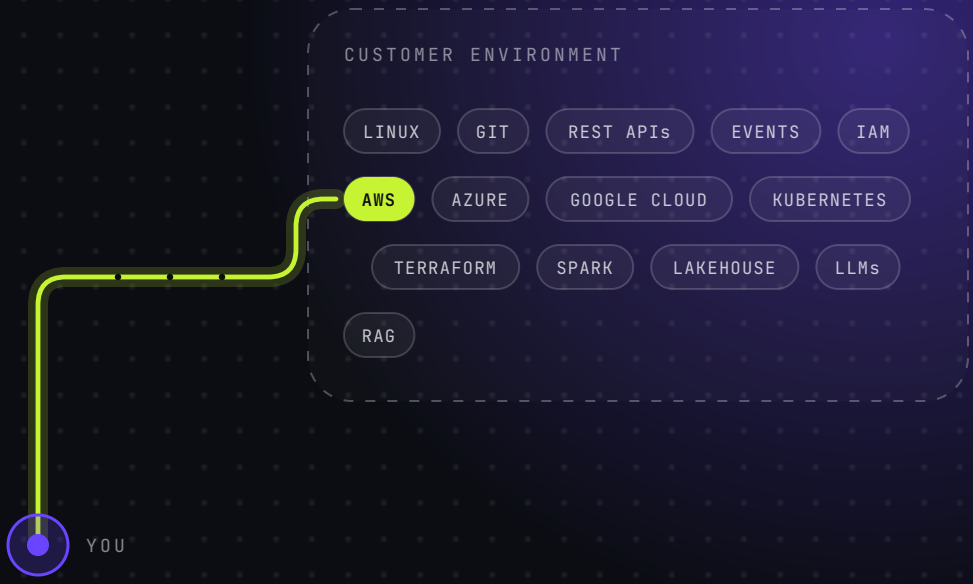
NB NEERAJ BHATYAL

VOL. 01 / 02



FROM SOFTWARE ENGINEERING TO AGENTIC AI SYSTEMS

The Forward Deployed Engineer

Volume 1 — Foundations: Software, Cloud, Data, and AI

I The Role and Its Roots
II Engineering Fundamentals
III Cloud
IV Data and AI

Neeraj Bhatyal

The Forward Deployed Engineer

VOLUME 1

Free sample · full book at neerajbhatyal.com

The Forward Deployed Engineer
Volume 1 — Foundations: Software, Cloud, Data, and AI

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, 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 I — THE ROLE AND ITS ROOTS

1 The Forward Deployed Engineer 3

2 A Short History: From Mainframes to Agents 19

PART II — ENGINEERING FUNDAMENTALS

3 Architecture and Design That Can Change 37

4 Modern Development: Git, Testing, and AI Coding Assistants 55

5 APIs, Integration, and Event-Driven Systems 71

6 Distributed Systems, Databases, and Resilience 87

7 Security and Enterprise Identity 103

8 Field Basics: Linux, Shell, Networking, and Debugging 119

PART III — CLOUD

9 Cloud Fundamentals: What Every Cloud Shares 135

10 AWS Essentials 149

11 Microsoft Azure Essentials 177

12 Google Cloud Essentials 197

13 Cloud Networking, Hybrid, and On-Premises Connectivity 215

14 Containers and Kubernetes 229

15 Infrastructure as Code, CI/CD, and Platform Engineering 245

16 Observability, SRE, and Performance 261

17 Cloud Cost and FinOps 275

PART IV — DATA AND AI

18 Data Platforms for AI: Lakehouse, Spark, Databricks, and Snowflake 291

19 AI and LLM Foundations 309

20 Working With Models: Prompting, Fine-Tuning, and Serving 323

21 RAG and Enterprise Knowledge 337

22 Document AI and Multimodal Systems 351

Appendix A AWS, Azure, and Google Cloud Service Comparison 363

Appendix B Command-Line Quick Reference 368

Appendix C Templates for Field Engineering 371
Appendix D Further Reading 374
Glossary 377
About the Author 386
Index 387

Contents

PART I — THE ROLE AND ITS ROOTS

1 The Forward Deployed Engineer 3

"Can You Just Build Us an AI Agent?" 4

1.1 What a Forward Deployed Engineer Does 4

1.2 Where the Role Came From 5

1.3 Why AI Made the Role Essential 7

1.4 The FDE and Neighboring Roles 8

1.5 The FDE Delivery Loop 10

1.6 Three Kinds of Ownership 11

1.7 The Skills of an Effective FDE 12

1.8 What a Week Looks Like 13

1.9 Meet Acme and Priya 14

1.10 How This Book Is Organized 16

Hands-On Exercise: Your FDE Profile and Your First One-Pager 17

Checklist: Starting Any FDE Engagement 17

Chapter Summary 18

2 A Short History: From Mainframes to Agents 19

Archaeology on Day Three 20

2.1 Computing: From Calculation to Intelligence 20

2.2 Programming Languages: Why Every Enterprise Runs Several 22

2.3 Software Engineering: From Crisis to Discipline 24

2.4 Software Architecture: From Mainframes to Agents 25

2.5 The Internet and APIs: From ARPANET to Tool Calling 27

2.6 Infrastructure: From Data Centers to the Cloud and AI Compute 29

2.7 The Rise of AI 30

2.8 The Stack Is Cumulative 31

2.9 A Framework for Judging New Technology 32

Hands-On Exercise: Draw a Customer's Technology Strata 33

Checklist: Reading the Layers in a Customer Environment 33

Chapter Summary 34

PART II — ENGINEERING FUNDAMENTALS

3 Architecture and Design That Can Change 37

"It Should Be a Small Change" 38

3.1 Architecture, Design, and Tradeoffs 38

3.2 The Real Enemy: Uncontrolled Dependencies 39

3.3 Design Principles That Matter in Practice 41

3.4 Ports and Adapters: Keeping Business Logic Independent 42

3.5 Domain-Driven Design and Bounded Contexts 43
3.6 Monoliths, Modular Monoliths, and Microservices 44
3.7 Separating AI Judgment From Deterministic Control 46
3.8 Code That Others Can Read and Change 47
3.9 Changing Code Safely: Refactoring and Technical Debt 49
3.10 Design in the Age of AI-Generated Code 50
3.11 Recording Decisions: Architecture Decision Records 50
Hands-On Exercise: Untangle a Business Rule 51
Checklist: Design Review for Anything You Add to a Customer System 52
Chapter Summary 52

4 Modern Development: Git, Testing, and AI Coding Assistants 55

"Can You Just Push It?" 56
4.1 From Requirements to Testable Behavior 56
4.2 Version Control With Git 57
4.3 Code Review 59
4.4 A Practical Testing Strategy 59
4.5 Testing AI Components 61
4.6 Continuous Integration 62
4.7 Feature Flags and Configuration 63
4.8 Reproducible Development Environments 64
4.9 AI Coding Assistants 64
4.10 Using AI Assistants on Customer Engagements 66
4.11 Documentation as an Engineering Artifact 67
Hands-On Exercise: A Small Service With CI and an AI Assistant 68
Checklist: Development Practices on an Engagement 68
Chapter Summary 69

5 APIs, Integration, and Event-Driven Systems 71

"Just Integrate With the CRM" 72
5.1 Four Ways to Integrate 72
5.2 HTTP Essentials 74
5.3 Designing REST APIs 74
5.4 Contracts and Change 75
5.5 Beyond REST: GraphQL, gRPC, and Real-Time Communication 77
5.6 Calling APIs You Do Not Control 77
5.7 API Gateways and API Security 79
5.8 Asynchronous Messaging 79
5.9 Event-Driven Architecture 81
5.10 Integrating With Legacy Systems 82
5.11 APIs for AI Models and Agents 83
5.12 Integration Services Across the Clouds 83
Hands-On Exercise: A Resilient Integration 84
Checklist: Before Shipping Any Integration 85
Chapter Summary 85

6 Distributed Systems, Databases, and Resilience 87
Forty Seconds Behind 88
6.1 Why Distributed Systems Are Different 88
6.2 Consistency and Replication 90
6.3 Choosing a Database 91
6.4 Indexes, Query Plans, and Common Performance Problems 92
6.5 Scaling Data: Replication and Partitioning 93
6.6 Transactions Across Services: Sagas 94
6.7 Caching 95
6.8 Resilience Patterns 96
6.9 Data Engineering Foundations 98
6.10 Backups, Recovery, and Disaster Planning 98
Hands-On Exercise: Breaking and Protecting a Data Path 99
Checklist: Data and Resilience Review 100
Chapter Summary 101

7 Security and Enterprise Identity 103
"Where Does the Data Go?" 104
7.1 Security as Engineering 104
7.2 Identity: Authentication and Authorization 106
7.3 Enterprise Single Sign-On: SAML, OIDC, and SCIM 107
7.4 OAuth 2.0 and Access Tokens 108
7.5 Authorization Models and Least Privilege 109
7.6 Workload Identity: Credentials Without Secrets 110
7.7 Secrets Management 110
7.8 Data Protection 111
7.9 Secure Coding and the Software Supply Chain 112
7.10 A First Look at AI-Specific Threats 113
7.11 Logging, Audit, and Incident Response 113
7.12 Passing an Enterprise Security Review 114
Hands-On Exercise: Identity End to End 115
Checklist: Security Before Go-Live 116
Chapter Summary 116

8 Field Basics: Linux, Shell, Networking, and Debugging 119
"Connection Timed Out" 120
8.1 Linux Essentials 120
8.2 The Shell as a Power Tool 122
8.3 Networking Fundamentals 123
8.4 Following a Request Through Every Layer 125
8.5 A Systematic Debugging Method 126
8.6 The Failures You Will Meet Most Often 127
8.7 Reading Logs and Errors 128
8.8 Working in Locked-Down Environments 128
8.9 Debugging AI Systems 129
Hands-On Exercise: A Debugging Lab 130

Checklist: First Day in a New Environment 130
Chapter Summary 131

PART III — CLOUD

9 Cloud Fundamentals: What Every Cloud Shares 135
Three Clouds, One Customer 136
9.1 What Makes Something "Cloud" 136
9.2 Service Models and the Shared Responsibility Model 137
9.3 Deployment Models: Public, Private, Hybrid, and Multi-Cloud 138
9.4 Regions, Zones, and Where to Run 139
9.5 How Every Cloud Organizes Resources 139
9.6 Everything Is an API: Control Plane and Data Plane 140
9.7 The Core Building Blocks 141
9.8 Managed Services, Lock-In, and Portability 142
9.9 Well-Architected Frameworks 143
9.10 How Cloud Pricing Works 143
9.11 Landing Zones and Cloud Governance 144
9.12 Working Across Clouds as an FDE 145
Hands-On Exercise: One Task, Three Clouds 146
Checklist: Starting in Any Cloud 146
Chapter Summary 147

