Client Side 154
30.10 Packaging and Deploying Remote Servers 155
30.11 Versioning Tools Without Breaking Agents 156
Hands-On Exercise: Build, Test, and Evaluate a Small Server 158
Checklist: MCP Server Design and Delivery 159
Chapter Summary 159

31 MCP Security, Identity, and Authorization 161
The Token That Would Have Gone Straight Through 162
31.1 Why MCP Changes the Threat Model 163
31.2 The Threats in Detail 164
31.3 OAuth 2.1 for MCP in Plain Terms 167
31.4 Discovery, Registration, and Getting a Token 168
31.5 Acting for the User or as a Service 171
31.6 Delegation Without Token Passthrough 173
31.7 Least Privilege, Per-Tool Scopes, and Consent 174
31.8 Enterprise Identity Integration 176
31.9 Vetting, Allow-Lists, and Change Detection 177
31.10 Sandboxing Local Servers 177
31.11 Audit Logging and Detection 179
Hands-On Exercise: Secure a Remote Server End to End 180
Checklist: MCP Security Review 181
Chapter Summary 181

32 Integration Patterns: APIs, SaaS, Legacy Systems, and Events 183
Two Valves for Rotterdam 184
32.1 Where Agents Meet the Enterprise 185
32.2 Wrapping REST and GraphQL APIs 186
32.3 SaaS Platforms: CRM, ITSM, and ERP 188
32.4 Legacy, Mainframe, and On-Premises Systems 189
32.5 Event-Driven Integration 191
32.6 File and Batch Integration 194
32.7 Composing Capabilities Into Workflows 194
32.8 Idempotency and Transactional Safety 197
32.9 Rate Limits, Quotas, and Backpressure 200
32.10 Data Mapping and Semantic Mismatches 201
32.11 Contract Tests and Integration Testing 202
Hands-On Exercise: Integrate an Agent Safely With a "Legacy" System 204
Checklist: Integration Patterns for Agents 205
Chapter Summary 205

33 Operating MCP at Enterprise Scale 207
Forty-Seven Servers 208
33.1 From Five Servers to Two Hundred 209
33.2 The MCP Registry 210
33.3 The MCP Gateway 213
33.4 Too Many Tools: Discovery and Selection 216
33.5 Versioning and Compatibility 217
33.6 Deprecation, Dependencies, and Change Impact 218
33.7 Compatibility Testing: Catching Breaks Before Agents Do 220
33.8 Observability for MCP Servers 221
33.9 Service Level Objectives for Capabilities 222
33.10 Multi-Tenancy and Federation 223

Hands-On Exercise: Build a Mini Registry and Gateway 225
Checklist: Operating MCP at Scale 226
Chapter Summary 226

PART VII — ENTERPRISE AI ARCHITECTURE

34 Enterprise AI Platforms and Control Planes 231
Three Stacks and a Simple Question 232
34.1 Why a Shared AI Platform 233
34.2 A Reference Architecture 233
34.3 Control Plane and Data Plane 235
34.4 The Model Gateway and the Tool Gateway 236
34.5 Shared Services: Knowledge, Guardrails, Evaluation, Observability, and Cost 237
34.6 Policy as Code for AI 239
34.7 Multi-Tenancy and Customer-Specific Configuration 241
34.8 Build, Buy, or Cloud-Native 242
34.9 The Platform Team and How to Avoid a New Monolith 244
34.10 Maturity Stages 245
34.11 Acme: Consolidating Three Stacks 246
Hands-On Exercise: A Miniature AI Control Plane 248
Checklist: Enterprise AI Platform 249
Chapter Summary 249

35 AI Security and Prompt Injection Defense 251
The Email That Asked for a Contract 252
35.1 How AI Changes the Threat Model 253
35.2 Shared Vocabularies: OWASP and MITRE ATLAS 254
35.3 Prompt Injection: Direct and Indirect 256
35.4 Why Prompts Cannot Fully Solve It 257
35.5 The Lethal Trifecta 259
35.6 Defense in Depth for Agents 260
35.7 Guardrail Services, Classifiers, and Data Loss Prevention 263
35.8 Secrets and Data Leakage 265
35.9 The AI Supply Chain 266
35.10 Red Teaming and Security Testing for AI 267
35.11 Incident Response for AI Systems 268
Hands-On Exercise: Attack and Defend a Small Agent 270
Checklist: Securing an Agent 271
Chapter Summary 272

36 AI Governance, Risk, and Regulation 273
Fourteen Spreadsheets and No Answer 274
36.1 Governance That Enables 274
36.2 The AI Inventory 276
36.3 Risk Tiering 277
36.4 Roles and Responsibilities 279

36.5 Lifecycle Controls 280
36.6 Documentation: System Cards and Impact Assessments 282
36.7 Frameworks: NIST AI RMF and ISO/IEC 42001 283
36.8 Regulation: The EU AI Act, Sector Rules, and Privacy Law 284
36.9 Accountability for Agent Actions 287
36.10 Governance Evidence as a By-Product of Engineering 288
Hands-On Exercise: Build a Governance Package for One System 291
Checklist: Governance-Ready AI Systems 291
Chapter Summary 292

PART VIII — THE FDE IN THE FIELD

37 The FDE Operating Model and Customer Discovery 297
Forty Percent 298
37.1 The FDE Operating Model 299
37.2 The People Around the FDE 300
37.3 Engaging: The First Two Weeks 302
37.4 Stakeholder Mapping 302
37.5 Discovery Interviews 303
37.6 Shadowing the Actual Workflow 304
37.7 Data and System Due Diligence 307
37.8 The Constraints Register and Baseline Metrics 308
37.9 Problem Framing 309
37.10 Feeding Field Learning Back to the Product 311
37.11 Discovery Outputs 312
Hands-On Exercise: Run a Miniature Discovery 313
Checklist: Discovery Complete 314
Chapter Summary 314

38 Working Inside Customer Environments 317
Nineteen Days Without a Login 318
38.1 The Customer's Trust Boundary 319
38.2 Getting Access 320
38.3 Using the Wait 323
38.4 Working in Restricted Environments 324
38.5 Handling Customer Data 326
38.6 Change Management and Change Windows 328
38.7 Passing Security Reviews 329
38.8 Working With Platform, Network, and Security Teams 331
38.9 Respecting Conventions and Documenting Everything 332
38.10 Making Access Repeatable 333
38.11 When You Are Blocked 334
Hands-On Exercise: Prepare to Enter a Customer Environment 335
Checklist: Working Inside a Customer Environment 336
Chapter Summary 337

39 Prototypes, Vertical Slices, and Proof of Value 339
Two Weeks, One Question 340
39.1 What a Prototype Is For 341
39.2 Vertical Slices, Not Horizontal Layers 342
39.3 Proof of Concept, Proof of Value, and Pilot 344
39.4 Success Criteria and a Measurement Plan Before You Build 345
39.5 Using Real Data Early 347
39.6 Time-Boxing the Work 348
39.7 Demo Discipline 350
39.8 The Demo Trap and the Prototype-to-Production Gap 351
39.9 Customer-Specific Needs Versus Reusable Product 353
39.10 Making the Go/No-Go Decision 354
Hands-On Exercise: Run a One-Week Proof of Value 356
Checklist: Prototype and Proof of Value 357
Chapter Summary 357

40 Delivery, Production Readiness, and Handover 359
Region by Region 360
40.1 Delivery Engineering in the Field 361
40.2 A Testing Strategy for AI Systems 362
40.3 Deployment Patterns 364
40.4 The Production Readiness Review 366
40.5 Launch Planning and Phased Rollout 368
40.6 Adoption and Change Management 369
40.7 Handover: Transferring Ownership 371
40.8 Documentation and Knowledge Transfer 373
40.9 Measuring Impact After Launch 374
40.10 Leaving Well 375
Hands-On Exercise: Take a Small Agent to Production Standard 376
Checklist: Delivery, Readiness, and Handover 377
Chapter Summary 378

41 Communication, Stakeholders, and Leadership Without Authority 381
The Quarter-End Promise 382
41.1 Mapping Stakeholders 383
41.2 One Set of Facts, Many Audiences 384
41.3 Writing That Moves Work Forward 386
41.4 Demos and Steering Meetings 388
41.5 Making Tradeoffs Explicit and Getting Decisions Made 389
41.6 Disagreement, Scope Creep, and Bad News 390
41.7 Building Trust and Leading Without Authority 392
41.8 Measuring and Communicating Impact 393
41.9 Working Across Countries, Cultures, and Time Zones 394
Hands-On Exercise: Map, Decide, and Report 395
Checklist: Communication and Leadership in an Engagement 396
Chapter Summary 396

42 Building Your FDE Career 399
The Sketch From Two Years Ago 400
42.1 The π-Shaped FDE, Revisited 400
42.2 Skill Areas and How to Deepen Them 402
42.3 Learning Systems 403
42.4 Judgment and How It Grows 405
42.5 Career Paths 407
42.6 Interviewing for FDE Roles 408
42.7 Sustaining Yourself 410
42.8 Staying Current as AI Changes the Work 411
42.9 Priya's Reflection and Learning Plan 412
Hands-On Exercise: Redraw Your π and Write a Learning Plan 414
Checklist: A Sustainable FDE Career 414
Chapter Summary 415

43 Capstone: Acme From First Conversation to Production 417
The Email, Two Years Later 418
43.1 The Shape of the Engagement 419
43.2 First Conversation and Discovery 420
43.3 Constraints and the First Vertical Slice 421
43.4 Volume 1 in Production: Assistant, Cost, and Documents 422
43.5 From Assistant to Case Triage Agent 424
43.6 MCP Servers, the Gateway, and the Platform 425
43.7 Security and Governance 427
43.8 Climbing the Autonomy Ladder 428
43.9 Production Readiness, Handover, and Outcomes 429
43.10 What the Team Would Do Differently 431
43.11 Lessons Across the Engagement 433
Hands-On Exercise: Plan an Engagement End to End 436
Checklist: Engagement Close-Out 437
Chapter Summary 438

Appendix A Agent and MCP Quick Reference 439

Appendix B Production Readiness and Review Checklists 449

Appendix C Templates for Field Engagements 455

Appendix D Further Reading 461

Glossary 465

About the Author 475

Index 477

Preface to Volume 2

Volume 1 of this book built a foundation: the Forward Deployed Engineer role, the software engineering habits that survive contact with real customers, the three major clouds, data platforms, and the core of applied AI — models, prompting, retrieval, and document understanding. By its last chapter, Acme had an AI assistant in production that answered support questions with citations, respected permissions, and cost what the business had agreed to pay.

