

O'REILLY®

2nd Edition



Fundamentals of Software Architecture

A Modern Engineering Approach

Mark Richards & Neal Ford

"An indispensable resource for exploring modern software architecture through a contemporary lens. Whether you are an 'accidental' architect stepping into the role or a veteran seeking to refine your skills, this book offers the tools and knowledge you need to excel in your craft."

Raju Gandhi, author of *Head First Git* and coauthor of *Head First Software Architecture*

Fundamentals of Software Architecture

Developers who want to move beyond coding to advance their career aspire to become software architects, yet no real guide has existed to help them do so—until now. This updated edition provides a comprehensive overview of software architecture's many aspects, with several entirely new chapters covering the latest insights from the field. Existing and prospective architects alike will examine architectural characteristics, architectural patterns, component determination, diagramming architecture, governance, data, generative AI, team topologies, and many other topics.

Mark Richards and Neal Ford—hands-on practitioners who have taught software architecture classes professionally for years—focus on architecture principles that apply across all technology stacks. You'll explore software architecture in a modern light, taking into account all the innovations of the past decade.

This book examines:

- **Architecture styles and patterns:** Microservices, modular monoliths, microkernels, layered architectures, and many more
- **Components:** Identification, coupling, cohesion, partitioning, and granularity
- **Soft skills:** Effective team management, collaboration, business engagement models, negotiation, presentations, and more
- **Modern engineering practices:** Methods and operational approaches that have changed radically in the past few years, including cloud considerations and generative AI
- **Architecture as an engineering discipline:** Repeatable results, metrics, and concrete valuations that add rigor to software architecture

Mark Richards is an experienced hands-on software architect involved in the architecture, design, and implementation of microservices architectures and other distributed systems. He's the author of numerous O'Reilly technical books and videos.

Neal Ford is a director, software architect, and meme wrangler at Thoughtworks and an internationally recognized expert on software development and delivery. Neal's authored eight books (and counting), a number of magazine articles, and dozens of video presentations.

SOFTWARE ARCHITECTURE

US \$79.99 CAN \$99.99

ISBN: 978-1-098-17551-1



9 781098 175511

O'REILLY®

Praise for *Fundamentals of Software Architecture*

Mark and Neal have done it again—this revised and expanded second edition of their bestseller is an indispensable resource for exploring modern software architecture through a contemporary lens. With a nuanced understanding of what software architecture truly involves, this comprehensive guide starts with the critical importance of trade-off analysis, then delves into a wide range of architectural styles and the philosophies that underpin them, along with detailed examinations of data and team topologies. Whether you are an “accidental” architect stepping into the role or a seasoned veteran seeking to refine your skills, this book offers the tools and knowledge you need to excel in your craft.

—Raju Gandhi, Author of *Head First Git* and Coauthor of *Head First Software Architecture*

Neal and Mark aren’t just outstanding software architects; they are also exceptional teachers. With *Fundamentals of Software Architecture*, they have managed to condense the sprawling topic of architecture into a concise work that reflects their decades of experience. Whether you’re new to the role or you’ve been a practicing architect for many years, the updated edition of this book will help you be better at your job. I only wish they’d have written it earlier in my career. I’ve now used both editions with my architecture graduate students, and will continue to recommend it widely in this expanded form.

—Nathaniel Schutta,
Coauthor of *Fundamentals of Software Engineering*

Mark and Neal set out to achieve a formidable goal—to elucidate the many, layered fundamentals required to excel in software architecture—and they have once again completed their quest. The software architecture field continuously evolves, and the role requires a daunting breadth and depth of knowledge and skills. This updated book will serve as a guide for many as they navigate their journey to software architecture mastery.

*—Rebecca J. Parsons, Technology Advisor
and Former CTO/CTO Emerita, Thoughtworks*

Mark and Neal truly capture real-world advice for technologists to drive architecture excellence. They achieve this by identifying common architecture characteristics and the trade-offs that are necessary to drive success.

—Cassie Shum, Technical Director, Thoughtworks

SECOND EDITION

Fundamentals of Software Architecture

A Modern Engineering Approach

Mark Richards and Neal Ford

O'REILLY®

Fundamentals of Software Architecture

by Mark Richards and Neal Ford

Copyright © 2025 Mark Richards and Neal Ford. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Acquisitions Editor: Louise Corrigan

Indexer: WordCo Indexing Services, Inc.

Development Editor: Sarah Grey

Interior Designer: David Futato

Production Editor: Christopher Faucher

Cover Designer: Karen Montgomery

Copyeditor: Sonia Saruba

Illustrator: Kate Dullea

Proofreader: Piper Content Partners

| | |
|---------------|----------------|
| January 2020: | First Edition |
| March 2025: | Second Edition |

Revision History for the Second Edition

2025-03-12: First Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781098175511> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Fundamentals of Software Architecture*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the authors and do not represent the publisher's views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-17551-1

[LSI]

Table of Contents

| | |
|--------------------------------------|----------|
| Preface..... | xvii |
| 1. Introduction..... | 1 |
| Defining Software Architecture | 2 |
| Laws of Software Architecture | 6 |
| Expectations of an Architect | 8 |
| Make Architecture Decisions | 8 |
| Continually Analyze the Architecture | 9 |
| Keep Current with Latest Trends | 9 |
| Ensure Compliance with Decisions | 10 |
| Understand Diverse Technologies | 10 |
| Know the Business Domain | 11 |
| Possess Interpersonal Skills | 11 |
| Understand and Navigate Politics | 12 |
| Roadmap | 13 |

Part I. Foundations

| | |
|---------------------------------------|-----------|
| 2. Architectural Thinking..... | 17 |
| Architecture Versus Design | 17 |
| Strategic Versus Tactical Decisions | 18 |
| Level of Effort | 19 |
| The Significance of Trade-Offs | 19 |
| Technical Breadth | 20 |

| | |
|---|-----------|
| The 20-Minute Rule | 24 |
| Developing a Personal Radar | 25 |
| Analyzing Trade-Offs | 30 |
| Understanding Business Drivers | 33 |
| Balancing Architecture and Hands-On Coding | 33 |
| There's More to Architectural Thinking | 36 |
| 3. Modularity..... | 37 |
| Modularity Versus Granularity | 38 |
| Defining Modularity | 38 |
| Measuring Modularity | 40 |
| Cohesion | 41 |
| Coupling | 44 |
| Core Metrics | 45 |
| Distance from the Main Sequence | 46 |
| Connascence | 49 |
| From Modules to Components | 54 |
| 4. Architectural Characteristics Defined..... | 55 |
| Architectural Characteristics and System Design | 56 |
| Architectural Characteristics (Partially) Listed | 59 |
| Operational Architectural Characteristics | 59 |
| Structural Architectural Characteristics | 60 |
| Cloud Characteristics | 60 |
| Cross-Cutting Architectural Characteristics | 61 |
| Trade-Offs and Least Worst Architecture | 64 |
| 5. Identifying Architectural Characteristics..... | 67 |
| Extracting Architectural Characteristics from Domain Concerns | 67 |
| Composite Architectural Characteristics | 68 |
| Extracting Architectural Characteristics | 69 |
| Working with Katas | 70 |
| Kata: Silicon Sandwiches | 71 |
| Explicit Characteristics | 72 |
| Implicit Characteristics | 75 |
| Limiting and Prioritizing Architectural Characteristics | 77 |
| 6. Measuring and Governing Architecture Characteristics..... | 81 |
| Measuring Architecture Characteristics | 81 |

| | |
|---|------------|
| Operational Measures | 82 |
| Structural Measures | 83 |
| Process Measures | 86 |
| Governance and Fitness Functions | 86 |
| Governing Architecture Characteristics | 86 |
| Fitness Functions | 87 |
| 7. The Scope of Architectural Characteristics..... | 95 |
| Architectural Quanta and Granularity | 96 |
| Synchronous Communication | 99 |
| The Impact of Scoping | 100 |
| Scoping and Architectural Style | 102 |
| Kata: Going Green | 103 |
| Scoping and the Cloud | 105 |
| 8. Component-Based Thinking..... | 107 |
| Defining Logical Components | 107 |
| Logical Versus Physical Architecture | 110 |
| Creating a Logical Architecture | 112 |
| Identifying Core Components | 113 |
| Assigning User Stories to Components | 117 |
| Analyzing Roles and Responsibilities | 118 |
| Analyzing Architectural Characteristics | 120 |
| Restructuring Components | 120 |
| Component Coupling | 121 |
| Static Coupling | 121 |
| Temporal Coupling | 122 |
| The Law of Demeter | 123 |
| Case Study: Going, Going, Gone—Discovering Components | 125 |

Part II. Architecture Styles

| | |
|----------------------------|------------|
| 9. Foundations..... | 131 |
| Styles Versus Patterns | 131 |
| Fundamental Patterns | 133 |
| Big Ball of Mud | 133 |
| Unitary Architecture | 134 |
| Client/Server | 135 |

| | |
|---|------------|
| Architecture Partitioning | 137 |
| Kata: Silicon Sandwiches—Partitioning | 140 |
| Monolithic Versus Distributed Architectures | 143 |
| Fallacy #1: The Network Is Reliable | 144 |
| Fallacy #2: Latency Is Zero | 144 |
| Fallacy #3: Bandwidth Is Infinite | 145 |
| Fallacy #4: The Network Is Secure | 146 |
| Fallacy #5: The Topology Never Changes | 147 |
| Fallacy #6: There Is Only One Administrator | 148 |
| Fallacy #7: Transport Cost Is Zero | 148 |
| Fallacy #8: The Network Is Homogeneous | 149 |
| The Other Fallacies | 149 |
| Team Topologies and Architecture | 151 |
| On to Specific Styles | 152 |
| 10. Layered Architecture Style..... | 153 |
| Topology | 153 |
| Style Specifics | 155 |
| Layers of Isolation | 155 |
| Adding Layers | 157 |
| Data Topologies | 159 |
| Cloud Considerations | 159 |
| Common Risks | 159 |
| Governance | 159 |
| Team Topology Considerations | 160 |
| Style Characteristics | 161 |
| When to Use | 162 |
| When Not to Use | 163 |
| Examples and Use Cases | 163 |
| 11. The Modular Monolith Architecture Style..... | 165 |
| Topology | 165 |
| Style Specifics | 166 |
| Monolithic Structure | 167 |
| Modular Structure | 168 |
| Module Communication | 168 |
| Data Topologies | 170 |
| Cloud Considerations | 171 |
| Common Risks | 171 |

| | |
|--|------------|
| Governance | 172 |
| Team Topology Considerations | 174 |
| Style Characteristics | 175 |
| When to Use | 176 |
| When Not to Use | 177 |
| Examples and Use Cases | 177 |
| 12. Pipeline Architecture Style..... | 181 |
| Topology | 181 |
| Style Specifics | 182 |
| Filters | 182 |
| Pipes | 183 |
| Data Topologies | 184 |
| Cloud Considerations | 184 |
| Common Risks | 185 |
| Governance | 186 |
| Team Topology Considerations | 188 |
| Style Characteristics | 189 |
| When to Use | 190 |
| When Not to Use | 190 |
| Examples and Use Cases | 191 |
| 13. Microkernel Architecture Style..... | 193 |
| Topology | 193 |
| Style Specifics | 194 |
| Core System | 194 |
| Plug-In Components | 198 |
| The Spectrum of “Microkern-ality” | 200 |
| Registry | 201 |
| Contracts | 201 |
| Data Topologies | 202 |
| Cloud Considerations | 203 |
| Common Risks | 203 |
| Volatile Core | 203 |
| Plug-In Dependencies | 204 |
| Governance | 204 |
| Team Topology Considerations | 204 |
| Architecture Characteristics Ratings | 205 |
| Examples and Use Cases | 207 |

| | |
|---|------------|
| 14. Service-Based Architecture Style..... | 209 |
| Topology | 209 |
| Style Specifics | 211 |
| Service Design and Granularity | 213 |
| User Interface Options | 213 |
| API Gateway Options | 215 |
| Data Topologies | 215 |
| Cloud Considerations | 219 |
| Common Risks | 219 |
| Governance | 219 |
| Team Topology Considerations | 220 |
| Style Characteristics | 221 |
| Examples and Use Cases | 224 |
| 15. Event-Driven Architecture Style..... | 227 |
| Topology | 228 |
| Style Specifics | 232 |
| Events Versus Messages | 232 |
| Derived Events | 233 |
| Triggering Extensible Events | 235 |
| Asynchronous Capabilities | 235 |
| Broadcast Capabilities | 239 |
| Event Payload | 240 |
| The Swarm of Gnats Antipattern | 246 |
| Error Handling | 249 |
| Preventing Data Loss | 253 |
| Request-Reply Processing | 256 |
| Mediated Event-Driven Architecture | 258 |
| Data Topologies | 268 |
| Monolithic Database Topology | 269 |
| Domain Database Topology | 271 |
| Dedicated Data Topology | 273 |
| Cloud Considerations | 275 |
| Common Risks | 275 |
| Governance | 276 |
| Team Topology Considerations | 276 |
| Style Characteristics | 277 |
| Choosing Between Request-Based and Event-Based Models | 280 |
| Examples and Use Cases | 280 |

| | |
|--|------------|
| 16. Space-Based Architecture Style..... | 283 |
| Topology | 284 |
| Style Specifics | 286 |
| Processing Unit | 286 |
| Virtualized Middleware | 287 |
| Messaging Grid | 287 |
| Data Grid | 288 |
| Processing Grid | 296 |
| Deployment Manager | 297 |
| Data Pumps | 297 |
| Data Writers | 298 |
| Data Readers | 300 |
| Data Topologies | 302 |
| Cloud Considerations | 302 |
| Common Risks | 303 |
| Frequent Reads from the Database | 303 |
| Data Synchronization and Consistency | 304 |
| High Data Volumes | 304 |
| Data Collisions | 304 |
| Governance | 307 |
| Team Topology Considerations | 310 |
| Style Characteristics | 311 |
| Examples and Use Cases | 313 |
| Concert Ticketing System | 313 |
| Online Auction System | 313 |
| 17. Orchestration-Driven Service-Oriented Architecture..... | 315 |
| Topology | 315 |
| Style Specifics | 316 |
| Taxonomy | 317 |
| Reuse...and Coupling | 320 |
| Data Topologies | 322 |
| Cloud Considerations | 323 |
| Common Risks | 323 |
| Governance | 324 |
| Team Topology Considerations | 325 |
| Style Characteristics | 325 |
| Examples and Use Cases | 327 |

| | |
|---|------------|
| 18. Microservices Architecture..... | 329 |
| Topology | 330 |
| Style Specifics | 331 |
| Bounded Context | 331 |
| Granularity | 332 |
| Data Isolation | 333 |
| API Layer | 334 |
| Operational Reuse | 334 |
| Frontends | 338 |
| Communication | 339 |
| Choreography and Orchestration | 341 |
| Transactions and Sagas | 345 |
| Data Topologies | 348 |
| Cloud Considerations | 351 |
| Common Risks | 351 |
| Governance | 352 |
| Team Topology Considerations | 353 |
| Style Characteristics | 354 |
| Examples and Use Cases | 356 |
| 19. Choosing the Appropriate Architecture Style..... | 359 |
| Shifting “Fashion” in Architecture | 359 |
| Decision Criteria | 361 |
| Monolith Case Study: Silicon Sandwiches | 364 |
| Modular Monolith | 365 |
| Microkernel | 366 |
| Distributed Case Study: Going, Going, Gone | 367 |
| 20. Architectural Patterns..... | 371 |
| Reuse | 372 |
| Separating Domain and Operational Coupling | 372 |
| Communication | 375 |
| Orchestration Versus Choreography | 375 |
| CQRS | 378 |
| Infrastructure | 379 |
| Broker-Domain Pattern | 380 |

Part III. Techniques and Soft Skills

| | |
|--|------------|
| 21. Architectural Decisions..... | 387 |
| Architectural Decision Antipatterns | 387 |
| The Covering Your Assets Antipattern | 387 |
| Groundhog Day Antipattern | 389 |
| Email-Driven Architecture Antipattern | 389 |
| Architectural Significance | 390 |
| Architectural Decision Records | 391 |
| Basic Structure | 392 |
| Example | 398 |
| Storing ADRs | 400 |
| ADRs as Documentation | 401 |
| Using ADRs for Standards | 402 |
| Using ADRs with Existing Systems | 402 |
| Leveraging Generative AI and LLMs in Architectural Decisions | 402 |
| 22. Analyzing Architecture Risk..... | 405 |
| Risk Matrix | 405 |
| Risk Assessments | 406 |
| Risk Storming | 410 |
| Phase 1: Identification | 411 |
| Phase 2: Consensus | 412 |
| Phase 3: Risk Mitigation | 415 |
| User-Story Risk Analysis | 416 |
| Risk-Storming Use Case | 416 |
| Availability | 418 |
| Elasticity | 420 |
| Security | 421 |
| Summary | 422 |
| 23. Diagramming Architecture..... | 423 |
| Diagramming | 424 |
| Tools | 424 |
| Diagramming Standards: UML, C4, and ArchiMate | 426 |
| Diagram Guidelines | 428 |
| Summary | 431 |

| | |
|---|------------|
| 24. Making Teams Effective..... | 433 |
| Collaboration | 433 |
| Constraints and Boundaries | 435 |
| Architect Personalities | 436 |
| The Control-Freak Architect | 437 |
| The Armchair Architect | 437 |
| The Effective Architect | 438 |
| How Much Involvement? | 439 |
| Team Warning Signs | 443 |
| Process Loss | 443 |
| Pluralistic Ignorance | 444 |
| Leveraging Checklists | 445 |
| Developer Code-Completion Checklist | 447 |
| Unit and Functional Testing Checklist | 448 |
| Software-Release Checklist | 449 |
| Providing Guidance | 450 |
| Summary | 452 |
| 25. Negotiation and Leadership Skills..... | 453 |
| Negotiation and Facilitation | 453 |
| Negotiating with Business Stakeholders | 453 |
| Negotiating with Other Architects | 456 |
| Negotiating with Developers | 457 |
| The Software Architect as a Leader | 459 |
| The 4 Cs of Architecture | 459 |
| Be Pragmatic, Yet Visionary | 461 |
| Leading Teams by Example | 462 |
| Integrating with the Development Team | 465 |
| Summary | 468 |
| 26. Architectural Intersections..... | 469 |
| Architecture and Implementation | 470 |
| Operational Concerns | 471 |
| Structural Integrity | 472 |
| Architectural Constraints | 474 |
| Architecture and Infrastructure | 475 |
| Architecture and Data Topologies | 477 |
| Database Topology | 478 |
| Architectural Characteristics | 479 |

| | |
|---|------------|
| Data Structure | 479 |
| Read/Write Priority | 479 |
| Architecture and Engineering Practices | 480 |
| Architecture and Team Topologies | 481 |
| Architecture and Systems Integration | 482 |
| Architecture and the Enterprise | 482 |
| Architecture and the Business Environment | 483 |
| Architecture and Generative AI | 484 |
| Incorporating Generative AI into Architecture | 484 |
| Generative AI as an Architect Assistant | 484 |
| Summary | 485 |
| 27. The Laws of Software Architecture, Revisited. | 487 |
| First Law: Everything in Software Architecture Is a Trade-Off | 487 |
| Shared Library Versus Shared Service | 488 |
| Synchronous Versus Asynchronous Messaging | 490 |
| First Corollary: Missing Trade-Offs | 492 |
| Second Corollary: You Can't Do It Just Once | 494 |
| Second Law: Why Is More Important Than How | 494 |
| Out of Context Antipattern | 494 |
| The Spectrum Between Extremes | 495 |
| Parting Words of Advice | 496 |
| Appendix. Discussion Questions. | 497 |
| Index. | 507 |

Preface

Preface to the Second Edition

“Wow, there’s a lot there!”

When we set out to write the second edition of *Fundamentals of Software Architecture*, we had a few ideas of things we wanted to flesh out and improve from the first edition, but like a lot of software projects, it kept growing.

One of our met goals was to make the styles sections more consistent, making them more useful for comparisons. We also made some changes to our star ratings to add sections and a few new categories, and added new sections on cloud considerations, data topologies, team topologies, and governance for each architectural style. Along the way we made major additions to a number of chapters on popular topics, such as Chapters 15 and 18, and added a new chapter ([Chapter 11](#)) on the modular monolith architectural style.

We also added several entirely new chapters covering architectural patterns in [Chapter 20](#), the intersections of architecture in [Chapter 26](#), and revisiting our laws of software architecture (of which there is a new corollary and a new law) in [Chapter 27](#).

Preface to the First Edition

Axiom

A statement or proposition that is regarded as being established, accepted, or self-evidently true.

Mathematicians create theories based on axioms—assumptions for things indisputably true. Software architects also build theories atop axioms, but the software world is, well, *softer* than mathematics: fundamental things continue to change at a rapid pace, including the axioms we base our theories upon.

The software development ecosystem exists in a constant state of dynamic equilibrium: while it exists in a balanced state at any given point in time, it exhibits *dynamic* behavior over the long term. A great modern example of the nature of this ecosystem follows the ascension of containerization and the attendant changes: tools like [Kubernetes](#) didn't exist a decade ago, yet now entire software conferences exist to service its users. The software ecosystem changes chaotically: one small change causes another small change; when repeated hundreds of times, it generates a new ecosystem.

Architects have an important responsibility to question assumptions and axioms left over from previous eras. Many of the books about software architecture were written in an era that only barely resembles the current world. In fact, the authors believe that we must question fundamental axioms on a regular basis, in light of improved engineering practices, operational ecosystems, software development processes—everything that makes up the messy, dynamic equilibrium where architects and developers work each day.

Careful observers of software architecture over time witnessed an evolution of capabilities. Starting with the engineering practices of [Extreme Programming](#), continuing with continuous delivery, the DevOps revolution, microservices, containerization, and now cloud-based resources, all of these innovations led to new capabilities and trade-offs. As capabilities changed, so did architects' perspectives on the industry. For many years, the tongue-in-cheek definition of software architecture was “the stuff that's hard to change later.” Later, the microservices architecture style appeared, where *change* is a first-class design consideration.

Each new era requires new practices, tools, measurements, patterns, and a host of other changes. This book looks at software architecture in a modern light, taking into account all the innovations from the last decade, along with some new metrics and measures suited to today's new structures and perspectives.

The subtitle of our book is “A Modern Engineering Approach.” Developers have long wished to change software development from a *craft*, where skilled artisans can create one-off works, to an *engineering* discipline, which implies repeatability, rigor, and effective analysis. While software engineering still lags behind other types of engineering disciplines by many orders of magnitude (to be fair, software is a very young discipline compared to most other types of engineering), architects have made huge improvements, which we'll discuss. In particular, modern Agile engineering practices have allowed great strides in the types of systems that architects design.

We also address the critically important issue of *trade-off analysis*. As a software developer, it's easy to become enamored with a particular technology or approach. But architects must always soberly assess the good, bad, and ugly of every choice, and virtually nothing in the real world offers convenient binary choices—everything is a trade-off. Given this pragmatic perspective, we strive to eliminate value judgments

about technology and instead focus on analyzing trade-offs to equip our readers with an analytic eye toward technology choices.

This book won't make someone a software architect overnight—it's a nuanced field with many facets. We want to provide existing and burgeoning architects a good modern overview of software architecture and its many aspects, from structure to soft skills. While this book covers well-known patterns, we take a new approach, leaning on lessons learned, tools, engineering practices, and other input. We take many existing axioms in software architecture and rethink them in light of the current ecosystem, and design architectures, taking the modern landscape into account.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

Constant width

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.



This element signifies a tip or suggestion.



This element signifies a general note.



This element indicates a warning or caution.

Supplemental Material

Visit <http://fundamentalsofsoftwarearchitecture.com> to access the supplemental resources for this book.

If you have a technical question or a problem using the code examples, please send email to bookquestions@oreilly.com.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: “*Fundamentals of Software Architecture*, Second Edition, by Mark Richards and Neal Ford (O'Reilly). Copyright 2025 Mark Richards and Neal Ford, 978-1-098-17551-1.”

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

O'Reilly Online Learning



For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, please visit <http://oreilly.com>.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-889-8969 (in the United States or Canada)
707-827-7019 (international or local)
707-829-0104 (fax)
support@oreilly.com
<https://oreilly.com/about/contact.html>

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at <https://oreil.ly/fundamentals-of-software-architecture-2e>.

For news and information about our books and courses, visit <https://oreilly.com>.

Find us on LinkedIn: <https://linkedin.com/company/oreilly-media>.

Watch us on YouTube: <https://youtube.com/oreillymedia>.

Acknowledgments

Mark and Neal would like to thank all the people who attended our classes, workshops, conference sessions, and user group meetings, as well as all the people who listened to versions of this material and provided invaluable feedback. We would also like to thank the publishing team at O'Reilly, who made this as painless an experience as writing a book can be. In particular, we would like to thank our first edition editors, Alicia Young and Virginia Wilson, and our second edition editor, Sarah Grey.

Acknowledgments from Mark Richards

In addition to the preceding acknowledgments, I would like to thank my lovely wife, Rebecca. Taking everything else on at home and sacrificing the opportunity to work on your own book allowed me to do additional consulting gigs and speak at more conferences and training classes, giving me the opportunity to practice and hone the material for this book. You are the best.

Acknowledgments from Neal Ford

I would like to thank my extended family, Thoughtworks as a collective, and Rebecca Parsons and Martin Fowler as individual parts of it. Thoughtworks is an extraordinary group of people who manage to produce value for customers while keeping a keen eye toward why things work so we can improve them. Thoughtworks supported this book in many ways and continues to grow Thoughtworkers who challenge and inspire every day. I would also like to thank our neighborhood cocktail club for a regular escape from routine. Lastly, I would like to thank my wife, Candy, whose tolerance for things like book writing and conference speaking apparently knows no bounds. For decades she's kept me grounded and sane enough to function, and I hope she will for decades more as the love of my life.

CHAPTER 1

Introduction

So you're interested in software architecture. Perhaps you're a developer who wants to move to the next career step, or perhaps you are a project manager who wants to understand what happens when software architectures work. You may also be an "accidental architect": someone who makes architecture decisions (defined below) but doesn't have the title of "software architect"...yet.

Why delve into the realm of software architecture? Perhaps you have experience with lots of projects and want to understand more deeply how the larger parts of systems fit together, along with the numerous trade-offs. If so, software architecture is an obvious next career step.

This book is designed for all of you. It provides an overview of the extremely multifaceted job of "software architect."

Software architects must understand and analyze software systems deeply, in all their complexity, and must make important trade-off decisions, sometimes with incomplete information. Many software developers worried that generative AI might slowly replace them are considering moving to software architecture, a role much harder to replace. Software architects make exactly the kinds of decisions that AI cannot, evaluating trade-offs within complex, changing contexts.

Architecture, like much art, can only be understood in context. Architects base their decisions on the realities of their environment. For example, one of the major goals of late-20th-century software architecture was to use shared infrastructure and resources as efficiently as possible, because operating systems, application servers, database servers, and so on were all commercial and very expensive.

In 2002, trying to build an architecture like microservices would have been inconceivably expensive. Imagine strolling into a 2002 data center and telling the head of operations, "Hey, I have a great idea for a revolutionary style of architecture, where

each service runs on its own isolated machinery with its own dedicated database. I'll need 50 Windows licenses, another 30 application-server licenses, and at least 50 database server licenses." We can only build such architectures today because of the advent of open source and the updated engineering practices of the DevOps revolution. All architectures are products of their context—keep that in mind as you read this book.

Defining Software Architecture

So what is software architecture? [Figure 1-1](#) illustrates how we like to think about software architecture. This definition has four dimensions. The software architecture of a system consists of an *architecture style* as the starting point, combined with the *architecture characteristics* it must support, the *logical components* to implement its behavior, and finally the *architecture decisions* justifying it all. The system's structure is denoted by the heavy black lines supporting the architecture. We'll walk quickly through these dimensions in the order in which architects analyze them, and subsequent chapters will provide more details.

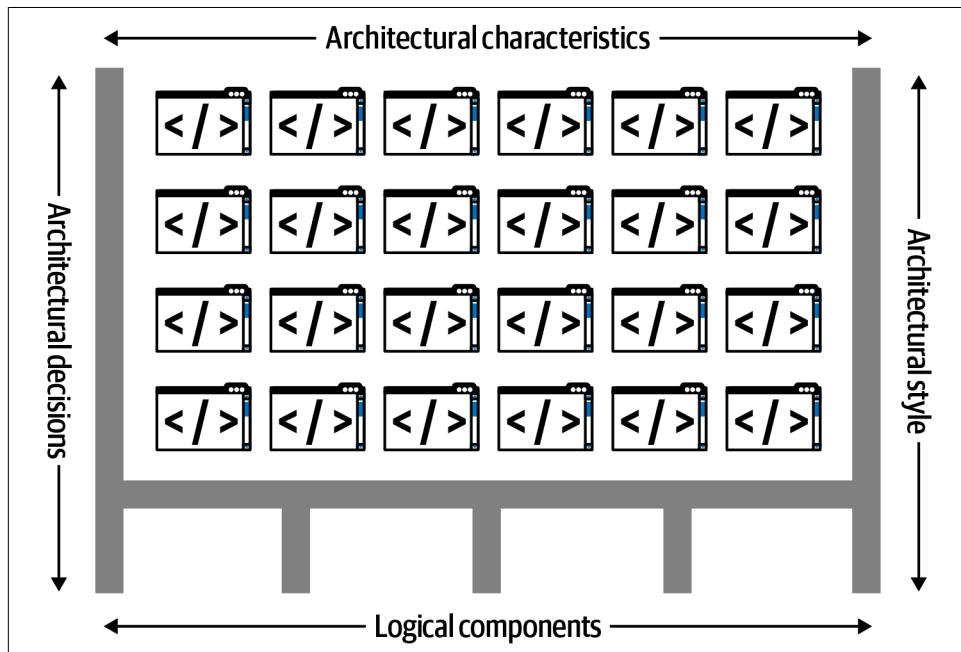


Figure 1-1. Architecture consists of the system's structure, combined with architecture characteristics ("-ilities"), logical components, architecture styles, and decisions

Architecture characteristics (see [Figure 1-2](#)) define the *capabilities* of a system (commonly abbreviated as “-ilities”) and the criteria for its success: in short, what the system should *do*. Architecture characteristics are so important that we’ve devoted several chapters in this book to understanding and defining them.