10 AWS Essentials 149
Day One at Acme 150
10.1 Why Start With AWS 151
10.2 The Global Footprint: Regions, Availability Zones, and the Edge 151
10.3 How Enterprises Organize AWS: Accounts and Organizations 152
10.4 Identity and Access: How IAM Thinks 154
10.5 A Mental Map of AWS Services 157
10.6 Networking: VPCs and How Traffic Flows 157
10.7 Compute: Choosing Where Code Runs 159
10.8 Storage and Databases 161
10.9 Integration: Queues, Events, and Workflows 162
10.10 Data and Analytics on AWS 163
10.11 AI on AWS: Bedrock, AgentCore, and Supporting Services 165
10.12 Security and Operations Services 167
10.13 Cost: Where AWS Bills Surprise People 168
10.14 Infrastructure as Code on AWS 170
10.15 Translating AWS to Azure and Google Cloud 170
Hands-On Exercise: From First Login to Your First Bedrock Call 173
Checklist: Your First Week in a Customer's AWS Environment 174
Chapter Summary 175

11 Microsoft Azure Essentials 177
A Guest in Brightline's Tenant 178
11.1 Why Azure Is Everywhere in the Enterprise 178

11.2 Geography: Regions, Zones, and Pairs 179
11.3 How Azure Is Organized 179
11.4 Identity: Microsoft Entra ID and Azure RBAC 181
11.5 A Mental Map of Azure Services 183
11.6 Networking: Hub and Spoke, Private Endpoints, and Private DNS 183
11.7 Compute: Choosing Where Code Runs 185
11.8 Storage and Databases 185
11.9 Integration: Messaging, Events, and APIs 186
11.10 Data and Analytics: Microsoft Fabric and Azure Databricks 186
11.11 AI on Azure: Microsoft Foundry and Friends 187
11.12 Security and Operations Services 189
11.13 Cost on Azure 190
11.14 Infrastructure as Code on Azure 191
11.15 Translating AWS to Azure 191
Hands-On Exercise: From First Login to Your First Foundry Call 192
Checklist: Your First Week in a Customer's Azure Environment 194
Chapter Summary 194

12 Google Cloud Essentials 197

A Query That Was Not Allowed to Leave 198
12.1 Why Google Cloud 198
12.2 Geography 199
12.3 How Google Cloud Is Organized 199
12.4 Identity and IAM 200
12.5 A Mental Map of Google Cloud Services 202
12.6 Networking: Global VPCs, Shared VPC, and Perimeters 203
12.7 Compute: Cloud Run First 205
12.8 Storage and Databases 205
12.9 Integration 206
12.10 Data and Analytics: BigQuery and Friends 206
12.11 AI on Google Cloud: Vertex AI and Gemini 208
12.12 Calling Google Cloud From Another Cloud 209
12.13 Security, Operations, and Infrastructure as Code 210
12.14 Cost on Google Cloud 210
12.15 Translating Between the Three Clouds 211
Hands-On Exercise: From First Login to Your First Gemini Call 212
Checklist: Your First Week in a Customer's Google Cloud Environment 213
Chapter Summary 214

13 Cloud Networking, Hybrid, and On-Premises Connectivity 215

Two Networks, One Address 216
13.1 Why Networking Sets the Timeline 216
13.2 IP Address Planning 217
13.3 Network Topologies 218
13.4 Hybrid Connectivity 219
13.5 Connecting Clouds to Each Other 220

13.6 Private Access to Services 221
13.7 DNS Across Clouds and Data Centers 222
13.8 Ingress: How Traffic Gets In 223
13.9 Egress: Controlling How Traffic Leaves 224
13.10 From Perimeters to Zero Trust 224
13.11 Troubleshooting Cloud Networks 225
13.12 Requesting Network Changes 225
Hands-On Exercise: Connect a Service, Not a Network 226
Checklist: Network Design Review 227
Chapter Summary 227

14 Containers and Kubernetes 229

"All New Services Run on the Platform" 230
14.1 What Containers Are 230
14.2 Building Good Images 231
14.3 Running Containers Well 232
14.4 Kubernetes: Architecture 233
14.5 The Core Objects 234
14.6 How a Request Reaches a Pod 236
14.7 Resources, Probes, and Autoscaling 237
14.8 Kubernetes Security 238
14.9 Packaging and Delivery: Helm, Kustomize, and GitOps 238
14.10 Managed Kubernetes: EKS, AKS, and GKE 239
14.11 GPUs and AI Workloads on Kubernetes 240
14.12 When Not to Use Kubernetes 241
14.13 Debugging Kubernetes 241
Hands-On Exercise: Deploy, Break, and Fix 242
Checklist: Running a Workload on a Customer's Platform 243
Chapter Summary 244

15 Infrastructure as Code, CI/CD, and Platform Engineering 245

"Who Changed the Security Group?" 246
15.1 Why Infrastructure Belongs in Code 246
15.2 Terraform and OpenTofu 247
15.3 State: Terraform's Memory 249
15.4 Structuring Infrastructure Code 249
15.5 Other Infrastructure-as-Code Tools 251
15.6 Policy as Code 251
15.7 CI/CD From Commit to Production 252
15.8 Deployment Strategies and Rollback 252
15.9 Securing the Pipeline 254
15.10 Measuring Delivery 255
15.11 Platform Engineering 255
15.12 Delivering AI Changes 256
Hands-On Exercise: A Reviewed, Keyless Infrastructure Pipeline 257
Checklist: Infrastructure and Delivery Handover 258

Chapter Summary 258

16 Observability, SRE, and Performance 261

"The Assistant Is Slow" 262
16.1 From Monitoring to Observability 262
16.2 OpenTelemetry 263
16.3 Reading a Trace 264
16.4 What to Measure 265
16.5 SLIs, SLOs, and Error Budgets 266
16.6 Alerting 267
16.7 Incident Response 268
16.8 Runbooks 269
16.9 Performance Engineering 269
16.10 Observability for AI Systems 270
Hands-On Exercise: See It, Define It, Stress It 272
Checklist: Before Handing Over a Production System 273
Chapter Summary 273

17 Cloud Cost and FinOps 275

Four Times the Estimate 276
17.1 Why Cost Is an Engineering Concern 276
17.2 How Cloud Bills Are Structured 277
17.3 The FinOps Practice 278
17.4 Visibility: Seeing Where Money Goes 278
17.5 Estimating Before You Build 279
17.6 Token Economics 280
17.7 GPU Costs and the Self-Hosting Decision 282
17.8 Common Optimizations 283
17.9 Guardrails Against Surprises 284
17.10 Unit Economics and Value 284
Hands-On Exercise: Build and Calibrate a Cost Model 285
Checklist: Cost Readiness for an Engagement 286
Chapter Summary 286

PART IV — DATA AND AI

18 Data Platforms for AI: Lakehouse, Spark, Databricks, and Snowflake 291

Which Table Is the Truth? 292
18.1 Why Data Platforms Matter for AI 292
18.2 From Warehouse and Lake to Lakehouse 293
18.3 Open Table Formats 294
18.4 The Medallion Architecture and Data Contracts 295
18.5 Apache Spark 296
18.6 Databricks 297
18.7 Snowflake 298
18.8 Choosing and Combining Platforms 300
18.9 Orchestration and Transformation 300

18.10 Streaming Data 301
18.11 Data Quality 302
18.12 Governance: Catalogs, Lineage, and Access 302
18.13 Preparing Data for AI 303
18.14 Working With the Customer's Data Team 304
Hands-On Exercise: A Small Lakehouse Pipeline 305
Checklist: Before Building AI on Customer Data 306
Chapter Summary 306

19 AI and LLM Foundations 309
"Can It Just Know Our Products?" 310
19.1 The Vocabulary of AI 310
19.2 How Models Learn 311
19.3 Neural Networks and the Transformer 312
19.4 Tokens, Embeddings, and Context 313
19.5 How LLMs Are Trained 315
19.6 How LLMs Generate Text 316
19.7 Reasoning, Multimodality, and Tool Use 316
19.8 What LLMs Do Well — and Where They Fail 318
19.9 The Model Landscape 319
Hands-On Exercise: Getting a Feel for Models 320
Checklist: AI Foundations for Every Engagement 321
Chapter Summary 321

20 Working With Models: Prompting, Fine-Tuning, and Serving 323
The Prompt That Worked on Tuesday 324
20.1 Anatomy of a Prompt 324
20.2 Context Engineering 326
20.3 Structured Outputs 327
20.4 Prompting, Retrieval, or Fine-Tuning? 328
20.5 Fine-Tuning Methods 329
20.6 Accessing Models 330
20.7 Self-Hosting Open Models 331
20.8 Caching, Batching, and Rate Limits 332
20.9 Resilience and Operating Model Calls 333
Hands-On Exercise: From Chat Prompt to Production Call 334
Checklist: Production-Ready Model Use 335
Chapter Summary 335

21 RAG and Enterprise Knowledge 337
The Answer Was in the Wrong Document 338
21.1 What RAG Is and Why It Matters 338
21.2 Ingestion: Getting Content Ready 339
21.3 Retrieval 341
21.4 Vector Databases and Retrieval Services 342
21.5 Permission-Aware Retrieval 343
21.6 Generation With Citations 344

21.7 Advanced Retrieval Patterns 344
21.8 Evaluating RAG 345
21.9 Operating a RAG System 347
Hands-On Exercise: Build and Measure a Small RAG System 348
Checklist: Enterprise RAG 348
Chapter Summary 349

22 Document AI and Multimodal Systems 351
Four Thousand Warranty Claims a Week 352
22.1 Documents in the Enterprise 352
22.2 The Document Processing Pipeline 353
22.3 OCR and Layout Analysis 353
22.4 Cloud Document Services 354
22.5 Extraction With Multimodal LLMs 355
22.6 Schemas, Validation, and Confidence 356
22.7 Human Review 358
22.8 Images, Drawings, Audio, and Video 358
22.9 Measuring Accuracy 359
Hands-On Exercise: A Small Extraction Pipeline 361
Checklist: Document AI in Production 361
Chapter Summary 362

Appendix A AWS, Azure, and Google Cloud Service Comparison 363

Appendix B Command-Line Quick Reference 368

Appendix C Templates for Field Engineering 371

Appendix D Further Reading 374

Glossary 377

About the Author 386

Index 387

Preface

For most of my career, the hardest part of engineering has not been writing code. It has been everything around the code: understanding what a business actually needs, finding the data that answers the question, getting access to the systems that hold it, convincing a security team that a design is safe, making something work inside an environment that was built long before anyone imagined it, and leaving behind a system that other people can run after the project ends.