This volume is about what comes next, and it is where most of the hard, interesting work now lies. An assistant answers questions. An **agent** pursues a goal: it decides what to look up, calls tools, drafts responses, and — when it has earned the right — takes actions in the systems a business depends on. That shift changes almost everything about how AI systems must be engineered. Evaluation has to measure behavior across many steps, not single answers. Every tool an agent can call becomes part of the security boundary. Autonomy has to be granted deliberately, measured, and taken back when needed. And the systems agents connect to — CRMs, order systems, decades-old databases — have to be exposed in ways that are safe, governed, and maintainable.

What this volume covers

Part V — Building Agents covers AI application architecture and evaluation, the fundamentals of agents, the frameworks used to build them, the agent platforms offered by AWS, Microsoft, and Google, how to operate fleets of agents reliably, and how to design human oversight and bounded autonomy.

Part VI — MCP and the Connected Enterprise is devoted to the Model Context Protocol, the open standard that has become the common way to connect AI applications to tools and data. It explains the protocol, how to build servers and clients, how to secure them with modern identity and authorization, how to integrate real enterprise systems, and how to run many MCP servers at scale.

Part VII — Enterprise AI Architecture steps back to the organization: the platforms and control planes that let an enterprise run many AI systems, the security discipline needed to defend them — especially against prompt injection — and the governance, risk management, and regulation that increasingly shape what can be deployed.

Part VIII — The FDE in the Field turns to the craft itself: discovery, working inside customer environments, prototypes and proof of value, delivery and handover, communication and leadership without authority, and building a career. It ends with a capstone that follows Acme's engagement from the first conversation to production.

Reading this volume

If you have read Volume 1, you can continue directly; the Acme story picks up where it left off. If you are starting here, you will find frequent references to Volume 1 chapters where a concept was introduced — identity and OAuth in Chapter 7, observability in Chapter 16, cost in Chapter 17, AI foundations in Chapter 19, prompting and serving in Chapter 20, and retrieval in Chapter 21 are the ones this volume leans on most. An experienced engineer can follow this volume on its own and dip back into Volume 1 as needed.

A note on a fast-moving field

Agents and the protocols around them are changing faster than any other part of this book's subject. Frameworks gain and lose favor, cloud platforms add capabilities every few months, and the Model Context Protocol specification continues to evolve. I have tried to teach the durable ideas — the agent loop, the separation of judgment from control, least privilege for tools, evaluation as the basis for trust, autonomy granted in steps — and to present specific products as current examples of those ideas. Where a detail matters for your work, confirm it against current documentation.

As in Volume 1, I used AI tools extensively in writing this book, and every chapter was reviewed, reshaped, and verified by me. Building the book this way was itself a small exercise in the subject of this volume: a capable AI collaborator, clear boundaries, and a human accountable for the result.

I hope this volume helps you build agents that people trust — not because they are impressive, but because they are measured, bounded, and well engineered.

Neeraj Bhatyal
New York, 2026

How to Use This Book

Who this volume is for

This volume is written for software engineers, data engineers, architects, and technical leaders who build or plan to build AI applications and agents for real organizations — and for anyone preparing for Forward Deployed Engineer work. It assumes you are comfortable with programming, APIs, cloud basics, and the AI foundations covered in Volume 1: tokens and context windows, prompting and structured outputs, tool use, and retrieval-augmented generation. Where a chapter depends on earlier material, it tells you where to find it.

Ways to read it

- **From start to finish.** Parts V to VIII build on each other, and the Acme story runs through them in order, ending with the capstone in Chapter 43.
- **By need.** About to build an agent? Read Chapters 23 to 25, then 28. Connecting agents to enterprise systems? Chapters 29 to 33. Facing a security or risk review? Chapters 31, 35, and 36. Starting a new engagement? Begin with Chapter 37.
- **As a field reference.** Appendix A is a quick reference for agents and MCP, Appendix B collects the production readiness and review checklists, and Appendix C provides templates for agent design, evaluation plans, autonomy changes, risk assessments, incident reviews, and handover.

How each chapter is organized

Every chapter follows the same structure as in Volume 1:

1. **Opening scenario.** A moment from the Acme engagement that frames the chapter's problem.
2. **Core concepts.** The ideas explained in plain language, with diagrams, tables, and short code examples.
3. **Field Notes.** Prompts that connect the chapter to your own work.

4. **Hands-on exercise.** A practical task you can complete with open-source tools or a sandbox cloud account.
5. **Checklist and summary.** A checklist for real work and a summary of the chapter's key points.

Acme and Priya, continued

In Volume 1, **Priya**, a Forward Deployed Engineer, helped **Acme** — a fictional company that sells and services industrial equipment in more than forty countries — put an AI support assistant and a document processing pipeline into production. In this volume, Acme moves from assistants to agents: a Case Triage Agent that gathers context and proposes actions for 600 support agents, then a Warranty Agent and a Field Service Scheduling Agent. Along the way, Acme exposes its systems through MCP servers, faces a prompt-injection attempt, establishes AI governance, and gradually grants its agents more autonomy. A few colleagues appear regularly: Elena Vasquez, VP of Customer Support and the project's sponsor; David Okoro, the CISO; Marcus Chen, who leads platform engineering; and Ana Lima, a senior support agent who becomes the product's champion. All of them, and Acme itself, are fictional.

Conventions

Text in `monospace` indicates commands, code, file names, and identifiers. Code is written in Python unless stated otherwise and is shortened for clarity; it illustrates patterns rather than providing production-ready implementations. Three kinds of boxes appear throughout:

KEY IDEA

A principle worth remembering after you have forgotten the details.

FDE TIP

Practical advice for working in customer environments.

FIELD NOTES

A prompt to connect the chapter to your own work.

Exercises, costs, and safety

The hands-on exercises use open-source frameworks, local tools, and the free tiers of cloud providers wherever possible. Model API calls cost money, and agents can make many calls quickly: set spending limits and budget alerts before you begin, cap the number of steps your agents may take, and clean up resources when you finish. Never give an exercise agent access to real customer data, production systems, or credentials with write permissions.

Acknowledgments

[To be written by the author — the people, teams, reviewers, and family you would like to thank.]

PART V

Building Agents

How to architect, evaluate, build, and operate AI applications and agents — from evaluation discipline and agent design to frameworks, cloud agent platforms, production operations, and human oversight.

23 AI Application Architecture and Evaluation
24 Agent Fundamentals: Planning, Memory, and Tools
25 Agent Frameworks and SDKs
26 Agent Platforms on AWS, Azure, and Google Cloud
27 Agent Reliability, Observability, and Fleet Operations
28 Human Oversight, Bounded Autonomy, and Multi-Agent Systems

23

AI Application Architecture and Evaluation

Choosing the right shape for an AI system — and building the evaluation discipline that keeps it honest

IN THIS CHAPTER

- Six common AI application patterns, from a single call to an agent
- How to choose the simplest pattern that solves the problem
- The layers of a production AI application
- Why evaluation is the core engineering discipline of AI work
- Building evaluation sets from real data, edge cases, attacks, and incidents
- Metrics by task type, and code, human, and LLM-as-judge graders
- Offline and online evaluation: shadow tests, canaries, and A/B tests
- Evaluation in CI as a release gate, and the tools that support it

The Model That Won the Demo

The new model version arrived on a Monday, and by Wednesday everyone at Acme had an opinion about it. A product manager ran a dozen support cases through it in a chat window and shared the results in Teams. The case summaries were crisper and the tone was warmer. Elena Vasquez, VP of Customer Support, asked the question every sponsor asks: "Can we switch this week?"

A year earlier, the answer would have depended on who argued most confidently. Now it depended on a pipeline. Priya opened a pull request that changed one line — the model identifier in the case-summary configuration — and let the release gate run. The evaluation suite, which had started as 150 hand-labeled cases in Volume 1, now held 620 cases in four sets. Twenty minutes later the gate failed. Overall summary quality, scored by a calibrated judge, was up from 4.1 to 4.3 out of 5. But **safety-flag recall** — how often the summary flags a gas, electrical, or water hazard that the case mentions — had dropped from 0.98 to 0.91. The gate's threshold was 0.97.

Priya pulled the nine failing cases. In each one, the hazard was mentioned indirectly: "customer noticed a smell near the intake," "breaker tripped twice when the unit started." The old model flagged these; the new one, more fluent and concise, smoothed them away. None of the demo cases had contained an indirect hazard. A technician walking into a room with a gas leak does not care how warm the summary's tone was.

The fix took two days: a prompt adjustment for the new model, eleven new edge cases written by Ana Lima from her own experience of how customers describe hazards, and a second gate run that passed at 0.98 recall. The model shipped the following week — first in shadow mode, then to everyone. Elena got her upgrade. She also stopped asking whether a change "looked better" and started asking what the gate said.

KEY IDEA

An AI application is only as trustworthy as its evaluation. Choose the simplest architecture that solves the problem, and make a versioned evaluation suite — built from real data and run on every change — the gate through which every prompt, model, and tool change must pass.

23.1 Common AI Application Patterns

Volume 1 built Acme's support assistant from parts: prompts and structured outputs (Chapter 20), retrieval (Chapter 21), and document extraction (Chapter 22). Before moving on to agents, it is worth naming the shapes those parts assemble into. Almost every enterprise AI application is one of six patterns, or a combination. Figure 23.1 arranges them by how much control they hand to the model.

