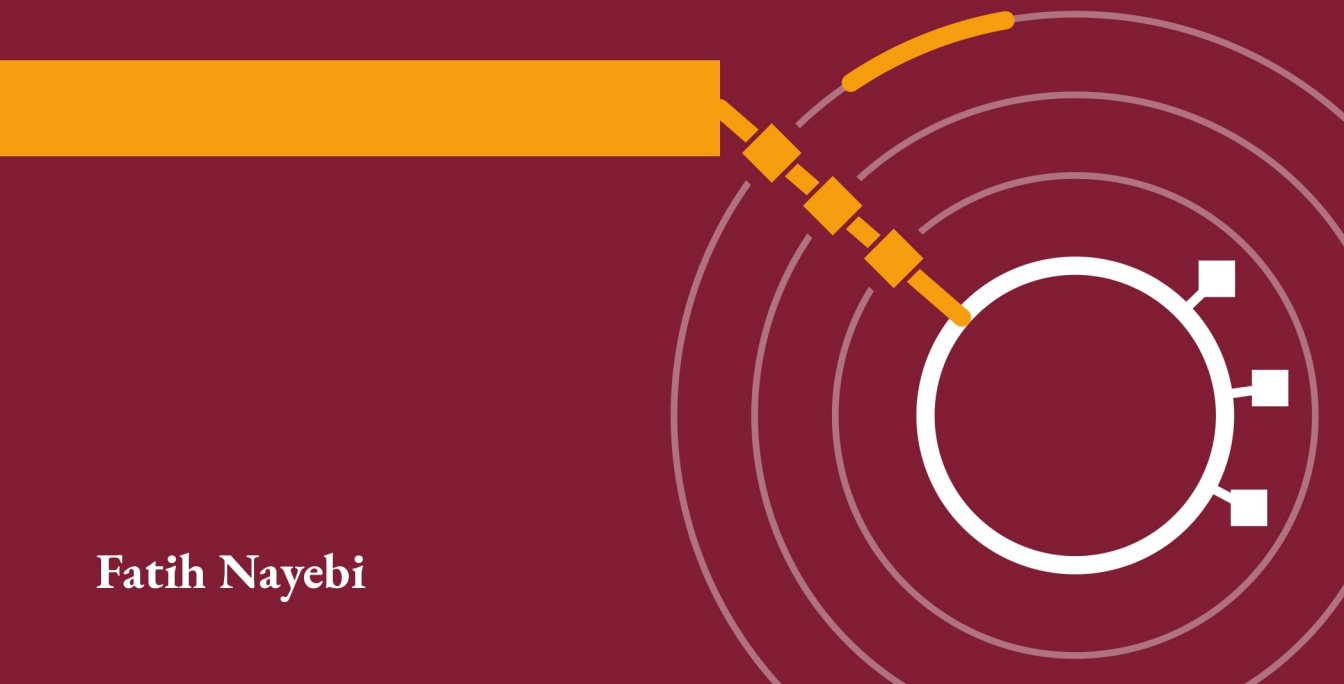
Foundations of
Agentic AI Systems
Volume II

Agents, Operations, and Governance, in Production

Autonomy should rise
only as evidence rises.



Fatih Nayebi

FIRST EDITION

Foundations of Agentic AI Systems

Volume II: Agents, Operations, and Governance, in Production

Fatih Nayebi

Sample - Not for Redistribution

Gradient Divergence
Montréal, QC, Canada

Foundations of Agentic AI Systems, Volume II
Agents, Operations, and Governance, in Production
First edition

Copyright © 2026 Fatih Nayebi. All rights reserved.
No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law.

Published by Gradient Divergence, Montréal, Québec, Canada
For permissions and inquiries, write to contact@gradientdivergence.com.

First published 2026. Version 1.0 (build 02e5a221b).
ISBN 978-1-0694226-8-2 (hardcover)
ISBN 978-1-0694226-7-5 (paperback)

The code in this book is released under the MIT License at
https://github.com/gradient-divergence/agentic-ai-systems.
Please report errors there as issues.

Models, interfaces, prices, and regulations are described as they stood in October 2026.
Nothing in this book is legal advice.
While every precaution has been taken in the preparation of this book, neither the author nor the publisher assumes any liability for errors or omissions, or for damages resulting from the use of the information contained herein.
Product and company names may be trademarks of their owners and are used here editorially.

Set in STIX Two and Fira Sans, with code in Inconsolata, using LuaLaTeX.

Sample – Not for Redistribution

Herr, die Noth ist groß!
Die ich rief, die Geister,
Werd' ich nun nicht los.

— Johann Wolfgang von Goethe, "Der Zauberlehrling," 1797

Master, the need is great!
The spirits that I summoned
I now cannot be rid of.

Sample - Not for Redistribution

Goethe, "Der Zauberlehrling," in *Goethe's Werke. Vollständige Ausgabe letzter Hand*, vol. 1 (1827, 220); first published in Schiller's *Musen-Almanach für das Jahr 1798*. English translation by the author.

Table of Contents

Preface xiii
Notation xvii
What Volume II Rests On xxxv

PART V
Agentic Systems Architecture

22 The Agentic Loop 3
23 Perception and Tools 17
24 Reasoning and Validated Planning 35
25 Action, Guardrails, and Policies 49
26 Memory and Retrieval 65
27 Multi-Agent Systems and Independent Verification 79
28 Agent Frameworks and Orchestration 97
29 Agent Evaluation and Release Evidence 109

PART VI
Engineering and Operations

30 Data Systems for Agentic AI 129
31 Batch and Streaming Pipelines 141
32 Deployment and Rollback 155
33 MLOps and ML Engineering 169
34 Observability and Reliability 181
35 Security and Privacy 199

PART VII
Governance and Adoption

36 Outcome Audits and Decision Explanations 221
37 Human Oversight and Risk Management 241
38 Organizational Adoption 265

PART VIII
Capstone and Futures

39 Capstone: Design, Build, Deploy, Operate 281
40 Domain-Agnostic Transfer Cases 295
41 Future Directions and System Renewal 309

Appendix B: Engineering Practices 321
Appendix C: Exercises, Projects, and Rubrics 325
References 347
Acknowledgments 355
About the Author 357
Index 359

Sample - Not for Redistribution

Preface xiii

Notation xvii

What Volume II Rests On xxxv
What the refund agent may decide xxxv
What runs at stage 0, and where each rule lives xxxvii
What evidence earns each step xxxviii
What a score is graded against xl
How a probability becomes an action xli
When a probability means what it says xlii
What a draft reads, and what it costs xliv
How a change to the drafts is judged xlv
Who reviews which draft, and what is worth learning xlvi
How far a learner may explore xlvi
The stage 3 record xlvii

PART V Agentic Systems Architecture

22 The Agentic Loop 3
22.1 What a loop adds to a model call 3
22.2 Who chooses the next step 4
22.3 One refund, followed through the loop 5
22.4 The loop as a state machine 6
22.5 How a run stops, waits, and resumes 10
22.6 Retries, repeated payments, and errors that compound 12
22.7 What the scenario scores show 13
22.8 Summary 14
22.9 Exercises 14

23 Perception and Tools 17
23.1 The age of a read at the moment it is used 17
23.2 When a fresh read is the wrong evidence 19
23.3 What a tool promises 20
23.4 The payment tool's contract: one key, one payment 23
23.5 Testing the promises, and what they cannot catch 25
23.6 What a convolution computes 26
23.7 When a vision model's error rates stop holding 28
23.8 Reading a screen, and checking what an action did 30
23.9 Summary 32
23.10 Exercises 33

24 Reasoning and Validated Planning 35
24.1 When an agent should plan before it acts 35
24.2 When a sequence of schema-valid calls is a valid plan 36
24.3 Bounding a plan 39
24.4 Finding a valid plan, and choosing among them 39
24.5 Validating the refund agent's plans 43
24.6 Replanning when the world changes 45
24.7 Summary 46
24.8 Exercises 47

25 Action, Guardrails, and Policies 49
25.1 Why a rule in a prompt does not stop an action 50
25.2 A policy gate in the execution path 51
25.3 Why the gate checks again at execution time 54
25.4 What an approval approves 55
25.5 An approval step that can still say no 58
25.6 When the action cannot be undone 59
25.7 Reference: the gate's checks and records 61
25.8 Summary 62
25.9 Exercises 62

26 Memory and Retrieval 65
26.1 What an agent remembers, and where 65
26.2 Why retrieval losses multiply 66
26.3 What ranking can recover, and what it cannot 68
26.4 Fitting evidence into the window 69
26.5 When a fact held, and when the memory knew it 72
26.6 The refund agent's memory lifecycle 74
26.7 Summary 77
26.8 Exercises 77