For a long time, that work did not have a single name. Some of it was called architecture, some consulting, some integration, some "just getting things done." Then the industry started using a term that had been coined years earlier at Palantir: the **Forward Deployed Engineer**. An FDE is an engineer who works directly inside a customer's world — their workflows, their data, their constraints, their cloud — and turns an ambiguous problem into a working, adopted, production system.

The role has existed in various forms for more than a decade. What changed is artificial intelligence. Large language models and AI agents made it possible to build systems that read documents, answer questions, and take actions in ways that were not practical a few years ago. But they also made the gap between a demonstration and a dependable production system wider than ever. A model can produce an impressive answer in a meeting. Connecting that model to an enterprise's real data, real identity system, real network, and real approval processes — safely, measurably, and at a cost the business accepts — is engineering work of the most demanding kind. That is why companies building AI platforms, and the enterprises adopting them, began hiring Forward Deployed Engineers in large numbers.

Why this book

When I looked for a book that prepared engineers for this work, I found excellent books on individual topics — cloud architecture, Kubernetes, data engineering, machine learning, large language models — but none that connected them the way the job requires. An FDE does not get to specialize in one layer. On a single engagement, you may need to read a

customer's AWS account structure, debug a failing network path, design a retrieval pipeline over messy documents, explain a model's failure modes to an executive, and write the runbook the customer's operations team will use at three in the morning.

This book is my attempt to provide that connected view. It is written for software engineers who want to become Forward Deployed Engineers, for experienced FDEs who want a structured reference, and for architects, engineering managers, and technical leaders who work alongside them.

Two volumes

The material is divided into two volumes.

Volume 1 — Foundations: Software, Cloud, Data, and AI builds the base. It begins with the FDE role itself and a short history of how we arrived at today's technology landscape. It then covers the software engineering fundamentals you will rely on in the field, gives extended, practical coverage of cloud computing — including dedicated chapters on AWS, Microsoft Azure, and Google Cloud — and ends with the data platforms and AI foundations on which modern intelligent systems are built.

Volume 2 — Agents, MCP, and Engineering in the Field builds on that base. It covers AI application architecture, agents, the Model Context Protocol, enterprise AI security and governance, and — in its final part — the craft of field engineering itself: discovery, scoping, prototyping, delivery, leadership, and building a career as an FDE.

Each volume stands on its own, but they are designed to be read together.

How this book was written

The views in this book come from two decades of building data and cloud systems in large, regulated enterprises. Throughout, you will find "Field Notes" — prompts that link each chapter to your own work — alongside a running fictional example: Acme, a company whose AI journey we follow from first discovery to production.

In keeping with the subject of the book, I used AI tools extensively while writing it: to explore structures, draft and revise material, check consistency, and produce diagrams. Every chapter was reviewed, reshaped, and verified by me, and the judgments in it are my own. I believe this is

how much engineering work will be done in the coming years — with AI as a powerful collaborator and humans accountable for the result — and I have tried to practice what the book recommends.

A note on currency

Cloud and AI services change quickly. Product names change, features are added, and prices fall. I have tried to focus on durable principles and to present specific services as examples of those principles. Where specific facts are given, they were checked at the time of writing. When you rely on a detail for real work, confirm it in the provider's current documentation — a habit every good FDE already has.

I hope this book helps you do the work well: to understand problems before solving them, to respect the systems and people you find in the field, and to leave every customer with something that works.

Neeraj Bhatyal
New York, 2026

How to Use This Book

Who this book is for

This book assumes you can program in at least one language and have built or maintained some software professionally. It does not assume you are an expert in cloud platforms, data engineering, or artificial intelligence. Each chapter begins from first principles and builds toward the judgment an experienced practitioner uses in the field.

Ways to read it

- **From start to finish.** The chapters build on each other, and the Acme story runs through them in order. This is the best path if you are preparing for an FDE role.
- **By need.** If you are about to work in a customer's Azure environment, go straight to Chapter 11 and then Chapter 13. If you are about to build a retrieval system, read Chapters 18 through 21. Cross-references point you to related material.
- **As a field reference.** The checklists at the end of each chapter, the comparison tables in Appendix A, the command reference in Appendix B, and the templates in Appendix C are designed to be used during real engagements.

How each chapter is organized

Every chapter follows the same structure so that you always know where to find things:

1. **Opening scenario.** A short situation from the Acme engagement that frames the chapter's problem.
2. **Core concepts.** The ideas explained in plain language, with diagrams.
3. **Cloud mapping.** Where it applies, how the concept appears on AWS, Azure, and Google Cloud.
4. **Field Notes.** Prompts that connect the chapter to your own work.
5. **Hands-on exercise.** A practical task you can complete in a personal or sandbox account.

6. **Checklist and summary.** A one-page checklist for real work and a summary of the chapter's key points.

Acme and Priya

Throughout both volumes, we follow **Priya**, a Forward Deployed Engineer, through her engagement with **Acme**, a fictional company that sells and services industrial equipment worldwide. Acme has the kind of environment you will find in many real enterprises: decades-old core systems alongside modern cloud services, a large customer support organization, data spread across many platforms, and a leadership team eager to use AI but cautious about risk. Acme's story gives every concept a concrete context.

Conventions

Text in `monospace` indicates commands, code, file names, and identifiers. 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 and costs

The hands-on exercises use the free tiers and low-cost services of the major cloud providers wherever possible. Always use a personal or sandbox account — never a customer's or employer's production environment — set a budget alert before you begin, and follow the cleanup steps at the end of each exercise.

Acknowledgments

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

PART I

The Role and Its Roots

What a Forward Deployed Engineer is, why the role matters now, and how we arrived at today's software, cloud, and AI landscape.

1 The Forward Deployed Engineer

2 A Short History: From Mainframes to Agents

1

The Forward Deployed Engineer

What the role is, where it came from, why it matters in the age of AI, and how this book prepares you for it

IN THIS CHAPTER

- What a Forward Deployed Engineer does — and what makes the role different
- Where the role came from, and why AI made it one of the most sought-after jobs in technology
- How an FDE differs from product engineers, solutions architects, consultants, and sales engineers
- The FDE delivery loop and the three kinds of ownership every engagement must settle
- The skills profile of an effective FDE
- Meet Acme and Priya, whose story runs through both volumes
- A map of the book

"Can You Just Build Us an AI Agent?"

The request arrives in a short email from Acme's Vice President of Customer Support:

THE REQUEST

"We want an AI agent that resolves support tickets automatically. Our competitors are doing it. Can you have something running by the end of the quarter?"

The request sounds clear. A capable engineer could pick a language model, connect it to the ticketing system, write a few prompts, and produce an impressive demonstration within a week. Many teams do exactly that. A surprising number of those demonstrations never become systems anyone relies on.

Look more closely, and the request contains almost none of the information needed to build something useful. What does "resolve" mean — answer the customer, close the ticket, issue a refund? Which tickets are in scope: password resets, or warranty disputes involving thousands of dollars? What happens when the system is unsure? Which systems hold the truth about an order, a warranty, or a customer's contract? Is the agent allowed to change anything, or only to suggest? What data may leave Acme's environment, and which country's laws apply to it? Who approves putting the system into production? And how will anyone know whether it worked?

The most important work has not started yet. Answering those questions — with the customer, inside the customer's real environment — and then building, validating, and handing over a system that actually works is the job of a **Forward Deployed Engineer**.

1.1 What a Forward Deployed Engineer Does

A Forward Deployed Engineer (FDE) is a software engineer who works directly with a customer to turn an ambiguous business problem into a working, adopted, production system. "Forward deployed" describes where

the work happens: not at a distance, building a product for thousands of users at once, but up close — in the customer's workflows, their data, their cloud accounts, their security reviews, and their meetings.

That sounds simple. In practice it combines responsibilities that most organizations split across several roles. On a single engagement, an FDE may need to:

- understand a business workflow well enough to question the customer's proposed solution;
- explore an unfamiliar technical environment without treating it as an obstacle to be replaced;
- translate operational needs into testable system behavior;
- build software that works within the customer's specific constraints — their identity system, network rules, approved platforms, and data policies;
- validate the system against real scenarios, including the ugly ones;
- explain technical decisions to executives, security teams, and the people who will use the system; and
- leave behind a system the customer can own, operate, and change after the FDE has gone.

The common thread is accountability for the *outcome*, not just the code. A product engineer's work is finished when a feature ships and works as designed. An FDE's work is finished when the customer's problem is measurably better and someone on the customer's side can keep it that way.

KEY IDEA

An FDE does not merely implement what a customer asks for. An FDE discovers the problem behind the request, validates the constraints, builds a working solution, and transfers an operable system.

1.2 Where the Role Came From

The title "Forward Deployed Engineer" was coined at Palantir in the early 2010s. Palantir built data integration and analysis software for governments and large companies, and it learned quickly that the software alone did not solve customers' problems. Every customer's data was different, every workflow had its own logic, and the hard part of each

deployment was connecting the product to that reality. Palantir's answer was to send engineers to work inside customer organizations, writing real code against real data. Internally the team was called "Delta." For several years, Palantir employed more of these engineers than traditional product engineers.

Palantir described the distinction with a memorable phrase. A product engineer works on *one capability for many customers*. A forward deployed engineer works on *many capabilities for one customer*. The two roles are complementary: FDEs discover what customers actually need and feed it back into the product, while product engineers turn repeated patterns into reusable capabilities.

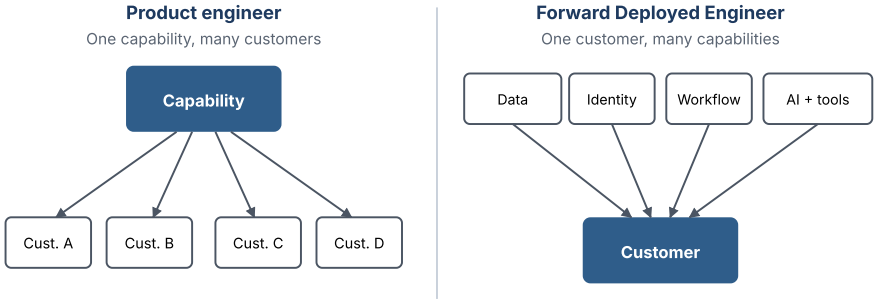


Figure 1.1 — Two complementary orientations. Product engineering builds one capability for many customers. Forward deployed engineering assembles many capabilities into a working system for one customer.