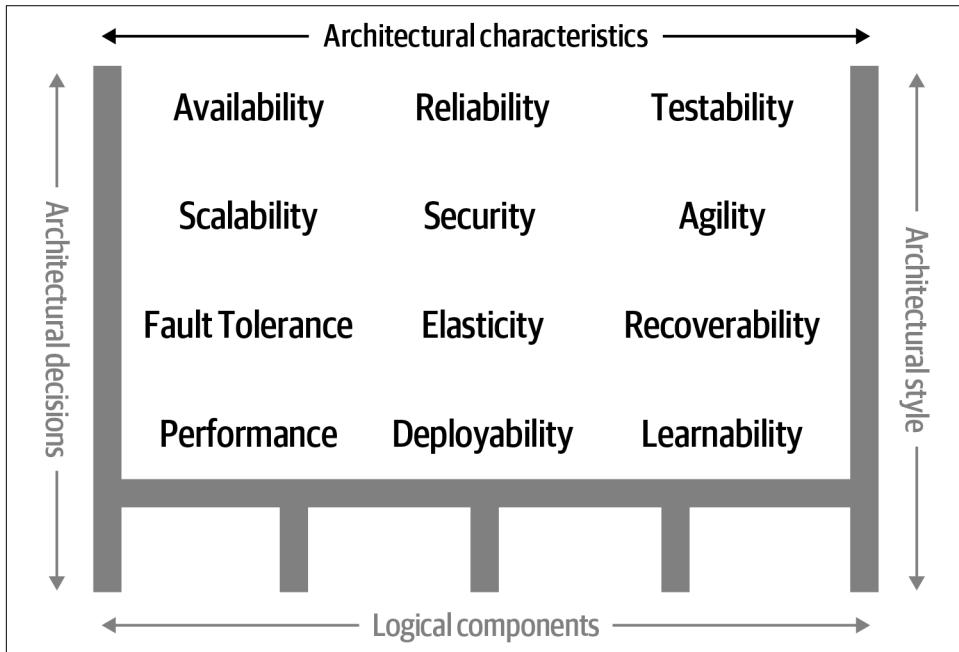


Figure 1-2. “Architecture characteristics” refer to the “-ilities” that the system must support

While architectural characteristics define a system's capabilities, *logical components* define its *behavior*. Designing logical components is one of the key structural activities for architects. In [Figure 1-3](#), the logical components form the domains, entities, and workflows of the application.

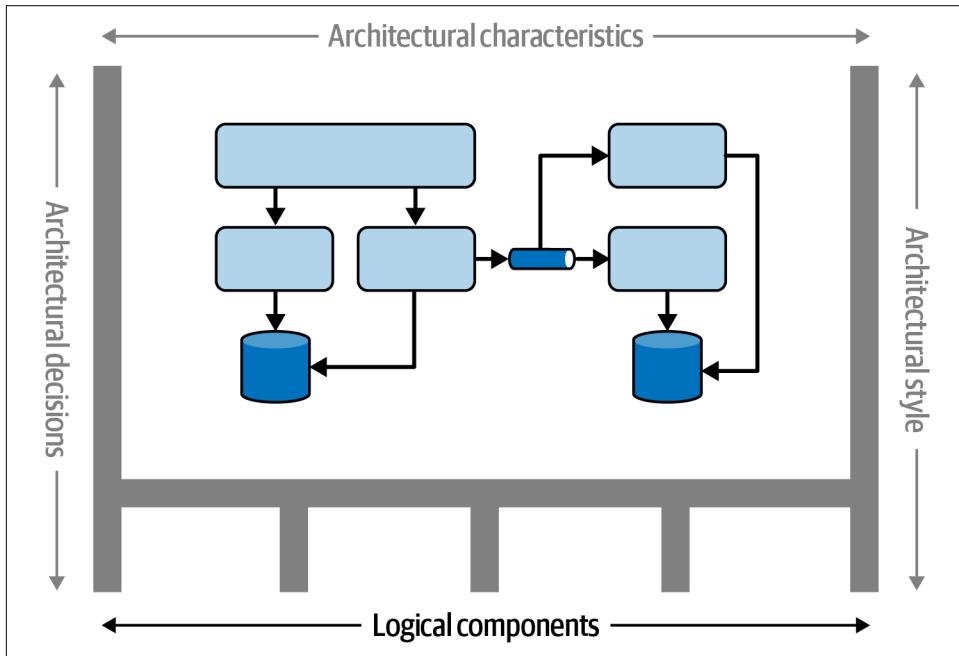


Figure 1-3. Logical components structure the behavior of the system

Once an architect has analyzed the architectural characteristics and logical components the system needs (both described in detail later), they know enough to choose an appropriate architecture style as a starting point for implementing their solution [Figure 1-4](#).

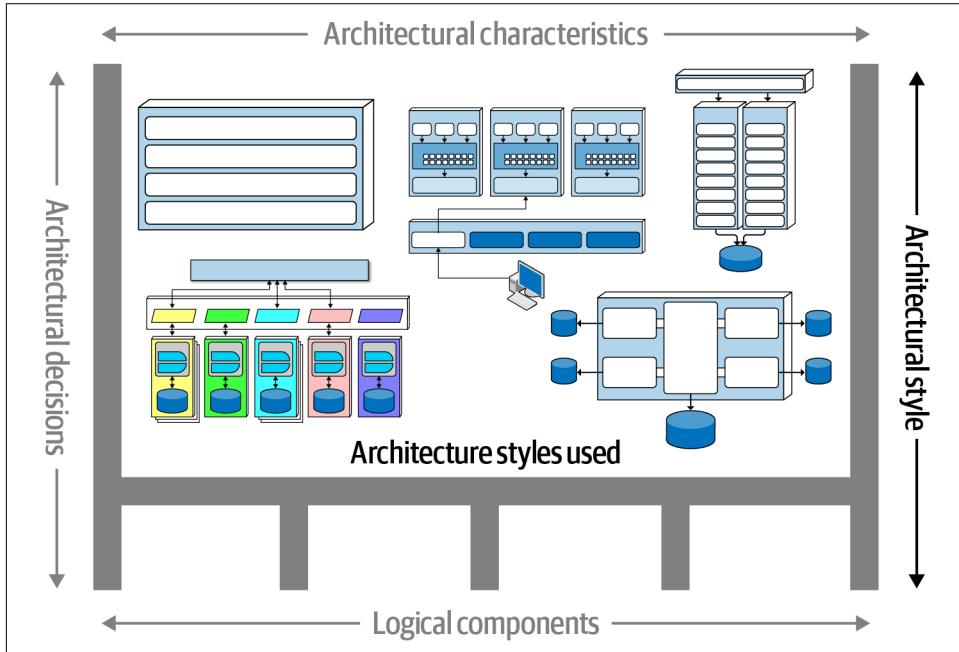


Figure 1-4. Choosing an architectural style involves finding the easiest implementation path for a given set of requirements

The fourth dimension that defines software architecture is *architecture decisions*, which define the rules for how a system should be constructed. For example, an architect might make a decision that only the Business and Services layers within a layered architecture can access the database (see [Figure 1-5](#)), restricting the Presentation layer from making direct database calls. Architecture decisions form the constraints of the system and direct the development teams on what is and what isn't allowed.

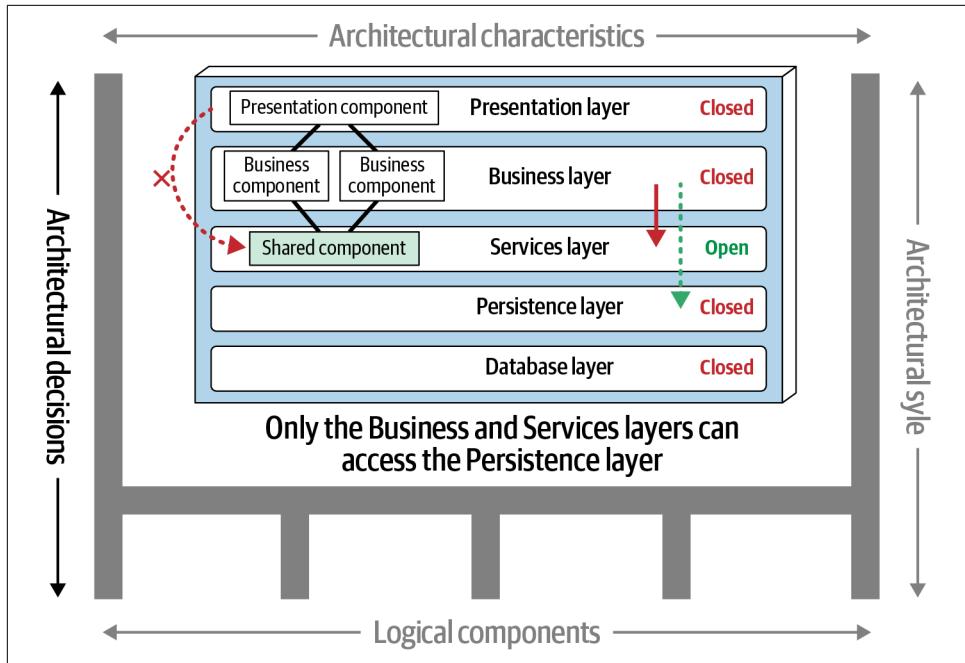


Figure 1-5. Architecture decisions are rules for constructing systems

We discuss architecture decisions and how to document them concisely in [Chapter 21](#).

Laws of Software Architecture

As your two authors set out to write the first edition of this book, we had an ambitious goal: we hoped to find things that seemed universally true about software architecture and codify them as “laws” of software architecture. As we wrote, we kept our eyes peeled for things we could capture; we had hoped to find maybe 10 or 15. To our surprise, we ended up identifying just two laws in the first edition, and uncovered one more while writing the second edition. True to our original intent, these three laws seem pretty universal and inform many important perspectives for working software architects.

We learned the First Law of Software Architecture by constantly stumbling across it, and we think it gets to the heart of why these universal truths seem so elusive:

Everything in software architecture is a trade-off.

—First Law of Software Architecture

Nothing exists on a nice, clean spectrum. Every decision a software architect makes must account for a large number of variables that take on different values depending on the circumstances. Trade-offs are the essence of software architecture decisions.

If you think you've discovered something that *isn't* a trade-off, more likely you just haven't *identified* the trade-off...yet.

—Corollary 1

You can't just do trade-off analysis once and be done with it.

—Corollary 2

Teams love standards, and it would be nice if architects could just do One Big Trade-off Jamboree to decide on our defaults for which architecture styles to use, how parts of the architecture should communicate, how to manage shared functionality, and a host of other sticky decisions. But we can't, because every situation requires us to re-evaluate all those trade-offs. (We've seen teams try to do this—for example, attempting to default to using only choreography in distributed workflows, only to discover that it works sometimes and is a spectacular disaster at other times. See “[Choreography and Orchestration](#)” on page 341 for our discussion.)

Architecture is broader than just the combination of structural elements, which is why our dimensional definition incorporates principles, characteristics, and so on. This is reflected in our Second Law of Software Architecture:

Why is more important than *how*.

—Second Law of Software Architecture

As an experienced architect, if someone shows me an architecture I've never seen before, I can understand *how* it works, but I might struggle with *why* the previous architect or team made certain decisions. Architects make decisions within extremely specific contexts, making sweeping and generic decisions difficult. *Why* an architect made a particular decision includes the trade-offs they considered, adding to the context of why this decision versus another. It turns out that one of the defining characteristics of software architecture decisions is they are rarely binary. This brings us to the Third Law of Software Architecture:

Most architecture decisions aren't binary but rather exist on a spectrum between extremes.

—Third Law of Software Architecture

Throughout the book, we highlight *why* architects make certain decisions along with trade-offs. We also highlight good techniques for capturing important decisions in [Chapter 21](#).

Readers will recognize these laws at work throughout the book, and we recommend keeping them in mind when evaluating software architecture decisions. We come back to these laws in [Chapter 27](#) with some additional examples.

With a working definition of software architecture in place, let's move into the role itself.

Expectations of an Architect

The role of a software architect can range from acting as an expert programmer to defining an entire company's strategic technical direction. This makes trying to define it a fool's errand, but what we can tell you more readily is what's *expected* of a software architect. We've identified eight core expectations for any software architect, irrespective of their role, title, or job description:

- Make architecture decisions
- Continually analyze the architecture
- Keep current with latest trends
- Ensure compliance with decisions
- Understand diverse technologies, frameworks, platforms, and environments
- Know the business domain
- Lead a team and possess interpersonal skills
- Understand and navigate organizational politics

Succeeding as a software architect depends on understanding and living up to each of these expectations. This section looks at all eight in turn.

Make Architecture Decisions

An architect is expected to define the architecture decisions and design principles used to guide technology decisions within the team, within the department, or across the enterprise.

Guide is the key word in this first expectation: architects should *guide* technology choices rather than *specify* them. For example, deciding to use React.js for frontend development is a technical decision rather than an architectural decision. Instead of making this decision, the architect should instruct the development teams to use a reactive-based framework for frontend web development, guiding them to choose Angular, Elm, React.js, Vue, or any of the other reactive-based web frameworks. The key is asking whether the architecture decision will *guide* teams in making the right technical choices or make the choice for them. That said, there are some occasions where architects need to make specific technology decisions in order to preserve a

particular architectural characteristic, such as scalability, performance, or availability. Architects often struggle with finding this balance, so [Chapter 21](#) is entirely about architecture decisions.

Continually Analyze the Architecture

An architect is expected to continually analyze the architecture and the current technology environment and recommend solutions for improvement.

Architecture vitality assesses how viable an architecture that was defined three or more years ago is *today*, given changes in both business and technology. In our experience, not enough architects focus their energies on continually analyzing existing architectures. As a result, most architectures experience elements of structural decay, which occurs when developers make coding or design changes that impact the required architectural characteristics, such as performance, availability, and scalability.

Other frequently forgotten aspects of this expectation are testing and release environments. Being able to modify code quickly is a kind of agility with obvious benefits, but if it takes weeks to test the changes and months to release, the overall architecture cannot achieve agility.

Architects must holistically analyze changes in the technology and the problem domain to determine the ongoing soundness of the architecture. This kind of consideration is what keeps applications relevant, even if it rarely appears in job postings.

Keep Current with Latest Trends

An architect is expected to keep current with the latest technology and industry trends.

Developers must keep up to date on the latest technologies, as well as those they use on a daily basis, to remain relevant (and to retain a job!). It's even more critical for architects to keep current on the latest technical and industry trends. Architects' decisions tend to be long-lasting and difficult to change. Understanding trends helps architects make decisions that will remain relevant into the future. For example, architects in recent years have needed to learn about cloud-based storage and deployment, and as we write this second edition, generative AI is having a massive impact on many parts of the development ecosystem.

Tracking trends and keeping current with those trends is hard, particularly for a software architect. In [Chapter 2](#), we discuss some techniques and resources for how to do this.

Ensure Compliance with Decisions

An architect is expected to ensure compliance with architecture decisions and design principles.

Ensuring compliance means continually verifying that the development teams are following the decisions and design principles defined, documented, and communicated by the architect.

Imagine that you, as an architect, make a decision to restrict access to the database in a layered architecture to only the Business and Services layers (not the Presentation layer). This (as you'll see in [Chapter 10](#)) means that the Presentation layer must go through all layers of the architecture to make even the simplest of database calls. However, a user interface (UI) developer disagrees with this decision and gives the Presentation layer direct access to the database for performance reasons. You made that architecture decision for a specific reason: so that changes to the database changes would not affect the Presentation layer. If you aren't ensuring compliance with your architecture decisions, violations like this can occur. The result can be that the architecture fails to provide the required characteristics, and the application or system does not work as expected. In [Chapter 6](#), we talk about measuring compliance using automated fitness functions and other tools.

Understand Diverse Technologies

An architect is expected to have exposure to multiple and diverse technologies, frameworks, platforms, and environments.

Every architect isn't expected to be an expert in every framework, platform, and language, but they should at least be familiar with a variety of technologies. Most environments these days are heterogeneous, so at a minimum, architects should know how to interface with multiple systems and services, whatever their language, platform, or technology.

One of the best ways to meet this expectation is to stretch outside your comfort zone, aggressively seeking out opportunities to gain experience in multiple languages, platforms, and technologies. Architects should focus on technical *breadth* rather than technical depth. Technical breadth includes the stuff you know about, but not at a detailed level, combined with the stuff you know a lot about. For example, it's far more valuable for an architect to be familiar with the pros and cons of 10 different caching products than to be an expert in only one of them.

Know the Business Domain

An architect is expected to have a certain level of business-domain expertise.

Effective software architects understand the business problem, goals, and requirements the architecture is intended to solve, collectively known as the *business domain* of a problem space. It's difficult to design an effective architecture if you don't understand the requirements of the business. Imagine being an architect at a major bank and not understanding common financial terms like *average directional index*, *aleatory contracts*, *rates rally*, or even *nonpriority debt*. You wouldn't be able to communicate with stakeholders and business users and would quickly lose credibility.

The most successful architects we know have broad, hands-on technical knowledge and a strong knowledge of a particular domain. They can communicate with C-level executives and business users in language these stakeholders know and understand, creating confidence that they know what they're doing and are competent to create an effective and correct architecture.

Possess Interpersonal Skills

An architect is expected to possess exceptional interpersonal skills, including teamwork, facilitation, and leadership.

As technologists, developers and architects tend to prefer solving technical problems, not people problems, so exceptional leadership and interpersonal skills are a difficult expectation. However, as [Gerald Weinberg](#) was famous for saying, "No matter what they tell you, it's always a people problem." The guidance that architects provide is not only technical—it includes leading the development teams through implementing the architecture. Leadership skills are *at least half* of what it takes to become an effective software architect, regardless of role or title.

The industry is flooded with software architects, all competing for a limited number of positions. Those with strong leadership and interpersonal skills stand out from the crowd. Conversely, we've known many software architects who are excellent technologists but struggle to lead teams, coach and mentor developers, or communicate ideas and architecture decisions and principles. Needless to say, those architects have difficulties holding down a job.

Understand and Navigate Politics

An architect is expected to understand the political climate of the enterprise and be able to navigate its politics.

It might seem strange to talk about office politics in a book about software architecture, but negotiation skills are crucial to the role. To illustrate, consider two scenarios.

In the first scenario, a developer decides to leverage a particular design pattern to reduce the complexity of a particular piece of convoluted code. That's a fine decision, and the developer does not need to seek approval for it. Programming aspects such as code structure, class design, design-pattern selection, and sometimes even language choice are all part of the art of programming.

In the second scenario, the architect responsible for a large customer relationship management (CRM) system is having difficulty controlling database access from other systems, securing certain customer data, and making changes to the database schema. All of these problems stem from too many other systems using the CRM database. The architect therefore decides to create *application silos*, where each application database is only accessible from the application that owns that database. This decision will give the architect better control over the customer data, security, and changes.

However, unlike the developer's decision in the first scenario, the architect can expect almost everyone in the company to challenge their decision (with the possible exception of the CRM application team). Other applications need the CRM data, and if they can no longer access the database directly, they must ask the CRM system for the data via remote access calls. Product owners, project managers, and business stakeholders may object to increases in their costs or effort, while developers may feel their approach is better. *Almost every decision an architect makes will be challenged.*

Whatever the objections, the architect must navigate organizational politics and apply negotiation skills to get most decisions approved. This can be very frustrating; most software architects started as developers and got used to making decisions without approval or even review. As architects, they're now finally able to make broad and important decisions, but they must justify and fight for almost every one. Negotiation skills, like leadership skills, are so necessary that we've dedicated an entire chapter to them ([Chapter 25](#)).

Roadmap

This book consists of three parts:

Part I: Foundations

Part I defines the key components of software architecture, focusing on the two key elements of architectural structure: architectural characteristics and logical components. Analyzing each of these requires different techniques, which we cover in depth. The outputs of these activities provide the software architect with enough information to choose an appropriate architectural style to provide a scaffolding for the general philosophy of the implementation.

Part II: Architecture Styles

Part II provides a catalog of *architectural styles*, the named topologies of software architecture. We show the structural and communication differences in each style, and provide a basis for comparison along a wide spectrum, including data topologies, teams, and physical architectures.

Part III: Techniques and Soft Skills

Parts I and II focus on technical aspects of the job, but a major part of the role of software architect involves what are traditionally called *soft skills*: skills that involve other people instead of technology. Ironically, soft skills are often the most difficult for burgeoning architects to acquire, because the main pathways to the software architect role typically involve more technical than interpersonal brilliance. However, once on the job, architects discover that these skills are vitally important. Thus, Part III of our book covers some critical soft skills to help you succeed in the role.

PART I

Foundations

To understand important trade-offs in architecture, developers must understand some basic concepts and terminology concerning components, modularity, coupling, and connascence.

CHAPTER 2

Architectural Thinking

Architectural thinking is about seeing things with an architect's eye—in other words, from an architectural point of view. Understanding how a particular change might impact overall scalability, paying attention to how different parts of a system interact, and knowing which third-party libraries and frameworks would be most appropriate for a given situation are all examples of thinking architecturally.

Being able to think like an architect first involves understanding what software architecture is and the differences between architecture and design. It then involves having a wide breadth of knowledge to see solutions and possibilities that others do not see; understanding the importance of business drivers and how they translate to architectural concerns; and understanding, analyzing, and reconciling trade-offs between various solutions and technologies.

In this chapter, we explore these aspects of thinking like an architect.

Architecture Versus Design

Take a moment and picture your dream house in your mind. How many floors does it have? Is the roof flat or peaked? Is it a big sprawling, single-story ranch house or a multistory contemporary house? How many bedrooms does it have? All of these things define the overall *structure* of the house—in other words, its architecture. Now take a moment to picture the inside of the house. Does it have carpeting or hardwood floors? What color are the walls? Are there floor lamps or lights hanging from the ceiling? All of these things relate to the *design* of the house.

Similarly, software architecture is less about a system's appearance and more about its structure, whereas design is more about a system's appearance and less about its structure. For example, the choice to use microservices defines the structure and

shape of the system (its architecture), whereas the look and feel of the user interface (UI) screen define the design of the system.

But what about decisions such as whether to break apart a service into smaller parts or deciding on a UI framework? Unfortunately, most decisions such as these fall somewhere on a *spectrum* between architecture and design, making it difficult to determine what should be considered architecture.

Leveraging the following criteria can help determine whether something is more about architecture or more about design:

- Is it more strategic in nature or more tactical?
- How much effort will it take to change or construct?
- How significant are the trade-offs?

These factors are shown in **Figure 2-1**, illustrating the spectrum between architecture and design to help determine where a decision lies and who should have the responsibility for it.

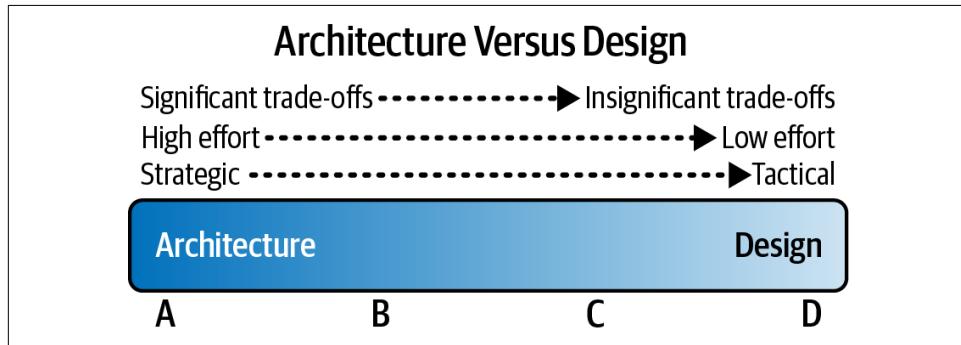


Figure 2-1. The spectrum between architecture and design

Strategic Versus Tactical Decisions

The more strategic a decision, the more architectural it becomes. Conversely, the more tactical a decision, the more it's likely to be about design. *Strategic* decisions are generally long-term, whereas *tactical* decisions are generally short-term and usually independent of other actions or decisions.

One good way to determine whether a decision is more strategic or tactical is to consider the following questions:

How much thought and planning is involved in the decision?

A decision that takes a couple of minutes is more likely to be tactical and hence more about design, whereas a decision requiring weeks of planning is likely to be more strategic and hence more about architecture.

How many people are involved in the decision?

A decision made alone or with a colleague is likely more tactical and on the design side of the spectrum, whereas a decision requiring lots of meetings with many different stakeholders is likely more strategic and on the architectural side of the spectrum.

Is the decision a long-term vision or a short-term action?

A decision that is likely to change soon is usually tactical in nature and would be more about design, whereas one that will last a very long time is typically more strategic and more about architecture.

While these questions are a bit subjective, they nevertheless help in determining whether something is strategic or tactical, and thus more about architecture or design.

Level of Effort

In his famous paper “[Who Needs an Architect?](#)”, software architect Martin Fowler writes that architecture is “the stuff that’s hard to change.” The harder something is to change, generally, the more effort is required, placing that decision or activity toward the architectural side of the spectrum. Conversely, something that requires minimal effort to implement or change places it more on the design side of the spectrum.

For example, moving from a monolithic layered architecture to microservices would require significant effort, so it would be more about architecture. Rearranging fields on a screen would require minimal effort, and therefore be more about design.

The Significance of Trade-Offs

Analyzing the trade-offs of a particular decision can help a lot in determining whether it is more about architecture or design. The more significant the trade-offs are, the more architectural the decision tends to be. For example, choosing to use the microservices architecture style provides better scalability, agility, elasticity, and fault tolerance. However, this architecture is highly complex, is very expensive, has poor data consistency, and doesn’t perform well due to service coupling. These are some pretty significant trade-offs. We can conclude that this decision is more on the architectural side of the spectrum than design.

Even design decisions have trade-offs. For example, breaking apart a class file provides better maintainability and readability—at the cost of managing more classes.

These trade-offs are not overly significant (especially compared to microservices' trade-offs), so this decision is more on the design side of the spectrum.

Technical Breadth

Unlike developers, who must have a significant amount of *technical depth* to perform their jobs, software architects must have a significant amount of *technical breadth* to see things from an architectural point of view. Technical depth is all about having deep knowledge of a particular programming language, platform, framework, product, and so on, whereas technical breadth is all about knowing a little bit about a lot of things.

To better understand the difference, consider the knowledge pyramid shown in [Figure 2-2](#). It encapsulates all the technical knowledge in the world, which can be broken down into three levels: *stuff you know*, *stuff you know you don't know*, and *stuff you don't know you don't know*.

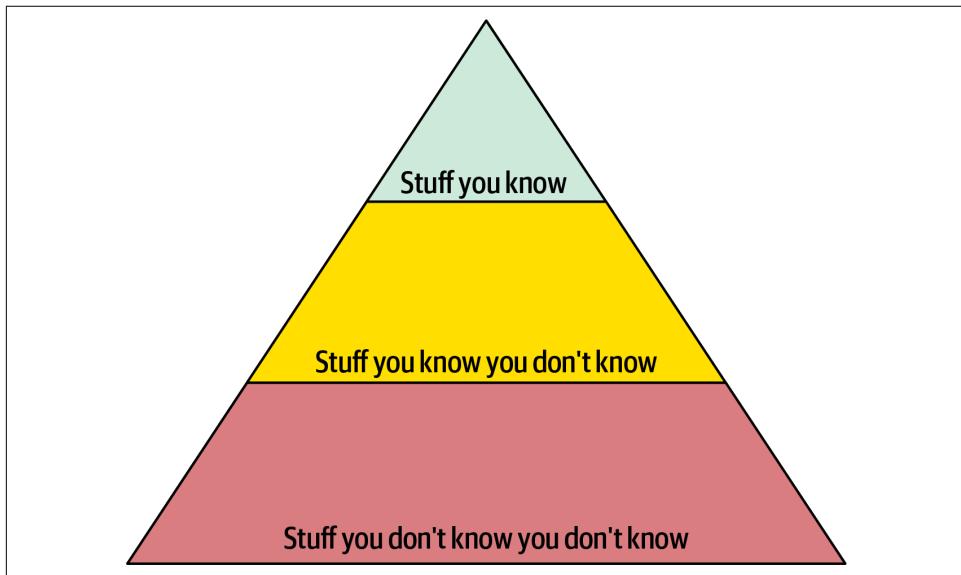


Figure 2-2. The pyramid representing all knowledge

Stuff you know includes the technologies, frameworks, languages, and tools technologists use on a daily basis to perform their jobs (such as a Java programmer knowing Java). They're good, or even expert, at all these things. Notice that this level of knowledge (represented by the top part of the pyramid) is the smallest and contains the fewest things. This is because most technologists have to pick and choose the areas in which to develop expertise—no one can be an expert at everything.