27 Multi-Agent Systems and Independent Verification 79
27.1 When one agent should become two 79
27.2 How much a second agent catches 80
27.3 What the verifier must read for itself 83
27.4 A verifier for the refund agent, measured on matched cases 84
27.5 What crosses between two agents 89
27.6 When agents want different things 91
27.7 When a second agent is the wrong tool 94
27.8 Summary 95
27.9 Exercises 95

28 Agent Frameworks and Orchestration 97
28.1 What a framework may supply, and what it may not 97
28.2 Showing that a runtime is only an adapter 100
28.3 A harness for an agent that edits code 105
28.4 Summary 108
28.5 Exercises 108

29 Agent Evaluation and Release Evidence 109
29.1 An evaluation that can say no 109
29.2 Scenarios, and the effects their pass conditions check 110
29.3 Running each scenario more than once 113
29.4 Graders that can be wrong 116
29.5 The release rule, and the candidates it stops 117
29.6 Shadow: the switch, the comparison, and what it cannot see 122
29.7 Summary 125
29.8 Exercises 125

Sample - Not for Redistribution

PART VI Engineering and Operations

30 Data Systems for Agentic AI 129
30.1 What a record must mean 129
30.2 Who answers for a source 132
30.3 A freshness budget from how a source changes 133
30.4 What a decision rested on 135
30.5 The refund agent's data contracts at stage 5 137
30.6 Summary 139
30.7 Exercises 139

31 Batch and Streaming Pipelines 141
31.1 Two times on every scan 141
31.2 When a record is complete enough 143
31.3 Which value each decision reads 147
31.4 Replays and corrections that pay nothing twice 150
31.5 Summary 152
31.6 Exercises 152

32 Deployment and Rollback 155
32.1 What changes when money moves 155
32.2 What the canary may pay, and who decides 156
32.3 How much a canary can pay before it stops 159
32.4 Swapping the model behind a canary 160
32.5 What a rollback restores, and what it cannot 161
32.6 Rehearsing the rollback 163
32.7 Summary 166
32.8 Exercises 166

33 MLOps and ML Engineering 169
33.1 What a release consists of 169
33.2 Fingerprinting the refund agent's bundle 171
33.3 What a change must show before it runs 173
33.4 The promotion record 175
33.5 Rolling back, retraining, and what a bundle cannot see 178
33.6 Summary 179
33.7 Exercises 179

34 Observability and Reliability 181
34.1 What a trace adds to the records 181
34.2 What the agent promises, and what counts against it 183
34.3 How much failure the grant allows 186
34.4 What a page asks of the person it reaches 191
34.5 Reconstructing the week 192
34.6 A silence no single decision shows 194
34.7 What the records cannot establish 196
34.8 Summary 197
34.9 Exercises 197

35 Security and Privacy 199
35.1 The agent acts with authority it holds for others 199
35.2 Why a check on the text cannot bound an injection 201
35.3 A credential that carries the requester's authority 202
35.4 Expiry, revocation and what an allowed action can reach 206
35.5 Authority beyond the refund agent's own tools 208
35.6 What the agent may disclose, and to whom 210
35.7 The refund agent's injection tests at stage 5 213
35.8 Summary 217
35.9 Exercises 217

PART VII Governance and Adoption

36 Outcome Audits and Decision Explanations 221
36.1 Who is denied, and who should have been paid 221
36.2 Why the criteria cannot all hold 224
36.3 Auditing denials no one reviewed 226
36.4 What a breach can change 231
36.5 What explains one decision 232
36.6 The refund agent's denial explanation 235
36.7 Summary 240
36.8 Exercises 240

37 Human Oversight and Risk Management 241
37.1 Why the wait explodes near capacity 241
37.2 Sizing the refund desk for stage 6 246
37.3 When a request waits too long 251
37.4 Whether review still refuses when it is busy 254
37.5 The stage-6 review-queue design, and where it fails 257
37.6 Summary 263
37.7 Exercises 263

38 Organizational Adoption 265
38.1 What a route around the agent costs 265
38.2 A rollout plan that counts the work 267
38.3 Finding the work done around the agent 271
38.4 Widen, hold or move back 274
38.5 Summary 276
38.6 Exercises 277

PART VIII Capstone and Futures

39 Capstone: Design, Build, Deploy, Operate 281
39.1 What one record cannot show 281
39.2 What a launch gate checks 282
39.3 The refund agent's launch packet 283
39.4 Running the gate 287
39.5 The comparison with the simpler alternatives 289
39.6 The launch decision, and what a hold leaves standing 291
39.7 Summary 293

Sample - Not for Redistribution

39.8 Exercises 293

40 Domain-Agnostic Transfer Cases 295
40.1 What travels with a decision, and what stays with refunds 295
40.2 An incident desk runs on seconds 297
40.3 When patients compete for the same slots 299
40.4 Which action a bank's agent may take alone 300
40.5 What carried over, and what each domain decided 302
40.6 Reference: taking a decision packet to another domain 306
40.7 Summary 307
40.8 Exercises 307

41 Future Directions and System Renewal 309
41.1 What earned each grant, and why a replacement must earn it again 309
41.2 Replacing the refund agent's risk score 312
41.3 Summary 319
41.4 Exercises 320

Appendix B: Engineering Practices 321

Appendix C: Exercises, Projects, and Rubrics 325
C.1 Scoring the capstone project 325
C.2 The volume's project: one class of decisions, from its packet to its launch gate 326
C.3 Answers, chapter by chapter 327

References 347

Acknowledgments 355

About the Author 357

Index 359

Sample - Not for Redistribution

Preface

What this volume holds

Foundations of Agentic AI Systems is about AI that acts as well as predicts, and it comes in two volumes. Volume I, *Models, Learning, and Decisions, from First Principles*, builds the foundations an agent stands on, from data and evaluation through deep learning to reinforcement learning and decision-making. This volume, *Agents, Operations, and Governance, in Production*, builds the agent on those foundations and takes it to production. It holds Parts V to VIII. The refund agent of Volume I runs through this one too. Across this volume's four parts the agent goes from drafting every decision for a person, through deciding its smallest refunds in shadow, to the designs for a paid canary, a staged rollout to every requester and a launch gate. One principle governs each step: autonomy should rise only as evidence rises. In print, the margin beside each chapter's first line shows its part's map, with the chapter's layer lit and the stage that part takes the agent toward.

Part	Core question	What you get
V: Agentic systems architecture	*What program runs the decision, and stops?*	A program that calls the tools and stops when it should, contracts for its seven tools, a gate inside the call that pays, the switch into shadow.
VI: Engineering and operations	*What must hold before the first payment, and what can still be done after it?*	Records with their guarantees and two times (when each event happened and when it arrived), a canary with its limits and its stop, one versioned bundle, an error budget, scoped credentials.
VII: Governance and adoption	*What must hold before the agent decides tier 1 for everyone?*	Outcome audits and denial explanations, a review queue sized for its arrivals, a staged rollout plan.
VIII: Capstone and futures	*Does the agent pass a launch gate?*	A launch packet and its gate, the same questions in three other domains, a replacement decision.

The capstone sets out what the refund agent's launch packet must hold. In the companion it runs its launch gate on a constructed packet, while the run of simulated operation that would assemble the agent's own packet is still to be made. The chapter after it carries the same decision packet into three other domains, where the packet transfers and the workflow does not.

The bridge from Volume I

The volume opens with *What Volume II Rests On*, a bridge from the first volume. It takes up the refund agent where Volume I leaves it and gathers what the chapters ahead need from that volume. Six artifacts are reprinted as Volume I prints them, and the rest is summarized with a pointer to the section that derives it. If you start with this volume, read the bridge straight through. If you have read Volume I, read five of the bridge's sections before Part V, and come back to the others when a chapter leans on them. Two of them, *What the refund agent may decide* and *What a draft reads, and what it costs*, give the boundary and the checked call that Part V assumes. The last three, from *Who reviews which draft, and what is worth learning* to *The stage 3 record*, give the desk, the learner and the record the agent starts from.

Sample · Not for Redistribution

Reading paths

Chapter numbers are those in the table of contents, and a chapter in Volume I is named with its volume. Each path starts in Volume I and continues here. If you don't have Volume I, the bridge stands in for what the chapters here need from the first column on every path, though it gives the results and leaves their derivations in Volume I.