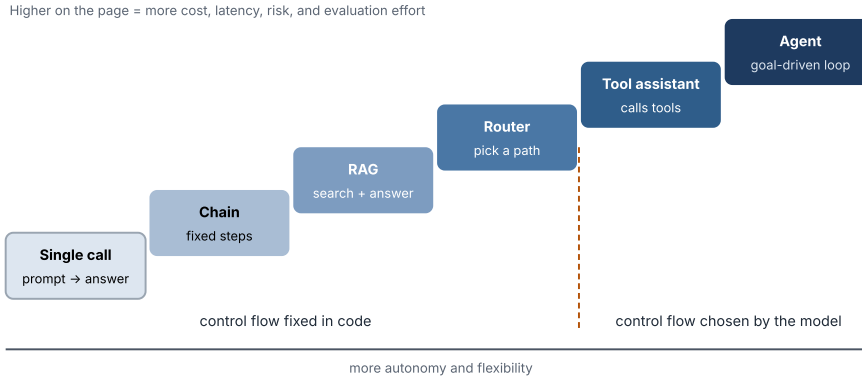


Figure 23.1 — Six AI application patterns. Moving right hands more decisions to the model, which buys flexibility at the price of cost, latency, unpredictability, and evaluation effort. The dashed line marks the point where the model, not your code, starts choosing what happens next.

- **Single call.** One prompt, one response, often with a structured output: classifying urgency, summarizing a case, extracting fields from an email. Cheap, fast, easy to evaluate — and enough for more problems than people expect.
- **Chain or workflow.** Several model calls and ordinary functions in a fixed sequence written in code: extract fields, look up the customer, then draft a reply using both. The path is decided by the developer; the model fills in steps.
- **Retrieval-augmented generation (RAG).** A specific, very common chain: search an index, then answer from what was found, with citations (Chapter 21). Acme's support assistant is a RAG application.
- **Router.** A classification step — by a model or by rules — sends each request down one of several paths: warranty questions to the warranty workflow, parts questions to the catalog, everything else to RAG.

Routers also choose models, sending easy requests to a small, cheap one (Volume 1, Chapter 17).

- **Tool-using assistant.** The model is given a set of tools (Chapter 19) and decides, turn by turn, whether to call one — look up an order, check stock — before answering a person. A human drives the conversation; the model chooses tools within each turn.
- **Agent.** The model works toward a goal in a loop: it plans, calls tools, observes the results, and decides what to do next until it judges the goal met or a limit is reached. Chapter 24 covers agents in depth.

The patterns combine freely: a router may send some requests to a RAG chain and others to an agent, and an agent may use RAG as one of its tools. Naming them forces a deliberate choice about where the model makes decisions.

23.2 Choosing the Simplest Pattern That Works

Customers often arrive asking for "an agent." Sometimes they need one; more often the problem is a workflow with one or two steps that need judgment, and the most valuable thing an FDE can do is say so. Every step to the right in Figure 23.1 adds failure modes: more model calls that can each go wrong, longer and less predictable latency, higher token cost, and a much larger space of behaviors to test. An agent that decides its own path can take a path nobody evaluated.

Table 23.1 — Choosing an application pattern

Pattern	Use when	Main risks	Acme example
Single call	One well-defined transformation of the input	Format drift; silent errors	Case summary; urgency label
Chain	Steps are known and stable	Errors compound across steps	Warranty claim extraction and checks
RAG	Answers need private, current, citable content	Wrong or missing passages	Technician support assistant
Router	Requests fall into distinct types	Misrouting; unseen types	Email intake: warranty, parts, fault
Tool assistant	A person needs live data in a conversation	Wrong tool or arguments	Agent desk: "where is my order?"
Agent	The path is unknown until work begins	Loops, cost, unsafe actions	Case Triage Agent (Chapter 24)

Work through four questions in order. *Can the steps be written down in advance*? If a support lead can draw the process as a flowchart, build the flowchart in code and use the model only inside boxes that need language understanding. *Does the answer depend on private or current knowledge*? Add retrieval or tools. *Do requests vary enough to need different handling*? Add a router. *Is the number and order of steps genuinely unknowable until the work begins* — investigating an unusual fault across cases, manuals, telemetry, and contracts? Only then consider an agent, and bound it carefully.

Record the choice and its reasoning in an architecture decision record (Volume 1, Chapter 3). The principle from that chapter — use AI for judgment, deterministic code for control — applies directly: the further right you go, the more important it becomes to wrap the model's choices in code that enforces limits.

23.3 The Layers of a Production AI Application

A prototype is a prompt and a model. A production application, whatever its pattern, has the layers shown in Figure 23.2. Knowing them helps you find the right place for each piece of logic, and spot what a customer's prototype is missing.

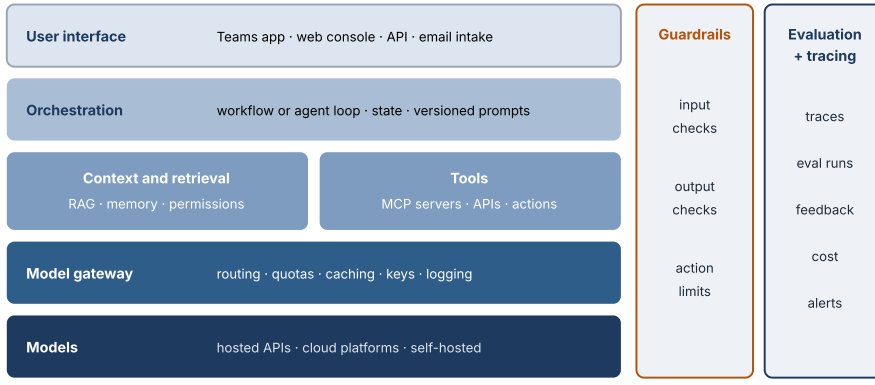


Figure 23.2 — Layers of a production AI application. Requests flow down from the interface through orchestration, context, and tools to the model gateway and models. Guardrails and evaluation with tracing are cross-cutting: they observe and constrain every layer.

- **User interface.** A Teams chat, a case-console panel, an API, an inbox. Good AI interfaces show sources, make uncertainty visible, and make output easy to correct or reject.
- **Orchestration.** The chain, router, or agent loop, with its state and versioned prompts. Most application logic lives here, and so do the frameworks of Chapter 25.
- **Context and retrieval.** Everything that assembles what the model sees, filtered by the user's permissions first (Volume 1, Chapters 20 and 21).
- **Tools.** The functions and services the model can call. At Acme these are increasingly exposed as Model Context Protocol (MCP) servers, the subject of Part VI.
- **Model gateway and models.** One internal entry point to all models — hosted, on platforms such as Amazon Bedrock, or self-hosted — handling credentials, routing, quotas, caching, and logging (Volume 1, Chapter 20). It lets you change models by configuration, which is what makes a release gate on model changes possible.
- **Guardrails.** Checks on inputs (prompt injection, personal data, off-topic requests), outputs (schema, policy, grounding), and actions (permissions, limits, approvals). Chapter 35 covers them in depth.
- **Evaluation and observability.** Traces, evaluation runs, feedback, cost, and alerts. Chapter 27 covers observability; the rest of this chapter covers evaluation.

Walk down these layers when you review a customer's prototype. The usual finding is an interface, a prompt, and a model — no gateway, no guardrails, no evaluation. That gap, not the model, stands between the prototype and production.

23.4 Why Evaluation Is the Core Discipline

Conventional software has specifications, deterministic outputs, and tests that point at bugs. AI components break each assumption. Outputs vary from run to run. The "specification" is a paragraph of intent ("a useful summary for a technician"). The same prompt behaves differently on a new model version, and improvements on one kind of input quietly cause regressions on another. Many outputs have no single correct answer.

Evaluation — systematically measuring an AI system's behavior against a representative set of inputs and explicit quality criteria — is the engineering response. It does three jobs at once. It is the *specification*: an evaluation set with labeled expectations is the most precise statement anyone will ever write of what the system should do. It is the *test suite*: it tells you whether a change made things better or worse, and where. And it is the *evidence*: it turns stakeholder debates about whether the system is "good enough" into conversations about numbers and examples.

Evaluation is not a phase at the end of a project. It is a loop that runs for the life of the system, fed by production, as Figure 23.3 shows.

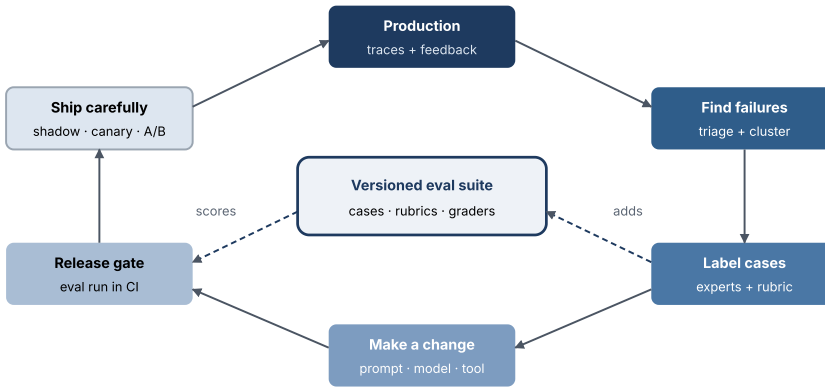


Figure 23.3 — The evaluation loop. Production traces and feedback reveal failures; experts label them into new cases; every change is scored against the suite before release; releases are rolled out carefully and observed in production. The suite grows with the system.

23.5 Building Evaluation Sets From Real Data

An evaluation set is a collection of **cases**: an input, any context the system needs, and the expectation the output is graded against — a reference answer, a label, facts that must appear, or a rubric. Everything downstream depends on whether the cases resemble what the system will really see.