Stuff you know you don't know (the middle part of the pyramid) includes things a technologist knows a little about or has heard of but in which they have little or no experience or expertise. For example, most technologists have *heard* of Clojure and know it's a programming language based on Lisp, but can't write source code in Clojure. This level of knowledge is much bigger than the top level. This is because people can become familiar with many more things than they can develop expertise in.

Stuff you don't know you don't know is the largest part of the knowledge pyramid. It includes the entire host of technologies, tools, frameworks, and languages that would be the perfect solution to a problem, if only the technologist trying to solve the problem knew that these solutions exist. The goal in any individual's career should be to move things from the *stuff you don't know you don't know* into the second area of the pyramid, the *stuff you know you don't know*—and, when expertise becomes necessary, to move things from the middle part of the pyramid to the top: the *stuff you know*.

Early on in a developer's career, expanding the top of the pyramid (Figure 2-3) means gaining valuable expertise. However, the *stuff you know* is also stuff you must *Maintain*—nothing is static in the software world. If a developer becomes an expert in Ruby on Rails, that expertise won't last if they ignore Ruby on Rails for a year or two. Keeping things at the top of the pyramid requires a time investment to maintain expertise. This top part represents the individual's *technical depth*: the things they know really well.

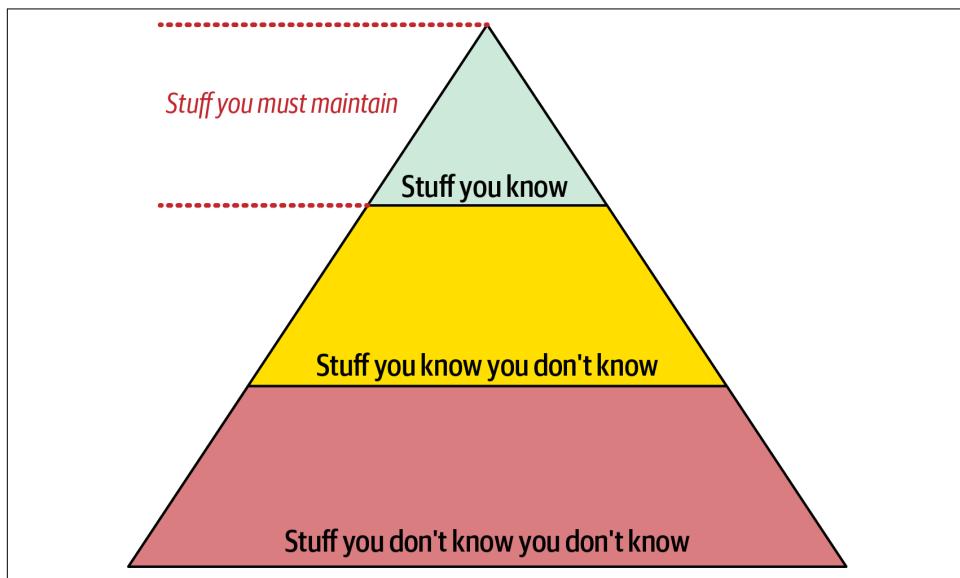


Figure 2-3. Developers must maintain expertise to retain it

However, the nature of knowledge changes as developers transition into the architect role. A large part of the value of an architect is that they have a *broad* understanding of technology and how to use it to solve particular problems. For example, it's better for an architect to know that five solutions exist for a particular problem than to have singular expertise in only one. The most important parts of the pyramid for architects are the top *and* middle sections; how far the middle section penetrates into the bottom section represents an architect's technical *breadth*, as shown in [Figure 2-4](#).

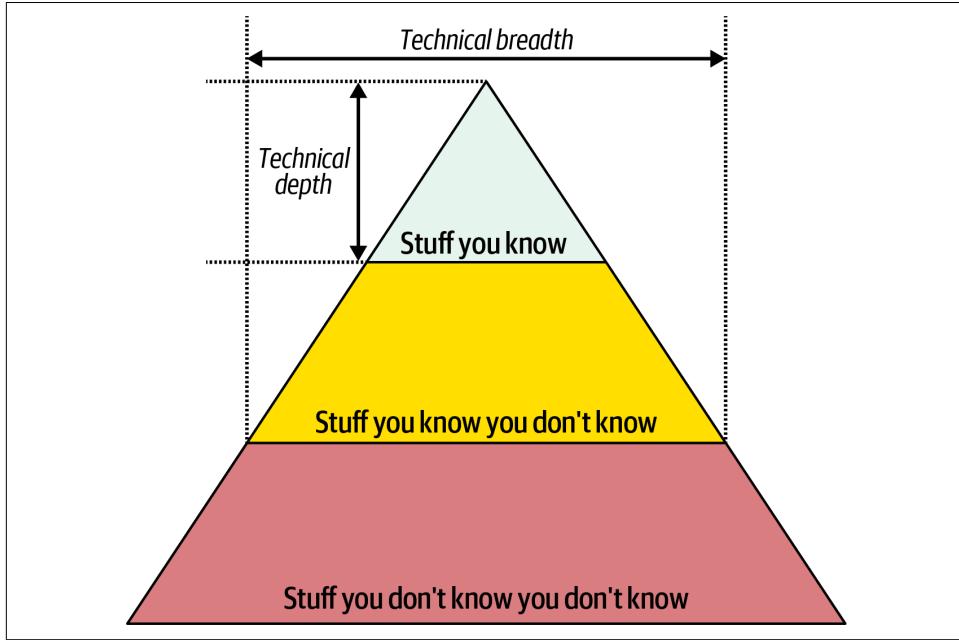


Figure 2-4. How much someone knows about a topic is technical depth, and how many topics someone knows is technical breadth

For an architect, *breadth* is more important than *depth*. Because architects must make decisions that match capabilities to technical constraints, a broad understanding of a wide variety of solutions is valuable. Thus, for an architect, the wise course of action is to sacrifice some hard-won expertise and use that time to broaden their portfolio, as shown in [Figure 2-5](#). Some areas of expertise will remain, probably in particularly enjoyable technology areas, while others usefully atrophy.

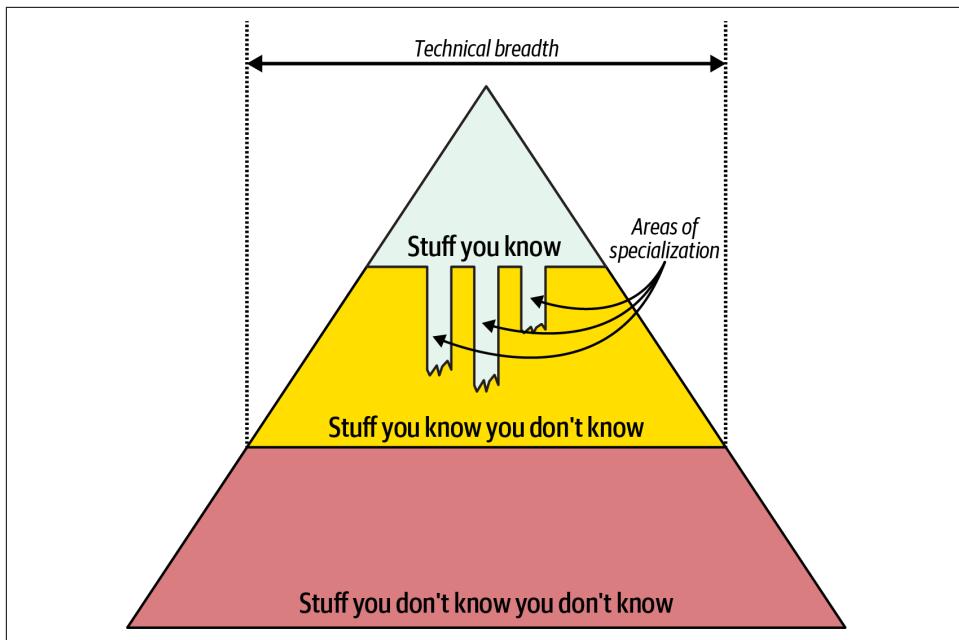


Figure 2-5. Enhanced breadth and shrinking depth for the architect role

Our knowledge pyramid illustrates how fundamentally different the roles of *architect* and *developer* are. Developers spend their whole careers honing expertise. Transitioning to the architect role means a shift in that perspective, which many individuals find difficult. This in turn leads to two common dysfunctions: first, an architect tries to maintain expertise in a wide variety of areas, succeeding in none of them and working themselves ragged in the process. Second, it manifests as *stale expertise* —the mistaken sensation that your outdated information is still cutting edge. We see this often in large companies where the developers who founded the company have moved into leadership roles, yet still make technology decisions using ancient criteria (see “Frozen Caveman Antipattern” on page 24).

Frozen Caveman Antipattern

An *antipattern* is what programmer Andrew Koenig defines as something that seems like a good idea when you begin, but leads you into trouble. A behavioral antipattern commonly observed in the wild, the Frozen Caveman antipattern, describes architects who revert to their pet irrational concern for every architecture. For example, one of Neal's colleagues worked on a system that featured a centralized architecture. Each time they delivered the design to the client architects, the persistent question was: "But what if we lose Italy?" Several years before, a freak communication problem had prevented the client's headquarters from communicating with its stores in Italy, causing great inconvenience. While the chances of this recurring were extremely small, the architects had become obsessed with this particular architectural characteristic.

Generally, this antipattern manifests in architects who have been burned in the past by a poor decision or unexpected occurrence, making them particularly cautious about anything related. While risk assessment is important, it should be realistic as well. Understanding the difference between genuine and perceived technical risk is part of the ongoing learning process. Thinking like an architect requires overcoming these Frozen Caveman ideas and experiences, seeing other solutions, and asking more relevant questions.

Architects should focus on technical breadth so that they have a larger quiver from which to draw arrows. Developers transitioning to the architect role may have to change the way they view knowledge acquisition. Balancing the depth and the breadth of their portfolio of knowledge is something every developer should consider throughout their career. But how does an architect gain technical breadth? The next sections provide a few techniques that will help you uncover the "stuff you don't know that you don't know."

The 20-Minute Rule

As illustrated in [Figure 2-5](#), technical breadth is more important to architects than technical depth. But how do you stay current on all the latest trends and buzzwords while also working full-time, developing your career, spending time with friends, and taking care of your family?

One technique we use is the *20-minute rule*. The idea is to devote *at least* 20 minutes a day to learning something new or diving deeper into a specific topic. [Figure 2-6](#) illustrates some good places to spend your 20 minutes, such as [InfoQ](#), [DZone Refcardz](#), and the [Thoughtworks Technology Radar](#). You can learn more about unfamiliar buzzwords by looking them up on the internet, promoting that knowledge from "the stuff you don't know you don't know" into the "stuff you know you don't know." You could even spend that time reading a book like this one. The point is to carve some

time out of your busy day to focus on developing your technical breadth and thus your career.

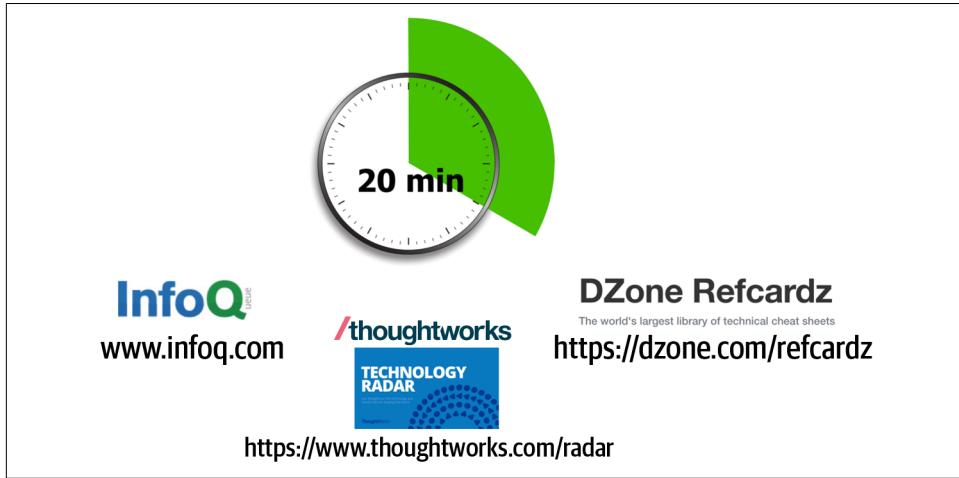


Figure 2-6. The 20-minute rule

Many technologists, when they first embrace this concept, plan their 20 minutes for lunch or after work; however, in our experience, these time periods rarely work. It's easy to start using lunchtimes to catch up on work rather than take a break, and evenings are even worse, with social plans, family time, and more after a long day. Instead, we strongly recommend taking your 20 minutes first thing in the morning—right after grabbing a cup of coffee or tea and, importantly, *before you check your email*. Once you check email, your morning is over and your day has started. Get your 20 minutes in while your mind is fresh and before distractions take over.

Following the 20-minute rule will increase your technical breadth and help you develop and maintain the knowledge that will make you an effective software architect.

Developing a Personal Radar

For most of the '90s and the beginning of the '00s, one of your authors was the CTO of a small training and consulting company. When he started there, the primary platform was Clipper, a rapid-application development tool for building DOS applications atop dBASE files. Until one day it vanished. The company had noticed the rise of Windows, but the business market was still DOS...until it abruptly wasn't. One coworker lamented that they couldn't take their enormous body of now-useless Clipper knowledge and replace it with something else. Has any group in history, the coworker wondered, learned and thrown away so much detailed knowledge within

their lifetimes as software developers do? The experience left a lasting impression: *ignore the march of technology at your peril.*

It also taught us an important lesson about technology bubbles. When developers, architects, and other technologists become heavily invested in a specific technology, pouring our work and thought into it, we tend to live in a memetic bubble. Inside the bubble, which also serves as an echo chamber, everyone knows and cares about that technology as much as we do. We might not even see honest appraisals from outside the bubble, especially if the bubble was created by a technology vendor in the first place. And when the bubble begins to collapse, there's no warning until it's too late.

What we lack in our bubble is a *technology radar*: a living document to help us assess the risks and rewards of existing and nascent technologies. The radar concept comes from [Thoughtworks](#), where Neal serves as a director and software architect. In this section, we'll describe how this concept came to be and then show you how to create a personal radar.

The Thoughtworks Technology Radar

The Technology Advisory Board (TAB) is a group of senior technology leaders within Thoughtworks that assists the CTO in making decisions about technology directions and strategies for the company and its clients. To stay current, this group began producing what's now a biannual [Technology Radar](#).

This had unexpected side effects. When Neal spoke at conferences, attendees began seeking him out to thank him for helping produce the Radar, often adding that their company had started producing its own version. Neal also realized that this was the answer to a question constantly asked at conference speaker panels: "How do you keep up with technology? How do you figure out what to pursue next?" The answer, of course, is that the speakers all have some form of internal radar.

Parts. The Thoughtworks Radar consists of four quadrants that attempt to cover most of the software-development landscape:

Tools

Everything from development tools like IDEs to enterprise-grade integration tools

Languages and frameworks

Computer languages, libraries, and frameworks, typically open source

Techniques

Any practice that assists software development overall, including processes, engineering practices, and advice

Platforms

Technology platforms, including databases, cloud vendors, and operating systems

Rings. The Radar has four rings, listed here from outer to inner:

Hold

The original meaning of the Hold ring was “hold off for now,” to represent technologies that were too new to reasonably assess yet—technologies that were getting lots of buzz but weren’t yet proven. The Hold ring has evolved, and now indicates something more like “don’t start anything new with this technology.” There’s no harm in using it on existing projects, but think twice about using it for new development.

Assess

The Assess ring indicates that a technology is worth exploring (such as through development spikes, research projects, or conference sessions) to see how it will affect the organization. For example, when mobile browsers became prominent, many large companies visibly went through this phase when formulating a mobile strategy.

Trial

The Trial ring is for technologies worth pursuing. If a capability is in this ring, it is important to understand how to build it. Now is the time to pilot a low-risk project.

Adopt

Thoughtworks feels strongly that the industry should adopt the items listed in the Adopt ring.

In the example view of the Radar in [Figure 2-7](#), each “blip” represents a different technology or technique. While Thoughtworks uses the radar to broadcast its collective opinions about the software world, many developers and architects also use it as a way of structuring their technology assessment process and organizing their thinking about what to invest time in. For personal use, we suggest altering the meanings of the quadrants to the following:

Hold

This can include not only technologies and techniques to avoid, but also habits you are trying to break. For example, if you’re an architect from the .NET world, you might be accustomed to reading the latest news and gossip on forums about team internals. While entertaining, this may be a low-value information stream. Placing it in the Hold ring serves as a reminder of what you want to avoid.

Assess

Use the Assess ring for promising technologies that you've heard good things about but haven't had time to assess for yourself yet. This ring forms a staging area for more serious future research.

Trial

The Trial ring indicates active research and development, such as performing spike experiments within a larger code base. These are technologies worth spending time on to understand more deeply so that you can include them effectively in trade-off analysis.

Adopt

Your personal Adopt ring represents the new things you're most excited about and best practices for solving particular problems.

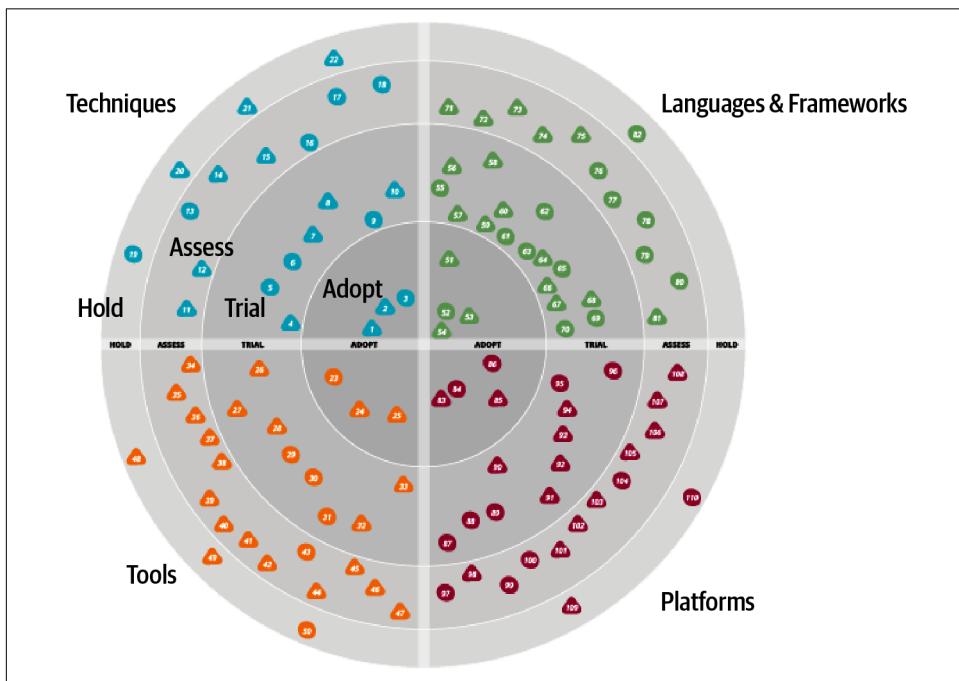


Figure 2-7. A sample Thoughtworks Technology Radar

Most technologists pick technologies on a more or less ad hoc basis, based on what's cool or what their employers are using—but it's dangerous to your career to adopt a laissez-faire attitude toward your technology portfolio. Creating a technology radar helps you formalize your thinking about technology and balance opposing decision criteria. (For example, it might be harder to get a new job focused on the “cooler” technology, whereas a more established technology might have a huge job market but offer less interesting work.)

Treat your technology portfolio like a financial portfolio: diversify! Choose some technologies and/or skills that are widely in demand, and track that demand. But you might also want to try some technology gambits, like generative AI or embedded IOT devices. Anecdotes abound about developers who freed themselves from cubicle-dwelling servitude by working late at night on open source projects that became popular and, eventually, purchasable. This is yet another reason to focus on breadth rather than depth.

Building a personal radar provides a good scaffolding for broadening your technology portfolio—but ultimately, the exercise is more important than the outcome. Creating the radar visualization gives you an excuse to carve out time from your busy schedule to think about these things, which is often the only way to accomplish this kind of thinking.

While the most important part of building your personal radar is the conversations it generates, it also produces some very useful visualizations. After much popular demand from technologists building their own radar visualizations, Thoughtworks released a tool called [Build Your Own Radar](#). With a Google spreadsheet as input, it generates radar visualization showing your personal radar. We encourage every technologist to leverage it.

Analyzing Trade-Offs

Thinking like an architect is about seeing trade-offs in every solution, technical or otherwise, and analyzing those trade-offs to determine the best solution. The reason this is one of the critical activities of an architect (and hence part of architectural thinking) is exemplified through the following quote by Mark (one of your authors):

Architecture is the stuff you can't Google or ask an LLM about.

—Mark Richards

Everything in architecture is a trade-off, which is why the famous answer to every architecture question in the universe is “It depends.” While this answer might be annoying, it is unfortunately true. You cannot Google the answer or ask an AI engine or large language model (LLM) whether REST or messaging would be better for your system or whether microservices is the right architecture style for your new product, because the answer *does* depend. It depends on the deployment environment, business drivers, company culture, budgets, time frames, developer skill set, and dozens of other factors. Everyone’s environment, situation, and problem will be different. That’s why architecture is so hard. To quote Neal, your other author:

There are no right or wrong answers in architecture—only trade-offs.

—Neal Ford

For example, consider an item auction system (Figure 2-8) where online bidders bid on items up for auction. The Bid Producer service generates a bid from the bidder and then sends that bid amount to the Bid Capture, Bid Tracking, and Bid Analytics services.

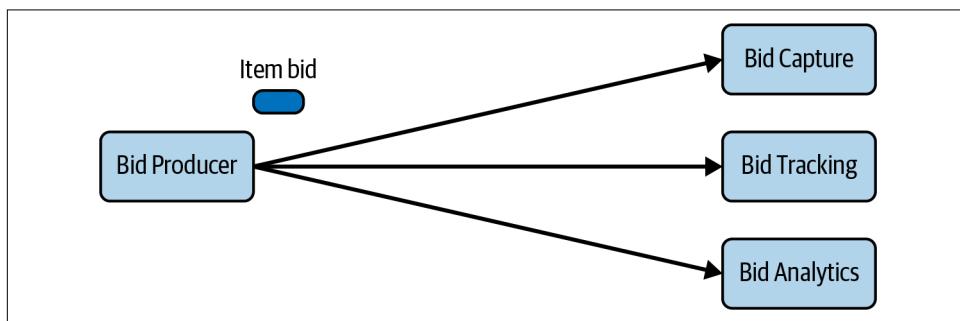


Figure 2-8. Auction system example of a trade-off—queues or topics?

For the asynchronous behavior in this system, an architect could use queues in a point-to-point messaging fashion, or use a topic in a publish-and-subscribe messaging fashion. Which one should they choose? They can’t Google the answer. Architectural thinking requires them to analyze the trade-offs associated with each option and select the best (or least worst) option given the specific situation.

The two messaging options for the item auction system are shown in [Figure 2-9](#), which illustrates using topics in a publish-and-subscribe messaging model, and [Figure 2-10](#), which depicts using queues in a point-to-point messaging model.

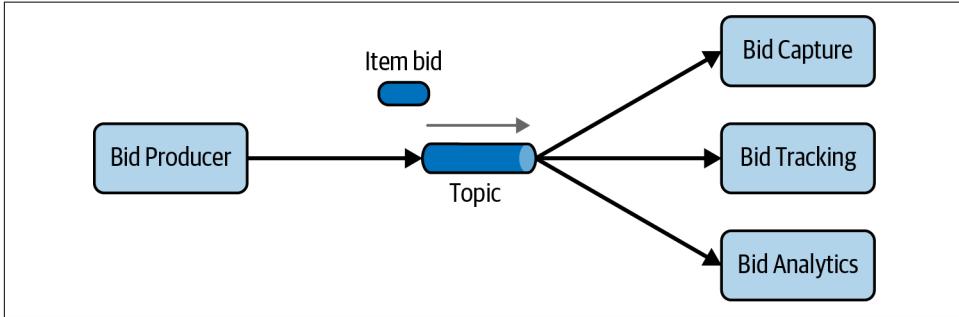


Figure 2-9. Using a topic for communication between services

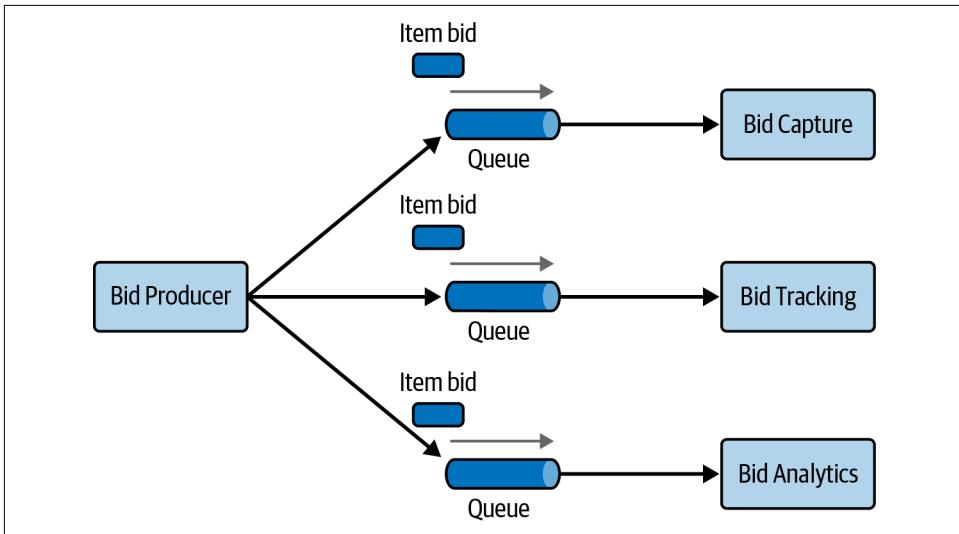


Figure 2-10. Using queues for communication between services

The clear advantage (and seemingly obvious solution) to this problem in [Figure 2-9](#) is *architectural extensibility*. The **Bid Producer** service only requires a single connection to a topic. Compare that to the queue solution in [Figure 2-10](#), where the **Bid Producer** needs to connect to three different queues. If a new service called **Bid History** were to be added to this system (to provide each bidder with a history of all of their bids), no changes would be needed to the existing services or infrastructure. The new **Bid History** service could simply subscribe to the topic that already contains the bid information.

However, with the queue option shown in [Figure 2-10](#), the Bid History service would require a new queue, and the Bid Producer would need to be modified to add an additional connection to the new queue. The point here is that using queues means that adding new bidding functionality requires significant changes to the services and infrastructure, whereas with the topic approach, no changes to the existing infrastructure are needed. Also, the Bid Producer is less coupled in the topic option, where the Bid Producer doesn't know how the bidding information will be used or by which services. In the queue option, the Bid Producer knows exactly how the bidding information will be used (and by whom), and hence is more coupled to the system.

So far, this trade-off analysis seems to make it clear that the topic approach using the publish-and-subscribe messaging model is the obvious and best choice. However, to quote [Rich Hickey](#), the creator of the Clojure programming language:

Programmers know the benefits of everything and the trade-offs of nothing. Architects need to understand both.

—Rich Hickey

Thinking architecturally means not only looking at the benefits of a given solution, but also analyzing the associated negatives, or trade-offs. Continuing with the auction system example, a software architect would analyze the negatives of the topic solution as well as the positives. In [Figure 2-9](#), you can see that with a topic, *anyone* can access bidding data, which introduces a possible issue with data access and data security. However, in the queue model illustrated in [Figure 2-10](#), the data sent to the queue can *only* be accessed by the specific consumer receiving that message. If a rogue service was to listen in on a queue, the corresponding service would not receive those bids, and a notification would immediately be sent about the loss of data (and possible security breach). In other words, it is very easy to wiretap a topic, but not a queue.

In addition to the security issue, the topic solution in [Figure 2-9](#) only supports homogeneous contracts. All services that receive the bidding data must accept the same data contract and set of bidding data. In the queue option, each consumer can have its own contract, specific to the data it needs. For example, suppose the new Bid History service requires the current asking price along with the bid, but no other service needs that information. In this case, the contract would need to be modified, impacting all other services using that data. In the queue model, this would be a separate channel, and thus a separate contract that does not impact any other service.

Another disadvantage of the topic model is that it does not support monitoring the number of messages in the topic, so it can't support autoscaling capabilities. However, with the queue option, each queue can be monitored individually and programmatic load balancing can be applied to each bidding consumer so that each can be automatically scaled independently. Note that this trade-off is technology specific:

the [Advanced Message Queuing Protocol \(AMQP\)](#) can support programmatic load balancing and monitoring because of the separation between an exchange (what the producer sends to) and a queue (what the consumer listens to).

Given this fuller trade-off analysis, which is the better option?

It depends! [Table 2-1](#) summarizes these trade-offs.

Table 2-1. Trade-offs for topics

| Topic advantages | Topic disadvantages |
|-----------------------------|---|
| Architectural extensibility | Data access and data security concerns |
| Service decoupling | No heterogeneous contracts |
| | Monitoring and programmatic scalability |

Again, *everything* in software architecture has trade-offs: advantages and disadvantages. Thinking like an architect means analyzing these trade-offs, then asking questions like: “Which is more important: extensibility or security?” An architect’s choice will always depend on the business drivers, environment, and a host of other factors.