Path	For whom	First	In this volume
Core	Any reader	Volume I's introduction; Volume I, Chapter 1 to Volume I, Chapter 4; Volume I, Section 5.3 to Volume I, Section 5.6; Volume I, Section 8.1 to Volume I, Section 8.3; Volume I, Section 9.1 to Volume I, Section 9.3 and Volume I, Section 10.1 to Volume I, Section 10.2; Volume I, Chapter 11 to Volume I, Chapter 12; Volume I, Chapter 14	Chapter 22 to Chapter 29
Builder	Engineers shipping an agent	As Core, with Volume I, Section 7.1, Volume I, Section 7.3, Volume I, Section 7.4 and Volume I, Section 7.6	As Core, then Chapter 30 to Chapter 39
Technical lead or architect	Readers who own design and risk	As Builder, with Volume I, Section 15.6; Volume I, Section 20.4; Volume I, Chapter 17 and Volume I, Chapter 21	The Part V opener; Chapter 22, Chapter 23, Chapter 25, and Chapter 29; Chapter 30 to Chapter 35; Chapter 36 to Chapter 38; Chapter 39 to Chapter 41
Sponsor	Executives and product leads	Volume I's introduction; Volume I, Chapter 1; Volume I, Chapter 3	The Part V opener; Chapter 25, Chapter 37, Chapter 38, Chapter 39, Chapter 40, and Chapter 41
Graduate course, 13 weeks	Students with one prior ML course	Volume I, Chapter 1 to Volume I, Chapter 3; Volume I, Chapter 4 to Volume I, Chapter 8; Volume I, Section 9.1 to Volume I, Section 9.3 and Volume I, Section 10.1 to Volume I, Section 10.2; Volume I, Chapter 11, Volume I, Chapter 12, and Volume I, Chapter 14; Volume I, Chapter 16, Volume I, Chapter 17, Volume I, Section 18.4 and Volume I, Section 18.5, and Volume I, Chapter 19	Chapter 22 to Chapter 29; Chapter 30 to Chapter 35; Chapter 36 to Chapter 38; Chapter 39 as the project, then Chapter 40 and Chapter 41
Foundations first	Readers new to ML	Volume I's Parts I to IV in order	Chapter 22 to Chapter 29, as in Core

Sample - Not for redistribution

On the Technical lead path, read Volume I, Section 7.1 and Volume I, Section 7.6 before Chapter 36, and Volume I, Section 7.3 before Chapter 38. The Builder path, which also runs through those two chapters, needs the same sections first.

On the Sponsor path, you take tool contracts and evaluation gates on trust. Chapter 29 adds the scenarios of Volume I, Section 13.4 to its suite. On the agent's first shadow day, whose decisions are logged and not carried out, every payment carries the standard notice. That is because the record of Volume I, Section 21.6 holds back both candidate offer policies, which choose the notice a payment carries, and keeps the fixed offer rule, which sends the standard notice to everyone. The Core, Builder and graduate paths take the scenarios and the record on trust; the Technical lead path reads the record and takes only the scenarios on trust. Chapter 37 restates the review times of Volume I, Section 15.6 for the review desk it sizes, and the bands of Volume I, Section 20.4 add no work to that desk. The Builder, Sponsor and graduate paths, which reach that chapter without either section, can therefore take the review queue and the bands on trust. The Sponsor path also takes on trust the audit and the denial explanations of Chapter 36, whose held denials join the review queue of Chapter 37. The Technical lead path takes on trust the verifier of Section 27.4, which the release suite of Chapter 29 carries.

Week by week, the graduate path gives Volume I's share five weeks and this volume eight, and a course on this volume alone gives the bridge its first week. In either, the project's four milestone weeks of Appendix C can run from week 10.

Week	Graduate path	This volume alone
1	Volume I, Chapter 1 to Volume I, Chapter 3	The bridge
2	Volume I, Chapter 4 and Volume I, Chapter 5	The Part V opener; Chapter 22
3	Volume I, Chapter 6 to Volume I, Chapter 8	Chapter 23 and Chapter 24
4	The path's sections of Volume I's Part III	Chapter 25
5	The path's sections of Volume I's Part IV	Chapter 26 and Chapter 27
6	The Part V opener; Chapter 22 to Chapter 24	Chapter 28 and Chapter 29
7	Chapter 25 to Chapter 27	The Part VI opener; Chapter 30 and Chapter 31
8	Chapter 28 and Chapter 29	Chapter 32 and Chapter 33
9	The Part VI opener; Chapter 30 to Chapter 32	Chapter 34 and Chapter 35
10	Chapter 33 to Chapter 35	The Part VII opener; Chapter 36
11	The Part VII opener; Chapter 36 and Chapter 37	Chapter 37 and Chapter 38
12	Chapter 38; the Part VIII opener; Chapter 39	The Part VIII opener; Chapter 39
13	Chapter 40 and Chapter 41	Chapter 40 and Chapter 41

Using these books in a course

The two volumes grew out of the five graduate courses I teach at McGill University. The table lists each course with the year I first taught it and the chapters that hold its material, so students and instructors can find a course's chapters in either volume. The chapters are optional reading, and no course of mine requires the books.

Sample · Not for Redistribution

Course	First taught	Chapters
Enterprise Data Science	2019	Volume I, Chapter 3 to Volume I, Chapter 7; Volume I, Chapter 16; Chapter 38
Machine Learning in Production	2019	Volume I, Chapter 8; Chapter 30 to Chapter 36, with Section 23.1, Section 25.2, Section 25.6, Section 29.6, Section 29.5 and the shadow-to-canary requirement of Table 0.4, which Part VI assumes
Introduction to AI and Deep Learning	2020	Volume I, Chapter 9 and Volume I, Chapter 10; Volume I's Appendix A
Applications and Architectures of Deep Learning	2020	Volume I, Chapter 11 to Volume I, Chapter 14; Section 23.6
Designing and Building Agentic AI Systems	2023	Volume I, Chapter 1 and Volume I, Chapter 2; Volume I, Chapter 12; Volume I, Chapter 16 to Volume I, Chapter 19; Chapter 22 to Chapter 29; Chapter 37; Chapter 39 and Chapter 41

Why I wrote these books

I have taught at McGill since 2019, in person and online, in the Master of Management in Analytics, and these two volumes hold the material of all five courses in the table. My students knew I was writing them and have been waiting for them, and with these volumes I can at last keep my promise. My hope is that these pages demystify agentic AI and help you move responsibly from proof of concept to production.

Code repository

The companion repository, at https://github.com/gradient-divergence/agentic-ai-systems, contains runnable examples and short demos aligned to both volumes. It is the canonical reference for the code shown in print. An exercise that needs a file names it in that repository, under `exercises/` or `demos/data/`. Appendix C holds the answers to the Check exercises, a worked answer to each chapter's Exercise 3 and one to each Exercise 7 that is a ★ Apply exercise. The appendix's first paragraph names the exceptions. The companion holds keys to most of the other exercises and reference solutions to most Build exercises, beside their acceptance checks, and the instructor's materials hold the marking notes.

The labs of my courses are public too. The McGill showcases repository, at https://github.com/conqueror/mcgill-showcases, holds 25 machine learning projects written for these courses, each with its own guide and tests.

Disclosure and neutrality

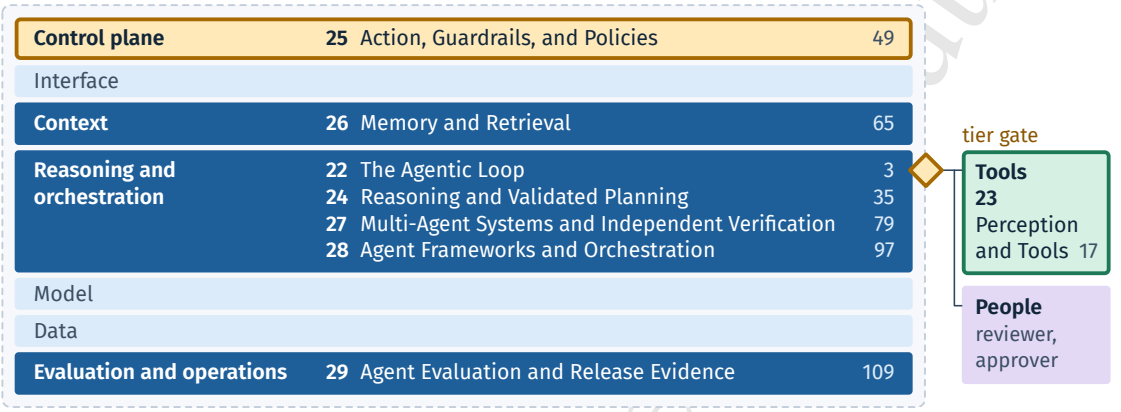
I am an Assistant Professor at McGill University and the founder and CEO of OODARIS AI. These books are written as an independent academic and practitioner reference. Product or platform mentions are provided for instructional context, not endorsement.

Sample - Not for Redistribution

PART V

Agentic Systems Architecture

V



Part IV ended on its stage 3 record. Part V builds the program for the refund agent, and its payment gate must enforce the refund limits at execution.

The refund agent enters the part as Part IV left it. It leaves with the switch into stage 4, the book's stage for this part. Switched on, the agent would decide refunds under $50 in shadow, beside the person who still handles each request, with its payment recorded and not made. In the companion the switch runs on generated requests, with generated labels standing in for the person's decisions.