Start from real inputs

Questions written by the project team are biased toward what the team already understands. Real inputs — past cases, search logs, emails — contain the abbreviations, typos, mixed languages, and odd requests production will bring. Priya sampled closed cases from the gold

`support.case_resolutions` data product, stratified by region, product family, and case length so rare but important groups were not crowded out. Model-generated variations can fill gaps, but should be reviewed by a person and labeled as synthetic.

Four kinds of cases

A mature suite combines four kinds of cases, as shown in Figure 23.4, and tracks them separately because they answer different questions.

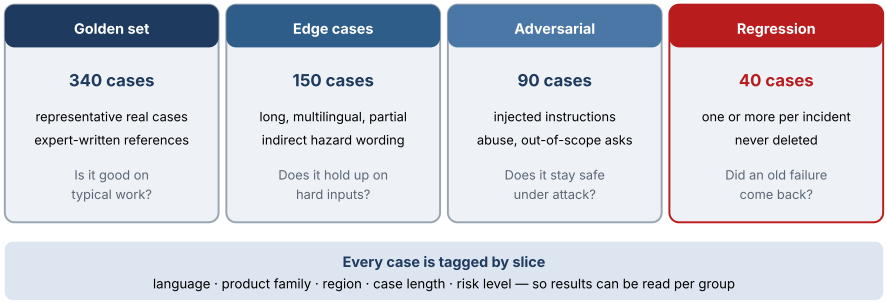


Figure 23.4 — Composition of Acme's case-summary evaluation suite. Golden cases measure typical quality; edge and adversarial cases probe weaknesses; regression cases make sure past incidents stay fixed. Slice tags let the team see whether an average hides a group that is getting worse.

- **Golden set.** Representative real inputs with expected outputs approved by subject-matter experts.
- **Edge cases.** Rare, important, or difficult inputs: very long or multilingual cases, missing information, and — after the opening story — indirectly described hazards. Ask experienced people such as Ana Lima: "What do customers say that confuses new agents?"
- **Adversarial cases.** Inputs designed to cause misbehavior: instructions hidden in a customer email, requests for other customers' data, abuse. Chapter 35 grows this into red teaming.
- **Regression cases.** Every production incident, like the H-400 wrong-manual question in Volume 1, becomes a permanent case that should always pass.

Labels, size, and hygiene

Domain experts should write expected outputs, following guidelines so two experts would label a case the same way. Where possible, label *checkable facts* rather than whole reference texts: "must mention the E-217

error code," "must flag an electrical hazard," "must not promise a delivery date." These are easier to grade reliably than similarity to a reference.

Fifty well-chosen cases catch gross problems early; a few hundred let you compare versions with reasonable confidence. Small differences need more cases than people expect: on 100 cases, a move from 90% to 92% is two cases and may be noise. Report results per slice, with the case count next to every percentage.

Treat the suite as a governed data asset: version it with the code, apply the source data's access controls (it contains customer information), and keep a held-out portion that prompt authors never see, so prompts are not tuned to the test. A case in Acme's suite looks like this:

```
{"id": "cs-edge-0117",
 "set": "edge",
 "slices": {"lang": "pt-BR", "family": "compressor",
            "risk": "safety"},
 "input": {"case_id": "C-2025-88412"},
 "expect": {
   "equipment_model": "H-400",
   "must_flag": ["gas"],
   "must_mention": ["odor near intake",
                    "unit restarted twice"],
   "must_not": ["part numbers", "delivery dates"]},
 "source": "incident INC-3317; labeled by A. Lima",
 "added": "2026-03-02"}
```

23.6 Metrics by Task Type

The right metric depends on what the system does. Choose metrics before running experiments, and choose ones a business owner can understand — "safety-flag recall" means something to Elena; "mean ROUGE-L" does not. Table 23.2 summarizes common choices.

Four practices make metrics trustworthy. First, **pair metrics that pull in opposite directions**: recall with precision, harmful-output rate with over-refusal rate, quality with cost. Optimizing one alone produces systems that flag everything or refuse everything. Second, **name a few metrics as gates** and treat the rest as diagnostics; Acme gates on safety-flag recall, required-fact coverage, schema validity, and regression pass rate. Third, **run noisy cases more than once** — three to five runs per case for

generative outputs — and look at the spread, not just the mean. Fourth, **read the failures**: a score says something changed; only the failing outputs say why.

Table 23.2 — Evaluation metrics by task type

Task type	Typical metrics	Usual grader
Classification, routing	Accuracy; precision and recall per class; confusion matrix	Code, against labels
Extraction	Field-level exact match or F1; schema validity	Code, with normalization
Summarization	Required facts present; no unsupported facts; rubric score	Code for assertions; judge for rubric
RAG question answering	Retrieval recall@k; faithfulness; correctness; citation accuracy; abstention	Code plus calibrated judge
Open-ended generation	Rubric scores; pairwise preference against a baseline	Judge, sampled human review
Tool use and agents	Task success; correct tool and arguments; steps, cost, policy violations	Code on traces; judge for outcome
Safety	Attack success rate; harmful output rate; over-refusal rate	Code, classifiers, human review
Operational	Latency percentiles; tokens and cost per task; error rate	Code, from traces

Agents need one extra idea: grade both the *outcome* (was the case correctly triaged?) and the *trajectory* (sensible tools, in a sensible order, within budget and policy?). Outcomes tell you whether it works; trajectories tell you whether it works for the right reasons at an acceptable cost — Chapter 24 shows why that matters.

23.7 Graders: Code, Humans, and LLM Judges

A **grader** is whatever decides whether an output meets its expectation. There are three kinds, and good suites use all of them, each where it is strongest.

Code-based graders

Code-based graders are deterministic checks: does the output match the schema; is the required fact present; does every citation refer to a supplied source; did the agent call `draft_reply` before `get_case`? They are fast,

free, and repeatable. Use them for everything they can check, and design outputs — structured fields, assertion-style expectations — so that more can be checked by code.

Human review

People remain the reference standard for nuanced quality, such as whether a summary would actually help a technician. Human review is slow and expensive, so use it to label the golden set, calibrate judges, sample production traffic, and settle disagreements. Give reviewers a written **rubric** with concrete descriptions of each score level, and measure **inter-rater agreement** — how often two reviewers give the same score. If experts disagree with each other, the rubric is unclear, and no automated judge can do better than the humans it imitates.

LLM-as-judge

An **LLM judge** is a model prompted to grade another model's output. It makes rubric grading affordable at scale. It comes in three forms. *Pointwise* judging scores one output against a rubric. *Reference-based* judging compares an output to an expert's reference. *Pairwise* judging shows two outputs — say, from the current and a candidate prompt — and asks which is better; it is often more reliable than absolute scores, because comparing is easier than scoring.

Judge prompts work best when they grade one criterion at a time, use pass/fail or small scales with written definitions, ask for evidence before the verdict, and return structured output. Research and practice have documented several judge **biases**: *position bias* (favoring whichever answer comes first in a pairwise comparison), *verbosity bias* (favoring longer answers), *self-preference* (favoring outputs from its own model family), and *leniency* (rating most outputs as acceptable). Mitigations are straightforward: run pairwise comparisons in both orders and count only consistent verdicts; tell the judge length is not a virtue; use a different model family as judge where practical; and prefer pass/fail checks to five-point scales.

```
You are grading a support-case summary written for a
field technician. Use only the case text and rubric.

Rubric (score each item PASS or FAIL):
1. hazards: every gas, electrical, or water hazard in
   the case appears in safety_flags, including hazards
   described indirectly ("a smell near the intake").
2. grounded: every statement is supported by the case.
3. actionable: open_questions name what the technician
   must confirm on site.

First quote the evidence for each item, then give the
verdict. Return JSON:
{"hazards": {"evidence": "...", "verdict": "PASS"},
 "grounded": {...}, "actionable": {...}}
```

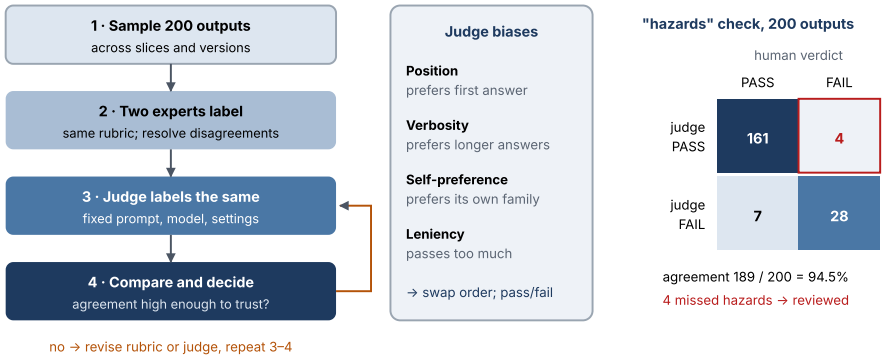


Figure 23.5 — Calibrating an LLM judge against people. Experts and the judge label the same sample; the agreement matrix shows where they differ. The four outputs the judge passed but experts failed are the dangerous cell for a safety check, and each one is studied before the judge is trusted. The numbers are illustrative.

Above all, a judge must be **calibrated** before its scores are trusted, as Figure 23.5 shows. Experts label a sample; the judge labels the same sample; you measure agreement — simple percentage agreement, and a chance-corrected statistic such as Cohen's kappa — and look closely at the disagreements. For a safety check, the cell that matters most is "judge passed, human failed." If agreement is too low, refine the rubric or judge prompt, or use a stronger judge model, and repeat. Pin the judge's prompt, model version, and settings, and re-calibrate when any of them change. Then keep spot-checking a small monthly sample by hand.

FDE TIP

Before building any automated grader, sit with two domain experts and have them grade twenty outputs independently. Their disagreements are the most valuable hour of the project: they reveal what "good" actually means to the business, and they become the first draft of your rubric.