Understanding Business Drivers

Thinking like an architect also means understanding the business drivers required for the success of the system and translating those requirements into architecture characteristics such as scalability, performance, and availability. This is a challenging task that requires the architect to have some business-domain knowledge and healthy, collaborative relationships with key business stakeholders. We’ve devoted four chapters in the book to this specific topic: in [Chapter 4](#), we define various architecture characteristics. In [Chapter 5](#), we describe ways to identify and qualify architecture characteristics. In [Chapter 6](#), we describe how to measure each characteristic to ensure the business needs of the system are met. And finally, in [Chapter 7](#) we discuss the scope of architectural characteristics and how they relate to coupling.

Balancing Architecture and Hands-On Coding

One of the difficult tasks an architect faces is how to balance hands-on coding with software architecture. We firmly believe that every architect should code and should maintain a certain level of technical depth (see “[Technical Breadth](#)” on page 20). While this may seem like an easy task, it is sometimes rather difficult to accomplish.

Our first tip for anyone striving to balance hands-on coding with being a software architect is avoiding the *Bottleneck Trap*. The Bottleneck Trap antipattern occurs when an architect takes ownership of code within the critical path of a system (usually the underlying framework code or some of the more complicated parts) and becomes a bottleneck to the team. This happens because the architect is not a full-time developer and therefore must balance development work (like writing and testing source code) with the architect role (drawing diagrams, attending meetings, and well, attending more meetings).

One way to avoid the Bottleneck Trap is for the architect to delegate the critical parts of the system to others on the development team and then focus on coding a minor piece of business functionality (such as a service or UI screen) one to three iterations down the road. This has three positive effects. First, the architect gains hands-on experience by writing production code while avoiding becoming a bottleneck on the team. Second, the critical path and framework code are distributed to the development team (where they belong), giving the team ownership and a better understanding of the harder parts of the system. Third, and perhaps most important, the architect is writing the same business-related source code as the development team. This helps them better identify with the development team's pain points around processes, procedures, and the development environment (and hopefully work to improve those things).

Suppose, however, that the architect can't develop code with the development team. How can they still remain hands-on and maintain some level of technical depth? The following are some tips and techniques for architects who want to continue deepening their technical abilities:

Frequent proofs-of-concept

Doing frequent proofs-of-concept (POCs) requires the architect to write source code; it also helps validate an architecture decision by taking the implementation details into account. For example, suppose an architect gets stuck trying to choose between two caching solutions. One effective way to help make this decision is to develop a working example in each caching product and compare the results. This allows the architect to see firsthand the implementation details and the amount of effort required to develop the full solution. It also allows them to better compare architectural characteristics between the different caching solutions, such as scalability, performance, and overall fault tolerance.

Whenever possible, the architect should write the best production-quality code they can. We recommend this practice for two reasons. First, quite often, “throw-away” proof-of-concept code goes into the source code repository and becomes the reference architecture or guiding example for others to follow. The last thing any architect would want is for their sloppy throwaway code to be treated as representing their typical quality of work. Second, writing production-quality

proof-of-concept code means getting practice at writing quality, well-structured code, rather than continually developing bad coding practices by creating quick, sloppy POCs.

Tackle technical debt

Another way architects can remain hands-on is to tackle some technical debt, freeing the development team to work on the critical functional user stories. Tech debt is usually low priority, so if the architect doesn't get the chance to complete a technical debt task within a given iteration, it's not the end of the world and generally won't impact the success of the iteration.

Fix bugs

Similarly, working on bug fixes within an iteration is another way of maintaining hands-on coding skills while helping the development team. While certainly not glamorous, this technique allows the architect to identify issues and weaknesses within the code base and possibly the architecture.

Automate

Leveraging automation by creating simple command-line tools and analyzers to help the development team with their day-to-day tasks is another great way to maintain hands-on coding skills while making the development team more effective. Look for repetitive tasks the development team performs and automate them. They'll be grateful for the automation. Some examples are automated source validators to help check for specific coding standards not found in other lint tests, automated checklists, and repetitive manual code-refactoring tasks.

Automation in the form of architectural analysis and fitness functions to ensure the vitality and compliance of the architecture is another great way to stay hands-on. For example, an architect can write Java code in [ArchUnit](#) in the Java platform to automate architectural compliance, or custom [fitness functions](#) to ensure architectural compliance while gaining hands-on experience. We talk about these techniques in [Chapter 6](#).

Do code reviews

A final technique to remain hands-on as an architect is to do frequent code reviews. While the architect is not actually writing code, this at least keeps them *involved* in the source code. Further benefits of code reviews include ensuring compliance with the architecture and identifying mentoring and coaching opportunities on the team.

There's More to Architectural Thinking

This chapter forms the foundational aspects of beginning to think like an architect. However, there's much more to thinking like an architect than what is described in this chapter. Thinking like an architect involves understanding the overall structure of a system (we cover that topic next in [Chapter 3](#)), understanding business concerns and translating those concerns to architectural characteristics (we have four chapters on that topic), and finally, seeing a system through its logical components—the building blocks of a system (we discuss that topic in [Chapter 8](#)).

CHAPTER 3

Modularity

Architects and developers have struggled with the concept of modularity for quite some time, as is evident in this quote from *Composite/Structured Design* (Van Noststrand Reinhold, 1978):

95% of the words [written about software architecture] are spent extolling the benefits of “modularity” and little, if anything, is said about how to achieve it.

—Glenford J. Myers

Different platforms offer different reuse mechanisms for code, but all support some way of grouping related code together into *modules*. While this concept is universal in software architecture, it has proven slippery to define. A casual internet search yields dozens of definitions, with no consistency (and some contradictions). This isn't a new problem. However, because no recognized definition exists, we must jump into the fray and provide our own definitions for the sake of consistency throughout the book.

Understanding modularity and its many incarnations in the development platform of choice is critical for architects. Many of the tools we have to analyze architecture (such as metrics, fitness functions, and visualizations) rely on modularity and related concepts. *Modularity* is an organizing principle. If an architect designs a system without paying attention to how the pieces wire together, that system will present myriad difficulties. To use a physics analogy, software systems model complex systems, which tend toward entropy (or disorder). In a physical system, energy must be added to preserve order. The same is true for software systems: architects must constantly expend energy to ensure structural soundness, which won't happen by accident.

Preserving good modularity exemplifies our definition of an *implicit* architecture characteristic: virtually no project requirements explicitly ask the architect to ensure

good modular distinction and communication, but sustainable code bases do require the order and consistency that this brings.

Modularity Versus Granularity

Developers and architects often use the terms *modularity* and *granularity* interchangeably, yet their meanings are very different. Modularity is about breaking systems apart into smaller pieces, such as moving from a monolithic architecture style (like the traditional n-tiered layered architecture) to a highly distributed architecture style, like microservices. Granularity, on the other hand, is about the *size* of those pieces—how big a particular part of the system (or service) should be. However, as stated in the following quote by one of your authors, it's with *granularity* where architects and developers get into trouble:

Embrace modularity, but beware of granularity.

—Mark Richards

Granularity causes services or components to be coupled to one another, creating complex, hard-to-maintain architecture antipatterns like *Spaghetti Architecture*, *Distributed Monoliths*, and the famous *Big Ball of Distributed Mud*. The trick to avoiding these architectural antipatterns is to pay attention to granularity and the overall level of coupling between services and components.

Defining Modularity

Merriam-Webster defines a *module* as “each of a set of standardized parts or independent units that can be used to construct a more complex structure.” By contrast, in this book, we use *modularity* to describe a logical grouping of related code, which could be a group of classes in an object-oriented language or a group of functions in a structured or functional language. Developers typically use modules as a way to group related code together. For example, the `com.mycompany.customer` package in Java should contain things related to customers. Most languages provide mechanisms for modularity (`package` in Java, `namespace` in .NET, and so on).

Modern programming languages feature a wide variety of packaging mechanisms, and many developers find it difficult to choose between them. For example, in many modern languages, developers can define behavior in functions/methods, classes, or packages/namespaces, each with different visibility and scoping rules. Some languages complicate this further by adding programming constructs, such as the [meta-object protocol](#), to provide even more extension mechanisms.

Architects must be aware of how developers package things, because packaging has important implications in architecture. For example, if several packages are tightly coupled, reusing one of them for related work becomes more difficult.

Modular Reuse Before Classes

Developers who trained in the days before object-oriented languages may puzzle over why there are so many different separation schemes. Much of the reason has to do with backward compatibility—not of code, but of how developers think about things.

In March 1968, the journal *Communications of the Association for Computing Machinery* (ACM) published a paper from computer scientist Edsger Dijkstra entitled “Go To Statement Considered Harmful.” He denigrated the common use of the GOTO statement, common in programming languages at the time, because it allowed nonlinear leaping around within code, making reasoning and debugging difficult.

Dijkstra’s paper helped usher in the era of *structured* programming languages in the mid-1970s, exemplified by Pascal and C, which encourage deeper thinking about how things fit together. Developers quickly realized that most programming languages offered no good way to group like things together logically. Thus, the short era of *modular* languages was born in the mid-1980s, such as Modula (Pascal creator Niklaus Wirth’s next language) and Ada. These languages embraced the programming construct of the *module*, much as we think about packages or namespaces today (but without the classes).

However, the modular programming era of the mid-1980s was short-lived because object-oriented languages became popular and offered new ways to encapsulate and reuse code. Still, language designers realized the utility of modules and retained them in the form of packages and namespaces. Many languages still contain compatibility features that seem odd today but were introduced to support these different paradigms. For example, Java supports modular paradigms (via packages and package-level initialization using static initializers), as well as object-oriented and functional paradigms, each with its own scoping rules and quirks.

In this book’s discussions about architecture, we use *modularity* as a general term to denote a related grouping of code: classes, functions, or any other grouping. This doesn’t imply a physical separation, merely a logical one. (The difference is sometimes important.) For example, lumping a large number of classes together in a monolithic application may be convenient; however, when it comes time to restructure the architecture, the coupling encouraged by loose partitioning can impede efforts to break the monolith apart. That’s why it’s useful to talk about modularity as a concept, separate from the physical separation that a particular platform forces or implies.

It is worth discussing the general concept of a *namespace*, which is separate from the technical implementation in the .NET platform that is also called a namespace. Developers often need precise, fully qualified names for different software assets (components, classes, and so on) to separate them from each other. The most obvious

example that people use every day is the internet, which relies on unique, global identifiers tied to IP addresses.

Most languages have some modularity mechanism that doubles as a namespace to organize things like variables, functions, or methods. Sometimes the module structure is reflected physically: Java package structures, for example, must reflect the directory structure of the physical class files.

A Language with No Name Conflicts: Java 1.0

The original designers of Java had extensive experience dealing with name conflicts and clashes, which were common in programming platforms at the time. Java 1.0 used a clever hack to avoid ambiguity when two classes had the same name—such as if the problem domain included a catalog *order* and an installation *order*, both named *order* but with very different connotations (and classes). The Java designers' solution was to create the package namespace mechanism, along with a requirement that the physical directory structure must match the package name. Because filesystems won't allow two files with the same name to reside in the same directory, this leveraged the operating system's inherent features to ambiguity and name conflicts. Thus, the original `classpath` in Java contained only directories.

However, as the language designers discovered, forcing every project to have a fully formed directory structure was cumbersome, especially as projects became larger. Plus, building reusable assets was difficult: frameworks and libraries had to be “exploded” into the directory structure. In the second major release of Java (1.2, but called Java 2), designers added the `jar` mechanism, which allows an archive file to act as a directory structure on a classpath. For the next decade, Java developers struggled with getting the classpath exactly right, as a combination of directories and JAR files. Their original intent had been broken: now two JAR files *could* create conflicting names on a classpath. This is why Java developers of that era tend to have numerous war stories about debugging class loaders.

Measuring Modularity

Since modularity is so important, architects need tools to help them better understand it. Fortunately, researchers have created a variety of language-agnostic metrics for this purpose. We'll focus here on three key concepts: *cohesion*, *coupling*, and *connascence*.

Cohesion

Cohesion refers to the extent to which a module's parts should be contained within the same module. In other words, it measures how related the parts are to one another. An ideal cohesive module is one where all parts are packaged together; breaking them into smaller pieces would require coupling the parts together via calls between modules to achieve useful results. The cautionary tale of modularity as it relates to cohesion is exemplified in the following quote from the book *Structured Design* (Pearson, 2008):

Attempting to divide a cohesive module would only result in increased coupling and decreased readability.

—Larry Constantine

Computer scientists have defined a range of cohesion, measured from best to worst:

Functional cohesion

Every part of the module is related to the other, and the module contains everything essential it needs to function.

Sequential cohesion

Two modules interact: one outputs data that becomes the input for the other.

Communicational cohesion

Two modules form a communication chain in which each operates on information and/or contributes to some output. For example, one adds a record to the database and the other generates an email based on that information.

Procedural cohesion

Two modules must execute code in a particular order.

Temporal cohesion

Modules are related based on timing dependencies. For example, many systems have a list of seemingly unrelated things that must be initialized at system startup; these different tasks are *temporally cohesive*.

Logical cohesion

The data within modules is related logically but not functionally. For example, consider a module that converts information from text, serialized objects, or streams into some other format. Its operations are related, but the functions are quite different. A common example of this type of cohesion exists in virtually every Java project in the form of the `StringUtils` package, a group of static methods that operate on `String` but are otherwise unrelated.

Coincidental cohesion

The elements in a module are unrelated other than being in the same source file. This represents the most negative form of cohesion.

Despite its many variants, *cohesion* is a less precise metric than *coupling*. Often, a particular module's degree of cohesion is determined at the discretion of a particular architect. Consider this module definition:

`Customer Maintenance`

- `add customer`
- `update customer`
- `get customer`
- `notify customer`
- `get customer orders`
- `cancel customer orders`

Should the last two entries reside in this module? Or should the developer create two separate modules? Here's what that would look like:

`Customer Maintenance`

- `add customer`
- `update customer`
- `get customer`
- `notify customer`

`Order Maintenance`

- `get customer orders`
- `cancel customer orders`

Which is the correct structure? As always, it depends:

- Are these the only two operations for `Order Maintenance`? If so, it may make sense to collapse those operations back into `Customer Maintenance`.
- Is `Customer Maintenance` expected to grow much larger? If so, perhaps developers should look for opportunities to extract behavior into a different (or new) module.
- Does `Order Maintenance` require so much knowledge of `Customer` information that separating the two modules would require a high degree of coupling to make it functional? (This relates back to the prior Larry Constantine quote.)

These questions represent the kind of trade-off analysis that lies at the heart of a software architect's job.

Computer scientists have developed a good structural metric to determine cohesion—specifically, *Lack of Cohesion* (which is a bit surprising, given how subjective this characteristic is). A well-known set of metrics named the **Chidamber and Kemerer Object-Oriented Metrics Suite** measures particular aspects of object-oriented software systems. It includes many common code metrics, such as Cyclomatic Complexity (see “[Cyclomatic Complexity](#)” on page 84) and several important coupling metrics, discussed in “[Coupling](#)” on page 44.

Chidamber and Kemerer have also developed a Lack of Cohesion in Methods (LCOM) metric, which measures the structural cohesion of a module. The initial version appears in [Equation 3-1](#).

Equation 3-1. LCOM, version 1

$$LCOM = \begin{cases} |P| - |Q|, & \text{if } |P| > |Q| \\ 0, & \text{otherwise} \end{cases}$$

In this equation, P increases by 1 for any method that doesn't access a particular shared field; Q decreases by 1 for methods that *do* share a particular shared field. If you find this formulation confusing, we sympathize—and it's gradually become even more elaborate. The second variation, introduced in 1996 (thus the name *LCOM96B*), appears in [Equation 3-2](#).

Equation 3-2. LCOM96B

$$LCOM96b = \frac{1}{a} \sum_{j=1}^a \frac{m - \mu(A_j)}{m}$$

We won't bother untangling the variables and operators in [Equation 3-2](#) because the following written explanation is clearer. Basically, the LCOM metric exposes incidental coupling within classes. A better definition of LCOM would be “the sum of sets of methods not shared via sharing fields.”

Consider a class with private fields `a` and `b`. Many of the methods only access `a`, and many other methods only access `b`. The *sum* of the sets of methods not shared via sharing fields (`a` and `b`) is high; therefore, this class incurs a high LCOM score, indicating a significant *lack of cohesion in methods*.

Consider the three classes shown in [Figure 3-1](#). Here, fields appear as single letters inside octagons, and methods appear as blocks. In Class X, the LCOM score is low, indicating good structural cohesion. Class Y, however, lacks cohesion; each of the

field/method pairs in Class Y could appear in its own class without affecting the system's behavior. Class Z shows mixed cohesion; the last field/method combination could be refactored into its own class.

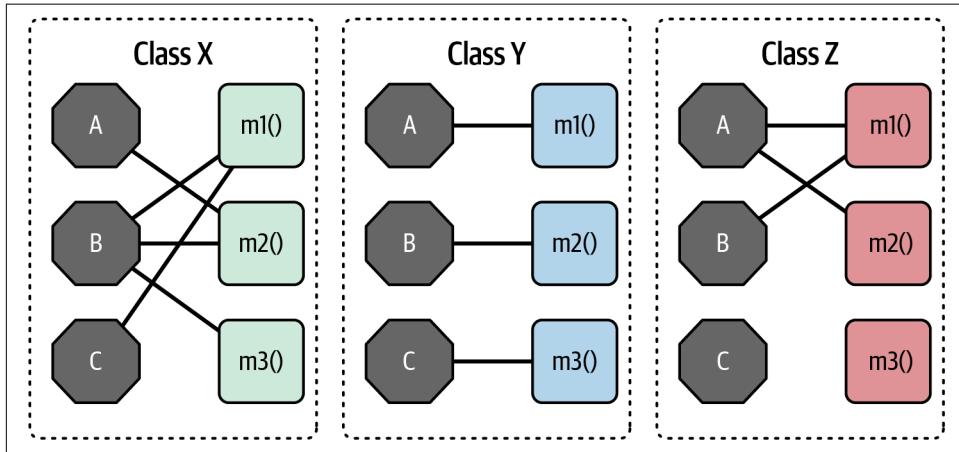


Figure 3-1. The LCOM metric, where fields are octagons, and methods are squares

The LCOM metric is useful to architects who are analyzing code bases in order to assist in restructuring, migrating, or understanding a code base. Shared utility classes are a common headache when moving architectures. Using the LCOM metric can help architects find classes that are incidentally coupled and should never have been a single class to begin with.

Many software metrics have serious deficiencies, and LCOM is not immune. All this metric can find is *structural* lack of cohesion; it has no way to determine if particular pieces fit together logically. This reflects back on our Second Law of Software Architecture: *why* is more important than *how*.

Coupling

Fortunately, we have better tools to analyze coupling in code bases. These are based in part on graph theory: because the method calls and returns form a call graph, it can be analyzed mathematically. Edward Yourdon and Larry Constantine's book *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design* (Prentice-Hall, 1979), which defined many core concepts, including the metrics *afferent coupling* and *efferent coupling*. *Afferent coupling* measures the number of *incoming* connections to a code artifact (component, class, function, and so on). *Efferent coupling* measures the *outgoing* connections to other code artifacts. There are tools for virtually every platform that allow architects to analyze the coupling characteristics of code.

Why Such Similar Names for Coupling Metrics?

Why are two critical metrics in the architecture world that represent *opposite concepts* named virtually the same thing, differing in only the vowels that sound the most alike? These terms originate from the book *Structured Design*. Borrowing concepts from mathematics, Yourdon and Constantine coined the now-common terms *afferent* and *efferent* coupling. They should really have been called *incoming* and *outgoing* coupling, but the authors leaned toward mathematical symmetry rather than clarity. Developers have come up with several mnemonics to help out. For instance, *a* appears before *e* in the English alphabet, just as *incoming* appears before *outgoing*. The letter *e* in *efferent* matches the initial letter in *exit*, which helps in remembering that it represents outgoing connections.

Core Metrics

While component coupling has raw value to architects, several other derived metrics allow for deeper evaluation. The metrics discussed in this section were created by software engineer [Robert C. Martin](#), and apply widely to most object-oriented languages.

Abstractness is the ratio of abstract artifacts (abstract classes, interfaces, and so on) to concrete artifacts (implementations). The Abstractness metric measures a code base's degree of abstractness versus implementation. For example, on one end of the scale would be a code base with no abstractions, just a huge, single function of code (as in a single `main()` method). The other end of the scale would be a code base with too many abstractions, making it difficult for developers to understand how things wire together. (For example, it takes developers a while to figure out what to do with an `AbstractSingletonProxyFactoryBean` abstract class, given its many layers of abstraction and ambiguous name.)

The formula for Abstractness appears in [Equation 3-3](#).

Equation 3-3. Abstractness

$$A = \frac{\sum m^a}{\sum m^c + \sum m^a}$$

Architects calculate Abstractness by calculating the ratio of the sum of abstract artifacts to the sum of the concrete and abstract ones. In the equation, m^a represents *abstract* elements (interfaces or abstract classes) with the module, and m^c represents *concrete* elements (nonabstract classes). This metric looks for the same criteria. The easiest way to visualize this metric is to consider an application with 5,000 lines of code, all in one `main()` method. Its Abstractness numerator would be 1, while the denominator would be 5,000, yielding an Abstractness score of almost 0. That's how this metric measures the ratio of abstractions in code.

Another derived metric, *Instability*, is defined as the ratio of efferent coupling to the sum of both efferent and afferent coupling, as shown in [Equation 3-4](#).

Equation 3-4. Instability

$$I = \frac{C^e}{C^e + C^a}$$

In the equation, c^e represents efferent (or outgoing) coupling, and c^a represents afferent (or incoming) coupling.

The Instability metric determines the *volatility* of a code base. A code base that exhibits high degrees of instability breaks more easily when changed because of high coupling. For example, if a class calls too many other classes to delegate work, the calling class will show high susceptibility to breakage if one or more of the called methods changes.

Distance from the Main Sequence

One of the few holistic metrics architects have for architectural structure is *Distance from the Main Sequence*, a derived metric based on Instability and Abstractness, shown in [Equation 3-5](#).

Equation 3-5. Distance from the Main Sequence

$$D = |A + I - 1|$$

In the equation, A = Abstractness and I = Instability.

Note that both Abstractness and Instability are fractions whose results will always fall between 0 and 1 (except in some extreme cases). Thus, graphing the relationship produces the graph in [Figure 3-2](#).

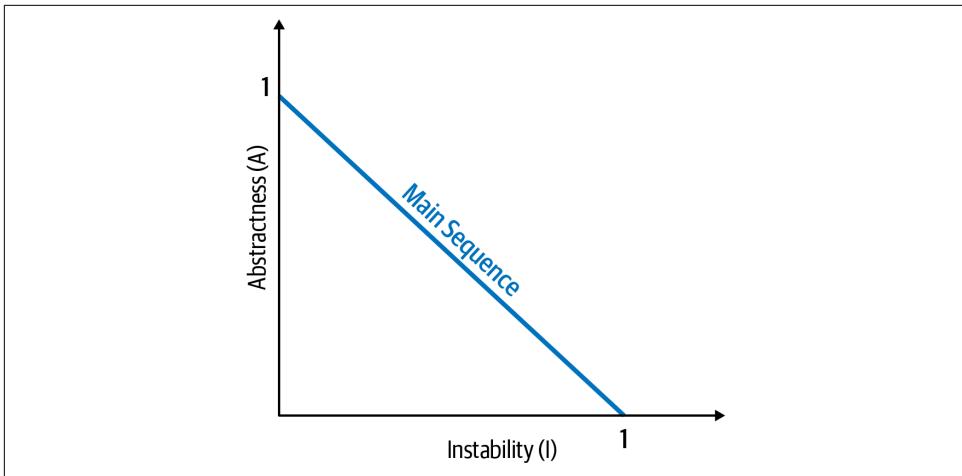


Figure 3-2. The main sequence defines the ideal relationship between Abstractness and Instability

The *Distance* metric imagines an ideal relationship between Abstractness and Instability; classes that fall near this idealized line exhibit a healthy mixture of these two competing concerns. For example, graphing a particular class allows developers to calculate the *Distance from the Main Sequence* metric, illustrated in Figure 3-3.

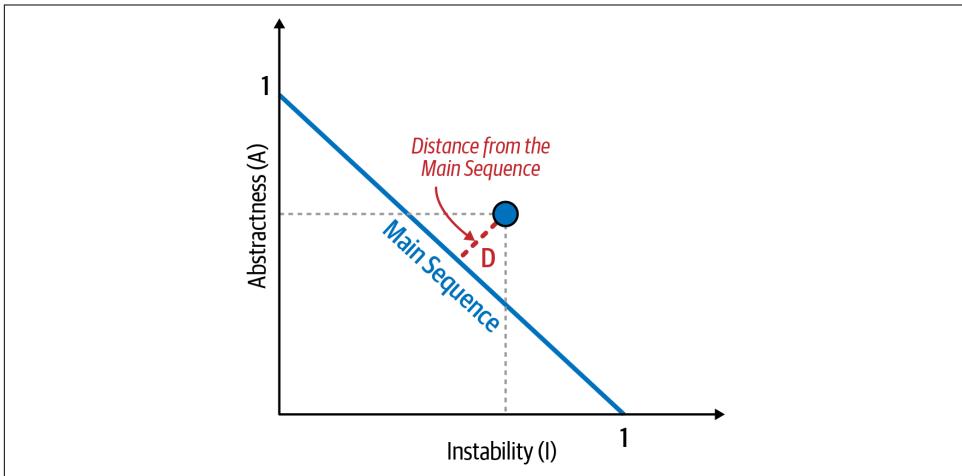


Figure 3-3. Normalized Distance from the Main Sequence for a particular class

The [Figure 3-3](#) metric graphs the candidate class, then measures its distance from the idealized line. The closer to the line, the better balanced the class. Classes that fall too far into the upper righthand corner enter what architects call the *Zone of Uselessness*: code that is too abstract becomes difficult to use. Conversely, code that falls into the lower lefthand corner, as illustrated in [Figure 3-4](#), enters the *Zone of Pain*: code with too much implementation and not enough abstraction becomes brittle and hard to maintain.

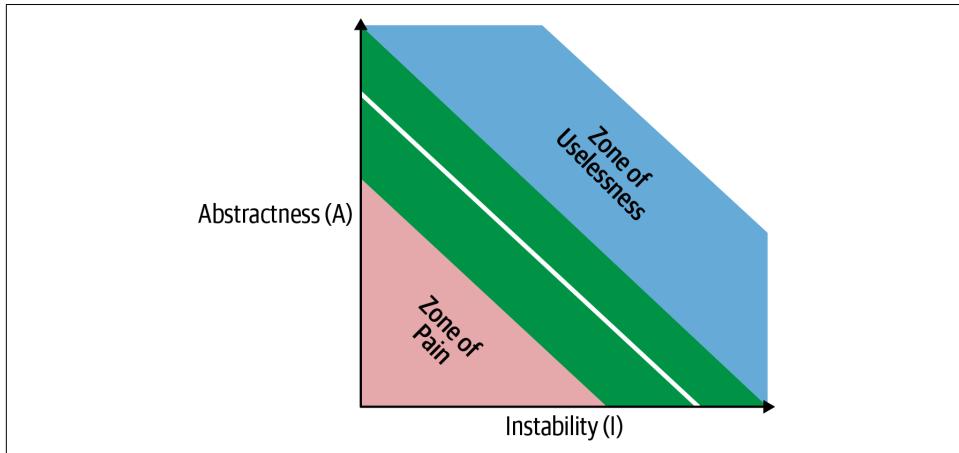


Figure 3-4. The Zones of Uselessness and Pain

Many platforms provide tools to calculate these measures, which assist architects when analyzing code bases to get familiar with them, prepare for a migration, or assess technical debt.

The Limitations of Metrics

While the industry has a few code-level metrics that provide valuable insight, our tools are extremely blunt compared to analysis tools in other engineering disciplines. Even metrics derived directly from the structure of code require interpretation. For example, Cyclomatic Complexity (see “[Cyclomatic Complexity](#)” on page 84) measures complexity in code bases, but this metric cannot distinguish between *essential* complexity (the code is complex because the underlying problem is complex) and *accidental complexity* (the code is more complex than it should be). Virtually all code-level metrics require interpretation, but it is still useful to establish baselines for critical metrics such as Cyclomatic Complexity so that architects can assess which type the code base exhibits. We discuss setting up just such tests in “[Governance and Fitness Functions](#)” on page 86.

Yourdon and Constantine's *Structured Design*, published in 1979, predates the popularity of object-oriented languages. It focuses instead on structured programming constructs, such as functions (not methods). It also defines other types of coupling that are outdated because of modern program language design. Object-oriented programming introduced additional concepts that overlay afferent and efferent coupling, including a more refined vocabulary to describe coupling called *connascence*.

Connascence

Meilir Page-Jones's book *What Every Programmer Should Know about Object-Oriented Design* (Dorset House, 1996) created a more precise *language* to describe different types of coupling in object-oriented languages. Connascence isn't a coupling metric like afferent and efferent coupling—rather, it represents a language that helps architects describe different types of coupling more precisely (and understand some common consequences of types of coupling).