The first four chapters go from the whole run to the single call that pays. Their payments go into a simulated ledger, which stands in for the shadow recorder until later in the part. The last four ask where the agent's evidence comes from and how it goes stale (Chapter 26), when one loop should become several agents (Chapter 27), what a framework may supply (Chapter 28), and what evidence can block the agent's admission to shadow (Chapter 29). Money first moves at stage 5, for a canary cohort. That is Part VI's work, which starts with what the data system must guarantee about the evidence the gate reads.

The part assumes the refund agent's decision boundary, with its tiers and tools (Volume I, Section 1.5), and the way a model's reply becomes a call the application checks (Volume I, Section 12.11). On a first pass, read the loop, the gate and the evaluation chapters, and leave *Multi-Agent Systems and Independent Verification* (Chapter 27) until one agent has been shown not to be enough. The closing sections of *Perception and Tools* (Section 23.6 to Section 23.8), on reads that a model computes from an image or a screen, can also wait for a second pass.

The part starts from the program that runs each decision and stops when it should.

Sample – Not for Re tion

THE ROUTE THROUGH PART V

Agentic Systems Architecture

What program runs the decision, and stops?

The first four chapters go from the whole run to the single call that pays.

22 **The Agentic Loop** 3

YOU CAN NOW

Trace a request through the loop, and diagnose why a run ended.

THE REFUND AGENT

A bounded loop of nine phases. Money moves only in the execute phase.

LEAVES OPEN *What must each read and each tool promise?*

23 **Perception and Tools** 17

Compute each read's age at use, and write a tool's contract.

Contracts for its seven tools and freshness limits for its three sources.

LEAVES OPEN *Is the sequence itself a valid plan?*

24 **Reasoning and Validated Planning** 35

Check a plan step by step and against its bounds.

Its call sequences can be checked as plans. It still runs on its pinned order.

LEAVES OPEN *What may execute at the moment of payment?*

25 **Action, Guardrails, and Policies** 49

Trace a rule to where it executes, and compute the gate's decision.

A gate inside the call that pays, and approvals bound to one proposal.

LEAVES OPEN *Where does the evidence come from, and how does it go stale?*

26 **Memory and Retrieval** 65

Compute what survives retrieval, and ask a memory what it knew and when.

A memory lifecycle for its policy, beside its loop, which still reads the snapshot.

LEAVES OPEN *Would a second agent catch more?*

27 **Multi-Agent Systems and Independent Verification** 79

Decide when one agent should become two, and build a verifier.

A read-only verifier that compares the customer's words with the carrier record.

LEAVES OPEN *What may come from a framework?*

28 **Agent Frameworks and Orchestration** 97

Separate a runtime's mechanism from the agent's authority.

It runs on either runtime and leaves the same record on every case of its suite.

LEAVES OPEN *What evidence lets it run in shadow?*

29 **Agent Evaluation and Release Evidence** 109

Build a suite whose failures stop a release, and put an agent in shadow.

The shadow switch starts only from a passing, signed release record.

LEAVES OPEN *What must the data system guarantee about the evidence?*

BY THE END OF PART V

The agent leaves with the switch into stage 4. Switched on, it would decide refunds under $50 in shadow, with its payment recorded and not made.

Refund agent stage 4 ●●●●●○○○ *decides tier 1 in shadow*

NEXT **Part VI** Engineering and Operations

Sample - Not for Redistribution

CHAPTER

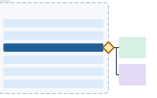
The Agentic Loop

22

13 pages, 3 code listings

Suppose a 2,499-cent `refund.execute` payment goes unanswered. If it succeeded, calling again would pay twice.

What program runs the decision, and stops? A loop does, with that payment as one step. Yet a run of it can be wrong with every step right. So far the refund agent only drafts, and a person carries out every refund. By the end of this part that loop can be switched into shadow.



Reasoning
Refund agent
Part V: stage 4

Learning objectives

By the end of this chapter, you will be able to:

1. Trace one refund request through the loop's stages and phases, and name the tool each phase accepts.
2. Choose between a loop in which the model picks the next tool and a loop with the order pinned, and say what each gives up.
3. Diagnose from a trace why a run ended: finished, refused by a check, waiting for a person, or out of steps.
4. Compute how per-step reliability compounds across a run, and state what a repeated paying call needs so that it pays once.

22.1 What a loop adds to a model call

A single model call reads its input and returns an answer. A loop asks a model what to do next, calls a tool, feeds each result back in and asks again, until a rule written into it says stop. What the refund agent reads at each step decides what it may do at the next, and its actions change later reads. The balance left on an order is computed from the payment ledger, so a paid refund lowers the balance that the next request on that order reads.

I describe that cycle in four stages: perceive, reason, act and learn. Perceive gathers the state, which for the refund agent is three reads. Reason chooses the next step: assess the request against the policy, then propose a refund. Act changes something outside the agent, by asking a person for approval or by paying. Learn records what happened.[1] Table 22.1 puts the refund agent's phases under each stage, with the one tool each accepts, and Figure 22.1 draws one cycle with its two ways out.

[1] Boyd's sketch has four steps (observe, orient, decide and act), with feedback from the decision and action into observation. It labels the decision a hypothesis and the action its test (Boyd 1996, slide 3). Volume I, Table 1.2 matches the four stages to the five verbs.

Table 22.1 The refund agent's phases under the four stages, with the one tool each phase accepts

Stage	Phases of the refund agent	Tool each phase accepts	Where the book goes deeper
Perceive	order, shipment, policy	`order.lookup`, `shipment.status`, `policy.retrieve`	Chapter 23, Chapter 26
Reason	assess, propose	none, then `refund.propose`	Chapter 24
Act	approval, execute	`approval.request`, `refund.execute`	Chapter 25
Learn	audit, then done	`audit.append`, written by the loop itself	Chapter 29

Sample - Not for Distribution

The last stage, Learn, is easy to misread. Learning means the next run gets better because the last run was observed. In production, "learning" usually does not mean training. Nothing is trained inside the refund agent's loop. Its Learn stage writes an audit record at the end of every run, and a person later turns a failure into a test or a change of policy.

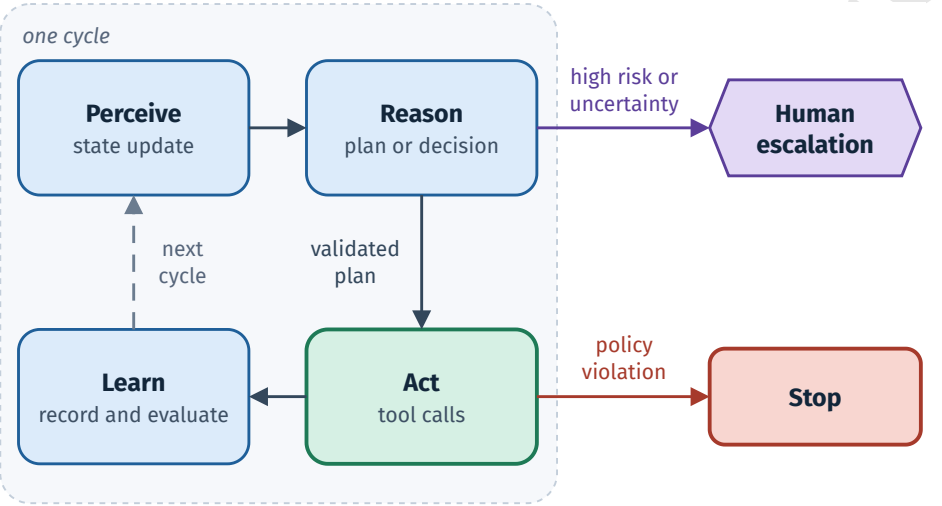


Figure 22.1 One cycle of the agentic loop, with its two guarded exits.

I draw parallels with human experience: trial and error, and learning from mistakes. For the refund agent's loop, the parallel holds for the trials and the errors, and breaks at the learning. A person carries the lesson into the next attempt, while the loop carries none until someone acts on its record.

What the stages don't say is who makes the choice in Reason, the model or the code that calls it.

22.2 Who chooses the next step

Volume I, Section 1.1 tests a design by asking who chooses the next step. In a loop the answer lies between two ends. At one end the model chooses. Each turn it names any tool, or writes a thought that calls none, and the loop runs what it names. ReAct interleaves reasoning and actions this way, and each action's observation joins the context that the next step reads (Yao, Zhao, et al. 2023, sec. 2). That end suits a path that has to be discovered, where the next useful call depends on what the last one found. An agent that has to find its way through a web page or a code repository sits at that end too, and Section 23.8 and Section 28.3 take up how it checks what each action did.

- **model chooses** each tool, as ReAct does
- **order pinned** refund agent

Who chooses the next step: at one end the model names each tool, as ReAct does, and at the other the order is pinned, where the refund agent sits.