For most of the 2010s, the role remained associated mainly with Palantir and a handful of companies selling complex enterprise software. That changed in 2025. As companies building AI models and platforms tried to move their customers from experiments to production, they found the same problem Palantir had found a decade earlier: the technology was powerful, but every enterprise deployment required deep, hands-on engineering in the customer's environment. OpenAI formed a forward deployed engineering team in early 2025, and AI companies, enterprise software vendors, and startups followed. Hiring for FDE roles rose sharply, and the venture capital firm Andreessen Horowitz called it one of the hottest jobs in technology.

1.3 Why AI Made the Role Essential

Earlier waves of enterprise technology were largely deterministic. A database returns the rows you ask for. An API either succeeds or returns an error. If you configure a system correctly, it behaves the same way every time. Deploying such systems was hard, but the difficulty was mostly in integration and scale.

AI systems add a new kind of difficulty. A language model produces fluent, plausible answers, and its quality depends heavily on the context it is given — which documents it retrieves, which tools it can call, which instructions it follows. It can be confidently wrong. Its behavior can change when the model is upgraded. It can be manipulated by text hidden in the documents it reads. And it is now often asked not only to answer questions but to *act*: to update records, send messages, or trigger workflows.

This creates a large gap between a demonstration and a production system. A demonstration needs a model and a few good examples. A production system needs answers to questions such as these:

- Which data is the system allowed to see, and how is that enforced for each user?
- How do we measure whether its answers are correct, and how often is "often enough"?
- What may it do on its own, and what requires a human's approval?
- How do we detect, investigate, and recover when it gets something wrong?
- What does it cost per request, and does that cost make business sense?
- Who is accountable for its behavior once the engineers who built it have moved on?

None of these questions can be answered from outside the customer's organization. They depend on the customer's data, policies, regulators, risk tolerance, and people. That is why AI turned the FDE from a niche role into a central one: the value of AI in an enterprise is realized — or lost — in exactly the space where FDEs work.

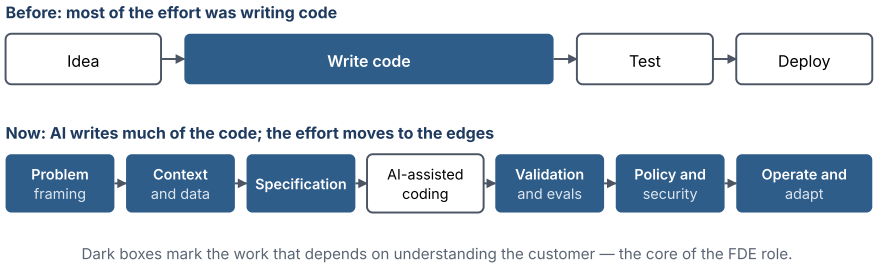


Figure 1.2 — The bottleneck moves. As AI tools make code faster to produce, the scarce skills become framing the problem, supplying the right context, validating behavior, enforcing policy, and operating the result.

There is a second reason AI elevates the FDE. AI coding assistants now generate a large share of routine code. That does not make engineers less important — but it shifts where their value lies. When code is cheap, the expensive mistakes are building the wrong thing, building it on the wrong data, or building it in a way the customer cannot safely operate. Figure 1.2 shows how the bottleneck moves. Every box that grew darker is a box where an FDE's judgment, not typing speed, determines the outcome.

1.4 The FDE and Neighboring Roles

The FDE role overlaps with several familiar roles, and in some organizations the titles are used loosely. It helps to understand the typical center of gravity of each.

Table 1.1 — The FDE and adjacent roles

Role	Primary focus	Typical output
Product engineer	Building and evolving a product for a broad base of users	Product features and maintainable code
Solutions architect	Designing a system that satisfies requirements and constraints	Architecture, design decisions, guidance
Consultant	Analyzing a problem and recommending a course of action	Assessments, recommendations, plans
Implementation engineer	Configuring and deploying a known product	A configured, integrated product
Sales engineer	Demonstrating technical fit during a sale	Demos, technical validation, customer confidence
Forward Deployed Engineer	Discovering, engineering, validating, and enabling a solution in the customer's real environment	A working system, evidence of value, and an operable handoff

An FDE regularly does some of each of these jobs. The difference is that the FDE connects them in one continuous loop and stays accountable from the first conversation to the measured outcome. Two comparisons come up often enough to deserve a closer look.

FDE versus solutions architect. Solutions architects typically design, advise, and sometimes build prototypes, often working outside the customer's systems. FDEs are more hands-on and work with more ambiguity: they write production code directly in the customer's environment, debug the customer's systems, and stay through deployment and adoption. A good FDE thinks like an architect; a good architect respects what FDEs learn in the field.

FDE versus consultant. Consultants are usually measured by the quality of their analysis and recommendations. FDEs are measured by whether something works. An FDE's recommendation is often a running system. And while a consulting engagement may end with a report, an FDE engagement should end with a customer team that can operate what was built — and, for the FDE's own company, with product insights that make the next deployment easier.

1.5 The FDE Delivery Loop

Because the customer's understanding of the problem changes as the work uncovers real data, real workflows, and real failure cases, FDE work is not a straight line from requirements to delivery. It is a loop, shown in Figure 1.3.

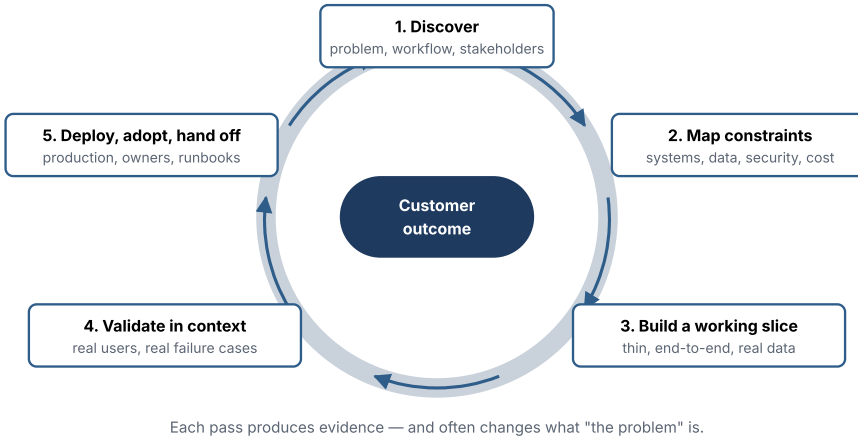


Figure 1.3 — The FDE delivery loop. Discovery, constraint mapping, building, validation, and handoff repeat in short cycles. Each cycle produces evidence that sharpens the next.

1. **Discover.** Understand the workflow and the outcome the customer actually needs, separate from the solution they proposed. Identify who uses, sponsors, approves, and will operate the system.
2. **Map constraints.** Learn the environment: systems of record, data quality, identity, network rules, approved platforms and models, regulations, budget, and timeline. Separate facts from assumptions.
3. **Build a working slice.** Deliver the smallest piece of real functionality that runs end to end on real data — not a slide, and not a disconnected prototype.
4. **Validate in context.** Put the slice in front of real users with real cases, including the difficult ones. Measure against the baseline.
5. **Deploy, adopt, and hand off.** Move what works into production with monitoring, runbooks, and named owners — and feed what you learned into the next cycle.

In a good engagement, the loop turns every week or two, and every turn produces evidence: a demonstration on real data, a measured improvement, a resolved security question, a decision made. Volume 2's

final part is devoted to running this loop well. For now, notice that only one of the five steps is primarily about building. The others are about understanding, proving, and transferring — the parts that make building worthwhile.

1.6 Three Kinds of Ownership

Customers often bring in FDEs because they lack the time, experience, or capacity to build something themselves. That creates a quiet trap: the FDE can become the only person who understands how the system works, why it behaves as it does, and how to recover it when it fails. When the engagement ends, the system slowly decays — or the FDE is never really allowed to leave.

The way to avoid this trap is to make ownership explicit from the start. Every engagement must settle three distinct kinds of responsibility, which often belong to different people.

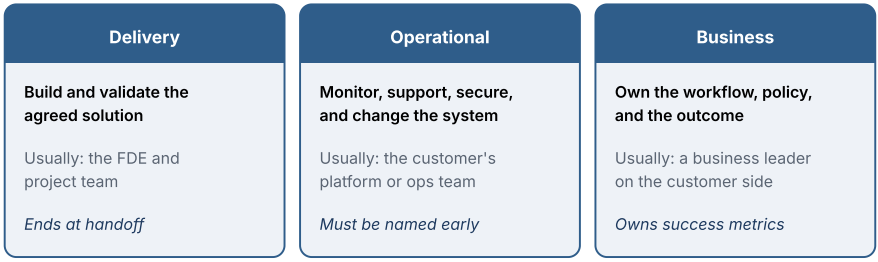


Figure 1.4 — Three kinds of ownership. An engagement is at risk whenever any of the three is left implicit — and especially when the FDE quietly holds all three.

Delivery responsibility is building and validating the agreed solution. This is usually the FDE's. **Operational responsibility** is monitoring, supporting, securing, and changing the system once it is live; it must belong to a team that will still be there next year. **Business responsibility** is owning the workflow, the policies the system enforces, and the outcome it is meant to improve. For Acme's support agent, that is likely the support organization's leadership — not engineering, and certainly not the FDE.

Naming these owners in the first weeks feels premature to many customers. It is not. The answers shape the design: a system that will be operated by a small platform team should be simpler than one operated by

a large SRE organization, and a workflow whose business owner wants to adjust policies weekly needs those policies to live in configuration, not code.

1.7 The Skills of an Effective FDE

The strongest FDE is rarely the person who knows the most technologies. It is the person who can enter an unfamiliar environment, work out what matters, learn what is necessary, make sound decisions, work well with the people there, and leave behind a system that survives. That is a different profile from a specialist's.

A useful way to picture it is the **π-shaped** engineer: broad knowledge across the whole system, connected to deep expertise in two or more areas. Breadth lets you understand how a customer's applications, data, cloud, security, and AI fit together. Depth lets you solve the hard problems that inevitably appear in at least one of those areas.