Two components are *connascent* if a change in one would require the other to be modified in order to maintain the overall correctness of the system. Page-Jones distinguishes two types of connascence: *static* and *dynamic*.

Static connascence

Static connascence refers to source-code-level coupling (as opposed to execution-time coupling, covered in “[Dynamic connascence](#)” on page 50). Architects view static connascence as the *degree* to which something is coupled via either afferent or efferent coupling. There are several types of static connascence:

Connascence of Name

Multiple components must agree on the name of an entity.

Method names and method parameters are the most common way that code bases are coupled and the most desirable, especially in light of modern refactoring tools that make system-wide name changes trivial to implement. For example, developers no longer change the name of a method in an active code base but rather *refactor* the method name using modern tools, affecting the change throughout the code base.

Connascence of Type

Multiple components must agree on the type of an entity.

This type of connascence refers to the common tendency in many statically typed languages to limit variables and parameters to specific types. However, this capability isn't purely for statically typed languages—some dynamically typed languages also offer selective typing, notably [Clojure](#) and [Clojure Spec](#).

Connascence of Meaning

Multiple components must agree on the meaning of particular values. It is also called *Connascence of Convention*.

The most common obvious case for this type of connascence in code bases is hardcoded numbers rather than constants. For example, it is common in some languages to consider defining somewhere that `int TRUE = 1; int FALSE = 0`. Imagine the problems that would arise if someone flipped those values.

Connascence of Position

Multiple components must agree on the order of values.

This is an issue with parameter values for method and function calls, even in languages that feature static typing. For example, if a developer creates a method `void updateSeat(String name, String seatLocation)` and calls it with the values `updateSeat("14D", "Ford, N")`, the semantics aren't correct, even if the types are.

Connascence of Algorithm

Multiple components must agree on a particular algorithm.

A common case for *Connascence of Algorithm* occurs when a developer defines a security hashing algorithm that must run and produce identical results on both the server and client to authenticate the user. Obviously, this represents a high degree of coupling—if any details of either algorithm change, the handshake will no longer work.

Dynamic connascence

The other type of connascence Page-Jones defines is *dynamic connascence*, which analyzes calls at runtime. The types of dynamic connascence include:

Connascence of Execution

The order of execution of multiple components is important.

Consider this code:

```
email = new Email();
email.setRecipient("foo@example.com");
email.setSender("me@me.com");
email.send();
email.setSubject("whoops");
```

It won't work correctly because certain properties must be set in a specific order.

Connascence of Timing

The timing of the execution of multiple components is important.

The common case for this type of connascence is a race condition caused by two threads executing at the same time, affecting the outcome of the joint operation.

Connascence of Values

Several values depend on one another and must change together.

Consider a case where a developer has defined a rectangle by defining four points to represent its corners. To maintain the integrity of the data structure, the developer cannot randomly change one of the points without considering the impact on the other points to preserve the shape of the rectangle.

A more common and problematic case involves transactions, especially in distributed systems. In a system designed with separate databases, when someone needs to update a single value across all of the databases, the values must either all change together or not change at all.

Connascence of Identity

Multiple components must reference the same entity.

A common example of *Connascence of Identity* involves two independent components that must share and update a common data structure, such as a distributed queue.

Connascence properties

Connascence is an analysis framework for architects and developers, and some of its properties help ensure that we use it wisely. These connascence properties include:

Strength

Architects determine the *strength* of a system's connascence by the ease with which a developer can refactor its coupling. Some types of connascence are demonstrably more desirable than others, as shown in [Figure 3-5](#). Refactoring toward better types of connascence can improve the coupling characteristics of a code base.

Architects should prefer static connascence to dynamic because developers can determine it by simple source-code analysis, and because modern tools make it trivial to improve static connascence. For example, *Connascence of Meaning* could be improved by refactoring to *Connascence of Name*, creating a named constant rather than a magic value.

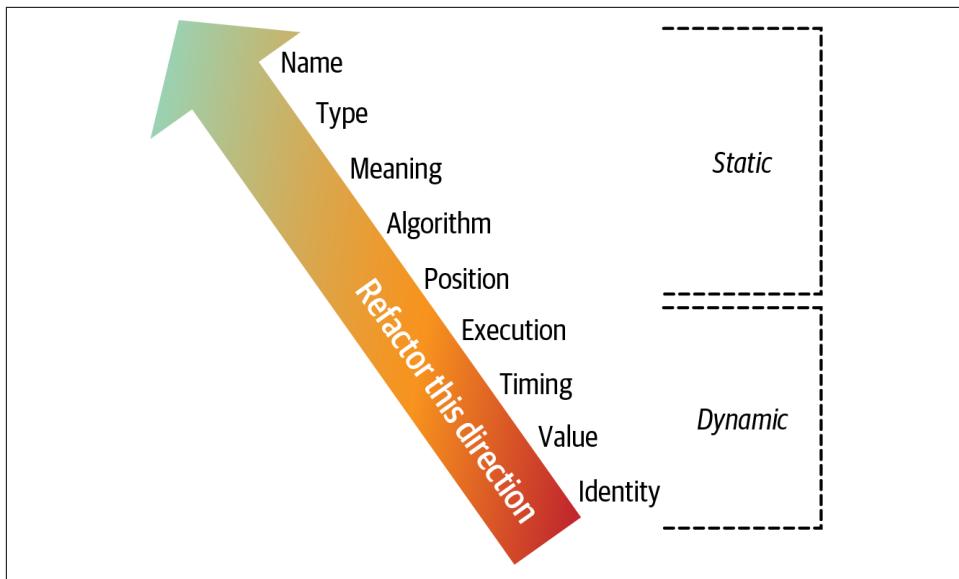


Figure 3-5. Connascence strength can be a good refactoring guide

Locality

The *locality* of a system's connascence measures how proximal (close) its modules are to each other in the code base. *Proximal code* (code in the same module) typically has more and higher forms of connascence than more separated code (in separate modules or code bases). In other words, forms of connascence that would indicate poor coupling when the components are far apart are fine when the components are closer together. For example, if two classes in the same module have *Connascence of Meaning*, it is less damaging to the code base than if those classes were in different modules.

Architects were largely unaware of the importance of this observation when the author first published it. In modern terms, he is suggesting that architects should limit the scope of implementation details (high coupling) as narrowly as practical, which is the same advice derived from domain-driven design's (DDD) bounded context idea. The architectural observation is the same—limit implementation coupling. Meilir-Page described a good design principle that was reintroduced more fully via DDD (see “[Domain-Driven Design’s Bounded Context](#)” on page 97 in Chapter 7).

It's a good idea to consider strength and locality together. Stronger forms of connascence within the same module represent less “code smell” than the same connascence spread apart.

Degree

The *degree* of connascence relates to the size of the impact of changing a class in a particular module—does that change impact a few classes or many? Lesser degrees of connascence require fewer changes to other classes and modules and hence damage code bases less. In other words, having high dynamic connascence isn't terrible if an architect only has a few modules. However, code bases tend to grow, making a small problem correspondingly bigger in terms of change.

In *What Every Programmer Should Know about Object-Oriented Design*, Page-Jones offers three guidelines for using connascence to improve system modularity:

- Minimize overall connascence by breaking the system into encapsulated elements.
- Minimize any remaining connascence that crosses encapsulation boundaries.
- Maximize connascence within encapsulation boundaries.

The legendary software architecture innovator [Jim Weirich](#), who repopularized the concept of connascence, offers two great rules from his talk on “[Connescence Examined](#)” during the Emerging Technologies for the Enterprise conference in 2012:

Rule of Degree: Convert strong forms of connascence into weaker forms of connascence.

Rule of Locality: As the distance between software elements increases, use weaker forms of connascence.

Architects benefit from learning about connascence for the same reason it's beneficial to learn about design patterns: connascence provides more precise language to describe different types of coupling. For example, an architect can tell someone, “We need a service and there can only be one instance,” or they can tell someone, “We need a Singleton service.” The Singleton design pattern nicely encapsulates the context and solution to a common problem with a simple name.

Similarly, when performing a code review, an architect can instruct a developer, “Don't add a magic string constant in the middle of a method declaration. Extract it as a constant instead.” Or they could say, “You have *Connascence of Meaning*; refactor it to *Connascence of Name*.”

From Modules to Components

We use the term *module* throughout this book as a generic name for bundles of related code. However, most architects refer to modules as *components*, the key building blocks for software architecture. This concept of a *component*, and the corresponding analysis of logical or physical separation, has existed since the earliest days of computer science, yet developers and architects still struggle to achieve good outcomes.

We'll discuss deriving components from problem domains in [Chapter 8](#), but we must first discuss another fundamental aspect of software architecture: architectural characteristics and their scope.

CHAPTER 4

Architectural Characteristics Defined

We now delve into the details of structural design, one of the key roles for software architects. This primarily consists of two activities: *architectural characteristics analysis*, covered in this chapter, and *logical component design*, covered in [Chapter 8](#). Architects may perform these two activities in any order (or even in parallel), but they come together at a critical join point.

When a company decides to solve a particular problem using software, it gathers a list of requirements for that system (there are many techniques for eliciting them, as covered in [Chapter 8](#)). We'll refer to these requirements as the *problem domain* (or just the *domain*) throughout the book. You learned in [Chapter 1](#) that *architecture characteristics* are the important aspects of a system that are independent from the problem domain and important to the system's success. In this chapter we'll dive deeper into defining that term, as well as specific architectural characteristics.

Architects often collaborate on defining the domain, but must also define, discover, and otherwise analyze all the things the software must do that isn't directly related to the domain functionality: *architectural characteristics*. Architects' role in defining architectural characteristics is part of what distinguishes software architecture from coding and design. They must also consider many other factors in designing a software solution, as illustrated in [Figure 4-1](#).

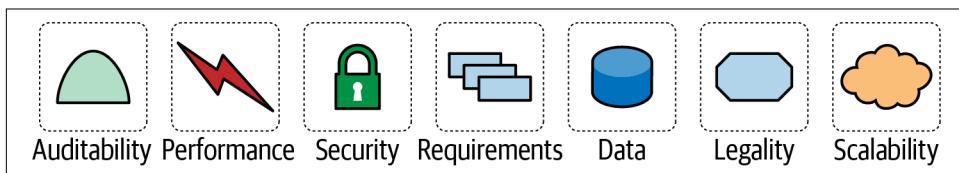


Figure 4-1. A software solution consists of both domain requirements and architectural characteristics

The Longevity of the Term “Non-Functional Requirements”

Many organizations describe architectural characteristics with a variety of terms, including *non-functional requirements*, a term created to distinguish architectural characteristics from *functional requirements*. We dislike that term because it is self-denigrating and has a negative impact from a language standpoint: how do you convince teams to pay enough attention to something that’s “non-functional”? Another popular term is *quality attributes*, which we dislike because it implies after-the-fact quality assessment rather than design.

We prefer the term *architectural characteristics* because it describes concerns that are critical to the success of the architecture, and therefore the system as a whole, without discounting the importance of these concerns. In *Head First Software Architecture* (O'Reilly, 2024), we refer to architectural characteristics as the *capabilities* of the system; in contrast, the domain represents the system's *behavior*.

Sometimes terms get “stuck,” and *non-functional requirements* seems to be a particularly sticky one among software architects. It's still common in many organizations. The term first started appearing in software engineering literature in the late 1970s, around the same time as *function point analysis*, an estimation technique that decomposes system requirements into “function points” that each represent some unit of work. Theoretically, teams could add up all the function points at the end of the analysis process and get some insight into the project. Sadly, it provided a veneer of certainty but was plagued by subjectivity, as many estimation schemes are, and is no longer in use.

However, one insight that remains with us from that time is that much of the effort involved to create a system is about that system's *capabilities* rather than its requirements. They called those efforts *non-function points*, which led to the term *non-functional requirements* becoming common.

Architectural Characteristics and System Design

To be considered an architectural characteristic, a requirement must meet three criteria. It must specify a nondomain design consideration, influence some structural aspect of the design, *and* be critical or important to the application's success. These interlocking parts of our definition are illustrated in [Figure 4-2](#), which consists of these three components and a few modifiers.

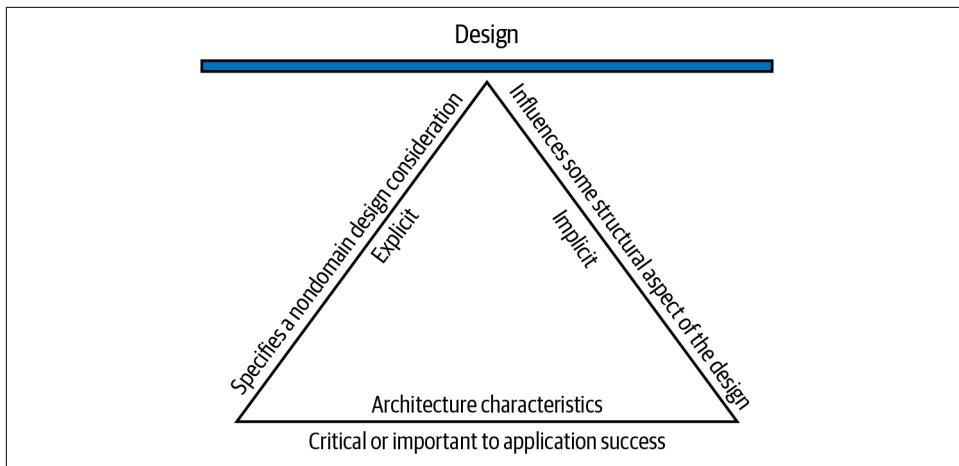


Figure 4-2. The differentiating features of architectural characteristics

Let's look more closely at these components:

An architecture characteristic specifies a nondomain design consideration.

Structural design in software architecture consists of two activities by an architect: understanding the problem domain and uncovering what kinds of capabilities the system needs to support to be successful. The domain design considerations cover the behavior of the system, and architectural characteristics define the capabilities. Taken together, these two activities define structural design.

While design requirements specify *what* the application should do, architectural characteristics specify *how* to implement the requirements and *why* certain choices were made: in short, the operational and design criteria for the project to succeed.

For example, specific levels of performance are often an important architectural characteristic, but often don't appear in requirements documents. Even more pertinent: no requirements document actually states that a design must "prevent technical debt," but it is a common design consideration. We cover this distinction between explicit and implicit characteristics in depth in "[Extracting Architectural Characteristics from Domain Concerns](#)" on page 67.

An architecture characteristic influences some structural aspect of the design.

The primary reason architects try to describe architectural characteristics on projects is to tease out important design considerations. Can the architect implement it via design, or does this architectural characteristic require special *structural* consideration to succeed?

For example, security is a concern in virtually every project, and all systems must take certain baseline precautions during design and coding. However, security rises to the level of an architectural characteristic when the architect determines that the architecture needs special structure to support it.

Consider two common architectural characteristics: security and scalability. Architects can accommodate security in a monolithic system by using good coding hygiene, including well-known techniques such as encryption, hashing, and salting. (Architectural fitness functions, which also fall under this umbrella, are discussed in [Chapter 6](#).) Conversely, in a distributed architecture such as microservices, the architect would build a more hardened service with stricter access protocols—a structural approach. Thus, architects can accommodate security via either design or structure. On the other hand, consider scalability: no amount of clever design will allow a monolithic architecture to scale beyond a certain point. Beyond that, the system must change to a distributed architectural style.

Architects pay close attention to operational architectural characteristics (discussed in [“Operational Architectural Characteristics” on page 59](#)) because they are the characteristics that most often require special structural support.

An architecture characteristic must be critical or important to application success.

Applications *can* support a huge number of architectural characteristics...but they shouldn't. Each architectural characteristic that a system supports adds complexity to its design. That's why architects should strive to choose the *fewest* possible architectural characteristics rather than the most.

We divide architectural characteristics into implicit versus explicit architectural characteristics. Implicit ones rarely appear in requirements, yet they're necessary for the project to succeed. Availability, reliability, and security underpin virtually all applications, yet they're rarely specified in design documents. Architects must use their knowledge of the problem domain to uncover these architectural characteristics during the analysis phase. For example, a high-frequency trading firm may not have to specify low latency in every system because the architects in that problem domain already know how critical it is. Explicit architectural characteristics appear in requirements documents or other specific instructions.

In [Figure 4-2](#), the choice of a triangle is intentional: each of the definition elements supports the others, which in turn support the overall design of the system. The fulcrum created by the triangle illustrates how these architectural characteristics often interact with one another. This is why architects use the term *trade-off* so much.

Architectural Characteristics (Partially) Listed

Architectural characteristics exist along a broad spectrum of complexity, ranging from low-level code characteristics (such as modularity) to sophisticated operational concerns (such as scalability and elasticity). There is no true universal standard, though people have attempted to codify one. Instead, each organization interprets these terms for itself. Additionally, because the software ecosystem changes so fast, new concepts, terms, measures, and verifications are constantly appearing, providing new opportunities to define architectural characteristics.

While the sheer volume and breadth of architecture characteristics make it hard to quantify them, architects do categorize them. The following sections describe a few such broad categories and provide some examples.

Operational Architectural Characteristics

Operational architectural characteristics cover capabilities such as performance, scalability, elasticity, availability, and reliability. **Table 4-1** lists some operational architectural characteristics.

Table 4-1. Common operational architectural characteristics

| Term | Definition |
|--------------------|---|
| Availability | How much of the time the system will need to be available; if that's 24/7, steps need to be in place to allow the system to be up and running quickly in case of any failure. |
| Continuity | The system's disaster recovery capability. |
| Performance | How well the system performs; ways to measure this include stress testing, peak analysis, analysis of the frequency of functions used, and response times. |
| Recoverability | Business continuity requirements: in case of a disaster, how quickly the system must get back online. This includes backup strategies and requirements for duplicate hardware. |
| Reliability/safety | Whether the system needs to be fail-safe, or if it is mission critical in a way that affects lives. If it fails, will it cost the company large sums of money? This is often a spectrum rather than a binary. |
| Robustness | The system's ability to handle error and boundary conditions while running, for example, if the internet connection or power fails. |
| Scalability | The system's ability to perform and operate as the number of users or requests increases. |

Operational architectural characteristics overlap heavily with operations and DevOps concerns.

Structural Architectural Characteristics

Architects are responsible for proper code structure. In many cases, the architect has sole or shared responsibility for the code's quality, including its modularity, its readability, how well coupling between components is controlled, readable code, and a host of other internal quality assessments. [Table 4-2](#) lists a few structural architectural characteristics.

Table 4-2. Structural architectural characteristics

| Term | Definition |
|-----------------------|---|
| Configurability | How easily end users can change aspects of the software's configuration through interfaces. |
| Extensibility | How well the architecture accommodates changes that extend its existing functionality. |
| Installability | How easy it is to install the system on all necessary platforms. |
| Leverageability/reuse | The extent to which the system's common components can be leveraged across multiple products. |
| Localization | Support for multiple languages on entry/query screens in data fields. |
| Maintainability | How easy it is to apply changes and enhance the system. |
| Portability | The system's ability to run on more than one platform (such as Oracle and SAP DB). |
| Upgradeability | How easy and quick it is to upgrade to a newer version on servers and clients. |

Cloud Characteristics

The software development ecosystem constantly changes and evolves; the most recent excellent example is the arrival of the cloud. When the first edition was published, cloud-based computing existed but wasn't pervasive. Now, most systems have some interaction with cloud-based systems in at least some capacity. A few of these considerations appear in [Table 4-3](#).

Table 4-3. Cloud provider architectural characteristics

| Term | Definition |
|-----------------------------------|---|
| On-demand scalability | The cloud provider's ability to scale up resources dynamically based on demand. |
| On-demand elasticity | The cloud provider's flexibility as resource demands spike; similar to scalability. |
| Zone-based availability | The cloud provider's ability to separate resources by computing zones to make for more resilient systems. |
| Region-based privacy and security | The cloud provider's legal ability to store data from various countries and regions. Many countries have laws governing where their citizens' data may reside (and often restricting it from storage outside their region). |

For this book's second edition, we've added a section to each of the architectural style chapters that describes how that style accommodates and facilitates cloud considerations.

Cross-Cutting Architectural Characteristics

While many architectural characteristics fall into easily recognizable categories, others fall outside them or defy categorization, yet form important design constraints and considerations. **Table 4-4** describes a few of these.

Table 4-4. Cross-cutting architectural characteristics

| Term | Definition |
|--------------------------|--|
| Accessibility | How easily all users can access the system, including those with disabilities like colorblindness or hearing loss. |
| Archivability | The system's constraints around archiving or deleting data after a specified period of time. |
| Authentication | Security requirements to ensure users are who they say they are. |
| Authorization | Security requirements to ensure users can access only certain functions within the application (by use case, subsystem, web page, business rule, field level, etc.). |
| Legal | The legislative constraints in which the system operates, such as data protection laws like GDPR or financial-records laws like Sarbanes-Oxley in the US, or any regulations regarding the way the application is to be built or deployed. This includes what reservation rights the company requires. |
| Privacy | The system's ability to encrypt and hide transactions from internal company employees, even DBAs and network architects. |
| Security | Rules and constraints about encryption in the database or for network communication between internal systems; authentication for remote user access, and other security measures. |
| Supportability | The level of technical support the application needs; to what extent logging and other facilities are required to debug errors in the system. |
| Usability/ achievability | The level of training required for users to achieve their goals with the application/solution. |

Any list of architectural characteristics will necessarily be incomplete; any software project may invent architectural characteristics based on unique factors. Many of the terms we've just listed are imprecise and ambiguous, sometimes because of subtle nuance or a lack of objective definitions. For example, *interoperability* and *compatibility* may appear to be equivalent, and that will be true for some systems. However, they differ because *interoperability* implies ease of integration with other systems, which in turn implies published, documented APIs. *Compatibility*, on the other hand, is more concerned with industry and domain standards. Another example is *learnability*: one definition is “how easy it is for users to learn to use the software,” and another definition is “the level at which the system can automatically learn about its environment in order to become self-configuring or self-optimizing using machine learning algorithms.”

Many definitions overlap: *availability* and *reliability*, for instance. Yet consider the internet protocol IP, which underlies TCP. IP is *available* but not *reliable*: packets may arrive out of order, and the receiver may have to ask for missing packets again.

There is no complete list of standards defining these categories. The International Organization for Standards (ISO) publishes a [list organized by capabilities](#) that overlaps with our list here, but mainly establishes an incomplete category list. Here are some of the ISO definitions, reworded to update terms and add categories to align with modern concerns:

Performance efficiency

Measure of the performance relative to the amount of resources used under known conditions. This includes *time behavior* (measure of response, processing times, and/or throughput rates), *resource utilization* (amounts and types of resources used), and *capacity* (degree to which the maximum established limits are exceeded).

Compatibility

Degree to which a product, system, or component can exchange information with other products, systems, or components and/or perform its required functions while sharing the same hardware or software environment. It includes *coexistence* (can perform its required functions efficiently while sharing a common environment and resources with other products) and *interoperability* (degree to which two or more systems can exchange and utilize information).

Usability

Users can use the system effectively, efficiently, and satisfactorily for its intended purpose. It includes *appropriateness recognizability* (users can recognize whether the software is appropriate for their needs), *learnability* (how easily users can learn how to use the software), *user error protection* (protection against users making errors), and *accessibility* (make the software available to people with the widest range of characteristics and capabilities).

Reliability

Degree to which a system functions under specified conditions for a specified period of time. This characteristic includes subcategories such as *maturity* (does the software meet the reliability needs under normal operation), *availability* (software is operational and accessible), *fault tolerance* (does the software operate as intended despite hardware or software faults), and *recoverability* (can the software recover from failure by recovering any affected data and reestablish the desired state of the system).

Security

Degree to which the software protects information and data so that people or other products or systems have the degree of data access appropriate to their types and levels of authorization. This family of characteristics includes *confidentiality* (data is accessible only to those authorized to have access), *integrity* (the software prevents unauthorized access to or modification of software or data),

nonrepudiation (can actions or events be proven to have taken place), *accountability* (can user actions of a user be traced), and *authenticity* (proving the identity of a user).

Maintainability

Represents the degree of effectiveness and efficiency to which developers can modify the software to improve it, correct it, or adapt it to changes in environment and/or requirements. This characteristic includes *modularity* (degree to which the software is composed of discrete components), *reusability* (degree to which developers can use an asset in more than one system or in building other assets), *analyzability* (how easily developers can gather concrete metrics about the software), *modifiability* (degree to which developers can modify the software without introducing defects or degrading existing product quality), and *testability* (how easily developers and others can test the software).

Portability

Degree to which developers can transfer a system, product, or component from one hardware, software, or other operational or usage environment to another. This characteristic includes the subcharacteristics of *adaptability* (can developers effectively and efficiently adapt the software for different or evolving hardware, software, or other operational or usage environments), *installability* (can the software be installed and/or uninstalled in a specified environment), and *replaceability* (how easily developers can replace the functionality with other software).

The last item in the ISO list addresses the functional aspects of software:

Functional suitability

This characteristic represents the degree to which a product or system provides functions that meet stated and implied needs when used under specified conditions. This characteristic is composed of the following subcharacteristics:

Functional completeness

Degree to which the set of functions covers all the specified tasks and user objectives.

Functional correctness

Degree to which a product or system provides the correct results with the needed degree of precision.

Functional appropriateness

Degree to which the functions facilitate the accomplishment of specified tasks and objectives.

However, we do not believe that functional suitability belongs in this list. It does not describe architectural characteristics but rather the motivational requirements to build the software. This illustrates how thinking about the relationship between architectural characteristics and the problem domain has evolved. We cover this evolution in [Chapter 7](#).

The Many Ambiguities in Software Architecture

A source of consistent frustration among architects is the lack of clear definitions for so many critical things, including the activity of software architecture itself! The absence of a standard leads companies to define their own terms for common things. This often leads to industry-wide confusion, because architects either use opaque terms or, worse yet, the same terms for wildly different meanings.

As much as we'd like, we can't impose a standard nomenclature on the software development world. However, to help avoid terminology-based misunderstandings, domain-driven design advises organizations to establish and use a *ubiquitous language* among employees. We follow and recommend that advice.

Trade-Offs and Least Worst Architecture

We said earlier that architects should support only those architectural characteristics that are critical or important to the system's success. Systems can only support a few of the architectural characteristics we've listed, for a variety of reasons. First, support for an architectural characteristic is rarely free. Each supported characteristic requires design effort from the architect, effort from developers to implement and maintain it, and perhaps structural support.

Second, architectural characteristics are synergistic with each other *and* the problem domain. As much as we would prefer otherwise, each design element interacts with all the others. For example, taking steps to improve *security* will almost certainly negatively impact *performance*: the application must do more on-the-fly encryption, indirection (for hiding secrets), and other activities that can degrade performance. Airplane pilots often struggle when learning to fly helicopters, which have controls for each hand and each foot. Changing one control impacts all of the others because they are synergistic. Flying a helicopter is a balancing exercise, which nicely describes the trade-off process when choosing architectural characteristics; they are similarly synergistic with both other architectural characteristics and the domain design: changing one often entails changing another. Like our belabored helicopter pilot, architects must learn to juggle interlocking items.

Third, as we discussed previously, the lack of standard definitions for architectural characteristics means that organizations struggle with ambiguity. While the industry

will never be able to create an immutable list of architectural characteristics (because new ones constantly appear), each organization can create its own list (or *ubiquitous language*) with objective definitions.

Finally, not only is the number of architectural characteristics constantly increasing, but the number of *categories* has increased over the last decade as well. For example, a few decades ago, architects cared little for operational concerns, which were considered a separate “black box.” However, with the increased popularity of architectures like microservices, architects and operations must collaborate more intensely and frequently. As software architecture becomes more complex, it tends to entangle with other parts of the organization.

Thus, it’s very rare for architects to be able to design a system and maximize every single architectural characteristic. More often, the decisions come down to trade-offs between several competing concerns.



Never strive for the *best* architecture; aim for the *least worst* architecture.

Trying to support too many architectural characteristics leads to generic solutions that attempt to solve every business problem. The design quickly becomes unwieldy, so such architectures rarely work.

Strive to design architecture that is as iterative as possible. The easier it is to change the architecture, the less everyone needs to stress about discovering the exact correct thing in the first attempt. One of the most important lessons of Agile software development is the value of iteration; this holds true at all levels of software development, including architecture.

Identifying Architectural Characteristics

To create an architecture or determine the validity of an existing architecture, architects must analyze two things: architectural characteristics and domain. As you learned in [Chapter 4](#), identifying the correct architectural characteristics (“-ilities”) for a given problem or application requires not only understanding the domain problem, but collaborating with stakeholders to determine what is truly important from a domain perspective.

There are at least three places to uncover what architectural characteristics a project needs: the domain concerns, the project requirements, and your implicit domain knowledge. In addition to generic implicit architectural characteristics, such as security and modularity, we've noted that some domains also include implicit architectural characteristics. For example, an architect working on medical software that reads from diagnostics equipment should already understand the importance of data integrity and the potential consequences of losing messages. Architects who work in that domain internalize data integrity, so it becomes implicit in every solution they design.

Extracting Architectural Characteristics from Domain Concerns

Most architectural characteristics come from listening to key domain stakeholders and collaborating with them to determine what is important from a business perspective. While this may seem straightforward, the problem is that architects and domain stakeholders speak different languages. Architects talk about scalability, interoperability, fault tolerance, learnability, and availability; domain stakeholders talk about mergers and acquisitions, user satisfaction, time to market, and competitive advantage. What happens is a “lost in translation” problem where the architect and domain

stakeholder don't understand each other. Architects have no idea how to create an architecture to support user satisfaction, and domain stakeholders don't understand why architects focus so much on the application's availability, interoperability, learnability, and fault tolerance.