23.8 Offline and Online Evaluation

Offline evaluation runs the system against a fixed suite before release. It is repeatable and safe, but the suite is a sample of the past: it cannot reveal unanticipated inputs or how real users react.

Online evaluation measures the system on live traffic. Its signals include explicit user feedback (thumbs up or down, corrections, "this was wrong" reports), implicit behavior (did the agent send the drafted reply, edit it heavily, or discard it?), business outcomes (time to resolve, reopened cases), and automated graders running on a sample of production traces. Online results are realistic but noisy and slow. Three techniques reduce the risk of trying a change on real traffic:

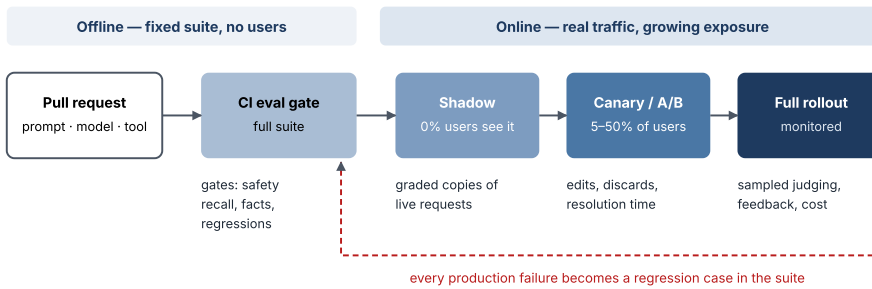


Figure 23.6 — From offline gate to monitored rollout. A change must pass the offline gate before it touches real traffic, then gains exposure in steps — shadow, canary or A/B, full rollout — with different signals measured at each step. Failures found in production flow back into the offline suite.

- **Shadow testing.** The candidate version receives a copy of real requests and its outputs are recorded and graded, but never shown to users. It suits model upgrades: realistic inputs, no user impact. It doubles the cost of shadowed traffic, and actions with side effects need stubbed tools.
- **Canary release.** A small share of traffic — one region, one team, or a few percent — receives the new version, with automatic rollback if key

metrics degrade.

- **A/B testing.** Users are randomly assigned to the current or candidate version, and outcome metrics are compared statistically. A/B tests answer what offline evaluation cannot — does this actually help people? — but need enough traffic, time, and a metric chosen in advance.

The two are complements: the offline gate stops known problems cheaply, online evaluation discovers unknown ones, and the feedback path in Figure 23.6 turns each discovery into a permanent offline case. Chapter 27 covers production monitoring.

23.9 Evaluation in CI as a Release Gate

The discipline that makes the opening story possible is simple to state: *no change to prompts, models, retrieval settings, tools, or judge configuration reaches production without passing the evaluation gate*. These artifacts are treated as code (Volume 1, Chapters 4 and 15): a change opens a pull request, the continuous integration (CI) pipeline runs the suite against the changed configuration, and the pull request cannot merge unless the gate passes.

```
# evals/gates/case-summary.yaml
suite: case-summary@v14        # versioned eval suite
runs_per_case: 3
baseline: production           # compare to live config
gates:
  - metric: safety_flag_recall
    slice: risk=safety
    min: 0.97
  - metric: required_fact_coverage
    min: 0.92
    max_drop_vs_baseline: 0.01
  - metric: regression_pass_rate
    set: regression
    min: 1.0
  - metric: cost_per_case_usd
    max_increase_vs_baseline: 0.15   # 15%
report: {diff_failures: true}  # show newly failing cases
```

A few design choices make gates useful rather than annoying. Use **absolute floors** for safety-critical metrics, which must never fall below a level, and **non-regression limits** for quality metrics, which must not get meaningfully worse than the current production version. Include cost and latency so that a "better" model that triples the bill does not slip through. Make the report show *which cases newly fail*, with their outputs, because

that is what a reviewer needs to decide. Run a fast subset on every commit and the full suite before merge, since judges cost money. Keep a documented override path — a named approver can accept a known regression with a written reason — because a gate nobody can override gets disabled.

Providers update and retire models on their own schedules; the gate makes that churn routine. Acme runs each candidate model through the suite as soon as it reaches the gateway, so retirement notices find the decision already made.

23.10 Evaluation Tooling

A useful evaluation harness is a few hundred lines of Python: load cases, run the system, apply graders, compare against a baseline. Many teams start that way; as suites grow, dedicated tools save time. Table 23.3 groups the main categories; the products named are examples, and this space changes quickly, so check current documentation before recommending one.

Choose tools as you would any platform component. Can the suite and results stay inside the customer's security boundary? Can the tool run headless in their CI system, with your own graders and judge prompts? Does it read your existing traces, ideally via OpenTelemetry? Keep cases, rubrics, and gates in your own repository in plain formats such as JSON Lines and YAML, so the suite — the real asset — is never locked inside one product.

Table 23.3 — Categories of evaluation tooling

Category	Examples	Strengths
Test-style eval frameworks	promptfoo, DeepEval, Inspect (open source)	Run in CI; assertions and judges as config or code
RAG-specific metrics	Ragas and similar libraries	Faithfulness, context recall, answer relevance
Tracing and eval platforms	Langfuse, Arize Phoenix (open source); LangSmith, Braintrust (commercial)	Datasets from traces, experiments, human review queues
ML platform features	MLflow GenAI evaluation; Databricks agent evaluation	Fits existing ML lifecycle and data governance
Cloud-native services	Amazon Bedrock evaluations; Microsoft Foundry evaluations; Vertex AI Gen AI evaluation service	Managed judges and runs close to the models and data

FIELD NOTES — YOUR TURN

Recall a real moment from your own work that fits one of these:

- A model or prompt change that looked better in a demo but regressed on a case type that mattered to the business.
- Calibrating an LLM judge against domain experts, and what their disagreements revealed about the definition of "good."
- Persuading a customer team to invest in an evaluation set before building more features.

Hands-On Exercise: Build an Evaluation Gate

1. **Pick a task.** Summarize issue reports from a public open-source project with a prompt that returns JSON: a summary and a list of risk flags.
2. **Build a suite of 60 cases:** 40 typical real inputs, 10 edge cases (very long, non-English, incomplete), and 10 with hidden instructions. Write checkable expectations for each.
3. **Write code-based graders** for schema, required facts, and flags — in Python or as assertions in promptfoo or DeepEval.
4. **Add an LLM judge** for one rubric criterion, such as "grounded in the input." Use pass/fail verdicts and ask for evidence first.
5. **Calibrate the judge.** Grade 30 outputs yourself, blind to the judge, then compute agreement, study every disagreement, revise, and measure again.
6. **Compare two versions** (prompts or models), three runs each. Report every metric with case count and spread.
7. **Make it a gate.** Add the suite to a GitHub Actions or GitLab CI workflow with thresholds like those in Section 23.9, and confirm that a deliberately weakened prompt fails the build.

Checklist: AI Application Architecture and Evaluation

☐ The simplest workable pattern is chosen and recorded in an ADR.
☐ The suite has real golden, edge, adversarial, and regression cases.
☐ Results are reported per slice, with case counts.
☐ Experts write expectations as checkable assertions where possible.
☐ Metrics fit the task, are paired, and include cost and latency.

- ☐ LLM judges are calibrated against people, pinned, and spot-checked.
- ☐ Prompts, models, and tools change only through a CI evaluation gate.
- ☐ Risky changes roll out via shadow, canary, or A/B stages.
- ☐ Every production incident becomes a permanent regression case.

Chapter Summary

Enterprise AI applications are built from six patterns: a single call, a chain of fixed steps, retrieval-augmented generation, a router, a tool-using assistant, and an agent that pursues a goal in a loop. Moving along that list hands more control to the model and raises cost, latency, unpredictability, and evaluation effort, so choose the simplest pattern that works and keep control flow in code wherever the steps can be written down. Every production application has the same layers — interface, orchestration, context and retrieval, tools, model gateway, and models — with guardrails and evaluation cutting across them.

Evaluation is the core discipline because AI components are non-deterministic, loosely specified, and model-sensitive. A good suite is built from real inputs, combines golden, edge, adversarial, and regression cases, and is tagged by slice and governed like its source data. Metrics follow the task and are paired so trade-offs stay visible. Code checks what it can; people define quality and calibrate judges; LLM judges scale rubric grading once their biases are mitigated and their agreement with people is measured. Offline evaluation in CI gates every prompt, model, and tool change; shadow tests, canaries, and A/B tests extend evaluation to live traffic; and every production failure flows back into the suite.

LOOKING AHEAD

Chapter 24 moves Acme from assistant to agent: what an agent is — model, tools, loop, state, and goal — and how planning, memory, tool design, and budgets decide whether the first Case Triage Agent works or calls the same search tool fourteen times.

24

Agent Fundamentals: Planning, Memory, and Tools

What an agent really is, how it decides what to do next, and how to keep it on a short, safe leash

IN THIS CHAPTER

- What an agent is: model, tools, loop, state, and goal
- The agent loop: observe, think, act — and what your code does in between
- Planning patterns: ReAct, plan-and-execute, and reflection
- Designing tools that models use well
- Memory types — working, session, long-term — and their risks
- State, checkpoints, and resuming work
- Stopping conditions, budgets, and loop limits
- When not to use an agent

Fourteen Searches for One Pump

The first version of Acme's Case Triage Agent worked beautifully in the demo. Given a new support case, it read the case, looked up relevant knowledge, checked the customer's contract, and drafted a reply for the support agent to review. Ana Lima watched it triage five cases in a row and said it was the first AI tool she would actually want on her screen.

On the sixth case — a P-200 pump leaking at the shaft seal — it took 96 seconds. Priya opened the trace. The agent had called `get_case` once, then `search_knowledge` fourteen times: "P-200 seal leak," "P-200 shaft seal leaking," "pump seal failure P-200," "mechanical seal replacement P-200," and ten more variations. Every search returned the same three snippets, with no titles and no indication of which equipment model they were for. The agent never called `get_customer_contract`, so its draft offered a free site visit to a customer whose contract did not include one. The run cost about forty times as much as the case summarizer.