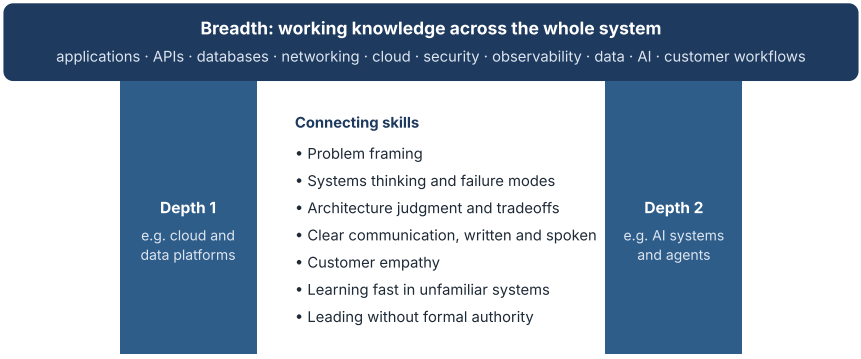


Figure 1.5 — The π-shaped FDE. Broad system knowledge, two or more areas of real depth, and a set of connecting skills that turn knowledge into outcomes.

There is no single correct combination of deep areas. One FDE may be deep in distributed systems and security; another in data engineering and AI evaluation. What matters is the distinction between *strategic depth* and *technology accumulation*. Knowing the names of twenty tools does not create judgment. The questions that create judgment are always the same: When should this technology be used, and when not? What does it cost to

operate? What failure modes does it introduce? Who will own it? What does the customer already have? What simpler option could work? What evidence would justify the choice?

The connecting skills in Figure 1.5 are what make an FDE effective rather than merely knowledgeable. Several of them — communication, empathy, leadership without authority — are often called "soft skills," which undersells them. In the field, a well-written one-page summary that gets a security approval in three days instead of three weeks is an engineering achievement. So is a demonstration that makes a skeptical operations team say, "We could run this."

Table 1.2 — Where this book builds each capability

Capability	What it looks like in the field	Where covered
Software engineering	Read and change unfamiliar codebases; build reliable services	Vol. 1, Part II
Cloud and infrastructure	Navigate AWS, Azure, or Google Cloud; deploy, network, and operate	Vol. 1, Part III
Data	Find, assess, and move the data a solution depends on	Vol. 1, Ch. 6, 18
AI systems	Choose models, ground them in knowledge, evaluate them	Vol. 1, Part IV; Vol. 2, Part V
Agents and MCP	Connect AI to tools and systems safely	Vol. 2, Parts V–VI
Security and governance	Pass security reviews; design within trust boundaries and regulation	Vol. 1, Ch. 7; Vol. 2, Part VII
Field craft	Discovery, scoping, prototyping, delivery, leadership, handoff	Vol. 2, Part VIII

1.8 What a Week Looks Like

To make the role concrete, here is a realistic week in the middle of an engagement. It is not glamorous, and that is the point: most of the value comes from many small, well-judged actions.

- **Monday.** A working session with three support agents, watching them handle real tickets. You discover that a third of "warranty" tickets are actually shipping delays — a category the original plan ignored.
- **Tuesday.** Your retrieval prototype returns outdated policy documents. You trace it to two copies of the knowledge base, one of which stopped syncing eight months ago. You raise it with the knowledge management owner and point the pipeline at the right source.

- **Wednesday.** A security review. You bring a one-page data-flow diagram showing exactly what leaves which network, which model endpoint is used, and how personal data is masked. The review takes forty minutes instead of the expected three weeks of back-and-forth.
- **Thursday.** You build an evaluation set of 150 real tickets with the support team's help, and run the current version against it. Accuracy on shipping questions is high; on warranty questions, poor. You narrow the next release to shipping questions only.
- **Friday.** A demonstration to the VP of Support using real tickets from that week, followed by a short written update: what improved, what did not, what decision you need from her, and what happens next.

Notice how little of that week involved choosing frameworks or tuning prompts, and how much involved understanding the workflow, finding the truth in the data, earning trust, and narrowing scope based on evidence. That balance is typical.

1.9 Meet Acme and Priya

Throughout both volumes, we follow one engagement from beginning to end. **Acme** is a fictional company that sells and services industrial equipment — pumps, compressors, and control systems — to customers in more than forty countries. It has about eight thousand employees, a customer support organization of six hundred agents, and a technology estate that reflects thirty years of growth.

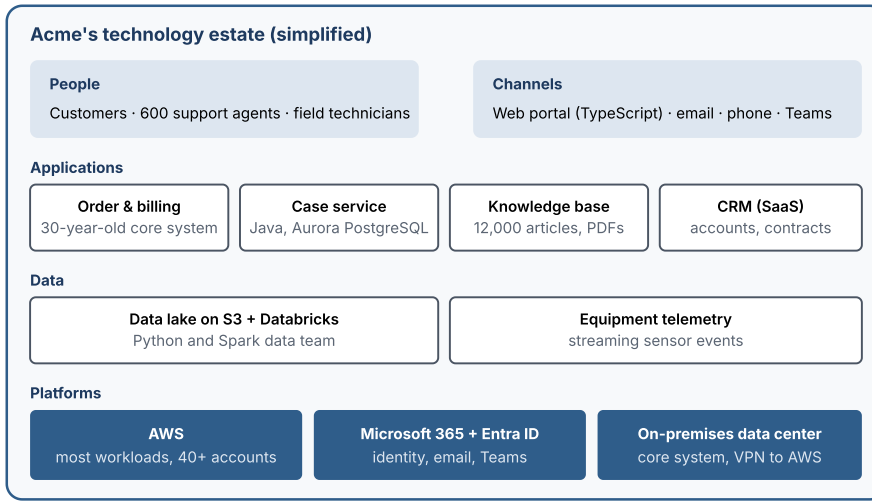


Figure 1.6 — Acme's technology estate. A realistic mix of old and new: a decades-old core system, Java services, a SaaS CRM, a data lake, AWS as the main cloud, and Microsoft for identity and collaboration.

Acme's environment, shown in Figure 1.6, is typical of the enterprises FDEs serve. Its order and billing system runs in its own data center and has been extended so many times that only a few people understand all of it. Support cases are handled by a Java service backed by a PostgreSQL database on AWS. Twelve thousand knowledge articles live in a content system, many as PDFs of uneven quality. Customer accounts and contracts are in a SaaS CRM. The data team has built a data lake on Amazon S3 with Databricks, and equipment in the field streams sensor data back to Acme. Employees sign in with Microsoft Entra ID and collaborate in Microsoft Teams.

Acme's leadership wants AI to reduce support costs and response times. Its security team, having watched peers suffer data leaks, is cautious. Its support agents are curious but worried about their jobs. Its platform team is already busy. Every one of those facts will shape the engagement.

Priya is the Forward Deployed Engineer assigned to Acme by the AI platform company Acme has chosen to work with. She has eight years of experience as a backend and data engineer and two years as an FDE. She is strong in cloud infrastructure and data systems, competent in AI, and still learning how to navigate a large company's politics. Over the course of both volumes, you will see her make good decisions, a few bad ones, and recover from them — as every real FDE does.

1.10 How This Book Is Organized

This book is published in two volumes that follow the shape of the FDE's knowledge, from foundations to field craft.

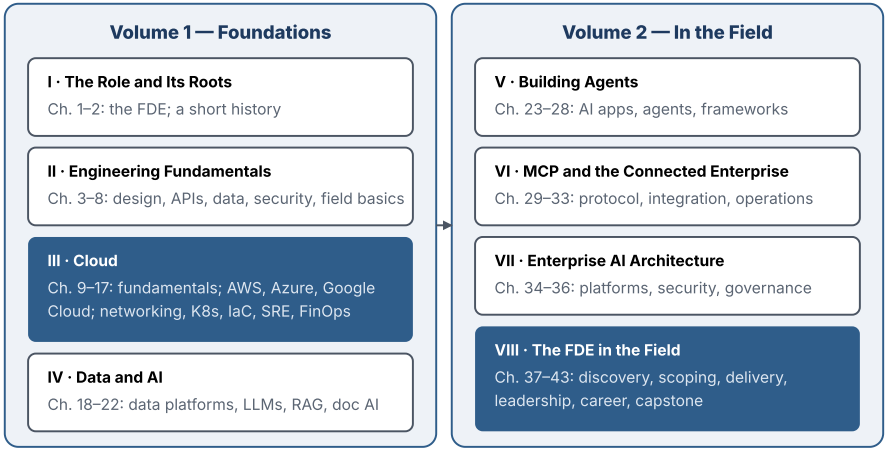


Figure 1.7 — The map of the book. Volume 1 builds the foundation; Volume 2 applies it to agents, connected systems, governance, and the craft of field engineering. The emphasized parts are where the book spends the most time.

Volume 1 builds the foundation. Part I introduces the role and gives a short history of the technologies an FDE inherits in every customer environment. Part II covers the software engineering fundamentals you will use daily, including a chapter on the practical field skills — Linux, the shell, networking, and debugging — that are rarely taught but constantly needed. Part III is the largest part of this volume: it explains the concepts all clouds share, gives a dedicated tour of AWS, Microsoft Azure, and Google Cloud, and covers networking, containers, infrastructure as code, operations, and cost. Part IV covers the data platforms that feed AI systems and the foundations of working with AI models, retrieval, and documents.

Volume 2 applies that foundation. It covers AI application architecture and agents, the Model Context Protocol and connected enterprise systems, AI security and governance, and finally the craft of field engineering: discovery, scoping, prototypes, delivery, leadership, and building a career as an FDE. It ends with a capstone that follows Acme's engagement from first conversation to production.

FIELD NOTES — YOUR TURN

Before you move on, take ten minutes to write down a real moment from your own work when understanding the actual problem mattered more than the technology. Prompts to start from:

- A request that turned out to be a different problem once you watched the real workflow.
- A project that technically succeeded but was not adopted — and what you learned about ownership.
- The moment you realized that most of your value on a project came from work outside the code.

Hands-On Exercise: Your FDE Profile and Your First One-Pager

This exercise has two parts. Neither needs a computer beyond a text editor, and both produce artifacts you will reuse throughout the book.

Part 1 — Map your π

Draw your own version of Figure 1.5. List the areas where you have real depth today — where you could debug a hard problem without help — and the areas where you have working breadth. Then mark one area you want to deepen in the next six months. Be honest: the goal is a learning plan, not a résumé. Keep this page; Chapter 42 returns to it.