Fortunately, it's possible to "translate" from domain concerns into the language of architectural characteristics. [Table 5-1](#) shows some of the more common domain concerns and the corresponding "-ilities" that support them. Understanding the key domain goals allows an architect to translate those domain concerns to "-ilities," which then forms the basis for correct and justifiable architecture decisions. For example, is the domain concern of time to market more important than scalability, or should the architect focus on fault tolerance, security, or performance? Perhaps the system requires all these characteristics combined.

Table 5-1. Translating domain concerns into architectural characteristics

| Domain concern | Architectural characteristics |
|--------------------------|---|
| Mergers and acquisitions | Interoperability, scalability, adaptability, extensibility |
| Time to market | Agility, testability, deployability |
| User satisfaction | Performance, availability, fault tolerance, testability, deployability, agility, security |
| Competitive advantage | Agility, testability, deployability, scalability, availability, fault tolerance |
| Time and budget | Simplicity, feasibility |

Composite Architectural Characteristics

When translating domain concerns into architectural characteristics, one common pitfall is making false equivalences, such as equating *agility* solely with *time to market*. Agility is an example of a *composite* architectural characteristic: one that has no single objective definition but rather is composed of other measurable things. There's no measure for agility, so architects must ask: what is agility composed of? It includes things like *deployability*, *modularity*, and *testability*, all of which are measurable.

A common antipattern arises when architects focus narrowly on just one part of a composite, often for convenience. This is like forgetting to put the flour in the cake batter. For example, a domain stakeholder might say something like, "Due to regulatory requirements, it is absolutely imperative that we complete end-of-day fund pricing on time."

An ineffective architect might respond by focusing solely on performance, because that seems to be the primary focus of the domain concern. However, that approach will fail, for many reasons:

- It doesn't matter how fast the system is if it isn't available when needed.
- As the domain grows and more funds are created, the system must be able to scale to finish end-of-day processing in time.
- The system must not only be available, it must also be reliable so that it doesn't crash as end-of-day fund prices are being calculated.
- What happens if the system crashes while end-of-day fund pricing is 85% complete? It must be able to recover and restart the pricing where it left off.
- Finally, the system may be fast, but are the fund prices being calculated correctly?

So, in addition to performance, this architect must focus equally on availability, scalability, reliability, recoverability, and auditability.

Many business goals start as composite architectural characteristics. Decomposing them, and giving the resulting characteristics objective definitions, is part of the architects' job. (We'll see the importance of this in [Chapter 6](#).)

Extracting Architectural Characteristics

Most architectural characteristics come from explicit statements in some form of requirements document. For example, lists of domain concerns commonly include explicit numbers of expected users and scale. Other characteristics come from architects' inherent domain knowledge (one of many reasons every architect should know their domain).

For example, suppose you're an architect designing an application that handles class registration for university students. To make the math easy, assume that the school has 1,000 students and 10 hours for registration. Should you make the implicit assumption that the students will distribute themselves evenly over those 10 hours, and design the system to handle a consistent scale? Or, based on your knowledge of university students' habits and proclivities, should you design a system that can handle all 1,000 students attempting to register in the last 10 minutes?

Anyone who understands how much students stereotypically procrastinate knows the answer to this question! Rarely will details like this appear in requirements documents—yet they inform design decisions.

The Origin of Architecture Katas

Over a decade ago, Ted Neward, a well-known architect, devised *architecture katas*, a clever method to allow nascent architects a way to practice deriving architectural characteristics from domain-targeted descriptions. The word *kata* derives from Japanese, where it refers to an individual martial arts training exercise that emphasizes proper form and technique.

How do we get great designers? Great designers design, of course.

—Fred Brooks

Big architectural projects take time, and it's common for architects to design perhaps half a dozen systems in their careers. So how are we supposed to get great architects? The key is giving aspiring architects opportunities to practice their craft. To provide a curriculum, Ted created the first architecture katas site, which your authors Neal and Mark adapted and updated on the companion site for this book at fundamentals-of-software-architecture.com. True to their original purpose, architecture katas provide a useful laboratory for aspiring architects.

Working with Katas

The basic premise is that each kata exercise provides a problem, stated in domain terms, a set of requirements, and some additional context (things that might not appear in the requirements, yet could influence the design). Small teams work on a design (consisting of architectural characteristics analysis and diagrams) for a set time. Then the groups share their results and vote on who came up with the best architecture.

Each kata has predefined sections:

Description

The overall domain problem the system is trying to solve.

Users

The expected number and/or types of users of the system.

Requirements

Domain requirements (as expected from domain users and experts).

Additional context

The authors updated the kata format on the previously mentioned site with an *additional context* section with important considerations, making the exercises more realistic.

We encourage readers to use the site to do their own kata exercise. Anyone can host a brown-bag lunch where a team of aspiring architects can solve a problem.

An experienced architect can evaluate the design and trade-off analysis, discussing missed trade-offs and alternative designs either on the spot or after the fact. The designs won't be elaborate because the exercise is timeboxed.

Next, we'll use an architecture kata to illustrate how architects derive architectural characteristics from requirements. Welcome to the *Silicon Sandwiches* kata.

Kata: Silicon Sandwiches

Description

A national sandwich shop wants to enable online ordering in addition to its current call-in service.

Users

Thousands, perhaps one day, millions.

Requirements

- Allow users to place an order and, if the shop offers delivery service, select pickup or delivery.
- Give pickup customers a time to pick up their order and directions to the shop (which must integrate with several external mapping services that include traffic information).
- For delivery service, dispatch the driver with the order to the user.
- Provide mobile device accessibility.
- Offer national daily promotions and specials.
- Offer local daily promotions and specials.
- Accept payment online, at the shop, or upon delivery.

Additional context

- Sandwich shops are franchised, each with a different owner.
- The parent company has near-future plans to expand overseas.
- The corporate goal is to hire inexpensive labor to maximize profit.

Given this scenario, how would you derive architectural characteristics? Each part of the requirement might contribute to one or more aspects of the architecture (and many will not). The architect doesn't design the entire system here—considerable effort must still go into crafting code to solve the domain design (covered in [Chapter 8](#)). Instead, look for things that influence or impact the design, particularly structural.

First, separate the candidate architectural characteristics into explicit and implicit characteristics.

Explicit Characteristics

Explicit architectural characteristics appear in a requirements specification as part of the necessary design. For example, a shopping website may aspire to support a particular number of concurrent users, which domain analysts specify in the requirements. Consider each part of the requirements to see if it contributes to an architectural characteristic. But first, consider domain-level predictions about expected metrics, as represented in the Users section of the kata.

One of the first details that should catch your eye is the number of users: currently thousands, perhaps one day, millions (this is a very ambitious sandwich shop!). Thus, *scalability*—the ability to handle a large number of concurrent users without serious performance degradation—is one of the top architectural characteristics. Notice that the problem statement didn’t explicitly ask for scalability, but rather expressed that requirement as an expected number of users. Architects must often decode domain language into engineering equivalents.

You’ll also probably need *elasticity*—the ability to handle bursts of requests. These two characteristics are often lumped together, but they have different constraints. Scalability looks like the graph shown in [Figure 5-1](#).

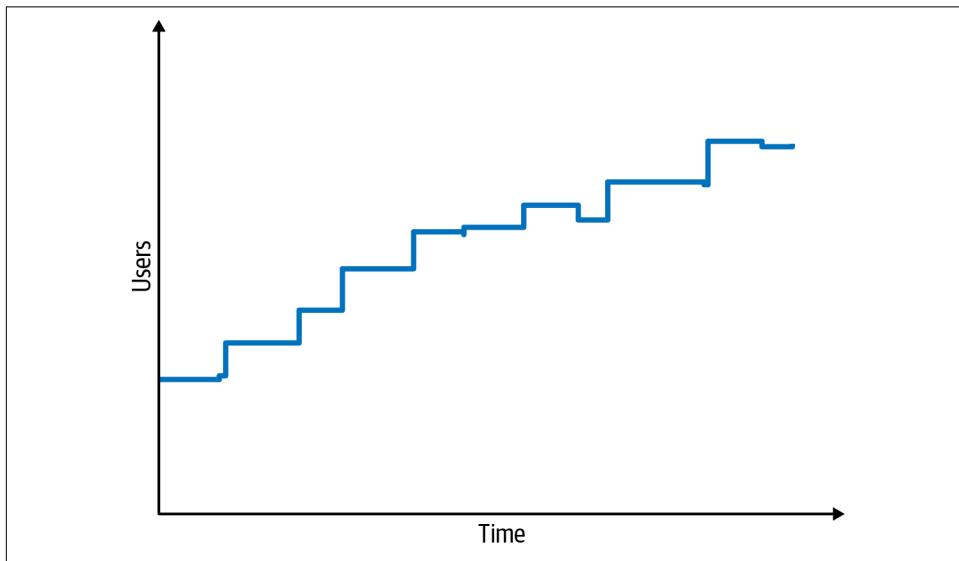


Figure 5-1. Scalability is a measure of the increase of users over time

Elasticity, on the other hand, measures bursts of traffic, as shown in [Figure 5-2](#).

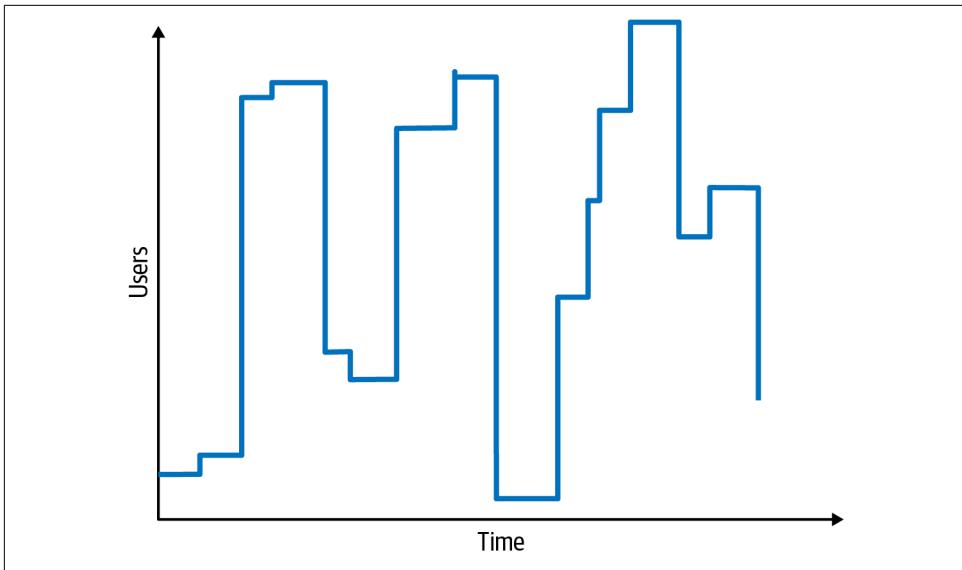


Figure 5-2. Elastic systems must withstand bursts of users

Some systems are scalable but not elastic. For example, a hotel reservation system's number of users, absent special sales or events, is probably predictably seasonal. In contrast, consider a concert-ticket booking system. As new tickets go on sale, fervent fans will flood the site, requiring high degrees of elasticity. Often, elastic systems also need *scalability*: the ability to handle bursts and high numbers of concurrent users.

Elasticity does not appear in the Silicon Sandwiches requirements, but you should still identify it as an important consideration. Requirements sometimes state architectural characteristics outright, but others lurk inside the problem domain. Is a sandwich shop's traffic consistent throughout the day, or does it get bursts of traffic around mealtimes? Almost certainly the latter, so identify this potential architectural characteristic.

Next, consider each of these business requirements in turn to see if it calls for architectural characteristics:

Users place their order, and, if the shop offers delivery service, select pickup or delivery.
No special architectural characteristics seem necessary to support this requirement.

Pickup customers are given a time to pick up their sandwich and directions to the shop (which must provide the option to integrate with external mapping services that include traffic information).

External mapping services imply integration points, which may impact aspects such as reliability. For example, say a developer builds a system that relies on

a third-party system. If those calls fail, it impacts the reliability of the calling system. Conversely, be wary of overspecifying architectural characteristics. What if the external traffic service is down? Should the Silicon Sandwiches site fail, or should it just offer slightly less efficiency without traffic information? Architects should always guard against building unnecessary brittleness or fragility into designs.

For delivery service, dispatch the driver with the order to the user.

No special architectural characteristics seem necessary to support this requirement.

Provide mobile device accessibility.

This requirement will primarily affect the user experience (UX) design of the application, and points toward building either a portable web application or several native web applications. Given the budget constraints and simplicity of the application, it would likely be overkill to build multiple applications, so the design points toward a mobile-optimized web application. Thus, you may want to define some specific performance-related architectural characteristics to optimize page load time and other mobile-sensitive characteristics.

You shouldn't act alone in situations like this. Collaborate with UX designers, domain stakeholders, and other interested parties to vet such decisions. For example, the business may require some behavior that's only possible if you build native applications.

Offer national daily promotions and specials; offer local daily promotions and specials.

Both of these requirements specify customizability across both promotions and specials. Requirement 1 also implies customized traffic information based on the user's address. Based on both of these requirements, you might consider customizability as an architectural characteristic. For example, the microkernel architectural style (covered in [Chapter 13](#)) supports customized behavior extremely well by defining a plug-in architecture. In this case, the default behavior appears in the core, and developers write the optional location-based custom parts as plug-ins. However, a traditional design can also accommodate this requirement via design patterns (such as the Template Method). This conundrum is common in architecture and requires architects to constantly weigh trade-offs between competing options. We discuss particular trade-offs in more detail in [“Design Versus Architecture and Trade-Offs” on page 76](#).

Accept payment online, at the shop, or upon delivery.

Online payments imply a need for security, but nothing in this requirement suggests a particularly high level of security beyond what's implicit. Architects can handle security with either design or architecture, making it a minimal architectural characteristics concern in this application.

Sandwich shops are franchised, each with a different owner.

This requirement may impose cost restrictions on the architecture. Check the feasibility, applying constraints like cost, time, and staff skill sets, to see if a simple or **sacrificial architecture** is warranted.

The parent company has near-future plans to expand overseas.

This requirement implies *internationalization* (often abbreviated as “i18n”). Many design techniques exist to handle this requirement, which shouldn’t require special structure to accommodate. This will, however, certainly drive UX decisions.

The corporate goal is to hire inexpensive labor to maximize profit.

This requirement suggests that usability will be important, but it, too, is more concerned with design than architectural characteristics.

The third architectural characteristic we can derive from the preceding requirements is *performance*: no one wants to buy from a sandwich shop that has poor performance, especially at peak times. However, *performance* is a nuanced concept. What kind of performance should you design for? (We cover the various nuances of performance in [Chapter 6](#).)

It’s also important to define performance numbers in conjunction with scalability numbers. In other words, establish a baseline of performance without a particular scale, and determine an acceptable level of performance given a certain number of users. Quite often, architectural characteristics interact, forcing architects to define them in relation to one another.

Implicit Characteristics

Many architectural characteristics aren’t specified in requirements documents, yet they make up important aspects of the design. One implicit architectural characteristic the system might want to support is *availability*: making sure users can access the site. Closely related to availability is *stability*: making sure the site stays up during interactions. No one wants to purchase from a site that drops connections, forcing them to log in again.

Security appears as an implicit characteristic in every system: no one wants to create insecure software. However, you might give it different priority depending on how critical it is, which illustrates the interlocking nature of our definition. You can consider security an architectural characteristic if it influences some structural aspect of the design and is critical or important to the application.

For Silicon Sandwiches, you might assume that payments should be handled by a third party. Thus, as long as developers follow general security hygiene (not passing credit card numbers as plain text, not storing too much information, and so on), good design in the application will suffice; you shouldn’t need any special structural design to accommodate security. Remember, architectural characteristics

are *synergistic*—each architectural characteristic interacts with the others. This is why overspecifying architectural characteristics is such a common pitfall. Overspecifying is just as damaging as underspecifying characteristics because it overcomplicates the system design.

The last major architectural characteristic that Silicon Sandwiches needs to support, *customizability*, is composed of several details from the requirements. Several parts of the problem domain offer custom behavior: recipes, local sales, and directions that may be locally overridden. Thus, the architecture should support custom behavior. Normally, this would fall under application design. However, as our definition specifies, when part of the problem domain relies on custom structure to support it, it moves into the realm of an architectural characteristic. This design element isn't critical to the application's success, though. Remember, there are no correct answers in choosing architectural characteristics, only incorrect ones—or:

There are no wrong answers in architecture, only expensive ones.

—One of Mark's famous quotes

Design Versus Architecture and Trade-Offs

In the Silicon Sandwiches kata, you would likely identify customizability as a part of the system, but the question then becomes: architecture or design? The architecture implies some structural component, whereas design resides within the architecture. For Silicon Sandwiches, you could choose an architecture style like microkernel and build structural support for customization. However, if you choose another style because of competing concerns, developers could implement the customization using the Template Method design pattern, which allows parent classes to define workflows that can be overridden in child classes. Which option is better?

Like everything in architecture, it depends. First, are there good reasons *not* to implement a microkernel architecture (such as performance and coupling)? Second, are other desirable architectural characteristics more difficult in one design versus the other? Third, how much would it cost to support all the architectural characteristics in the architecture versus in the design? This type of trade-off analysis is an important part of an architect's role.

Above all, it is critical to collaborate with the developers, project manager, operations team, and other co-constructors of the software system. No architecture decision should be made in isolation from the implementation team (which leads to the dreaded Ivory Tower architecture antipattern). In the case of Silicon Sandwiches, the architect, tech lead, developers, and domain analysts should collaborate to decide how best to implement customizability.

You shouldn't stress too much about discovering the exactly correct set of architectural characteristics. Developers can implement functionality in a variety of ways. However, correctly identifying important structural elements may facilitate a simpler or more elegant design. You could design an architecture that doesn't accommodate customizability structurally, instead requiring the design of the application itself to support that behavior (see [“Design Versus Architecture and Trade-Offs” on page 76](#)). Remember: there is no best design in architecture, only a least worst collection of trade-offs.

Prioritize these architectural characteristics when trying to find the simplest required sets. Once the team has made a first pass at identifying the architectural characteristics, a useful exercise is to try to determine the least important one. If you had to eliminate one, which would it be? Generally, architects are more likely to cull the explicit architectural characteristics, since many of the implicit ones support general success. The way we define what's critical or important to success assists in determining if the application *truly requires* each architectural characteristic. Attempting to determine the least applicable one can help you determine critical necessity.

In the case of Silicon Sandwiches, which of the architectural characteristics we've identified is least important? Again, no absolute correct answer exists. However, in this case, the solution could lose either customizability or performance. We could eliminate customizability as an architectural characteristic and plan to implement that behavior as part of application design. Of the operational architectural characteristics, performance is likely the least critical for success. Of course, that doesn't mean the developers are setting out to build an application that has terrible performance; it simply means this design doesn't prioritize performance over other characteristics, such as scalability or availability.

Limiting and Prioritizing Architectural Characteristics

One tip when collaborating with domain stakeholders to define the driving architectural characteristics: work hard to keep the final list as short as possible. A common antipattern entails trying to design a *generic architecture*: one that supports *all* the architectural characteristics. Each characteristic the architecture supports complicates the overall system design, so supporting too many architectural characteristics leads to greater and greater complexity. And that's before the architect and developers have even started addressing the problem domain—the original motivation for writing the software. Don't obsess over the number of characteristics, but remember the motivation: keeping the design simple.

Case Study: The *Vasa*

The original story of overspecifying architectural characteristics to the point of ultimately killing a project must be the *Vasa*. It was a Swedish warship built between 1626 and 1628 by a king who wanted the most magnificent ship ever created. Up until that time, ships were either troop transports or gunships—the *Vasa* would be both. Most ships had one deck—the *Vasa* had two! All of its cannons were twice the size of those on similar ships. Despite some trepidation, the expert shipbuilders couldn't say no to King Adolphus.

To celebrate finishing construction, the *Vasa* sailed out into Stockholm harbor and shot a cannon salute off one side. Unfortunately, because the ship was top-heavy, it capsized and sank to the bottom. In 1961, salvagers rescued the ship, which now resides in a museum in Stockholm.

Architect and business stakeholder collaboration is crucial during architectural characteristics analysis. However, if the architect provides an extensive list of possible architectural characteristics and asks the business stakeholders which ones they want the architecture to support, what will the answer be every time? “*All of them!*”

Thus, architects need a technique for determining which architectural characteristics drive structural decisions and are critical to success. The authors have developed a number of techniques to facilitate this effort, including an architectural characteristics “worksheet,” illustrated in [Figure 5-3](#) (and downloadable at <https://developertoarchitect.com/resources.html>).

The worksheet is meant to be used in an interactive session managed by an architect, who solicits stakeholders’ opinions as to the number and details of the required architectural characteristics. On the left, notice the seven slots to list the desired architectural characteristics.

Why seven? Why not? Seriously, an architect needs to restrict the list to some reasonable number—six or eight would also work (although there is some interesting psychology behind [the number seven](#)). The second column includes some implicit architectural characteristics; these appear in most systems, but sometimes architects prioritize them as driving concerns of the application that require special design and consideration. In those cases, the architect “pulls” the implicit characteristics into the first column. If the first column fills and a better item appears to displace an existing one, the architect moves it to the “Others Considered” category.

| Architecture characteristics worksheet | | |
|--|---------------------------------|--|
| System/project: _____ | | |
| Architect/team: _____ | | |
| Top 3 Driving characteristics | Implicit characteristics | |
| <input type="checkbox"/> _____ | Feasibility (cost/time) | |
| <input type="checkbox"/> _____ | Security | |
| <input type="checkbox"/> _____ | Maintainability | |
| <input type="checkbox"/> _____ | Observability | |
| <input type="checkbox"/> _____ | Others considered | |
| <input type="checkbox"/> _____ | | |
| <input type="checkbox"/> _____ | | |

Figure 5-3. Architectural characteristics worksheet

The last step is to collaboratively choose the top three highest-priority architectural characteristics, in any order (check the box next to each one). This exercise allows architects to derive a shortened, prioritized list of driving forces they can use to drive design decisions and trade-off analysis.

Many architects and domain stakeholders want to prioritize getting to a final list of architectural characteristics that the application or system must support. While this is certainly a desirable outcome, in most cases, it is a fool's errand that will not only waste time, but produce a lot of unnecessary frustration and disagreement with the key stakeholders. Rarely will all stakeholders agree on the priority of each and every characteristic. A better approach is to have the domain stakeholders select the top three most important characteristics from the final list (in any order). This makes it much easier to gain consensus and fosters discussions about what is most important. All of this helps the architect analyze trade-offs when making vital architectural decisions.

Measuring and Governing Architecture Characteristics

Architects must deal with an extraordinarily wide variety of architecture characteristics across all different aspects of software projects. Operational aspects like performance, elasticity, and scalability commingle with structural concerns, such as modularity and deployability. It benefits architects to understand how to measure and govern architectural characteristics, rather than drown in ambiguous terms and broad definitions. This chapter focuses on concretely defining some of the more common architecture characteristics and discusses how to build governance mechanisms for them.

Measuring Architecture Characteristics

Architects struggle to define architectural characteristics for a number of reasons:

They aren't physics

Many architecture characteristics in common usage have vague meanings. For example, how does an architect design for *agility* or *deployability*? What about *wicked fast performance*? People around the industry have wildly differing perspectives on common terms—sometimes driven by legitimate differing contexts, sometimes accidental.

Wildly varying definitions

Even within the same organization, different departments may disagree on the definitions of critical characteristics such as *performance*. Until developers, architects, operations, and others can unify on a common definition, how can they have a proper conversation?

Too composite

Many desirable architecture characteristics are really collections of other characteristics at a smaller scale, as you might remember from our discussion of composite architectural characteristics in [Chapter 5](#). For example, agility breaks down into characteristics such as modularity, deployability, and testability.

Decomposing composite architectural characteristics into their constituent parts is an important part of establishing objective definitions for architecture characteristics, which solves all three of these problems.

When an organization agrees that everyone will use standard, concrete definitions for architecture characteristics, they create a *ubiquitous language* around architecture. This standardization allows them to unpack composite characteristics to uncover *objectively measurable* features.

Operational Measures

Many architecture characteristics have obvious direct measurements, such as performance or scalability. Yet even these offer many nuanced interpretations, depending on the team's goals. For example, perhaps your team measures the average response time for certain requests—a good example of a measure for an operational architecture characteristic. But if your team only measures the average, what happens if some boundary condition causes 1% of requests to take 10 times longer than others? If the site has enough traffic, the outliers may not even show up. To catch outliers, you might also want to measure the maximum response times.

High-level teams don't just establish hard performance numbers; they base their definitions on statistical analysis. For example, say a video streaming service wants to monitor scalability. Rather than set an arbitrary number as the goal, the engineers measure the scale over time and build statistical models, then raise alarms if the real-time metrics fall outside the prediction models. If they do, the failure can mean two things: the model is incorrect (which teams like to know) or something is amiss (which teams also like to know).

The kinds of characteristics that teams measure evolve rapidly in conjunction with tools, targets, devices, and capabilities. For example, recently many teams have focused on performance budgets for metrics such as *first contentful paint* and *first CPU idle*, both of which speak volumes about performance issues for web page users on mobile devices. As these and myriad other things change, teams will find new things and ways to measure them.

The Many Flavors of Performance

Many of the architecture characteristics we describe have multiple definitions. Performance is a great example. Many projects look at general performance: for example, how long request and response cycles take for a web application. However, through tremendous work, architects and DevOps engineers in many organizations have established specific performance budgets for specific parts of an application. For example, many organizations have researched user behavior and determined that the optimum time for first-page render (the first visible sign of progress for a web page in a browser or mobile device) is some fraction of a second. Most applications fall in the double-digit range for this metric. But for modern sites attempting to capture as many users as possible, this is an important metric to track, and the organizations behind such sites have built extremely nuanced measures.

Some of these metrics have additional implications for application design. Many forward-thinking organizations place *K-weight budgets* for page downloads; that is, they allow a maximum number of bytes' worth of libraries and frameworks on a particular page. Their rationale derives from physics constraints: only so many bytes can travel over a network at a time, especially for mobile devices in low-bandwidth areas.

Structural Measures

Some objective measures are not so obvious as performance. What about internal structural characteristics, such as well-defined modularity? This is a great example of an implicit architectural characteristic—architects are responsible for defining components and interactions, and want to build a sustainable structure for overall high quality. Unfortunately, there are not yet any comprehensive metrics to assess the quality of an architecture. However, there are metrics and common tools that allow architects to address some critical aspects of code structure, albeit along narrow dimensions.

One measurable aspect of code is complexity, defined by the *Cyclomatic Complexity* metric.

Cyclomatic Complexity

Cyclomatic Complexity (CC) is a code-level metric designed by Thomas McCabe Sr., in 1976 to provide an objective measure for the complexity of code at the function/method, class, or application level. It is computed by applying graph theory to code—specifically, *decision points*, which cause different execution paths. For example, if a function has no decision statements (such as `if` statements), then CC = 1. If the function has a single conditional, then CC = 2 because there are two possible execution paths.

The formula for calculating the CC for a single function or method is $CC = E - N + 2$, where N represents *nodes* (lines of code), and E represents *edges* (possible decisions). Consider the C-like code shown in [Example 6-1](#).

Example 6-1. Sample code for Cyclomatic Complexity evaluation

```
public void decision(int c1, int c2) {  
    if (c1 < 100)  
        return 0;  
    else if (c1 + c2 > 500)  
        return 1;  
    else  
        return -1;  
}
```

The Cyclomatic Complexity for [Example 6-1](#) is 3 ($3 - 2 + 2$), shown in [Figure 6-1](#).

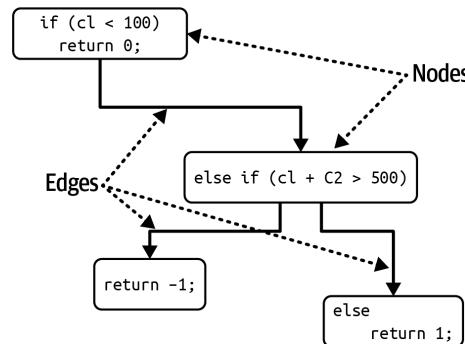


Figure 6-1. Cyclomatic Complexity graph for the decision function

The number 2 appearing in the Cyclomatic Complexity formula represents a simplification for a single function/method. For fan-out calls to other methods (known as *connected components* in graph theory), the more general formula is $CC = E - N + 2P$, where P represents the number of connected components.

Architects and developers universally agree that overly complex code represents a “code smell”—something that appears in code that is so bad that it has an imaginary odor. It harms virtually every one of the desirable characteristics of code bases: modularity, testability, deployability, and so on. If teams don’t keep an eye on gradually growing complexity, it will dominate the code base.

Cyclomatic Complexity is a great example of the bluntness of the metrics that architects have available. While it measures the complexity of code, it cannot determine whether that complexity is *essential* (because we’re solving a complicated problem) or *accidental* (because we’ve implemented a poor design). Metrics like CC are extremely useful for assessing code, whether it’s written by developers or generative AI. Generative AI tends to solve problems by brute force, often leading to accidental complexity.

What’s a Good Value for Cyclomatic Complexity?