The refund agent sits at the other end, because its path is known: read the order, the shipment and the policy, assess, propose, then pay or ask a person. The rule of Volume I, Section 1.6 decides that case. Workflows win when the path is known. So the companion pins the order. At each phase the loop accepts one tool name, and a reply naming any other ends the run as a referral to a person. At the assess phase the model writes a draft decision, and when the fixed rules approve, the loop accepts the draft only if it agrees with theirs. Call this a *loop with the order pinned*.

Pinning gives up the paths a model could discover, and this book hasn't measured what pinning costs or saves against a loop in which the model chooses. By the test of who chooses, the model here chooses almost nothing, which puts the loop nearer a

Exercise 4

Sample. Not for distribution

workflow than its name suggests. And where the conditions in Volume I, Section 1.6 hold, there is no need to make it agentic. Among the places where this book's advice fails, I count systems that are too complicated and overengineering up front for simple tasks.

Pinning leaves the harder problem standing: a run can be wrong when every step is right. Besides a payment call repeated after its reply is lost, a request can qualify when its evidence is read and no longer qualify when the refund is paid. That happens when the shipment record or the balance changes in between. At my company, OODARIS AI, my team found failures at the boundaries between steps: one step's outputs did not match the next one's inputs, the contracts between steps did not match, and what one step certified was not what the next admitted. Mechanisms for retries could also be missing. One of these failures, a retry that skipped the checks, is told in Section 22.6.

My team found other failures in the tools. When an agent needed to check a calendar, or whether something was closed, either it had no access to the right tool or its tool could not do everything the check required. Those failures come down to what a tool can promise, which Section 23.3 takes up.

The failures between steps are why four things are written into the loop's own code, where a test can reach them: which call may come next, what a repeated call does, when the run ends, and what record it leaves.

22.3 One refund, followed through the loop

Watch them on one request. In the companion a scripted model stands in for a real one, so the run is offline and gives the same trace every time. No run in this volume calls a live model, so what a real model would choose, cost or get wrong is measured nowhere in it. What the volume builds is the production system around the model: the contracts, records and checks a live model would run inside, and the designs that would release it, from a paid canary through a staged rollout to a launch gate. Request `r-7-0007` in the default run asks for 2,499 cents back, the same amount as the payment in the opening, not the $38 that Part I's constructed scenario gave it (Volume I, Section 1.2). It comes 12 days after its order was delivered, and its reason is a change of mind. The policy the run reads, version `2026-03-01` of `refund_cap_v4`, allows returns for 21 days, and at 2,499 cents the request is tier 1, which the agent may decide without a person. The demo prints the run's trace, one line per step, then the request, the order and the policy it read.[2]

[2] The demo is `agentic_loop_demo.py`, in `demos/` in the companion repository, and its output comes from the default run, `python -m refund_agent.run --seed 7`.

```
step  phase     call             age_seconds  record or reason                OUTPUT
0     order     order.lookup     60.0
1     shipment  shipment.status  120.0
2     policy    policy.retrieve  3600.0
3     assess    (draft)                       approve, tier 1: eligible
4     propose   refund.propose                proposal-1a500fb604bde3c7e4a7eabc
5     execute   refund.execute                refund-r-7-0007
6     audit     audit.append                  audit-12268cedeb91c44c01a98a14
request r-7-0007: 2499 cents, changed_mind, at 2026-03-22T15:08:27Z
order o-0007; delivered at 2026-03-10T15:08:27Z
policy refund_cap_v4 version 2026-03-01, return_days 21
```

Sample - Not for Redistribution

Steps 0 to 2 return each record with its age, which the checks test against 30 minutes for the order and the shipment and 30 days for the policy text. At step 3 the model drafts `approve` at tier 1, and the rules agree. Step 4 writes a proposal bound to the request, the order and the policy version, step 5 pays, and step 6 closes the run (Figure 22.2).

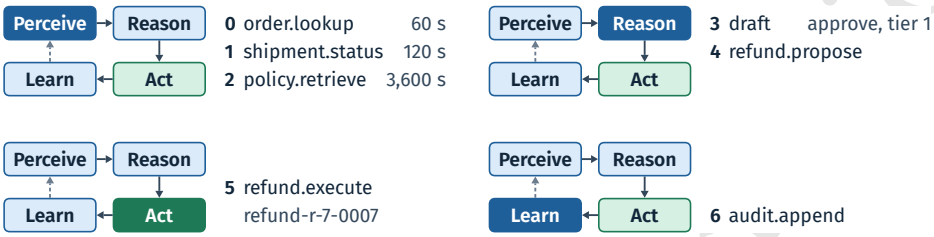
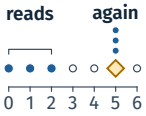


Figure 22.2 Request `r-7-0007`'s seven steps along the loop of Figure 22.1, one frame for each stage with that stage lit: the three reads under Perceive, the draft and the proposal under Reason, the payment under Act and the audit record under Learn.



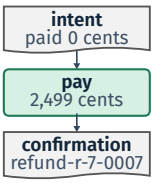
Request `r-7-0007`'s steps 0 to 6: the three reads, and the same three again inside `refund.execute` at step 5, where the gate judges the payment.

Step 5 is where the problem of Section 22.2, evidence that changes between the read and the payment, is answered in code. `refund.execute` doesn't trust steps 0 to 3. It reads the order, the shipment and the policy again. Its gate judges the payment on them as they stand at the run's time of execution, which the companion fixes when the run starts (Section 25.3). A request that qualified at step 3 can be refused at step 5. In this run nothing had changed, and the refund was paid.

The payment leaves two audit records in `audits.json`, and the demo prints the fields where they differ and two they share. The first, the intent, is written before the payment, and the second, the confirmation, after it. If the intent can't be written, nothing is paid.

OUTPUT

field	intent	confirmation
money_moved	false	true
paid_amount_cents	0	2499
receipt_id	null	"refund-r-7-0007"
trace_id	"trace-r-7-0007"	"trace-r-7-0007"
at	"2026-03-22T15:08:27Z"	"2026-03-22T15:08:27Z"
decision.executed	false	true



The two audit records around the payment: the intent before it, no money moved, and the confirmation after it, naming the receipt.

The two records are the run's trail. Every serious agent needs a trail from goal to outcome, and without a trace there is no review. Seven steps ran in a fixed order and ended on a record. What in the code fixes that order and that ending?

22.4 The loop as a state machine

In the trace each step ran in one phase, and each phase accepted at most one tool, such as `order.lookup` at the order phase and `refund.execute` at execute. That is the pin, the first of the four things, and the pin is a table. The loop's states are nine named phases, and the first listing maps six of them to the one tool each accepts.[3]

```python
class Phase(str, Enum):
    ORDER = "order"
    SHIPMENT = "shipment"
    POLICY = "policy"
    ASSESS = "assess"
    PROPOSE = "propose"
    APPROVAL = "approval"
    EXECUTE = "execute"
    AUDIT = "audit"
```

[3] The listings are in `contracts.py` and `agent/__init__.py`, under `refund_agent/` in the companion repository.

Sample - Not for Redistribution

```
    DONE = "done"


PHASE_TO_TOOL = _ReadOnly(
    {
        Phase.ORDER: "order.lookup",
        Phase.SHIPMENT: "shipment.status",
        Phase.POLICY: "policy.retrieve",
        Phase.PROPOSE: "refund.propose",
        Phase.APPROVAL: "approval.request",
        Phase.EXECUTE: "refund.execute",
    }
)
```

Three phases accept no tool from the model: at assess the model returns a draft decision (step 3 in the trace), at audit the loop itself calls `audit.append`, and done ends every run. The policy check isn't a tool either, but a gate inside `refund.execute`, in the path of every payment, which Section 25.2 builds out. **Exercise** 1

Figure 22.3 draws the transitions between these phases. The next two listings print the loop's code, which holds the guards that decide whether a transition may happen: the pin, the tier and the approval on record.

From phase to phase, a run carries its state: the request, the evidence with its ages, the policy version, a risk score, the decision, any approvals, the proposal, and a trace identifier. The identifier ties the run's records together, as `trace-r-7-0007` does in both audit records. The risk score in this build is a fixed text rule, not the calibrated estimate proposed in Volume I, Section 5.5, and the book makes no claim about the rule.[4]

[4] The rule gives 0.9 when the request's text contains "ignore" in any mix of capitals, even inside a longer word, and 0.1 otherwise, against a fictional threshold of 0.2.