Part 2 — Reframe a request

Take the Acme request from the start of this chapter — "an AI agent that resolves support tickets automatically" — or a real request from your own work. Write a one-page engagement summary using the template in Appendix C.1. You will not be able to fill in every section. That is the point: list the questions you would need answered, who you would ask, and what you would want to see in the first two weeks. Compare your list with the questions in Section 1.3.

Checklist: Starting Any FDE Engagement

☐ I can state the customer's problem as a workflow and an outcome, not just as the requested solution.

- ☐ I know who uses, sponsors, approves, and will operate the system.
- ☐ Delivery, operational, and business ownership each have a named owner.
- ☐ I have separated what I know from what I am assuming.
- ☐ I know how success will be measured, and what the baseline is today.
- ☐ The first working slice is small, end to end, and uses real data.
- ☐ I have agreed how and when I will show progress — with evidence, not slides.
- ☐ I know what "done" looks like for my involvement, and how I will hand over.

Chapter Summary

A Forward Deployed Engineer turns an ambiguous customer problem into a working, adopted, production system, working directly in the customer's environment and staying accountable for the outcome. The role was created at Palantir in the early 2010s and became one of the most sought-after roles in technology from 2025 onward, as AI companies and enterprises discovered that the value of AI is realized — or lost — in the gap between a demonstration and a dependable system.

FDEs overlap with product engineers, solutions architects, consultants, implementation engineers, and sales engineers, but differ in connecting discovery, engineering, validation, and handoff in one continuous loop. Every engagement must settle three kinds of ownership — delivery, operational, and business — so that the system outlives the FDE's involvement. The effective FDE is π-shaped: broad across the system, deep in two or more areas, and strong in the connecting skills of framing, judgment, communication, and leadership without authority. As AI makes code cheaper to produce, those connecting skills become the scarce ones.

LOOKING AHEAD

Every customer environment is a museum of decisions made over decades: a core system from the mainframe era, services from the web era, a data lake from the big-data era, and AI from this one. Chapter 2 tells the short story of how those layers came to be — so that when you meet them in the field, you understand why they exist and what they can and cannot do.

2

A Short History: From Mainframes to Agents

How the layers of technology you meet in every customer environment came to be — and why each still matters

IN THIS CHAPTER

- Why history is a practical skill for an FDE
- Computing hardware: from calculating machines to GPUs
- Programming languages and why enterprises run several at once
- How software engineering became a discipline: from the software crisis to DevOps and AI-assisted development
- Software architecture: from mainframes to microservices and agents
- The internet and APIs: from ARPANET to tool calling
- Infrastructure: from data centers to cloud, containers, and AI compute
- A five-question framework for judging any new technology

Archaeology on Day Three

On her third day at Acme, Priya traces how a single support question is answered today. A customer asks why a replacement compressor has not arrived. An agent looks up the case in a Java web application. The application calls a service that queries the order system — which turns out to be a program written in COBOL, running on a mainframe in Acme's own data center, reached through a messaging bridge built in 2009. Shipping status comes from a logistics partner's REST API. The customer's contract terms are in a SaaS CRM. The agent then pastes an answer into an email template stored in a shared drive.

In one workflow, Priya has touched technology from five decades. None of it is a mistake. Each layer was a reasonable answer to the problems of its time, and each still works well enough that replacing it has never been the top priority. The AI assistant she is about to build will not replace these layers. It will have to work *with* all of them.

This is why history is a practical skill for an FDE. When you understand why a technology was created, you understand what it is good at, what it assumes, how it tends to fail, and what it will take to connect something new to it. This chapter is deliberately short. It does not try to tell the full history of computing. It gives you just enough of each story to recognize the layers when you meet them in the field.

KEY IDEA

Enterprise technology is cumulative. New layers are added on top of old ones far more often than old ones are removed. An FDE's job is usually to connect the newest layer to all the others.

2.1 Computing: From Calculation to Intelligence

The idea of a general-purpose computing machine is older than electronics. In the 1830s, Charles Babbage designed the Analytical Engine, a mechanical computer that could, in principle, be programmed with punched cards. Ada Lovelace, writing about it in 1843, recognized that such a machine could manipulate symbols of any kind, not only numbers — and

published what is often called the first computer program. In 1936, Alan Turing described a theoretical machine that could compute anything computable, laying the mathematical foundation for every computer since.

Electronic computing arrived during and after the Second World War. Machines such as ENIAC, completed in 1945, used thousands of vacuum tubes and had to be rewired to change their task. The decisive idea was the **stored-program computer**, described in 1945 and first demonstrated in 1948: a machine that keeps its instructions in the same memory as its data, so that changing its behavior means loading a new program rather than rewiring hardware. Software, as a separate thing from hardware, begins here.

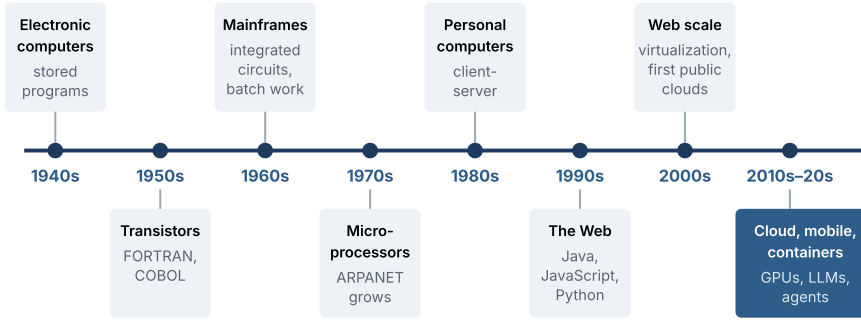


Figure 2.1 — Eight decades of computing in one line. Each era added a layer. Most large enterprises still run something from almost every one of them.

The rest of the hardware story is one of relentless miniaturization. The **transistor**, invented at Bell Labs in 1947, replaced fragile vacuum tubes. The **integrated circuit**, developed in the late 1950s, put many transistors on one chip. The **microprocessor**, introduced in 1971, put an entire processor on one chip. Each step made computers smaller, cheaper, and more reliable by orders of magnitude.

That progress produced the eras you still see in enterprises. **Mainframes**, typified by IBM's System/360 from 1964, became the backbone of banks, insurers, airlines, and governments, running batch jobs and high-volume transactions with extraordinary reliability. Many still do: if your customer processes payments, policies, or orders at scale, there is a good chance a mainframe is involved. The **personal computer**, popularized by the IBM PC

in 1981, put computing on every desk and led to client-server systems. **Networks** connected those computers, first inside companies and then globally through the internet.

Two more hardware shifts matter for today's work. **Virtualization** let one physical server run many isolated virtual machines, making hardware flexible and, eventually, rentable by the hour — the foundation of cloud computing. And **graphics processing units** (GPUs), originally built to render video games, turned out to be extremely good at the parallel arithmetic that neural networks require. When researchers trained a deep neural network on GPUs to win the ImageNet image recognition competition in 2012, modern AI's hardware foundation was set. Today, access to GPUs and similar accelerators shapes what AI systems cost and where they can run.

2.2 Programming Languages: Why Every Enterprise Runs Several

The first programs were written in **machine language** — raw numbers that the processor executes directly — and then in **assembly language**, which gave those numbers readable names. Both are tied to a specific processor and are slow to write. The breakthrough was the **high-level language**, which lets programmers express logic in a form closer to mathematics or English and relies on a compiler to translate it.

FORTRAN (1957) made scientific and engineering computation practical. **COBOL** (1959) was designed for business data processing — records, files, and arithmetic on money — and was written to be readable by managers as well as programmers. Billions of lines of COBOL still run core systems in banking, insurance, and government. When an FDE hears "the mainframe," COBOL is usually what is running on it.

Later languages introduced ideas that shape how code is organized today. **Structured programming** in the 1960s and 1970s replaced tangled jumps with clear loops and functions. **C** (1972) gave programmers efficient, portable access to hardware and became the language of operating systems. **Object-oriented programming**, popularized by C++ and later Java, organized code around objects that combine data and behavior. **Functional programming** ideas — functions without side effects, immutable data — moved from academic languages into mainstream ones.

Three languages dominate the environments in this book, and FDEs should be able to read all three:

- **Java** (1995) runs on the Java Virtual Machine and became the standard language for large enterprise back-end systems. Its strong typing, mature tooling, and frameworks such as Spring make it the language of many systems of record. Acme's case service is written in Java.
- **JavaScript** (1995) began as a scripting language for web pages and, through Node.js, also runs on servers. **TypeScript** (2012) adds static types to JavaScript and has become the default for serious web front ends and many back-end services. Acme's customer portal is TypeScript.
- **Python** (1991) became the language of data science, machine learning, and automation because it is easy to read and has an enormous ecosystem of libraries. Almost every AI framework is Python-first. Acme's data team writes Python and Spark.

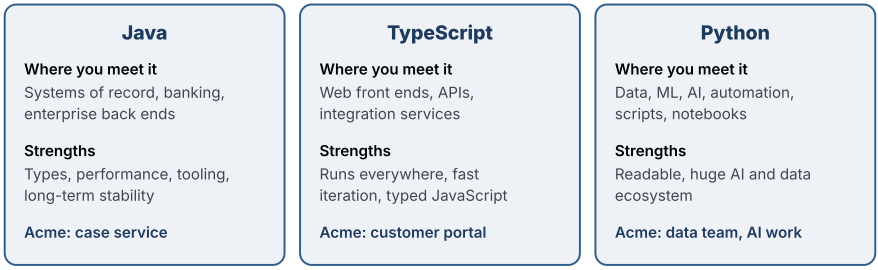


Figure 2.2 — Three languages, three ecosystems. Most enterprises are polyglot. An FDE who can read all three can follow a request across the whole system.

The practical lesson is that **language choice is an architectural decision**, not a matter of taste. Each language brings its own build tools, libraries, deployment patterns, and talent pool. When Acme's business rule for "warranty eligibility" exists in Java in the case service, in TypeScript in the portal, and in Python in a data pipeline, the three copies can quietly disagree — a problem known as *semantic drift*. Many real AI failures begin here: the assistant retrieves one version of a rule while the system of record enforces another. Part of an FDE's discovery work is finding where the authoritative version of each rule lives.

AI coding assistants have changed how code is written, but not this lesson. They can generate code in any language quickly, which makes it even easier to add a fourth copy of a business rule in a fourth language. Chapter

4 returns to how to use these tools well.