A common question the authors receive when talking about this subject is: what’s a good threshold value for CC? Of course, like all questions in software architecture, the answer is: it depends! Specifically, it depends on the complexity of the problem domain. For example, if you have an algorithmically complex problem, the solution will yield complex functions. Some of the key aspects of CC for architects to monitor: are functions complex because of the problem domain or because of poor coding? Alternatively, is the code partitioned poorly? In other words, could a large method be broken down into smaller, logical chunks, distributing the work (and complexity) into more well-factored methods?

In general, the industry thresholds for CC suggest that a value under 10 is acceptable, barring other considerations such as complex domains. We consider that threshold very high and would prefer code to fall under five, indicating cohesive, well-factored code. A metrics tool in the Java world, [Crap4J](#), attempts to determine how poor (crappy) your code is by evaluating a combination of CC and code coverage; if CC grows to over 50, no amount of code coverage rescues that code from crappiness. The most terrifying professional artifact Neal ever encountered was a single C function that served as the heart of a commercial software package whose CC was over 800! It was a single function with over 4,000 lines of code, including liberal use of GOTO statements (to escape impossibly deeply nested loops).

Engineering practices like test-driven development (TDD) have the beneficial side effect of generating smaller, less complex methods on average for a given problem domain. When practicing TDD, developers try to write a simple test, then write the smallest amount of code to pass the test. This focus on discrete behavior and good test boundaries encourages well-factored, highly cohesive methods that exhibit low CC.

Process Measures

Some architecture characteristics intersect with software development processes. For example, agility often appears as a desirable feature. However, it is a composite architecture characteristic, made up of features such as testability and deployability.

Testability is measurable through code-coverage tools for virtually all platforms that report on what percentage of the code the tests execute. Like all software checks, though, these cannot replace thinking and intent. For example, a code base can have 100% code coverage, yet use poor assertions that don't actually provide confidence in the code's correctness.

Testability is an objectively measurable characteristic, as is deployability. Deployability metrics include percentage of successful deployments, how long deployments take, and issues/bugs raised by deployments. Every team must arrive at a good set of measurements that capture useful qualitative and quantitative data for their organization's and team's priorities and goals.

While agility and its related parts clearly relate to the software development process, that process may influence the structure of the architecture. For example, if ease of deployment and testability are high priorities, the architect would emphasize good modularity and isolation at the architecture level—an example of an architecture characteristic driving a structural decision. Virtually anything within the scope of a software project can rise to the level of an architecture characteristic if it manages to meet our three criteria, forcing an architect to make significant decisions to account for it.

Governance and Fitness Functions

Once architects establish and prioritize architecture characteristics, how can they make sure that developers will respect those priorities and implement their designs correctly and safely, regardless of schedule pressure? On many software projects, urgency dominates, yet architects still need tools and techniques to provide architectural *governance*. Modularity is a great example of an aspect of architecture that is important but not urgent.

Governing Architecture Characteristics

Governance, derived from the Greek word *kubernan* (to steer) is an important responsibility of the architect role. As the name implies, its scope covers any aspect of the software development process that architects want to influence. For example, ensuring software quality falls under architectural governance because neglecting it can lead to disastrous quality problems.

Fortunately, architects have increasingly sophisticated solutions to this problem—a good example of incremental growth within the software development ecosystem's capabilities. The drive toward automation spawned by **Extreme Programming** created continuous integration (CI). CI led to further automation in operations, which we now call DevOps, and this chain continues through to architectural governance. The book *Building Evolutionary Architectures* (O'Reilly, 2022) by Neal Ford et al. describes a family of techniques called *fitness functions* used to automate many aspects of architecture governance. We'll spend the rest of this chapter looking at fitness functions.

Fitness Functions

The word *evolutionary* in the title *Building Evolutionary Architectures* comes more from evolutionary computing than from biology. One of the authors, Dr. Rebecca Parsons, spent some time in the evolutionary computing space, including working with tools like genetic algorithms. When a developer designs a genetic algorithm to produce some beneficial outcome, they often want to guide the algorithm by providing an objective measure of the quality of its outcome. That guidance mechanism is called a *fitness function*: an object function used to assess how close the output comes to achieving its aim.

For example, suppose you need to solve the **traveling salesperson problem**, a famous problem used as a basis for machine learning. Given a salesperson, a list of cities they must visit, and the distances between those cities, what is the optimum possible route—minimizing distance, time, and cost? If you design a genetic algorithm to solve this problem, you might use one fitness function to evaluate the length of the route and another to evaluate the overall cost associated with the route. Yet another might evaluate the time the traveling salesperson is away.

Practices in evolutionary architecture borrow this concept to create an *architectural fitness function*: any mechanism that provides an objective integrity assessment of some architecture characteristic or combination of architecture characteristics.

Fitness functions are not some new framework for architects to download. Rather, they offer a new perspective on many existing tools. Notice in the definition the phrase *any mechanism*—the verification techniques for architecture characteristics are as varied as the characteristics are. Fitness functions overlap many existing verification mechanisms, depending on the way they are used: in chaos engineering or as metrics, monitors, or unit testing libraries (see [Figure 6-2](#)).

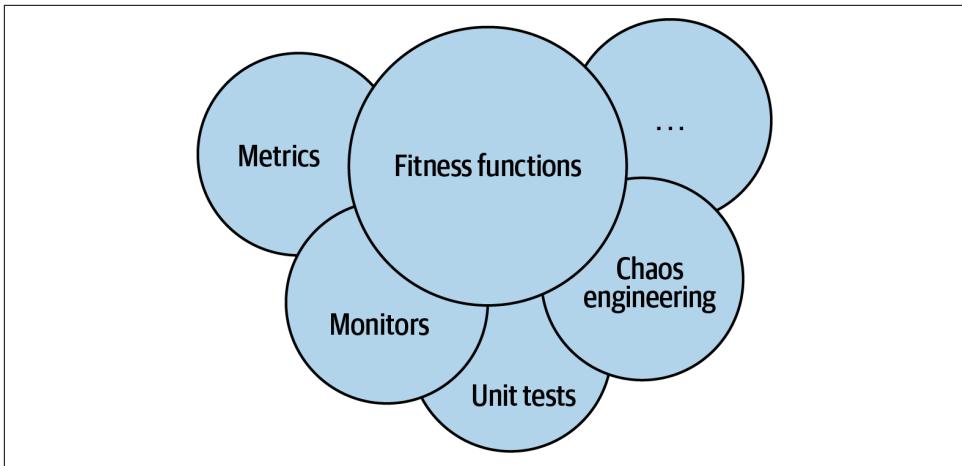


Figure 6-2. The mechanisms of fitness functions

Many different tools may be used to implement fitness functions, depending on the architecture characteristics. Let's look at a couple of examples of fitness functions that test various aspects of modularity.

Cyclic dependencies

Modularity is an implicit architecture characteristic that most architects care about. Because poorly maintained modularity harms the structure of a code base, architects generally place a high priority on maintaining good modularity. However, on many platforms, there are forces working against those good intentions. For example, when coding in any popular Java or .NET development environment, as soon as a developer references a class not already imported, the IDE helpfully presents a dialog asking if they would like to auto-import the reference. This occurs so often that most programmers reflexively swat the auto-import dialog away. But arbitrarily importing classes between components can spell disaster for modularity. For example, [Figure 6-3](#) illustrates *Cyclic Dependencies*, a particularly damaging antipattern that architects aspire to avoid.

In [Figure 6-3](#), each component references something in the other components. Having a network like this damages modularity, because it's impossible to reuse a single component without bringing the others along. And if those other components are coupled to other components? The architecture will tend more and more toward the [Big Ball of Mud](#) antipattern. How can architects govern this behavior without constantly looking over the developers' shoulders? Code reviews help, but happen too late in the development cycle to be fully effective. If the development team imports rampantly across the code base for a week until the code review, they've already done serious damage to the code base.

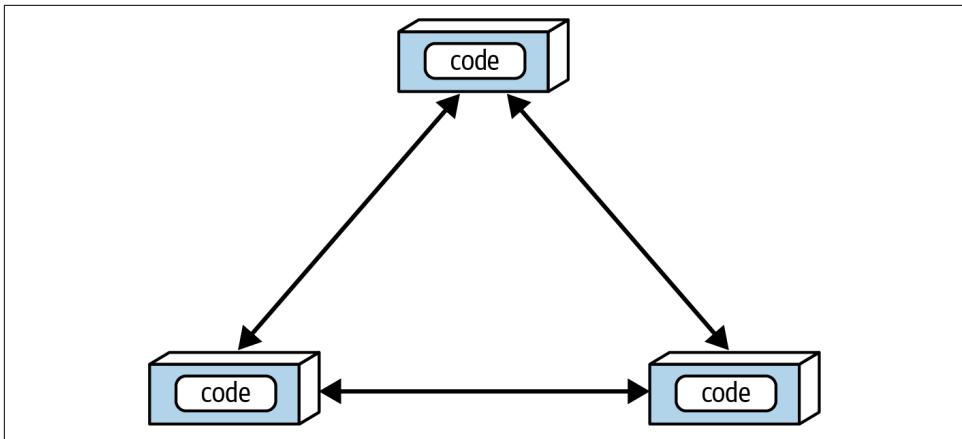


Figure 6-3. Cyclic dependencies between components

The solution to this problem is to write a fitness function to look after cycles, as shown in [Example 6-2](#).

Example 6-2. Fitness function to detect component cycles

```

public class CycleTest {
    private JDepend jdepend;

    @BeforeEach
    void init() {
        jdepend = new JDepend();
        jdepend.addDirectory("/path/to/project/persistence/classes");
        jdepend.addDirectory("/path/to/project/web/classes");
        jdepend.addDirectory("/path/to/project/thirdpartyjars");
    }

    @Test
    void testAllPackages() {
        Collection packages = jdepend.analyze();
        assertEquals("Cycles exist", false, jdepend.containsCycles());
    }
}

```

This code uses the metrics tool **JDepend** to check the dependencies between packages. The tool understands the structure of Java packages and fails the test if it finds any cycles. An architect can wire this test into a project's continuous build on and stop worrying about trigger-happy developers accidentally introducing cycles. This is a great example of a fitness function guarding the *important rather than urgent* practices of software development: it's an important concern for architects, yet has little impact on day-to-day coding.

Distance from the Main Sequence fitness function

In “[Coupling](#)” on page 44, we introduced the more esoteric metric of *Distance from the Main Sequence*, which can also be verified using fitness functions, as shown in [Example 6-3](#).

Example 6-3. Distance from the Main Sequence fitness function

```
@Test
void AllPackages() {
    double ideal = 0.0;
    double tolerance = 0.5; // project-dependent
    Collection packages = jdepend.analyze();
    Iterator iter = packages.iterator();
    while (iter.hasNext()) {
        JavaPackage p = (JavaPackage)iter.next();
        assertEquals("Distance exceeded: " + p.getName(),
            ideal, p.distance(), tolerance);
    }
}
```

This code uses JDepend to establish a threshold for acceptable values, failing the test if a class falls outside the range. (The tool ArchUnit, highlighted in the next section, allows architects to create similar fitness functions.) This example of an objective measure for an architecture characteristic illustrates the importance of collaboration between developers and architects when designing and implementing fitness functions. The intent is not for a group of architects to ascend to an ivory tower and develop esoteric fitness functions that developers cannot understand—it’s to implement automated governance rules that ensure the quality of the code base.



Architects must ensure that developers understand the purpose of a fitness function before imposing it on them.

The sophistication of fitness function tools has increased over the last few years, especially as some special-purpose tools have emerged. One such tool is [ArchUnit](#), a Java testing framework inspired by—and using—several parts of the [JUnit](#) ecosystem. ArchUnit provides a variety of predefined governance rules, codified as unit tests, and allows architects to write specific tests that address modularity. Consider the layered architecture illustrated in [Figure 6-4](#).

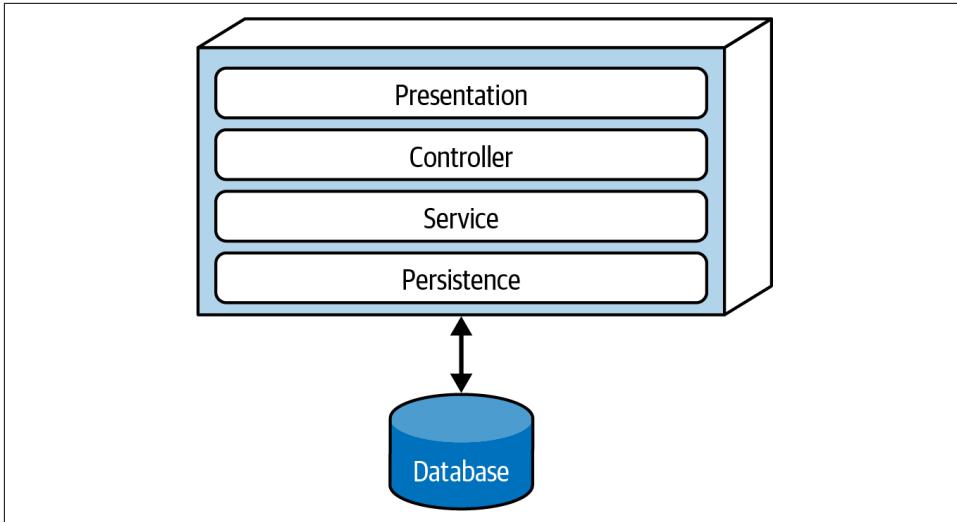


Figure 6-4. Layered architecture

When designing a layered monolith such as the one in [Figure 6-4](#), the architect defines the layers for good reasons (we describe these motivations, trade-offs, and other aspects in [Chapter 10](#)). However, some developers may not understand the importance of these patterns, while others may adopt a “better to ask forgiveness than permission” attitude because of some overriding local concern, such as performance. But allowing them to erode the reasons for the architecture will hurt the long-term health of the architecture.

ArchUnit allows architects to address this problem via a fitness function, shown in [Example 6-4](#).

Example 6-4. ArchUnit fitness function to govern layers

```
layeredArchitecture()
    .layer("Controller").definedBy(..controller..)
    .layer("Service").definedBy(..service..)
    .layer("Persistence").definedBy(..persistence..)

    .whereLayer("Controller").mayNotBeAccessedByAnyLayer()
    .whereLayer("Service").mayOnlyBeAccessedByLayers("Controller")
    .whereLayer("Persistence").mayOnlyBeAccessedByLayers("Service")
```

In [Example 6-4](#), the architect defines the desirable relationship between layers and writes a verification fitness function to govern it.

There is a similar tool in the .NET space, [NetArchTest](#). [Example 6-5](#) shows a layer verification in C#.

Example 6-5. NetArchTest for layer dependencies

```
// Classes in the presentation should not directly reference repositories
var result = Types.InCurrentDomain()
    .That()
    .ResideInNamespace("NetArchTest.SampleLibrary.Presentation")
    .ShouldNot()
    .HaveDependencyOn("NetArchTest.SampleLibrary.Data")
    .GetResult()
    .IsSuccessful;
```

Our discussion of testability highlights a problem with any metric or measurement: the possibility that developers will try to game the system. Once they learn how architects are measuring compliance, they may code to the metric rather than build the correct thing. For example, a common failing of testability occurs when developers who are taking shortcuts write unit tests, but without any assertions. “Touching” the code but not verifying that it works amounts to cheating on the code-coverage metric. Writing governance code using tools like ArchUnit helps architects prevent this behavior by ensuring that every unit test includes at least one assertion. Of course, dedicated rule-breakers will always find a way, but fitness functions like this prevent accidental lapses.

Another example of fitness functions is Netflix’s “Chaos Monkey” and the attendant [Simian Army](#). When Netflix decided to move its operations to Amazon’s cloud, the architects no longer had control over operations, which worried them: what would happen if a defect appeared operationally? To solve this problem, they spawned the discipline of *chaos engineering*. The Chaos Monkey essentially acts as a production fitness function, simulating general chaos to see how well the system can endure it. Latency was a problem with some AWS instances, so the Chaos Monkey would simulate high latency. (In fact, this was such a problem, they eventually created a specialized Latency Monkey). This led them to develop additional tools: for instance, the Chaos Kong, which simulates an entire Amazon datacenter failure, has helped Netflix avoid such outages when they occur for real.

In particular, the *Conformity*, *Security*, and *Janitor* “Monkeys” (fitness functions) exemplify the automated-governance approach. The Conformity Monkey allows Netflix architects to define governance rules enforced by the monkey in production. For example, if they decide that each service should respond without errors for all requests, they build that check into the Conformity Monkey. The Security Monkey checks each service for well-known security defects, like ports that shouldn’t be active and configuration errors.

Finally, the Janitor Monkey looks for instances to which no other services route anymore. Netflix has an evolutionary architecture, so developers routinely migrate to newer services, leaving old services running with no collaborators. Because running

services on the cloud consumes money, the Janitor Monkey looks for orphan services and disintegrates them out of production.

Chaos engineering offers an interesting new perspective on architecture: it's not a question of *if* something will eventually break, but *when*. Anticipating those breakages and testing to prevent them makes systems much more robust. A book by some of the Netflix innovators, Casey Rosenthal and Nora Jones, called *Chaos Engineering* (O'Reilly, 2020) highlights this approach.

Atul Gawande's influential book *The Checklist Manifesto* (Metropolitan, 2009) describes how professionals such as airline pilots and surgeons use checklists. (Sometimes they're even legally mandated to do so.) It's not because those professionals don't know their jobs or are forgetful. Rather, when professionals do a highly detailed job over and over, it becomes easy for details to slip by them; a succinct checklist forms an effective reminder.

This is the correct perspective on fitness functions—they are not a heavyweight governance mechanism, but a mechanism for architects to express and automatically verify important architectural principles. Developers know that they shouldn't release insecure code, but that competes with dozens or hundreds of other priorities. Tools like the Security Monkey, and fitness functions generally, allow architects to build important governance checks into the substrate of the architecture.

The Scope of Architectural Characteristics

Software architects' thinking must evolve with our ecosystem, and nowhere is that more apparent than in scoping architectural characteristics. Many of the outdated frameworks for determining architectural characteristics had a fatal flaw: assuming one set of architectural characteristics for the entire system. While that is sometimes still true, many modern architectures, like microservices, contain different architectural characteristics at the service and system levels.

The scope of architectural characteristics is a useful measure for architects, especially in determining the most appropriate architecture style to use as a starting point for implementation. When we were writing our book *Building Evolutionary Architectures*, we needed a technique to measure the structural evolvability of particular architecture styles. None of the existing measures offered the correct level of detail. The section “[Structural Measures](#)” on page 83 discusses a variety of code-level metrics that allow architects to analyze structural aspects of an architecture—but none of these metrics reflects scope. They reveal low-level details about the code, but can’t evaluate dependent components outside the code base (such as databases) that affect many architectural characteristics, especially operational ones. No matter how much effort an architect puts into designing a code base to be performant or elastic, if the system’s database doesn’t match those characteristics, their effort won’t be successful.

Failing to find a good measure of scope, we developed one. We call it *architecture quantum*.

Architectural Quanta and Granularity

Component-level coupling isn't the only thing that binds software together. Many business concepts bind parts of the system together semantically, creating *functional cohesion*. To design, analyze, and evolve software successfully, architects and developers must consider all the coupling points that could break.

Plurals for Latin-Derived Technical Terms

Quantum originates in Latin, which means that the plural ends with an *a*. If you have more than one quantum, you have *quanta*—like the more common *data*, also derived from Latin. Architects rarely discuss a single datum. The *Chicago Manual of Style* usage guide notes: “Though originally this word was a plural of *datum*, it is now commonly treated as a mass noun and coupled with a singular verb” (18th edition, 2024, p. 336).

Many science-literate architects know of the concept of a quantum from physics, where it refers to the smallest possible amount of something—usually energy. The word *quantum* derives from Latin, meaning “how great” or “how much.” In general usage, it tends to mean “small, unbreakable thing,” which is how we define an *architecture quantum*. Another common informal definition of *architecture quantum* is “the smallest part of the system that runs independently.” For example, microservices often form architecture quanta, which exemplifies this definition: a service can run independently within the architecture, including its own data and other dependencies.

An *architecture quantum* establishes the scope for a set of architectural characteristics. It features:

- Independent deployment from other parts of the architecture
- High functional cohesion
- Low external implementation static coupling
- Synchronous communication with other quanta

This definition contains several parts. Let's break them down:

Establishes the scope for a set of architectural characteristics

Architects use the architecture quantum as a boundary delineating a set of architectural characteristics, particularly operational ones. Because it is independently deployable and has high functional cohesion (discussed next), the architecture quantum provides a useful measure of architecture modularity.

Independently deployable

An architecture quantum includes all the necessary components to function independently from other parts of the architecture. For example, if an application uses a database, that database is part of the quantum, because the system won't function without it. This requirement means that virtually all legacy systems that are deployed using a single database, by definition, form a quantum of one. However, in the microservices architecture style, each service includes its own database (part of the *bounded context* driving philosophy, described in detail in [Chapter 18](#)). This creates multiple quanta within that architecture, because each service has its own architectural characteristics scope.

High functional cohesion

Cohesion in component design refers to how unified the contained code is in its purpose. For example, a *Customer* component, with properties and methods all pertaining to a *Customer* entity, exhibits high cohesion. A *Utility* component with a random collection of miscellaneous methods would not. High functional cohesion implies that an architecture quantum does something purposeful. This distinction matters little in traditional monolithic applications with a single database, where the cohesion is essentially the entire system. However, in distributed architectures like event-driven or microservices, architects are more likely to design each service to match a single workflow (a *bounded context*, as described in [“Domain-Driven Design’s Bounded Context”](#)), so that service would exhibit high functional cohesion.

Domain-Driven Design’s Bounded Context

Eric Evans's book *Domain-Driven Design* (Addison-Wesley Professional, 2003) has deeply influenced modern architectural thinking. [Domain-driven design \(DDD\)](#) is a modeling technique that allows architects to decompose complex problem domains in an organized way. DDD defines a *bounded context*, where everything related to a portion of the domain is visible internally but opaque to other bounded contexts.

Before DDD, architects sought to reuse code holistically across common entities within the organization. But they found that creating common shared artifacts causes a host of problems, such as tight coupling, more difficult coordination, and increased complexity. The bounded-context concept recognizes that each entity works best within a localized context. Thus, instead of creating a unified *Customer* class across the entire organization, each problem domain can create its own *Customer* class and reconcile the differences at communication points with other domains.

To clarify the next parts of the definition, we need to make some finer distinctions about the types of coupling:

Semantic coupling

Semantic coupling describes the natural coupling of the problem for which an architect is building a solution. For example, an order-processing application's inherent coupling includes things like inventory, catalogs, shopping carts, customers, and sales. The nature of the problem that motivates building a software solution defines this coupling. Architects have few techniques that prevent changes to the domain from rippling out through the system: a change to the domain (and therefore the semantics) means we're changing the requirements for the system. While architects can adapt to that, no magical architecture pattern prevents changing the core problem from affecting the architecture.

Implementation coupling

Implementation coupling describes how an architect and team decide to implement particular dependencies. In an order-processing application, the team must consider a variety of constraints in setting domain boundaries. For example, should all the data reside in a single database, or should some of it be split apart for better scalability or availability? Should we build a monolith or a distributed architecture? The answers to these questions have little effect on the system's semantic coupling, but greatly affect architectural decisions.

Static coupling

Static coupling refers to the “wiring” of an architecture—how services depend upon one another. Two services are part of the same architecture quantum if they both depend on the same coupling point. For example, let's say that two microservices, **Catalog** and **Shipping**, need to share address information, so they both create a dependency to a shared component. Because both services are coupled to that dependency, they are part of the same architecture quantum.

Here's an easy way to think about coupling in software architecture: two things are coupled if changing one might break the other. Static coupling defines the scope dependencies in an architecture. For example, if several services use the same relational database, they are part of the same quantum.

Dynamic coupling

Dynamic coupling describes the forces involved when architecture quanta must communicate with each other. For example, when two services are running, they must communicate to form workflows and perform tasks within the system. Architects must consider the trade-offs when communicating between services in a distributed architecture, which is discussed in detail in [Chapter 15](#).

With these definitions in hand, we can complete our *architecture quantum* definition:

Low external implementation static coupling

The level of implementation coupling between architecture quanta should be low. This characteristic is also derived from the DDD philosophy of low coupling between bounded contexts. Quanta form the operational building blocks of architecture that sit one level of abstraction higher than components. They often overlap with service boundaries. This goal reflects architects' general preference for loose coupling between different parts of the architecture.

In general, tight coupling is desirable when high cohesion is required, such as within a service or subsystem. We call an architecture “brittle” when a single implementation change can cause unexpected rippling side effects that break many other (ostensibly unrelated) things. The broader the scope of the system’s coupling, the more loose coupling helps to make the architecture less brittle. For example, an architect might rename a field in a service call from `State` to `StateCode`, thinking it will only affect one caller, only to discover that many other dependencies have just broken unexpectedly.



Higher coupling is allowed for narrower scopes; the broader the scope, the looser the coupling should be.

Synchronous Communication

Communication refers to dynamic coupling—when architecture quanta call each other, which is common in distributed architectures. This particularly concerns the family of operational architectural characteristics described in [“Operational Architectural Characteristics” on page 59](#), because they often determine important timing and blocking in distributed architectures.

For example, consider a microservices architecture with a **Payment** service and an **Auction** service. When an auction ends, the **Auction** service sends payment information *synchronously* to the **Payment** service. However, let's say that the **Payment** service can only handle one payment every 500 ms. What happens when a large number of auctions end at once? The two services have different operational architectural characteristics. The first call will work, but then calls will start failing because the **Payment** service can't handle the number of requests—it isn't as scalable as the **Auction** service.

We call out synchronous communication here because asynchronous communication has less potential impact. For example, if the **Auction** service calls the **Payment** service asynchronously, using a message queue, the queue can act as a buffer to allow the two systems to operate. Of course, if the **Auction** service continually sends more messages than **Payment** can handle, it will eventually overflow the message queue. However, when the flood of messages comes in bursts, the queue can hold on to pending messages until the receiver is ready. Synchronous communication is unforgiving in distributed architectures, especially if parts of the architecture have different architectural characteristics. [Chapter 15](#) thoroughly covers the types of communication in event-driven architectures.

The concept of architecture quantum provides a new way to think about scope. In modern systems, architects define architectural characteristics at the quantum level rather than the system level. This provides important information when analyzing a new problem domain.

The Impact of Scoping

Architects can use the scope of architectural characteristics to help determine appropriate service boundaries illustrated overall in the decision tree in [Figure 7-1](#) and further detailed in the steps.

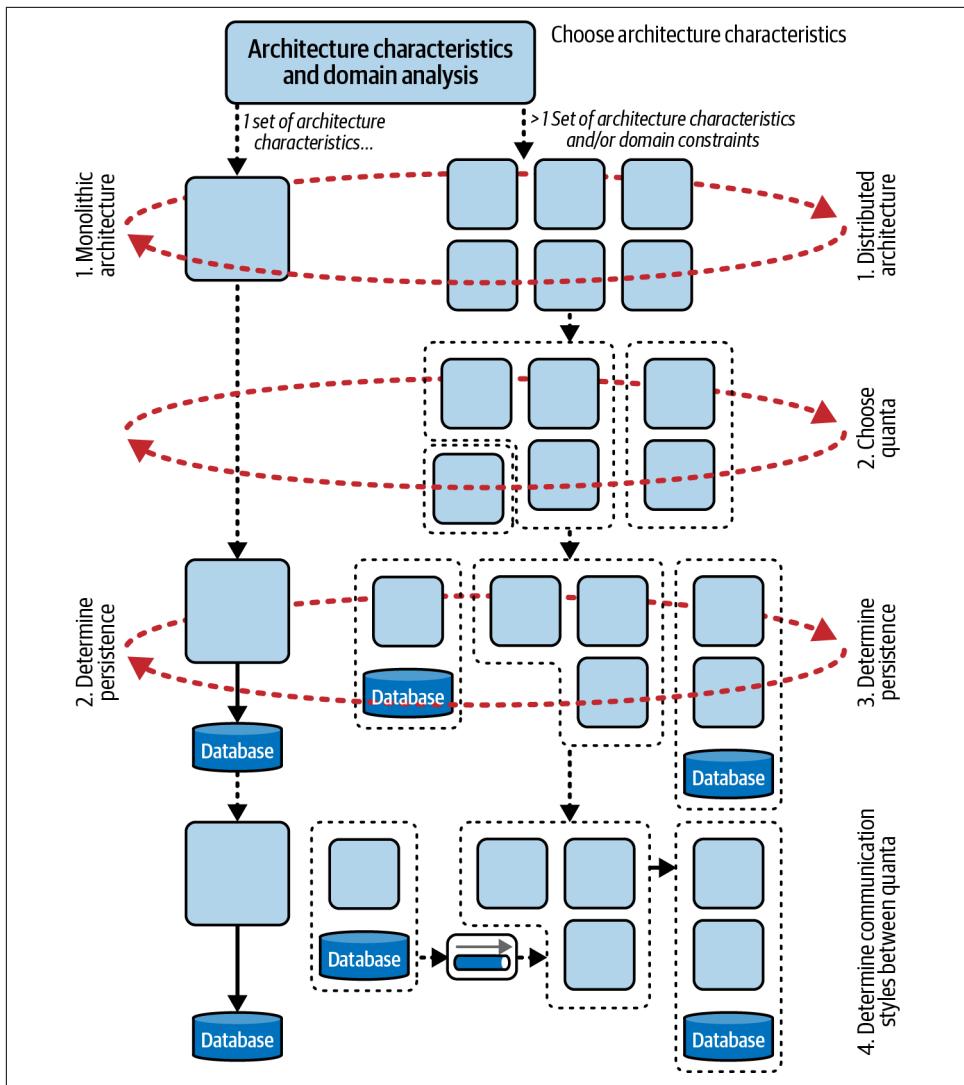


Figure 7-1. A decision tree that uses architectural characteristics scope to help determine architectural style

Scoping and Architectural Style

Determining the quantum boundaries of the problem domain helps in choosing an architectural style: is a monolithic architecture or distributed architecture most suitable? Consider the first part of the decision chart shown in [Figure 7-2](#).