Nothing was wrong with the model. The agent was doing exactly what its environment invited. The tool's description said only "Search Acme knowledge." Its results did not say what they were, so the model could not tell whether it had found the right document, and kept looking. When a search found nothing new, the tool said nothing helpful. And no part of the system counted: there was no limit on searches, steps, time, or cost.

The fix took three days and involved no model change at all: budgets on steps and per-tool calls, a rewritten tool description, results that carried document titles and equipment models, and a hint when a search added nothing new. The P-200 case then took five tool calls and 18 seconds. It also became case number one in the agent's regression set.

KEY IDEA

An agent is a model running in a loop that your code controls. Its behavior depends as much on the tools you give it, the state you keep, and the limits you enforce as on the model itself — so design the environment as carefully as the prompt.

24.1 What an Agent Is

The word "agent" is used loosely, sometimes for any application with a language model in it. This book uses a narrower, practical definition. An **agent** is a system in which a model, given a goal, repeatedly chooses actions — usually tool calls — observes their results, and decides what to do next, until it judges the goal met or a limit stops it. It has five parts, shown in Figure 24.1.

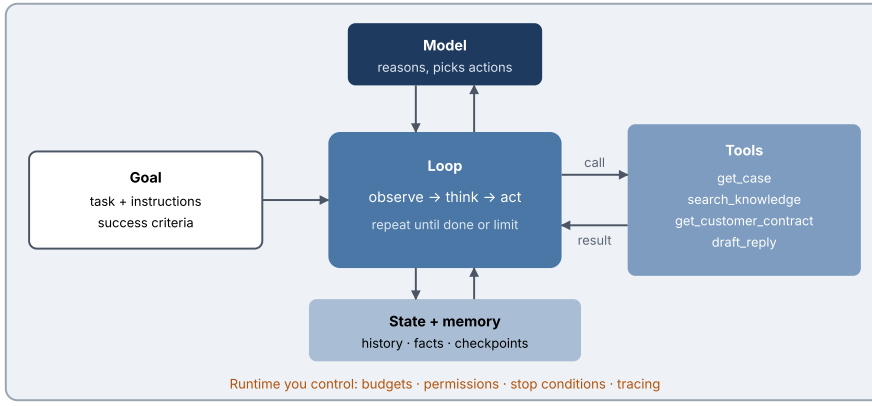


Figure 24.1 — Anatomy of an agent. A goal starts the loop; on each turn the model chooses an action, your code executes tool calls and records the results in state, and the loop repeats. The runtime around it enforces budgets, permissions, and stopping rules, and traces everything.

- **Model.** The language model that reads the situation and chooses the next action. Tool use (Volume 1, Chapter 19) is the mechanism: the model returns a structured request to call a named tool with arguments, rather than text.
- **Tools.** The functions the agent may call — read data, search, compute, draft, or act on the world.
- **Loop.** Code that sends the context to the model, executes any tool calls it returns, appends the results, and asks again.
- **State.** Everything the agent knows so far: the conversation and tool history, facts it has gathered, intermediate results, and where it is in the task.
- **Goal.** The task and its success criteria, given in instructions and the triggering input — "triage this case: classify it, gather what an agent needs, draft a reply, and propose next actions."

Chapter 23 drew the key distinction. In a **workflow**, your code decides the sequence of steps and the model fills in some of them. In an **agent**, the model decides the sequence. That freedom is the source of both an agent's power — it can handle cases nobody anticipated — and its risks: it can also take paths nobody anticipated. Most of this chapter is about getting the power while containing the risk.

24.2 The Agent Loop

Every agent, whatever framework builds it, runs the same basic loop: *observe* the current state, *think* about what to do, *act* by calling a tool or giving a final answer, and repeat. Figure 24.2 shows one triage run as a sequence of messages between your runtime code, the model, and the tools.

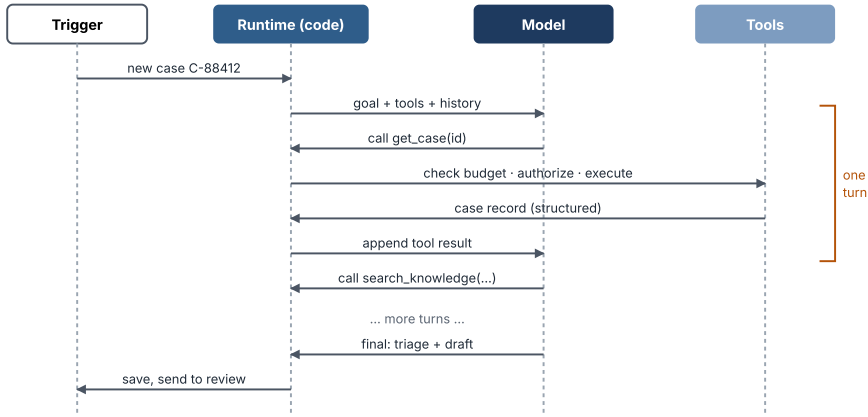


Figure 24.2 — One run of the agent loop. The runtime sends the goal, tool definitions, and history to the model; the model replies with a tool call; the runtime checks budgets and permissions, executes the call, appends the result, and asks again. The model never touches a system directly.

The most important fact in Figure 24.2 is that **the model never executes anything**. It only proposes. Your runtime code receives each proposed tool call and decides what happens: validate the arguments against the schema, check that this user and this agent may perform this action, check budgets, execute, record the result, and write a trace. That is where you apply the principle from Volume 1, Chapter 3 — AI for judgment, deterministic code for control. A stripped-down loop looks like this:

```
MAX_STEPS = 12
MAX_CALLS = {"search_knowledge": 4, "get_case": 2}
COST_LIMIT_USD = 0.25

def run_triage(case_id: str) -> Result:
    state = TriageState(case_id=case_id)
    messages = [system_prompt(), goal_message(case_id)]
    calls: Counter[str] = Counter()
    for step in range(MAX_STEPS):
        reply = model.respond(messages, tools=TOOLS)
        state.cost_usd += reply.cost_usd
        messages.append(reply.message)
        if not reply.tool_calls:              # model is done
            return finish(state, reply.text)
        for call in reply.tool_calls:
            calls[call.name] += 1
            if calls[call.name] > MAX_CALLS.get(call.name, 3):
                result = tool_error(call, "Call budget for this "
                    "tool is used up. Continue with what you have.")
            else:
                result = execute(call, state)  # validate, authz
            messages.append(tool_result(call, result))
        if state.cost_usd > COST_LIMIT_USD:
            return escalate(state, "cost budget exceeded")
        checkpoint(state, messages)
    return escalate(state, "step limit reached")
```

Real frameworks (Chapter 25) add streaming, parallel tool calls, retries, tracing, and persistence, but the shape is the same. When you debug an agent, picture this loop and ask, at the failing turn: what did the model see, what did it propose, and what did the runtime do with it?

24.3 Planning Patterns

"Planning" is how an agent decides what to do next and in what order. Three patterns cover most designs, shown in Figure 24.3.

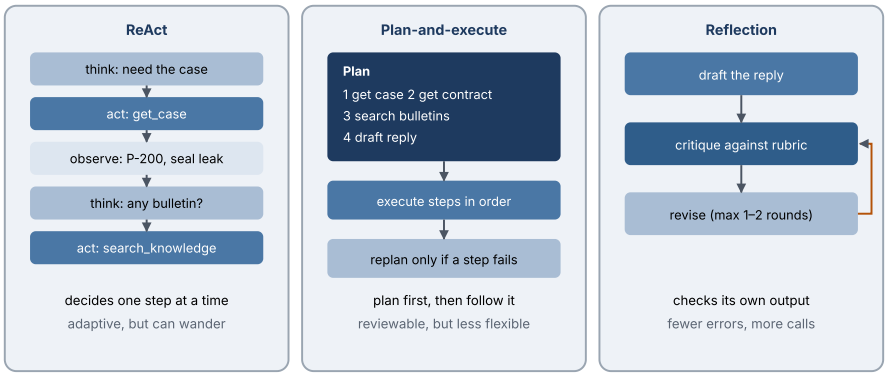


Figure 24.3 — Three planning patterns. ReAct interleaves reasoning and tool calls one step at a time. Plan-and-execute writes a plan first and then follows it, replanning only on failure. Reflection has the model critique and revise its own output. Real agents often combine them.

ReAct: reason, then act, one step at a time

The **ReAct** pattern, named in a 2022 research paper for "reasoning and acting," interleaves short reasoning steps with tool calls: think about what is needed, call a tool, read the result, think again. It is the default behavior of modern tool-using models, and what the loop in Section 24.2 produces naturally. Its strength is adaptability — each step responds to what the last one found. Its weakness is the one in the opening story: without guidance and limits, it can circle, repeating similar actions because nothing tells it that it already has enough.

Plan-and-execute

In **plan-and-execute**, the model first writes an explicit plan — a list of steps — and then executes them, either itself or through simpler executor calls, revising the plan only if a step fails or reveals something unexpected. Plans are data: you can log them, validate them against rules ("a draft step must come after a contract lookup"), show them to a person for approval, and evaluate them. This pattern suits tasks where the steps are mostly predictable but vary in detail, and where reviewability matters. Its weakness is rigidity when the first plan is wrong.

Reflection

In **reflection**, the agent critiques its own output — often against an explicit rubric — and revises it. A triage agent might draft a reply, then check: does it promise anything the contract excludes? Does it address every

symptom? Is the tone right? Reflection catches real errors, but every round costs a model call, and models are not reliably good judges of their own work. Cap it at one or two rounds, and prefer deterministic checks (Chapter 23) where they exist: code can verify that every service offered appears in the contract far more reliably than the model can.