2.3 Software Engineering: From Crisis to Discipline

By the late 1960s, software projects were routinely late, over budget, and unreliable. Programs had grown far larger than the methods used to build them. At a NATO conference in 1968, participants described this as a "software crisis" and popularized the term **software engineering** — the idea that building software should be a disciplined engineering activity, not an art.

The first widely adopted response was the **waterfall** model: gather all requirements, design everything, build, test, and deploy, in that order. Waterfall made large projects plannable and auditable, which is why it suited government contracts and regulated industries. Its weakness is that it assumes requirements can be known completely in advance and that problems found late are rare. In practice, the most important things are often discovered only when real users see working software.

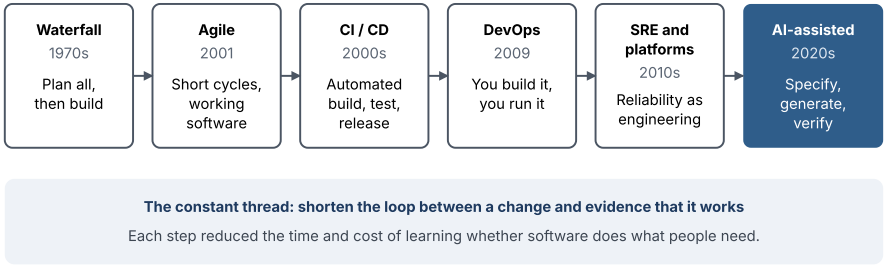


Figure 2.3 — How software delivery evolved. Each practice built on the last, and each shortened the time between making a change and knowing whether it works.

Through the 1990s, iterative methods such as Scrum and Extreme Programming emerged, and in 2001 a group of practitioners published the **Agile Manifesto**, which valued working software, collaboration with customers, and responding to change. Agile changed the unit of progress from a document to a working increment. Practices such as test-driven development and **continuous integration** — merging and testing every change automatically, many times a day — made those increments reliable.

The next barrier was the wall between the people who wrote software and the people who ran it. **DevOps**, a term that spread from 2009, broke that wall down: teams took responsibility for operating what they built, and automated the path to production with **continuous delivery**. Google's **Site Reliability Engineering** (SRE) practice, developed from 2003 and published in 2016, treated operations as a software problem, with explicit reliability targets and error budgets. More recently, **platform engineering** has packaged these practices into internal platforms so that product teams can deploy safely without becoming infrastructure experts.

The latest step is **AI-assisted and agentic software engineering**. Coding assistants now suggest and write code, generate tests, explain unfamiliar codebases, and — in agentic forms — carry out multi-step tasks such as fixing a failing test across several files. This does not remove the need for engineering discipline; it raises it. When code can be produced faster than humans can read it, the controls that verify code — tests, reviews, evaluation, and CI pipelines — become the most important parts of the process. Chapters 4 and 15 cover these practices in depth.

Each of these eras survives in enterprises today. An FDE may work with a bank whose core-system changes follow a quarterly waterfall release with formal sign-offs, while the team next door deploys a web application thirty times a day. Neither is wrong for its context. The skill is recognizing which world you are in and designing your delivery to fit it.

2.4 Software Architecture: From Mainframes to Agents

Architecture is the set of decisions that are expensive to change: how a system is divided, how its parts communicate, where its data lives, and how it behaves when things fail. Like languages and processes, architectural styles arrived in waves, and each wave left systems behind.

Centralized	Mainframe runs everything; terminals display results	1960s–
Client-server	Desktop applications talk to a shared database server	1980s–
N-tier web	Browser → web server → application server → database	1990s–
SOA	Shared enterprise services over a service bus (SOAP, XML)	2000s–
Microservices	Small, independently deployed services; APIs and events	2010s–
Cloud-native	Containers, serverless functions, managed services, events	2010s–
AI and agentic	Models, retrieval, tools, and agents added to existing systems	2020s–

Figure 2.4 — Architectural waves. Read the dates as "since," not "until." Every one of these styles is still in production somewhere — often inside the same company.

In the **centralized** era, one mainframe ran all the logic and stored all the data; users worked on terminals that simply displayed results. **Client-server** systems moved the user interface and some logic onto personal computers, which talked to a shared database. The web produced **n-tier** architectures: a browser, a web server, an application server holding business logic, and a database, each in its own layer.

As companies accumulated many applications, they tried to share capabilities between them. **Service-oriented architecture** (SOA) in the 2000s exposed business functions as enterprise services, typically using SOAP and XML over a central enterprise service bus. SOA introduced valuable ideas — contracts, reuse, loose coupling — but often became heavy and centrally governed.

Microservices, popularized in the 2010s by companies operating at internet scale, divided applications into small services that could be developed, deployed, and scaled independently by separate teams, communicating through lightweight APIs and events. **Domain-driven design** supplied the method for deciding where service boundaries should go. Microservices made large organizations faster, but at a cost: a distributed system has many more ways to fail than a single program. Many companies have since moved back toward a **modular monolith** — one deployable application with strong internal boundaries — when they did not need independent scaling.

Cloud platforms added **serverless** and **event-driven** styles, in which small functions respond to events and managed services replace self-run infrastructure. And now **AI and agentic architectures** add models, retrieval systems, and agents that can call tools. These do not replace earlier architectures; they sit on top of them and depend on them. An agent that "updates an order" is only as reliable as the order service it calls — which, at Acme, means a messaging bridge to a mainframe.

Chapter 3 covers architecture and design in depth. The historical lesson is this: **every architectural style is a set of tradeoffs, not a level of maturity**. A monolith is not primitive and microservices are not advanced. The right architecture is the one whose tradeoffs fit the customer's problem, team, and constraints — and the one they can operate after you leave.

2.5 The Internet and APIs: From ARPANET to Tool Calling

The internet began as **ARPANET**, a research network funded by the United States Department of Defense that connected its first computers in 1969. Its key idea was *packet switching*: breaking messages into small packets that travel independently across a network and are reassembled at the destination, so that the network keeps working even if parts of it fail. On January 1, 1983, ARPANET switched to the **TCP/IP** protocols, which still carry nearly all internet traffic. The **Domain Name System** (DNS), introduced shortly after, mapped human-readable names to numeric addresses.

In 1989, Tim Berners-Lee at CERN proposed the **World Wide Web**: documents linked by URLs, retrieved using **HTTP**, and written in HTML. The web made the internet usable by everyone, and HTTP — with its methods such as GET and POST and its status codes such as 200 and 404 — became the universal language of networked software.

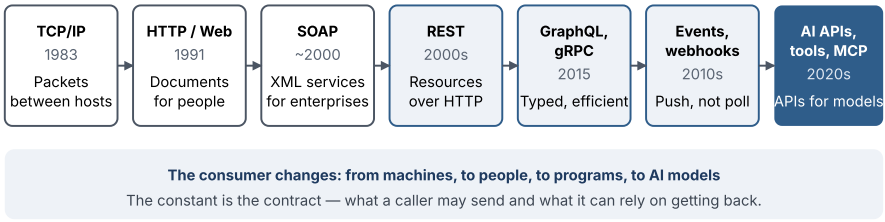


Figure 2.5 — The evolution of machine-to-machine communication. Each generation kept the earlier ones running. AI models are the newest kind of API consumer.

As businesses moved online, programs began talking to programs. **SOAP** web services, built on XML with formal contracts, became the enterprise standard around 2000 and still run inside many banks and insurers. **REST**, described by Roy Fielding in 2000 and widely adopted through the 2000s, used plain HTTP and URLs to represent resources, and became the default style for public APIs. Later, **GraphQL** let clients request exactly the data they need, and **gRPC** provided fast, strongly typed communication between internal services. **Webhooks**, **WebSockets**, and event streams added the ability to push information instead of repeatedly asking for it.

The newest consumer of APIs is the AI model itself. Model providers expose their models through APIs. Models, in turn, can now be given descriptions of other APIs — *tools* — and decide when to call them, a capability called **tool calling** or function calling. In late 2024, Anthropic introduced the **Model Context Protocol** (MCP), an open standard for connecting AI applications to tools and data sources, which was quickly adopted across the industry. MCP is the subject of Part VI in Volume 2.

Through all of these changes, one idea has remained central: **an API is a contract**. It defines what a caller may send, what it will get back, how errors are reported, and how the interface may change over time. Contracts are what allow systems built by different teams, in different decades and different languages, to work together. When an AI agent calls an API, it inherits every property of that contract — including its quirks. Chapter 5 covers APIs and integration in depth.

2.6 Infrastructure: From Data Centers to the Cloud and AI Compute

For most of computing history, running software meant owning hardware. Companies bought servers, installed them in **data centers** with power, cooling, and network connections, and installed an operating system and applications on each one. Capacity had to be planned months ahead, and a server bought for one application often sat mostly idle.

Virtualization changed the economics. A *hypervisor* lets one physical server run many isolated virtual machines, each with its own operating system. Companies such as VMware brought virtualization into enterprise data centers from the late 1990s onward, and utilization improved dramatically. Just as important, a virtual machine is software: it can be created, copied, and moved in minutes.

Amazon Web Services turned that idea into a public service. In 2006, it launched S3 for storage and EC2 for virtual servers, rented by the hour through an API. **Cloud computing** was born: infrastructure as a service (IaaS), then platform services (PaaS) that managed databases, queues, and runtimes, and later software as a service (SaaS) for complete applications. Microsoft Azure and Google Cloud followed. The cloud removed the months-long wait for hardware and replaced capital expense with operating expense — but also introduced new disciplines, from identity design to cost management, that Part III of this book covers in detail.

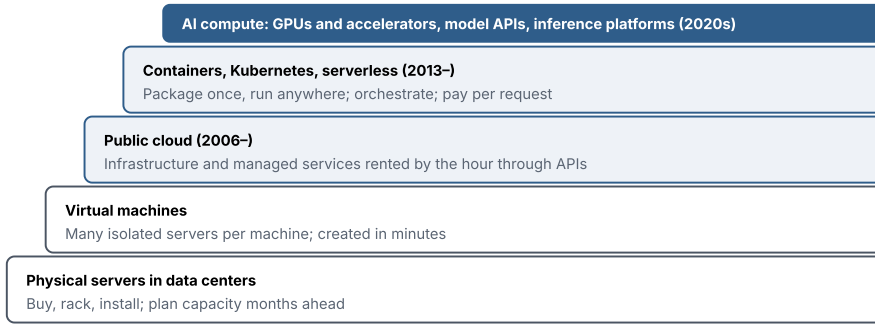


Figure 2.6 — Infrastructure layers. Each layer abstracts the one below it. The higher you build, the less you manage — and the more you depend on the provider's choices.