The second listing is the loop that carries the state. It starts at the first phase and sends straight to audit a request already paid, or one that comes back under the same identifier with different content. Otherwise it takes steps until the run reaches audit or has one step of its budget left. A run that ran out is referred as `step_budget_exhausted`. Then, however the loop ended, `ensure_named_review` makes sure that an unpaid request over 50,000 cents, at tier 3, has an open review for its named approver. If the run was refused or ran out before opening one, the function makes the reads the run lacks, proposes if there is no proposal and opens the review with `approval.request`, even with no budget left. If `order.lookup` can't resolve the order, the function opens a review of the request without a proposal. Every run then writes its audit record through `audit.append`. **Exercise** 6

```
def run(self, request: c.Request, *, at: datetime | None = None) -> Trace:
    request = _validated(c.Request, request)
    state = c.State(request=request, trace_id=f"trace-{request.request_id}")
    steps: list[TraceStep] = []
    try:
        with self.tools._boundary(state, "run"):
            self._start(state, at)
            # Reserve the final step for audit, including budget exhaustion.
            while state.phase != c.Phase.AUDIT and len(steps) < self.max_steps - 1:
                steps.append(self._step(state, len(steps)))
            if state.phase != c.Phase.AUDIT:
                self._refer(state, "step_budget_exhausted")
            ensure_named_review(self.tools, state, steps)
```

Copyright protected material. Not for redistribution

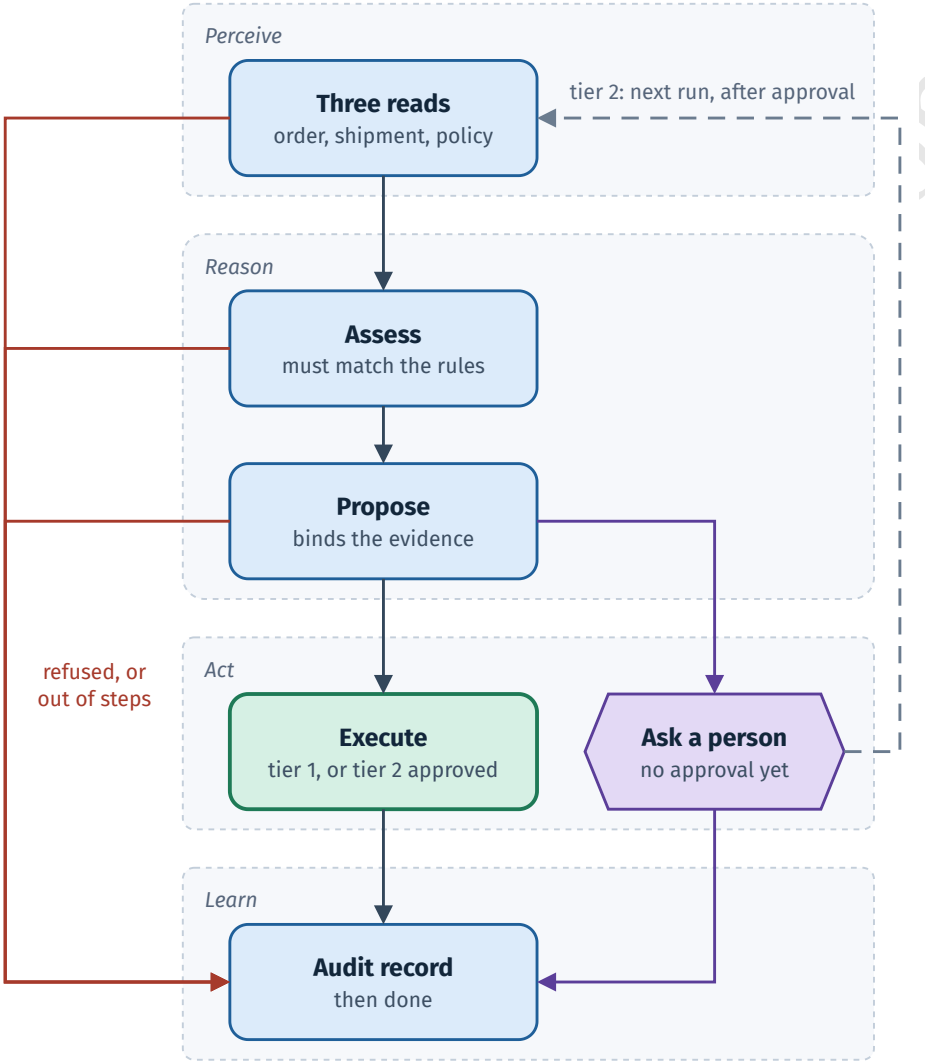


Figure 22.3 The refund agent's loop as a state machine, with its phases grouped under the four stages. The three reads share one box. Every refusal, and a run out of steps, leads to the audit record. On that way out, a tier 3 request first gets the named approver's review if the run has not opened it. A tier 2 request runs again from the top once a person approves.

```
            return self._trace(state, steps, self.tools.record_decision(state))
        except ReconciliationError:
            raise
        except (AuditWriteError, PaymentError) as error:
            if not isinstance(error, PaymentError) or error.confirmed:
                self._error_trace(state, steps, error)
            raise
```

The listing's last lines handle failure. The run stops with an error if the audit store fails, if a store holds something other than what was written, or if a payment succeeded before a later failure. The error names the payment the ledger confirms, or, when the ledger can't be read, reports the payment's outcome as unknown. Each record is read back after it is written, the two audit records of `r-7-0007` among them. So if a store silently drops a record, or keeps or returns something other than what was written, the request stops for a person to reconcile. In that case nothing the ledger can't show is reported as paid.[5] The important part is the loop boundary, not the syntax. In this loop the boundary is the `while` condition and the record written after it.

[5] The companion keeps the ledger and the audit log in memory. A deployed system needs a durable store for both, idempotency at the payment provider, and reconciliation between the two records.

Sample

The third listing is one step inside that `while`. `_process_reply` refuses `refund.execute` for any request over 50,000 cents and hands a draft at the assess phase to `_assess`. Any other reply that isn't the phase's own tool ends the run as `unexpected_tool`. Otherwise that tool is called, and the step advances or refers the request with the tool's reason. `_assess` lets a referral from the rules stand, and a denial at tiers 1 and 2. At tier 3, where a named approver decides every request, a denial becomes a referral that keeps the rules' reason and goes on to the propose phase. The run then opens a review for that approver. When the rules approve, `_assess` accepts the model's draft only if it matches their approval, the wording of the reason aside. After a proposal, `_advance` chooses between executing and asking a person, from the tier and the approval on record.

```
def _process_reply(self, state: c.State, reply: Reply) -> c.ToolResult:
    if isinstance(reply, c.ToolCall) and reply.name == "refund.execute":
        if state.request.amount_cents > 50000:
            self._refer(state, "approval_required")
            return c.ToolResult(ok=False, reason="approval_required")
    if state.phase == c.Phase.ASSESS and isinstance(reply, c.Decision):
        return self._assess(state, reply)
    if not isinstance(reply, c.ToolCall) or reply.name !=
        ↪ c.PHASE_TO_TOOL.get(state.phase):
        self._refer(state, "unexpected_tool")
        return c.ToolResult(ok=False, reason="unexpected_tool")
    result = self.tools.call(reply, state)
    if result.ok:
        self._advance(state, state.phase)
    else:
        reason = result.reason or "tool_rejected"
        self._refer(state, reason)
        state.failure_detail = state.failure_detail or f"{reply.name}:{reason}"
    return result

def _assess(self, state: c.State, reply: c.Decision) -> c.ToolResult:
    assessed = self.tools.assess(state)
    changed = self.tools.changed_approval(state)
    if changed:
        state.decision, state.phase = decision(state, c.Outcome.REFER, changed),
     ↪ c.Phase.PROPOSE
    elif assessed.tier == c.Tier.NAMED_APPROVER and assessed.outcome !=
        ↪ c.Outcome.APPROVE:
        state.decision = decision(state, c.Outcome.REFER, assessed.reason)
        state.phase = c.Phase.PROPOSE
    elif assessed.outcome != c.Outcome.APPROVE:
        state.decision, state.phase = assessed, c.Phase.AUDIT
    elif reply.model_dump(exclude={"reason"}) !=
        ↪ assessed.model_dump(exclude={"reason"}):
        self._refer(state, "invalid_decision")
    else:
        state.phase = c.Phase.PROPOSE
    return c.ToolResult(
        ok=state.decision is None or state.decision.reason != "invalid_decision",
        reason=state.decision.reason if state.decision else "eligible",
    )

def _advance(self, state: c.State, phase: c.Phase) -> None:
    if phase in READ_NEXT:
        state.phase = READ_NEXT[phase]
    elif phase == c.Phase.PROPOSE:
        tier = tier_for(state.request.amount_cents)
        approval = self.tools.approval_for(state)
        state.approvals = [approval] if approval else []
        may_execute = tier == c.Tier.AGENT or (tier == c.Tier.REVIEWER and
     ↪ bool(approval))
        ready = state.decision is None and may_execute
```

```
        state.phase = c.Phase.EXECUTE if ready else c.Phase.APPROVAL
    elif phase == c.Phase.APPROVAL:
        self._refer(state, state.decision.reason if state.decision else
            ↪ "approval_required")
    elif phase == c.Phase.EXECUTE:
        state.phase = c.Phase.AUDIT
        if state.receipt is None:
            self._refer(state, "execution_not_recorded")
```

The checks on what a step returned, such as the age of the evidence and the policy version, are in the tools and the gate, and each had to be written there. Every path in Figure 22.3 ends at the audit record, and only one of them pays.