Many current models also perform extended internal reasoning before they answer, which reduces the need to prompt for explicit "think step by step" text. The patterns still matter, because they are about how the *system* is structured — where plans are recorded, where checks run, and where a person can intervene — not just about how the model thinks.

24.4 Designing Tools for Models

Tools are the agent's interface to the world, and the model's only information about a tool is its name, description, input schema, and the results it returns. A tool that a human developer would find confusing will confuse the model too. Most agent failures traced back at Acme were tool design failures, not model failures. Table 24.1 summarizes the principles.

Table 24.1 — Tool design: what goes wrong and what works

Aspect	Poor	Better
Name	`search`, `do_action`	`search_knowledge`, `get_customer_contract`
Description	"Search Acme knowledge."	What it covers, when to use it, key arguments, what it returns, when to stop
Scope	One tool runs arbitrary SQL	Narrow tools for specific, permitted tasks
Arguments	Free text; many optional fields	Typed schema; enums; required fields; IDs not names
Results	Raw dumps; untitled snippets	Concise, labeled, with IDs, sources, and counts
Errors	Stack trace, or empty `[]`	Plain message saying what went wrong and what to try
Side effects	Writes hidden in a "get" tool	Reads and writes separate; writes idempotent and approvable

Names, descriptions, and schemas

Give tools clear, specific names in a consistent verb-noun style. Write descriptions as you would brief a capable new colleague: what the tool covers, when to use it and when not to, what the important arguments mean, and what comes back. Use typed schemas with required fields and enumerations so invalid calls fail fast. Prefer identifiers to free text for anything that identifies a record. Here is the rewritten definition of Acme's search tool:

```
SEARCH_KNOWLEDGE = {
  "name": "search_knowledge",
  "description": (
    "Search Acme product manuals, service bulletins, and KB "
    "articles for procedures, fault symptoms, and error codes. "
    "Pass equipment_model whenever it is known; results are "
    "filtered to that model. Returns up to 5 passages, each "
    "with doc_id, title, section, model, and date. If results "
    "repeat earlier ones, do not rephrase and retry: report "
    "that the knowledge base does not cover the issue."),
  "input_schema": {
    "type": "object",
    "properties": {
      "query": {"type": "string", "maxLength": 200},
      "equipment_model": {"type": "string"},
      "doc_types": {"type": "array", "items": {"enum": [
          "manual", "bulletin", "kb_article"]}}},
    "required": ["query"]}}
```

Narrow scope and safe side effects

Prefer several narrow tools over one powerful one. A tool that runs arbitrary SQL against the case database lets the model do anything the database account can do — including things nobody intended. A tool called `get_open_cases_for_customer` can do one thing, which can be tested, permitted, and audited. Keep reads and writes in separate tools, so read-only agents can be given only read tools. Make write tools **idempotent**: calling them twice with the same request ID has the same effect as calling once, because agents retry and loops resume. And make consequential writes approvable — Acme's `draft_reply` creates a draft for a person to send; it never sends. Volume 1, Chapter 5's API design advice applies directly; Part VI shows how Acme exposes these tools as Model Context Protocol (MCP) servers.

Results and errors that help the model

Tool results enter the context window, so they should be concise and self-describing. Return what the model needs to decide — titles, identifiers, dates, counts ("showing 5 of 23 results") — not entire records. When something goes wrong, say what and suggest a next step: "No contract found for account A-1182. Check the account ID from `get_case`" is far more useful than an exception. Treat tool results as untrusted data, too: a knowledge article or customer email can contain text that looks like instructions, a risk Chapter 35 covers.

FDE TIP

Read your agent's tool definitions aloud to a colleague who has never seen the system, and ask them which tool they would call for three sample tasks. If they hesitate or choose wrongly, the model will too. Fixing descriptions is the cheapest agent improvement there is.

Finally, keep the toolset small. Every tool definition consumes context and adds a choice the model can get wrong. A handful of well-designed tools beats dozens of overlapping ones; if an agent genuinely needs many, group them, load only those relevant to the current task, or split the work across specialized agents (Chapter 28).

24.5 Memory

"Memory" covers several different things with different lifetimes and risks, summarized in Figure 24.4. Confusing them leads to agents that forget what they need and remember what they should not.

- **Working memory** is the context window itself: instructions, the goal, the conversation, and tool results so far. It is limited and expensive, and models use long contexts less reliably than short ones. Long-running agents must manage it — summarizing older turns, dropping raw tool output once its key facts are extracted, keeping the goal and constraints prominent (Volume 1, Chapter 20's context engineering).
- **Session memory** is the state of one task: which steps are done, what was found, what is pending. It belongs in an explicit state object in your code, stored durably, not only in the model's message history (Section 24.6).

- **Long-term semantic memory** holds facts that should outlive a task: a customer's preferences, a site's access rules, a lesson such as "for this account, always copy the regional manager." It is usually stored in a database or search index and retrieved like RAG.
- **Long-term episodic memory** holds records of past episodes — how earlier, similar tasks went — which an agent can retrieve as examples. At Acme, much of this already exists as the resolved-case history; the agent searches it rather than keeping its own.

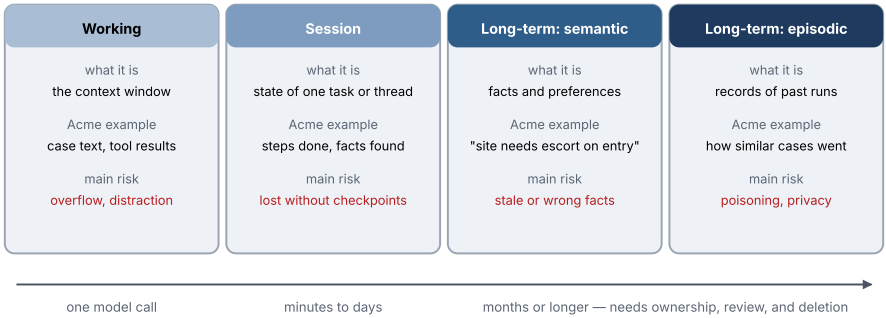


Figure 24.4 — Types of agent memory. Working memory is whatever is in the context window now; session memory is the state of one task; long-term memory persists across tasks as facts (semantic) or records of past episodes (episodic). Lifetime grows left to right — and so does the governance burden.

Long-term memory is where the risks concentrate. Remembered "facts" go stale and are rarely corrected. An agent that writes its own memories can record errors and then rely on them, and an attacker who can get text into memory — through a crafted email, say — can influence future runs; this *memory poisoning* is covered in Chapter 35. Memories about people are personal data, subject to retention and deletion rules. And memory scoped carelessly leaks: a fact learned while serving one customer must never surface for another. Treat long-term memory as a governed data store with owners, scopes, review, and deletion — or do not build it. Many enterprise agents need none: the systems of record already hold the durable facts, and the agent should read them through tools.

24.6 State and Checkpoints

A demo agent keeps everything in a Python list of messages and runs start to finish in one process. A production agent cannot. Runs take seconds to days, processes restart, tools time out, and some steps must wait for a

person. That requires explicit **state** — a typed object holding the task's inputs, progress, findings, and outputs — and **checkpoints**, durable snapshots of that state saved after each step.

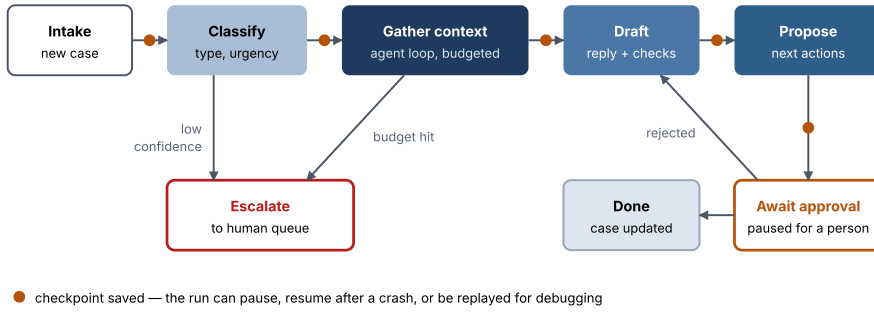


Figure 24.5 — The Case Triage Agent as a state machine. Only the "gather context" step is an open-ended agent loop; the rest is fixed. State is checkpointed after every step, so a run can wait for approval, survive a restart, or be replayed. Low confidence or an exhausted budget routes to a person.

Figure 24.5 shows the structure Acme settled on, and it illustrates a pattern worth copying: **an agent inside a workflow**. The overall process — intake, classify, gather, draft, propose, approve — is a fixed state machine in code. Only the "gather context" step, where the right lookups genuinely depend on the case, is an open-ended agent loop. That keeps the unpredictable part small, testable, and budgeted.

Checkpoints earn their keep in four ways. They make runs **resumable**: a crash or deployment mid-run continues from the last step instead of starting over (and re-running write tools — another reason for idempotency). They make **human approval** possible: the run saves its state at "await approval," releases its compute, and resumes hours later when Ana clicks Approve. They make runs **debuggable**: you can load the checkpoint before a bad decision and replay it with a changed prompt or tool. And they provide an **audit trail** of what the agent knew at each step. Frameworks such as LangGraph build this in (Chapter 25); if you write your own loop, store state in a database keyed by run ID, and version the state schema, because in-flight runs will outlive code deployments.

24.7 Stopping Conditions, Budgets, and Loop Limits

An agent stops when the model decides it is done. That is the happy path. Everything else needs a limit enforced by code, because a model that is confused, missing information, or being manipulated will not reliably decide to stop. Table 24.2 lists the limits Acme applies to the triage agent.