Two further abstractions followed. **Containers**, made popular by Docker in 2013, package an application with everything it needs to run, so that it behaves the same on a laptop, in a test environment, and in production.

Kubernetes, released by Google as open source in 2014, became the standard way to run large numbers of containers across clusters of machines. At about the same time, AWS Lambda introduced **serverless** functions that run only when called and are billed by the millisecond. Infrastructure itself became code, described in files and created by tools such as Terraform — the practice called **infrastructure as code**.

The newest infrastructure layer is **AI compute**. Training large models requires enormous clusters of GPUs and specialized accelerators; running them — called *inference* — requires fast hardware close to users. Most enterprises do not train large models. Instead, they consume models through APIs, run smaller models on their own infrastructure, or use managed AI platforms from the major clouds. Where the model runs, and where the data goes to reach it, has become one of the first questions in any enterprise AI security review.

2.7 The Rise of AI

Artificial intelligence has its own long history, which Chapter 19 tells in more detail. The term was coined for a workshop at Dartmouth College in 1956. Early AI focused on logic and hand-written rules; in the 1980s, rule-based *expert systems* enjoyed commercial success before proving brittle and expensive to maintain. **Machine learning** took a different approach: instead of writing rules, let the system learn patterns from examples. Through the 1990s and 2000s, machine learning quietly powered spam filters, recommendations, and fraud detection.

Deep learning — large neural networks trained on large datasets using GPUs — produced dramatic gains from 2012 onward in images, speech, and translation. In 2017, researchers at Google introduced the **Transformer** architecture, which proved remarkably effective at learning from text at scale. Transformers led to **large language models** (LLMs), and the public release of ChatGPT in November 2022 brought them to hundreds of millions of people within months.

Since then, the pace has been extraordinary. Models have become more capable, cheaper, and able to work with images, audio, and documents as well as text. They have gained the ability to call tools and act as **agents**, carrying out multi-step tasks. And enterprises have moved from experimentation toward production — which is exactly where the work of the Forward Deployed Engineer begins.

2.8 The Stack Is Cumulative

Put these histories together and a pattern emerges. Each generation of technology solved real problems and was layered on top of what already existed. Very little was ever fully retired. Figure 2.7 shows how this looks at Acme.

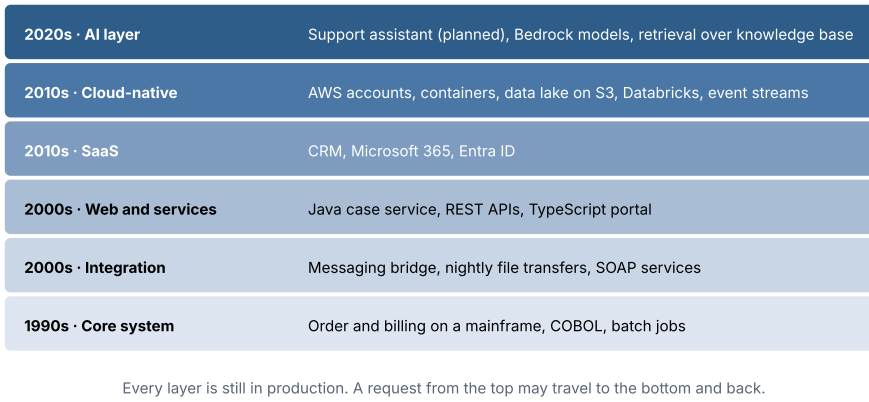


Figure 2.7 — Acme's technology strata. Like geological layers, each era of technology sits on the ones before it. The AI assistant at the top depends on every layer below.

For an FDE, this pattern has three practical consequences.

First, legacy is not a dirty word. The oldest systems in an enterprise are usually its most important, because they hold the records the business runs on. They have survived precisely because they work. Treat them with respect, learn their constraints, and connect to them through their established interfaces. Proposing to replace them in order to make an AI project easier is almost never the right first move.

Second, every new layer inherits the properties of the layers beneath it. If the order system updates only in a nightly batch, an AI assistant cannot report real-time order status, however capable its model. If the knowledge base contains outdated articles, retrieval will surface outdated answers. Many apparent AI problems are really data, integration, or process problems lower in the stack.

Third, people and processes come with each layer. The mainframe has a team with its own change windows. The data lake has owners with their own priorities. The security team has controls designed for each era.

Understanding which team owns which layer — and how each prefers to work — is as important as understanding the technology.

2.9 A Framework for Judging New Technology

Because new technologies arrive constantly, an FDE needs a quick, reliable way to evaluate them — both the ones customers are excited about and the ones vendors are selling. Five questions, drawn from the history in this chapter, serve well.

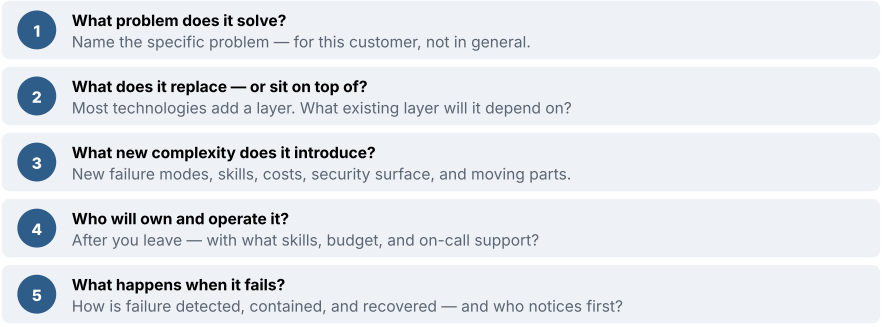


Figure 2.8 — Five questions for any new technology. If a proposal cannot answer all five, it is not ready for a customer's production environment.

Apply these questions to Acme's request for an AI agent that resolves tickets. It solves a real problem — slow, costly support — but only for some ticket types. It does not replace the case service, order system, or knowledge base; it depends on all three. It introduces new complexity: model behavior, evaluation, prompt injection, and token costs. It needs an owner on Acme's side who can monitor and adjust it. And when it fails — gives a wrong answer about a refund, for instance — someone must notice, correct it, and prevent recurrence. None of these answers makes the project a bad idea. Together, they turn it into a project that can succeed.

FIELD NOTES — YOUR TURN

Before you move on, write down a real moment from your own work about meeting "old" technology in a modern project. Try one of these:

- A mainframe, batch job, or file transfer that turned out to be the real constraint on a cloud or data project.
- A modernization effort that tried to replace a legacy system too early — and what worked instead.
- How understanding why a legacy design existed changed your approach.

Hands-On Exercise: Draw a Customer's Technology Strata

Choose a system you know well — at your current employer, a past one, or a well-documented open-source project. In about an hour:

1. **Trace one request end to end.** Pick a common user action (placing an order, opening a support case, running a report) and list every system it touches, in order.
2. **Date each layer.** For each system, note roughly when it was built, in which language, and on what infrastructure.
3. **Draw the strata.** Arrange the systems in layers by era, as in Figure 2.7.
4. **Mark the constraints.** For each layer, note one property an AI system built on top would inherit — such as data freshness, an interface limit, or a change window.
5. **Apply the five questions.** Imagine adding an AI assistant at the top. Answer the five questions in Figure 2.8.

Keep your diagram. You will extend it in later chapters as you learn more about each layer.

Checklist: Reading the Layers in a Customer Environment

☐ I have traced at least one important workflow through every system it touches.

☐ I know which systems are the authoritative sources for the data my solution needs.

- ☐ I know which layers are batch and which are real-time, and how fresh each data source is.
- ☐ I know where key business rules are implemented — and whether they are implemented more than once.
- ☐ I know which teams own each layer, and their change processes.
- ☐ I have identified the established interfaces for connecting to legacy systems.
- ☐ I have applied the five questions to every new technology in my proposal.

Chapter Summary

Computing evolved from mechanical designs and room-sized electronic machines to mainframes, personal computers, networks, virtualization, cloud, and today's GPU-powered AI. Programming languages evolved from machine code to COBOL, C, Java, JavaScript, TypeScript, and Python, and most enterprises now run several at once — which makes language choice an architectural decision and duplicated business rules a common source of error. Software engineering evolved from waterfall to agile, continuous delivery, DevOps, SRE, platform engineering, and now AI-assisted development, each step shortening the loop between change and evidence. Architecture moved from centralized systems through client-server, n-tier, SOA, microservices, and cloud-native styles to AI and agentic systems — each a set of tradeoffs rather than a level of maturity. Networks and APIs moved from TCP/IP and HTTP through SOAP, REST, GraphQL, gRPC, and events to APIs consumed by AI models, with the contract as the constant.

The key lesson is that enterprise technology is cumulative. New layers inherit the properties of the old ones, and the people and processes that come with them. An FDE who can read those layers, respect them, and evaluate new technology with five simple questions is equipped to connect the newest layer — AI — to everything beneath it.

LOOKING AHEAD

Part II turns from history to practice. Chapter 3 begins with the decisions that shape every system you will touch: how software is divided into parts, how those parts depend on each other, and how to design systems that can change safely — including the systems you add to a customer's environment.

// END OF THE FREE SAMPLE

Enjoyed it? Get the full book.

This sample includes the front matter and Chapters 1 and 2. The full book continues with every remaining chapter, the appendices and the index — 413 pages in all.

The Forward Deployed Engineer, Volume 1

PDF · 413 PAGES · INSTANT DOWNLOAD

$19

Buy at neerajbhatyal.com

neerajbhatyal.com/books/forward-deployed-engineer-vol-1.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 1 OF 2

Engineering that works where the customer works.

A Forward Deployed Engineer works inside a customer's world — their workflows, data, identity systems, networks, and clouds — and turns an ambiguous request into a working, adopted, production system. AI made the role essential, and it widened the gap between an impressive demo and a system a business can depend on.

Volume 1 builds the foundations the job demands: the software engineering you rely on in the field, practical coverage of AWS, Microsoft Azure, and Google Cloud, and the data platforms and AI foundations that modern intelligent systems are built on. Every chapter follows Priya, an FDE on the Acme engagement, with hands-on exercises, checklists, and Field Notes that connect the material to your own work.

INSIDE THIS VOLUME

PART I	The Role and Its Roots	Ch. 1–2
PART II	Engineering Fundamentals	Ch. 3–8
PART III	Cloud	Ch. 9–17
PART IV	Data and AI	Ch. 18–22



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

NEERAJBHATYAL.COM/BOOKS