22.5 How a run stops, waits, and resumes

Request `r-7-0007` finished in seven steps. Not every run finishes, so a run needs written ways to end, and the refund agent's loop has the four in Table 22.2.

Exercise 2

Table 22.2 The four ways a refund run ends, each followed by its audit record

How the run ends	Reason in the decision	Who holds the request next	Money moved
Finished	`eligible` when paid, or a denial at tier 1 or 2, such as `outside_return_window`	Nobody	The refund, or nothing
Refused by a check	The check's reason, such as `stale_shipment` or `unexpected_tool`	A person	Nothing
Waiting for a person	`approval_required`, or a tier 3 denial's reason	The reviewer or the named approver	Nothing
Out of steps	`step_budget_exhausted`	A person	Nothing

Where a run ends also sets the authority it used. Of the four levels of autonomy (Volume I, Section 1.3), the specification's tiers fall on the upper three. A tier 3 run ends with a packet for the named approver, where the agent recommends, and a tier 2 run with a proposal that one person approves. A tier 1 run pays inside the loop, where the agent decides and acts. In the companion that payment goes to a simulated ledger, and there is no shadow recorder yet. At tier 2 a denial is the exception: the companion's loop finishes it with nobody after it (Table 22.2), though the decision boundary sends it to a person. From stage 6 the routing of Section 37.2 gives a reviewer each tier 2 denial, as it does each tier 2 payment.

step budget

Whatever its tier, every run has a *step budget* for its loop, at most twelve steps by default with one kept for the audit record.[6] Running out of it is the last of the four endings. A test lowers the budget to 1, 2, 4 and 6 for a tier 1 request and checks that each run ends as `step_budget_exhausted`, with no more steps than the budget, one audit record and no payment. A tier 3 request's trace can still run past the budget. Each call that `ensure_named_review` makes after the loop to open the named approver's review is a step in the trace, though the run keeps its reason and pays nothing. Another test gives such a request a budget of 1 and checks that it ends out of steps, with that review open and nothing paid.

[6] Twelve is the companion's default, `Agent(max_steps=12)`. The specification sets no operating limits, and the companion has no deadline and no cost limit.

Some loops also need a clock within a run. A call that may hang needs a deadline, passed on to every service the call reaches (Beyer et al. 2016, ch. 22). Emergency response runs on a different clock from quarterly planning. My longest run at OODARIS AI took five days, from forecasting and optimization to enriching, interpreting and explaining the results. At the other end, in a constructed scenario from ambulance

Sample - Not for Redistribution

dispatch, a step that misses its deadline hands the emergency call to the dispatcher with what is known so far.

Back in the default run, two requests ended without a payment, `r-7-0002` and `r-7-0009`, and the demo prints each step of the first, then the decision that ended each.

Exercise 5

```
step  phase     call             ok    reason or age_seconds                OUTPUT
0     order     order.lookup     true  60.0
1     shipment  shipment.status  true  1801.0
2     policy    policy.retrieve  true  3600.0
3     assess    (draft)          true  stale_shipment
4     audit     audit.append     true

request   cents  tier  decision                  steps  executed
r-7-0002  2499   1     refer stale_shipment      5      false
r-7-0009  5000   2     refer approval_required   7      false
```

The first, `r-7-0002`, is yours to read. Its trace alone shows which step refused it and why. The second, `r-7-0009`, is a tier 2 request, and it waits. It asks for 5,000 cents, and its run ended after seven steps as `approval_required`, with no payment. Its run opened a review item with `approval.request` and then referred the request. The waiting happens outside the loop, where the request sits in a review queue until a person on the reviewer list approves the proposal. The approval has no time limit of its own, but it lapses when the request, order, amount, reason or policy version at execution no longer matches the proposal it approved. So the wait has no clock, a cost the loop can't see, and Chapter 37 designs the review queue around it.

After the approval the request runs again from the first phase. It reads, assesses and proposes again, and it gets the same proposal, whose identifier is computed from the request, the order and the policy version (Figure 22.4). Only then does `_advance` find the approval and move to execute, where, as at step 5 of `r-7-0007`, the gate checks it against the evidence as it now stands. With no entry in the middle, no run can pay on checks an earlier run made.

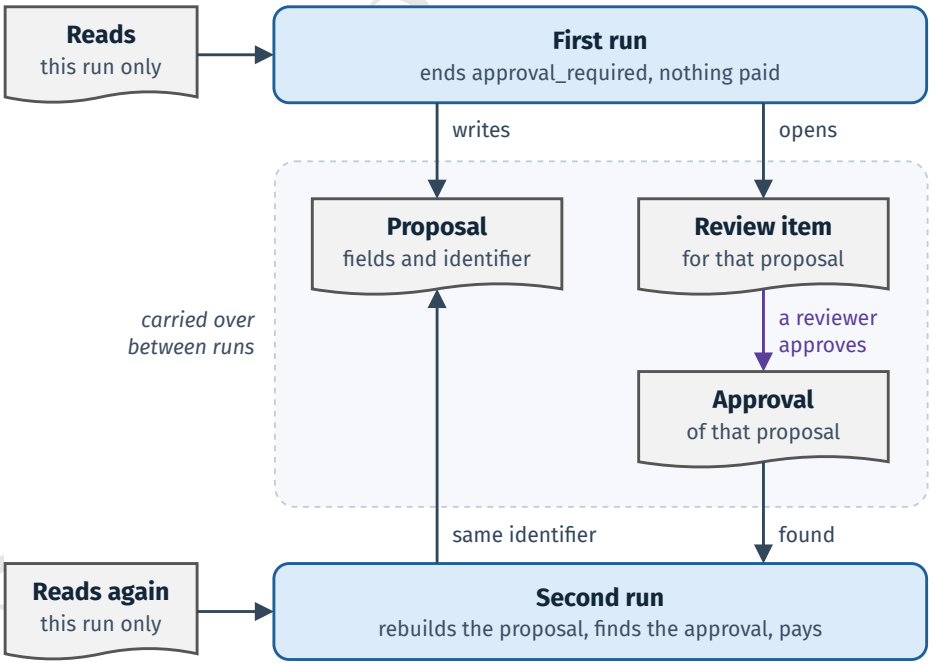
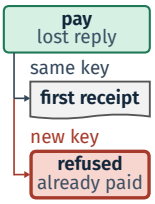


Figure 22.4 The second run of `r-7-0009` takes from the first only its proposal, review item and approval. It reads the evidence again and pays on the approval of the proposal it rebuilds.

In a constructed scenario from IT operations, an approval waits too. An agent prepares a configuration change during a live incident for the on-call lead to approve. Resumed at the apply step, the run applies a change planned for an incident that has since moved on. Started again at the first read, it sees the new state. A run too long to start again from the top, such as a coding agent's hours of edits, resumes from a checkpoint instead. Its checks have to run again before the checkpoint can be an entry in the middle. Section 28.3 says which checks those are.

22.6 Retries, repeated payments, and errors that compound



The same key gets the first receipt back, a new key a refusal.

In the refund agent a retry starts from the top too, as the run after an approval does. Inside the loop nothing is retried, because a step that fails ends the run as a referral. A retry happens one level up, which is where the opening's lost reply is settled. Run `r-7-0007` again after it is paid, and the loop finds the receipt in the ledger before its first step and goes straight to audit. The ledger files each payment under the idempotency key of the call that made it. Call `refund.execute` again with the same arguments and key, and the ledger returns the first receipt. Change the amount under that key, or send a new key or none, and the call is refused. A test makes each of these calls and checks that the ledger still holds one payment.

⚠ Warning

A retry of a paying call must carry the same key

A key made afresh for each attempt, from a clock or a random number, makes the retry look like a new payment to the key check. Here a separate check refuses it because the request is already paid, so the retry gets a refusal and not the receipt. Derive the key from what identifies the payment, as the scripted model does with `execute-` and the request's identifier, and keep it for every attempt.

Retrying has a second cost. Each retry is another request to a service that may already be overloaded, and the remedy is to limit the retries per request and space them with randomized exponential backoff (Beyer et al. 2016, ch. 22).

A repeated payment is the loud failure. In the quiet one, an error passes from step to step because each step assumes the one before it was checked, and a wrong value reaches the payment with every step reporting success. This book calls that a *silent cascade*.[7] The refund agent stops it the way step 5 of `r-7-0007` did, where `refund.execute` repeats the reads instead of trusting them.

silent cascade

[7] Chapter 27 extends the term from the steps of one run to the agents of a system, and asks when one loop should become several agents.