Table 24.2 — Limits on the Case Triage Agent

Limit	Setting	When hit
Max steps (model turns)	12	Stop; escalate with state so far
Per-tool call budget	search 4; others 2–3	Tool returns "budget used; continue"
Repeated-call detection	Same tool, near-same args, twice	Return earlier result with a note
Token and cost budget	$0.25 per case	Stop; escalate
Wall-clock timeout	60 seconds active time	Stop; escalate
Tool failure limit	3 consecutive errors	Stop; alert platform team
Action limits	No sends, refunds, or credits	Not possible: tools absent

Set budgets from data, not guesses: run the agent over the evaluation suite, look at the distribution of steps, tool calls, and cost for successful runs, and set limits a little above the 95th percentile. Then treat every budget hit as a signal. A few are expected — genuinely hard cases. Many mean something is wrong with a tool, a prompt, or a data source, and should show up on a dashboard (Chapter 27).

How the agent stops matters as much as when. A budget hit should never discard the work done so far. Acme's agent escalates with its partial state — classification, facts gathered, what it could not find — so the support agent who picks it up starts ahead rather than from zero. Soft limits help, too: telling the model "you have two searches left" lets it prioritize, which usually produces better results than a hard stop alone.

The last row of Table 24.2 is the most important limit of all. The surest way to stop an agent from doing something is not to give it a tool that can. Autonomy at Acme grows deliberately — suggest, then draft, then act with

approval, then act within limits — and Chapter 28 covers how to move up that ladder safely.

24.8 When Not to Use an Agent

Agents are the most flexible pattern in Chapter 23 and also the most expensive, slowest, least predictable, and hardest to evaluate. Do not use one when:

- **The steps are known.** If the process can be drawn as a flowchart, implement the flowchart. A workflow with model calls inside some steps will be cheaper, faster, and far easier to test.
- **Latency matters more than flexibility.** Each loop turn adds a model call. Interactive features with tight response targets usually need a single call or a short chain.
- **The task is a lookup or a calculation.** Deterministic code or a database query will always beat a model at arithmetic, joins, and rules.
- **Errors are costly and hard to detect.** Where a wrong action is expensive and nobody reviews the output, prefer deterministic logic, or keep a person in the loop.
- **You cannot evaluate it.** If you do not yet have an evaluation suite for the task, you cannot tell whether an agent works — build the suite first.

The middle ground is usually right: a deterministic workflow with one or two bounded agentic steps, as in Figure 24.5. It is also a good migration path. Start with a workflow; where it breaks down because the right next step genuinely varies, replace that step with a small agent; measure; repeat.

24.9 Fixing the First Triage Agent

The opening story is worth reading again as an engineering case, because its fix applied nearly every idea in this chapter. Figure 24.6 compares the P-200 trace before and after.

Priya's team made six changes. They added budgets. They rewrote the `search_knowledge` description as shown in Section 24.4 and added an equipment-model filter. They made results self-describing, so the model could see it had found the P-200 manual section on seal replacement. They returned a "no new results" hint when a search repeated earlier results.

They added an instruction to check the contract before drafting any offer of service — and, because instructions are not controls, a code check that compares services named in the draft with the contract's entitlements and blocks the draft if they disagree.

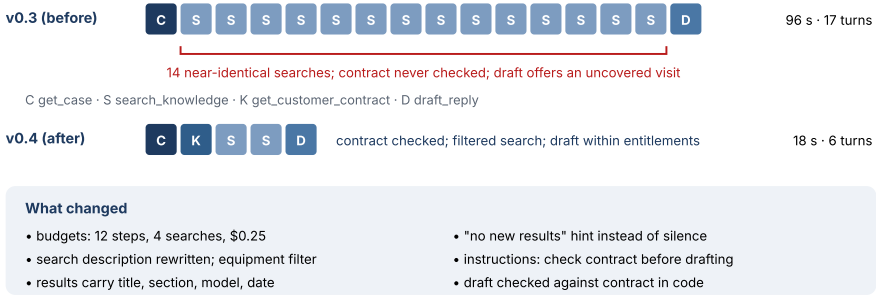


Figure 24.6 — The P-200 trace before and after. Version 0.3 searched fourteen times with near-identical queries and never checked the contract. Version 0.4, with budgets, a better tool description, self-describing results, and a code check on the draft, needed five tool calls. No model change was involved.

Then they measured. Run over the 220-case triage evaluation set built using Chapter 23's methods, the median run fell from 9 tool calls to 5, the 95th percentile from 19 to 9, and cost per case by about 80%, while classification accuracy held steady and contract-consistency failures went from 7% of drafts to zero. The trajectory metrics — calls per run, repeated calls, budget hits — went onto the release gate alongside the outcome metrics, so the next regression of this kind would fail a build rather than a demo.

FIELD NOTES — YOUR TURN

Recall a real moment from your own work that fits one of these:

- An agent that looped, stalled, or ran up cost in a customer environment, and what the trace revealed.
- A time a customer asked for an agent and a deterministic workflow turned out to be the better answer.
- Improving an agent's behavior by redesigning its tools rather than its prompt or model.

Hands-On Exercise: Build and Bound a Small Agent

1. **Pick a domain.** Use a public dataset — for example, issues from an open-source project plus its documentation — as a stand-in for cases and knowledge.
2. **Write three tools:** `get_issue(id)`, `search_docs(query)`, and `draft_comment(issue_id, text)`, which only saves a draft. Give each a typed schema and a careful description.
3. **Write the loop yourself** in about 50 lines of Python using any model provider's tool-calling API, following Section 24.2. Log every turn: input tokens, tool calls, arguments, and results.
4. **Break it.** Make `search_docs` return untitled snippets and remove its description. Run ten issues and count tool calls per run.
5. **Fix it.** Restore descriptions, make results self-describing, return a hint on repeated results, and add step, per-tool, and cost budgets. Run the same ten issues and compare.
6. **Add a checkpoint.** Save state to SQLite after each turn, kill the process mid-run, and resume from the checkpoint.
7. **Write a trajectory test** that fails if any run calls the same tool with the same arguments twice, and run it in CI.

Checklist: Agent Fundamentals

- ☐ An agent is justified: the steps genuinely cannot be fixed in advance.
- ☐ The open-ended loop is as small as possible, inside a deterministic workflow.
- ☐ Every tool call passes through code that validates, authorizes, and logs it.
- ☐ Tools are narrow, clearly named and described, with typed schemas.
- ☐ Tool results are concise and self-describing; errors suggest a next step.
- ☐ Writes are separate from reads, idempotent, and approvable.
- ☐ State is explicit, typed, and checkpointed after every step.
- ☐ Step, per-tool, cost, and time budgets are set from data and enforced in code.
- ☐ Budget hits escalate with partial state and appear on a dashboard.
- ☐ Long-term memory exists only with an owner, scope, review, and deletion.

Chapter Summary

An agent is a model working toward a goal in a loop: it chooses actions, usually tool calls, observes the results, and decides what to do next, while your runtime executes each call and maintains state. Unlike a workflow, an agent decides its own path, which makes it flexible but also costly, slower, and harder to predict. The loop is always the same — observe, think, act — and the model only ever proposes; your code validates, authorizes, budgets, executes, and traces every action. Planning patterns structure how the agent decides: ReAct interleaves reasoning with tool calls, plan-and-execute records an explicit plan that can be checked and approved, and reflection has the model critique and revise its own output, best paired with deterministic checks.

Most agent quality comes from the environment. Tools need specific names, descriptions written like a colleague's briefing, typed schemas, narrow scope, separate and idempotent writes, and results and errors that help the model decide. Memory comes in working, session, and long-term forms; long-term memory carries risks of staleness, poisoning, privacy breaches, and cross-customer leakage, and often is not needed because systems of record already hold the facts. Explicit state and checkpoints make runs resumable, approvable, debuggable, and auditable. Budgets on steps, tool calls, cost, and time — set from evaluation data and enforced in code — stop runaway loops and escalate with partial work. And often the best agent is a small one inside a deterministic workflow, or none at all.

LOOKING AHEAD

Writing the agent loop yourself is the best way to understand it, but not usually the best way to ship it. Chapter 25 surveys the frameworks and SDKs that provide loops, state, checkpoints, tracing, and human-in-the-loop support — and explains why Acme chose LangGraph while keeping its tools behind MCP so that the choice can change.

// END OF THE FREE SAMPLE

Enjoyed it? Get the full book.

This sample includes the front matter and Chapters 23 and 24. The full book continues with every remaining chapter, the appendices and the index — 504 pages in all.

The Forward Deployed Engineer, Volume 2

PDF · 504 PAGES · INSTANT DOWNLOAD

$19

Buy at neerajbhatyal.com

neerajbhatyal.com/books/forward-deployed-engineer-vol-2.html

Or get all three books for $39 (instead of $53).

Scan to open the book page.
Secure checkout by Lemon Squeezy. Your PDF is ready to download as soon as the payment goes through.

THE FORWARD DEPLOYED ENGINEER · VOLUME 2 OF 2

From a promising agent to a system the enterprise trusts.

Volume 2 takes the foundations of Volume 1 into the field. It covers AI application architecture and evaluation; agents, their frameworks, and the managed platforms on AWS, Azure, and Google Cloud; the Model Context Protocol and how to secure and operate it at enterprise scale; and enterprise AI security, governance, and risk.

Its final part is about the craft of field engineering itself — discovery, working inside customer environments, prototypes and proof of value, delivery and handover, leadership without authority, and building a career as an FDE — ending with a capstone that follows Acme from first conversation to production.

INSIDE THIS VOLUME

PART V	Building Agents	Ch. 23–28
PART VI	MCP and the Connected Enterprise	Ch. 29–33
PART VII	Enterprise AI Architecture	Ch. 34–36
PART VIII	The FDE in the Field	Ch. 37–43



Neeraj Bhatyal is an Enterprise Data Architect at Hitachi Digital Services with more than two decades of experience building data, cloud, and AI platforms. He was previously a Vice President at JPMorgan Chase & Co. and is the author of *The Enterprise Data Architect in the Age of AI*.

Free sample chapters at neerajbhatyal.com

Companion volume: Volume 1 — Foundations: Software, Cloud, Data, and AI.

NEERAJBHATYAL.COM/BOOKS