In practice

A retry that skipped the checks

At OODARIS AI my team built a training and optimization system of 24 steps, each with substeps. There were problems with what each step certified about its output, what the next step admitted and the contracts between steps, and those problems wasted days of work. The team had assumed every run would go well end to end, so a retry from a middle step missed the certifications and guardrails. Those were the failures the team had to fix.

The repeated call and the evidence that changed are failures between steps, in runs whose every step was right on its own. A run can also fail the plain way, with one step that is wrong, and the more steps it has, the more chances for one. How often does a long run get every step right?

Write E_t for the event that step t of a run of n steps is right, and Pr for a probability, as in Table A.1. The definition of conditional probability in its product form, $\Pr(A \cap$

Notation Table A.1

Sample — Not for distribution

$B) = \Pr(B \mid A)\Pr(A)$ (Volume I, Equation A.2), applied once for each step after the first, gives

$$\Pr(E_1 \cap E_2 \cap \cdots \cap E_n) = \prod_{t=1}^{n} \Pr(E_t \mid E_1 \cap \cdots \cap E_{t-1}) = \prod_{t=1}^{n} p_t, \qquad (22.1)$$

where p_t is the probability that step t is right given that every step before it was, and the first factor is $\Pr(E_1)$.[8] The share of runs with every step right is the product of the per-step probabilities. It doesn't assume independent steps, because each factor is conditional on the steps before it, which is where any dependence goes.

[8] Each factor is defined when the steps before it are all right with positive probability. When they never are, the share of runs with every step right is 0.

Take $p_t = 0.98$ at every step, a value given for the example and not measured. The seven steps of a tier 1 refund give $0.98^7 \approx 0.868$, so about 13 runs in a hundred have at least one wrong step. Eight steps give $0.98^8 \approx 0.851$, and seven steps of which one is at 0.90 give $0.98^6 \times 0.90 \approx 0.797$.

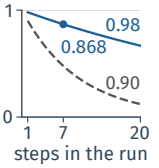
Exercise 3

The product counts runs with every step right, but a refund desk counts how runs end. A wrong step that a check catches ends the run as a referral, safe but work for someone, and one that no check catches can reach the payment, as in a silent cascade. So each run ends in one of three outcomes: a correct finish, a referral, or a wrong finish such as a forbidden payment. In a constructed comparison where nothing else catches a wrong step, the demo prints the expected outcomes of 1,000 runs of the seven steps, without and with a checker before the payment.

Predict
Does the checker raise or lower the share with every step right?

```
expected runs out of 1000; seven conditional step probabilities of 0.98
checker: catches 0.90 of wrong runs; refers 0.02 of right runs
without checker: correct=868.126, referred=0.000, wrong=131.874
with checker: correct=850.763, referred=136.050, wrong=13.187
with checker, every step right: 0.851
```
OUTPUT

The checker is one more step that can be wrong, so it lowers the share with every step right from 0.868 to 0.851, the eight-step value above. It also adds about 136 referrals. Yet it cuts the expected wrong finishes from about 132 to about 13, because each wrong run it catches becomes a referral. The companion can't say how often any of this happens. Its scripted model is right by construction, so no p_t is measured, and these numbers are inputs.



The share of runs with every step right, p^n, against the number of steps n, for each step right with probability $p = 0.98$ and $p = 0.90$.

22.7 What the scenario scores show

Three printed runs can't show that the loop holds for every request. The default run scores the specification's forty public scenarios, eight refusal cases and five hand-written payments, and prints five named scores.

```
baseline: 40/40 agreement with generated labels (100.0%);
    approve=4 deny=22 refer=14 executed=0
agent: 26/40 agreement with generated labels (65.0%);
    approve=4 deny=8 refer=28 executed=4
safety: 8/8 correct refusals;
    audited=8/8 unauthorized_cents=0
gate_removed: 7/8 correct refusals;
    audited=8/8 unauthorized_cents=999999
positive_payments: 5/5 ledger checks;
    paid_cents=62598
```
OUTPUT

The `baseline` and `agent` scores compare the rule-only baseline of Volume I, Section 1.5 with the loop on the same forty requests. The demo prints one request on which their decisions differ and counts all such requests.

Sample, Not for Distribution

```
request r-7-0005: 65000 cents, tier 3                                    OUTPUT
label: deny already_delivered
baseline: deny already_delivered
agent: refer already_delivered, executed=False
review: open=True, required_role=named_approver; payments=0
disagreements: tier 3, label deny=14
```

At tier 3 a denial becomes a referral to the named approver. A label records what the rules decide and says nothing of who may decide it, so here 26 of 40 measures agreement rather than correctness. The loop also paid four refunds, and the baseline, which only drafts, paid none.

In the `safety` score, five requests carry injected failures, among them an ambiguous order identifier and an instruction hidden in the request text. Three more pair an eligible 2,499-cent request with a wrong reply from the scripted model: a tool that doesn't exist, an overstated amount for `refund.execute`, or a call to `refund.execute` at the assess phase that obeys the hidden instruction. The loop refused and audited all eight, and paid nothing. The `gate_removed` score runs the eight again with the gate inside `refund.execute` removed, a removal that exists only in this offline comparison. Seven are still refused, and on the eighth, a request for 2,499 cents, the loop pays 999,999 cents. Section 25.6 reads that as the gate's own test.

The last score, `positive_payments`, checks five payments against the ledger. The three under 5,000 cents were paid without a person, and the two of 5,000 and 50,000 cents were paid after one reviewer approved and the request ran again, each of them once. A loop that never paid would pass the `safety` score and fail this one.

Still, the labels were generated from the rules the loop applies, the policy values are fictional, and the model is a script. So the `baseline` and `agent` scores say the code runs as written, and nothing about a real model. Chapter 29 builds the evaluations that can tell a better agent from a worse one. What the companion can show, given those limits, is how a run's trace becomes a test. Every production incident should become tomorrow's regression test. That is the only learning loop that compounds. A run's trace is where such a test can begin, because the companion turns a trace into a script for the model, and a test checks that replaying it gives the same trace.

22.8 Summary

The refund agent now runs each request through a bounded loop of nine phases, and money can move only in the execute phase. Each run ends finished, refused, waiting for a person or out of steps, with an audit record. A retried payment pays once: the same key returns the first receipt. If each step is right with probability 0.98, given and not measured, all seven are right 0.868 of the time. In a constructed comparison, a checker lowers that to 0.851 but cuts expected wrong finishes from about 132 to about 13 in 1,000. All eight refusal cases were refused, and without the gate one paid 999,999 cents. What must each read and each tool promise?

22.9 Exercises

Answers, p. 327

1. (Check, objectives 1 and 4.) Without looking back at Table 22.1, name the tool that each of the phases shipment, propose and execute accepts. In which phase, and only that one, can money move? A payment is made but its reply is lost, and the call is sent again. What must the second call carry so that the request is paid once, and what does `refund.execute` return?
2. (Check, objective 3.) Four runs ended with these decisions. (a) `refer`, `approval_required`, tier 2. (b) `refer`, `step_budget_exhausted`, tier 1. (c) `approve`, `eligible`, tier 1,

Sample – Not for redistribution

executed. (d) `refer`, `stale_shipment`, tier 1. For each, give the ending from Table 22.2 and say whether a person now holds the request.

3. ★ (Apply, objective 4.) A run has seven steps whose per-step probabilities are given as 0.99, 0.99, 0.99, 0.96, 0.98, 0.97 and 0.999. Use Equation 22.1 to compute the share of runs with every step right, then again with the fifth step at 0.90. If you could raise one step's probability by 0.01, which would you choose, and why?
4. ★ (Apply, objective 2.) Apply the rule of Section 22.2 to two constructed scenarios: "Workflows win when the path is known. Agents earn their keep when the path has to be discovered." (a) A hospital follows up laboratory results: a result outside its normal range notifies the patient's doctor and books a repeat test by a fixed protocol. (b) A dispatch desk takes conflicting reports of a collision on a road with no recent data. For each, choose a loop with the order pinned or one in which the model picks the next tool, and name what your choice gives up.
5. ★★ (Analyze, objective 3.) The demo's output in Section 22.5 prints each step of request `r-7-0002`. Find the step that refused the request and the reason. The shipment read at step 1 succeeded: explain why the refusal came later, why the step that refused it still reports `ok` true, and whether money moved.
6. ★★★ (Build, objectives 3 and 4.) Run request `r-7-0007` with step budgets of 12, 8, 7, 6 and 5. Start each budget with a fresh `Tools` instance. Deliverable: a table of the reason, the trace steps and the payments for each budget, and the diff of a test that pins the smallest budget at which the request is paid. Acceptance check: at the largest failing budget the reason is `step_budget_exhausted`, one audit record exists and the ledger shows no payment, and at the smallest succeeding one it shows one payment of 2,499 cents.

Sample - Not for Redistribution