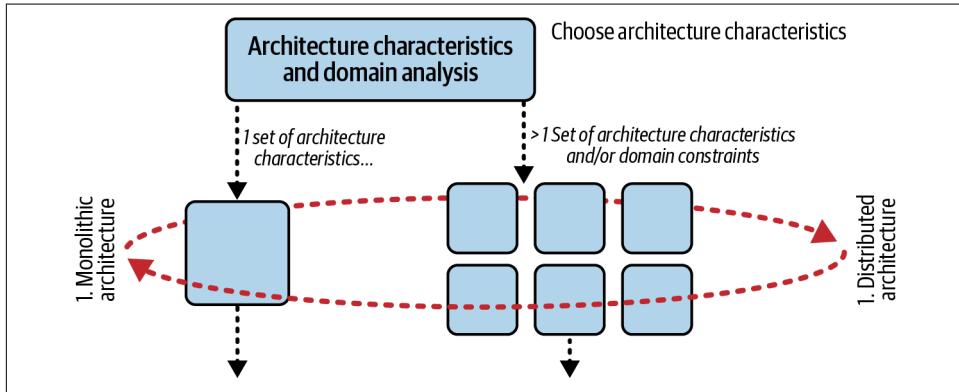


Figure 7-2. Choosing an appropriate architectural style based on architectural characteristics

As we discussed in [Chapter 4](#), architects must analyze architectural characteristics and the domain to determine the most appropriate architectural style. Part of this analysis concerns whether the solution needs multiple groups of architectural characteristics. In step one in [Figure 7-2](#), the architect determines if the system can succeed with a single set of architectural characteristics, or if more than one group is required (see "[Kata: Going Green](#)" on page 103 for an example).

If the architect determines that a single set of architectural characteristics will suffice, they can choose a monolithic architecture, reducing the number of subsequent choices. If they choose a distributed architecture, the next step is to determine its quantum boundaries, as shown in [Figure 7-3](#).

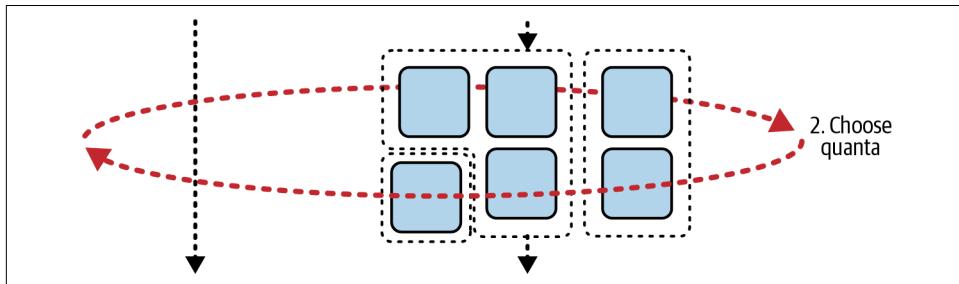


Figure 7-3. In distributed architectures, architects must choose the appropriate quantum boundaries

We provide some guidelines for determining granularity in [Chapter 18](#). The next step is choosing a persistence mechanism, which concerns both architecture families, shown in [Figure 7-4](#).

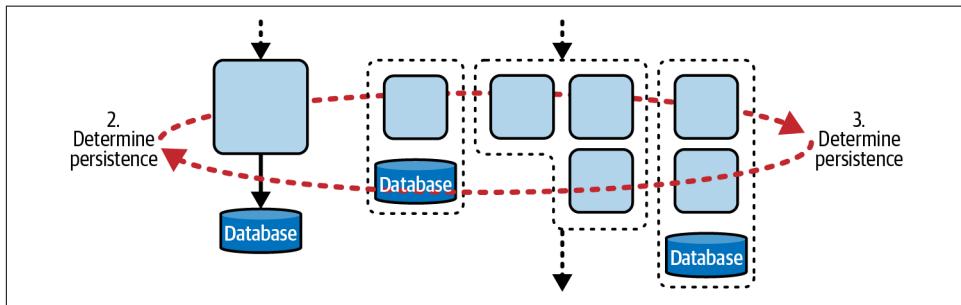


Figure 7-4. Both architecture styles generally require persistence of some kind

For monolithic architectures, a single monolithic database is generally suitable. The architecture and database are developed and deployed together, in lockstep. This completes the process—the architect can move on to choosing the most appropriate monolithic style.

Distributed architectures, on the other hand, can use a single database (common in event-driven architectures) or partition the data along the lines of the service granularity (as in microservices architectures). One further step remains: determining what type of communication to use between quanta, synchronous or asynchronous. (We discuss this question in detail in [Chapter 15](#).) Remember that, in some systems, choosing synchronous communication can change the quantum boundaries established through static coupling; the two types of coupling interact frequently.

Kata: Going Green

Let's try analyzing a problem by using architecture quantum as a scope for architectural characteristics. The problem, called *Going Green*, is illustrated in [Figure 7-5](#).

You are working with Going Green (GG), a business that recycles and resells old electronics, such as cell phones. Its system uses both public kiosks and a website, all running the same system. A user can upload their device's model number and condition, and GG will bid for it. If the user accepts the bid, they can deposit it in the kiosk or, if they go through the website, GG will send them a box to mail it in. Upon receiving the device, GG assesses it and sends the user their payment. GG then estimates the value of the device and either recycles or resells it. Its system also produces reports and other analytics.

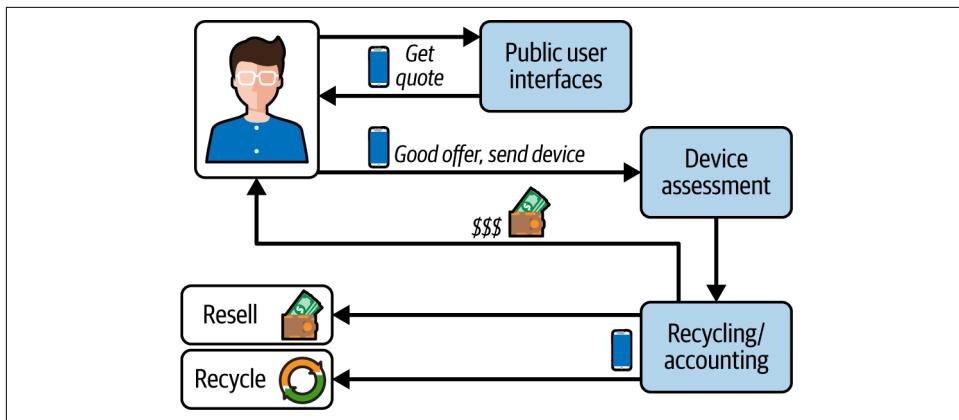


Figure 7-5. Requirements overview for Going Green

As you perform your architectural characteristics analysis, you notice three distinct clusters forming, as shown in Figure 7-6.

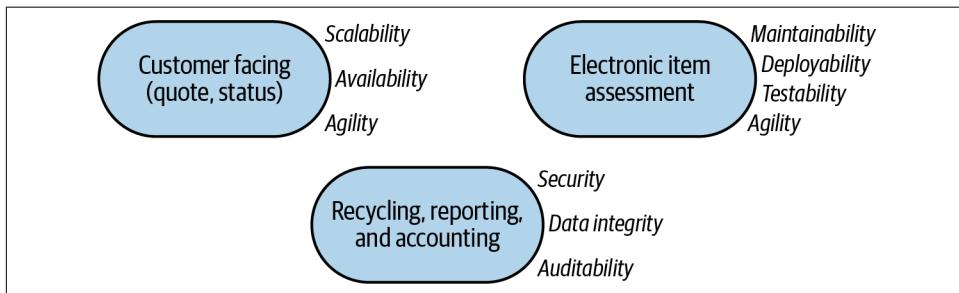


Figure 7-6. The GG architectural characteristics analysis yields three clusters of capabilities

The public-facing parts of the application need *scalability*, *availability*, and *agility*; the back-office functions need *security*, *data integrity*, and *auditability*; and the assessment part needs *maintainability*, *deployability*, and *testability* (which make up the composite architectural characteristic of *agility*). Why does the assessment part need a separate set of architectural characteristics? This is a good example of how business drivers intersect with architectural concerns. GG's business model relies on reselling the highest-value used electronics, and new models come out in a constant stream. The faster they can update their device assessments, the more they can get newer (and therefore more valuable) devices to resell.

Could you design a system that meets all these criteria: scalability, availability, security, data integrity, auditability, maintainability, deployability, and testability? It's possible...but it would be difficult. These architectural characteristics often counteract one another: for instance, achieving fast deployability is more difficult when you're

also prioritizing back-office concerns like auditability. And the UI requires a vastly different level of scalability than other parts of the system.

Instead of trying to do it all, you can use the clusters of architectural characteristics as your guide to separating quanta. In [Figure 7-7](#), the dotted lines illustrate each architectural quantum. Using characteristic scope as a guide for service granularity is a good first step in determining the most beneficial set of trade-offs. (We explore this topic further in [Chapter 18](#).)

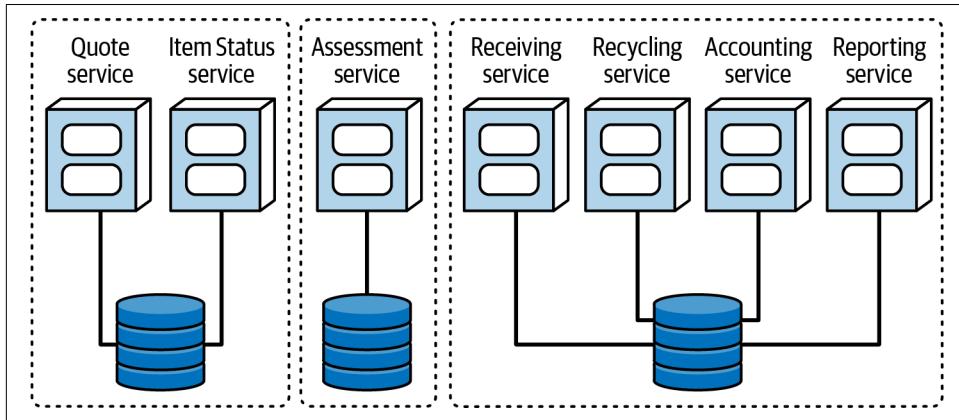


Figure 7-7. This GG architecture captures each set of architectural characteristics as an architectural boundary

Scoping and the Cloud

Cloud-based resources complicate the picture, because they encapsulate so many of the operational architectural characteristics of a system. Architects must consider at least two scenarios when using cloud-based resources for all or part of an application, depending on the deployment model:

Using the cloud to host containers

Many development teams use the cloud as an alternate operations center, running (and orchestrating) containers for servers. In such situations, architects must consider the architectural characteristics of the containers and the constraints introduced by the orchestration tool (for example, [Kubernetes](#)).

Using cloud-provider resources as system components

Another flavor of a cloud-based system pieces applications together using the cloud provider's building blocks, such as triggered functions, databases, and so on. In this case, architects must look at the capabilities the provider advertises (and hopefully maintains) for insights on how to build suitable capabilities for that particular context.

Many of the capabilities we know today as cloud providers' configuration settings, such as elasticity, were hard-won by the previous generation of architects working on physical systems. Their major concerns have become easier—but we have our own modern set of trade-offs, such as provider availability and heightened security concerns. The details of software architecture change a lot, but the job of analyzing trade-offs remains constant.

Component-Based Thinking

In [Chapter 3](#), we introduced the concept of a *module* as a collection of related code. In this chapter, we dive much deeper into this concept, focusing on the architectural aspect of modularity in terms of *logical components*—the building blocks of a system.

Identifying and managing logical components is a part of *architectural thinking* (see [Chapter 2](#)), so much so that we call this activity *component-based thinking*. Component-based thinking is seeing the structure of a system as a set of logical components, all interacting to perform certain business functions. It's at this level (not the class level) that an architect “sees” the system.

In this chapter, we define logical components within software architecture, how to identify them, and how to arrive at an appropriate level of granularity through analyzing what's known as *cohesion* (we define what that means a little later on in this chapter). We also discuss coupling between components and how and why to create loosely coupled systems.

Defining Logical Components

Think about the floor plan of a typical Western house, as illustrated in [Figure 8-1](#). Notice the floor plan is made up of various rooms (such as a kitchen, bedrooms, bathrooms, a living room, an office, and so on), each serving a different purpose. These rooms represent the building blocks—the *components*—of the house.

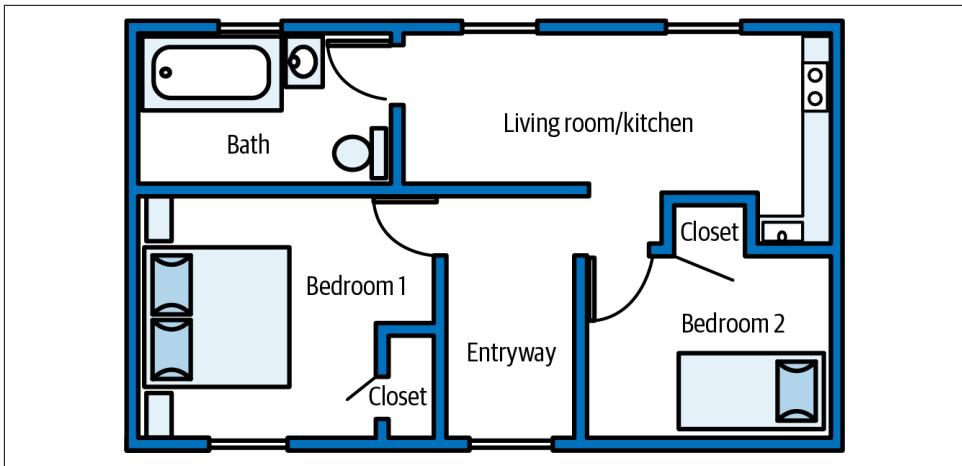


Figure 8-1. The various rooms represent the components of the house

In the same way, the major functions a system performs represent the components of that system, as shown in [Figure 8-2](#). Like the rooms of a house, each component performs a specific function, such as managing inventory, shipping orders, or processing payments. Together they make up the system. Each component contains source code that implements that particular business function.

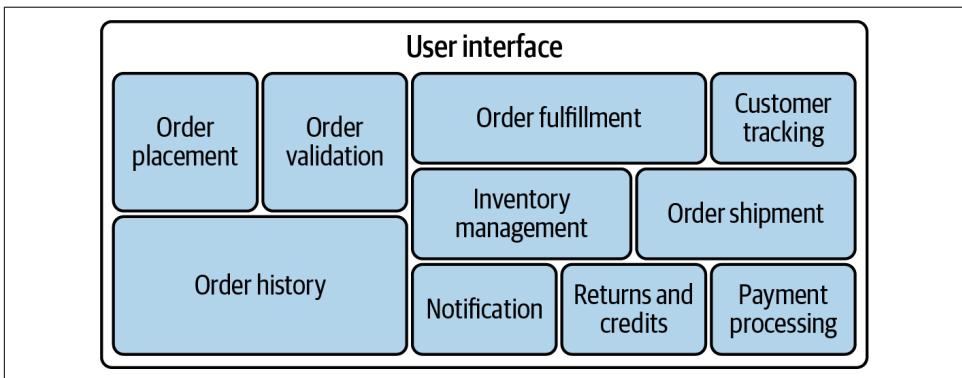


Figure 8-2. The various major functions represent the components of the system

Logical components in software architecture are usually manifested through a name-space or a directory structure containing the source code that implements that particular functionality. Typically, the *leaf nodes* of the directory structure or name-space containing source code represent the logical components in the architecture, and higher-level directories or namespace nodes represent the system's domains and subdomains. In the directory structure illustrated in [Figure 8-3](#), the directory path *order_entry/ordering/payment* represents the Payment Processing component, and the path *order_entry/processing/fulfillment* represents the Order Fulfillment component. The source code underlying the directories implements these logical components.

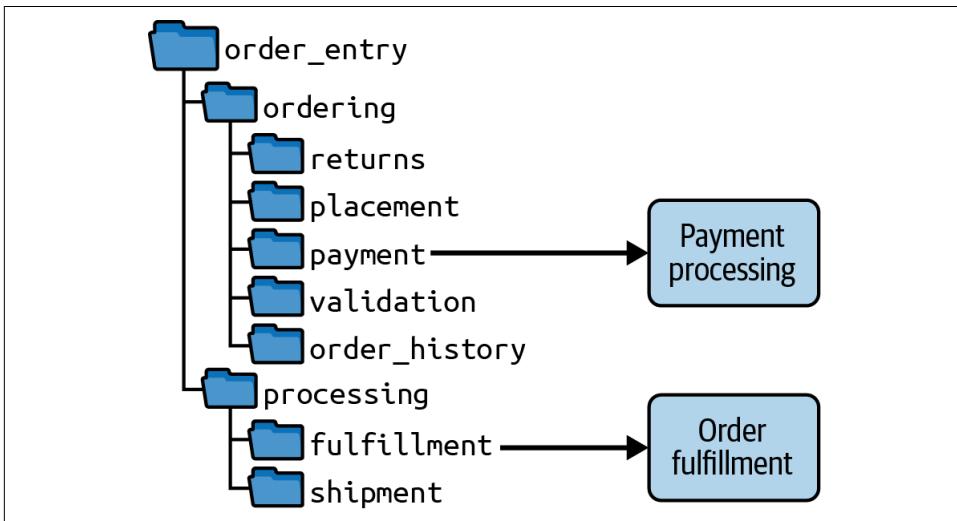


Figure 8-3. The leaf nodes of the directory represent the components of the system

An architect can analyze a software system's directory structures or namespaces to understand its internal structure—in other words, its *logical architecture*. We describe the differences between logical and physical architectures in the next section.

Logical Versus Physical Architecture

A *logical architecture* consists of a system's logical components (building blocks) and how they interact with one another. It also usually shows their interactions with various *actors* (users who interact with the system), and may also show a repository (not a database) to clarify where data is used or transferred between components. [Figure 8-4](#) illustrates an example of a logical architecture.

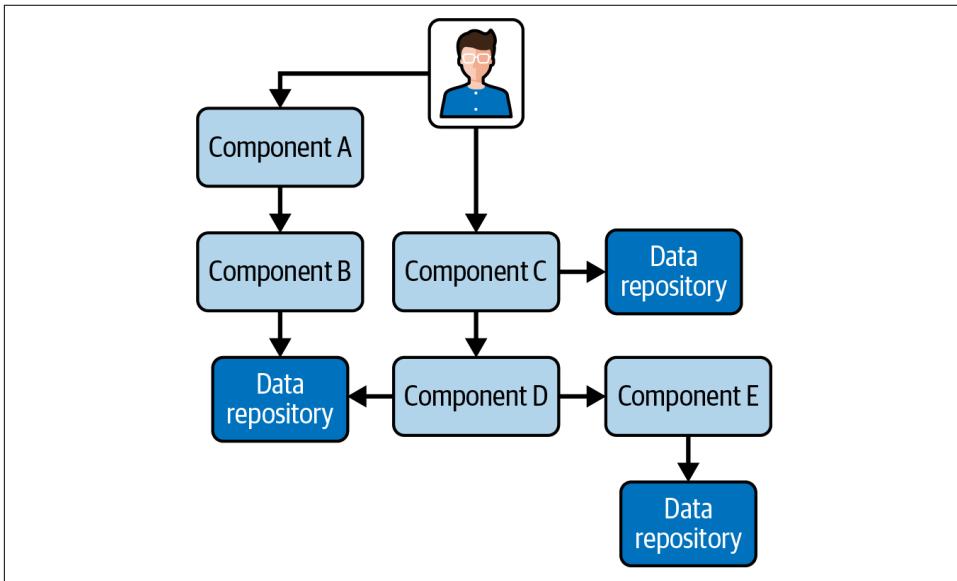


Figure 8-4. Diagram of a logical architecture

A logical architecture diagram doesn't typically show a user interface, databases, services, or other physical artifacts. Rather, it shows the logical components and how they interact, which should match the directory structures and namespaces that organize the code.

A *physical architecture*, on the other hand, includes such physical artifacts as services, user interfaces, databases, and so on. [Figure 8-5](#) is an example of a physical architecture diagram.

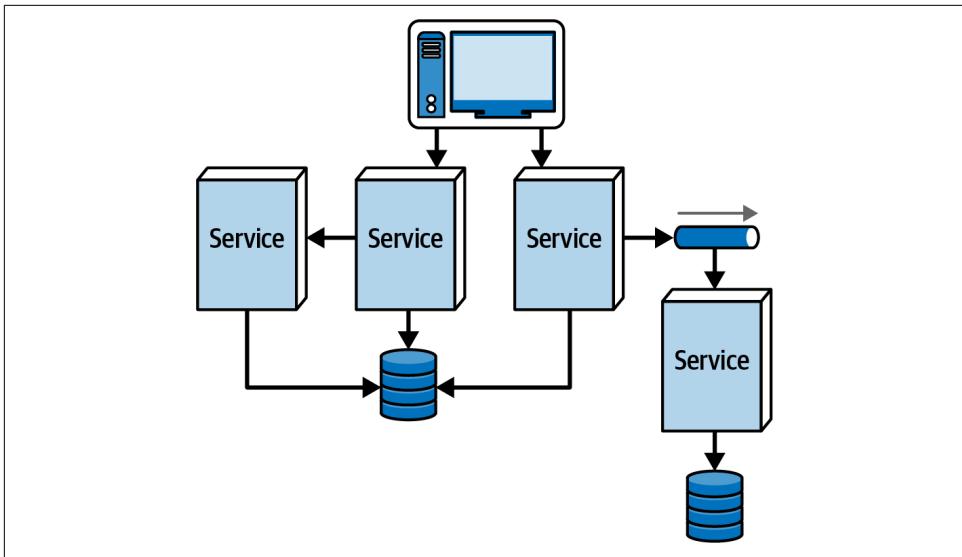


Figure 8-5. Diagram of a physical architecture

The system's physical architecture should closely represent one (or more) of the many architectural styles described in [Part II](#) of this book: microservices, layered architecture, event-driven architecture, and so on.

Many architects choose to bypass creating a logical architecture and start directly with a physical architecture. However, we advise against this practice, because a physical architecture doesn't always show where the system's functionality is and how it all fits together. For example, in a physical architecture, payment-processing functionality may be spread across multiple services, making it difficult to see how it interacts with other parts of the system. Furthermore, a physical architecture doesn't provide development teams with any guidance on how to build a monolithic or distributed system, or organize its code, resulting largely in unstructured architectures that are hard to maintain, test, and deploy.

Generally, a system's logical architecture is independent of its physical architecture. In other words, when creating a logical architecture, the focus is more on what the system does, how that functionality is demarcated, and how the functional parts of the system interact, rather than on its physical structure. For example, the architect creating a logical architecture like that shown in [Figure 8-4](#) may not have determined yet whether those components will all go into a monolithic architecture (a single deployment unit) or will be deployed as services (separate deployment units), nor have they necessarily decided what kind of architecture style would be most appropriate.

Creating a Logical Architecture

Creating a logical architecture involves continuously identifying and restructuring logical components. *Component identification* works best as an iterative process. It involves producing candidate components, then refining them through a feedback loop, as illustrated in [Figure 8-6](#).

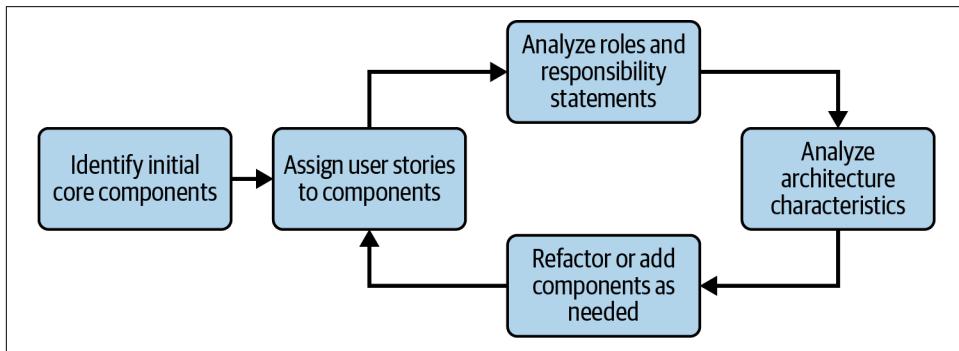


Figure 8-6. The component identification and refactoring cycle

The first activity an architect does in developing a logical architecture is to identify the initial core components, and then assign user stories or requirements to them. After this, they analyze the component's roles and responsibilities to make sure that the user stories or requirements assigned to it make sense. Then the architect looks at the architectural characteristics the system must support and determines whether any refactoring is needed to possibly break apart or combine the component based on those characteristics. Finally, the architect optionally refines the components based on this analysis. This process is a feedback loop that essentially never stops.

The workflow in [Figure 8-6](#) can be used for greenfield (new) systems or anytime a feature is added to or changed in the system. For example, suppose you need to enable an order-entry system to allow users to pick items up at a store rather than having them shipped to their homes. This will involve adding scheduling code as well as changing the existing ordering process. This change may require new components, changes to existing ones, or both. As components change, their roles and responsibilities may change, prompting you to restructure your code or create new components for it.

We describe each of these steps in detail in the following sections.

Identifying Core Components

The challenge in starting out with a new logical architecture (or making major modifications to an existing one) is determining what the initial core components should be. One mistake many software architects make is spending too much effort trying to get the initial logical components perfect the first time. A better approach is to make a “best guess” as to what the initial core components might look like, based on the core functionalities of the system, and refine them through the workflow steps outlined in [Figure 8-6](#). In other words, it’s better to iterate over the logical components as you learn more about the system than to try to get it all perfect the first time, when you know least about the system and its specific requirements.

In most cases, the architect does not need to know all (or sometimes any) of the system’s requirements and specifications to start creating logical components. Typically, the initial core components are based on major actions users might take or the system’s major processing workflows.

We like to think of the initial core components as empty buckets. The bucket doesn’t do anything until the architect starts “filling” it—that is, assigning user stories or requirements to that component. As illustrated in [Figure 8-7](#), the component name should represent its proposed role and responsibility. However, until the architect assigns user stories to it (the next step in the workflow), it’s essentially a placeholder—a best guess at a functional building block.

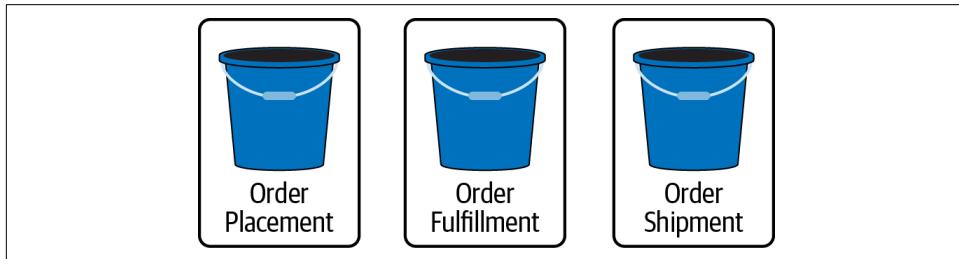


Figure 8-7. In the beginning, initial components are like empty buckets

The following are three common approaches to creating initial core components; the first two approaches (the Workflow and Actor/Action) are ones we find useful, and the third (the Entity Trap) is an antipattern we caution against.

The Workflow approach

A common approach architects use for identifying the initial core components of a logical architecture is the *Workflow* approach. As the name suggests, this approach leverages the major happy-path (nonerror) workflows a user might take through the system (or its main request-processing workflow). If the architect has some sense of the general flow, they can develop components based on those steps.

For example, assume you are building a new order-entry system for your company. You don't know the specific requirements and specifications yet, but you at least know the general workflow for processing a new order. You can thus assign a component to each step in the workflow:

1. User browses the catalog of items → **Item Browser**
2. User places an order → **Order Placement**
3. User pays for the order → **Order Payment**
4. Send user an email with order details → **Customer Notification**
5. Prepare the order → **Order Fulfillment**
6. Ship the order → **Order Shipment**
7. Email customer that order has shipped → **Customer Notification**
8. Track shipment → **Order Tracking**

In the Workflow approach, not every step in the major workflow yields a new component. In the workflow above, steps 4 and 7 both use the **Customer Notification** component. The architect models as many major workflows or user journeys they can for the system, identifying corresponding components from those steps.

This is, again, only a “best guess” as to what the logical architecture might look like. These components represent empty buckets and have no responsibilities yet, so they will likely change as they evolve (as illustrated in [Figure 8-6](#)). This is perfectly normal and part of the iterative nature of software architecture. Don’t worry about trying to model every workflow in your system. Instead, focus on the major workflows. The rest will evolve as you learn more about the system and start gathering user stories and requirements.

The Actor/Action approach

Another way architects identify initial core components is the *Actor/Action* approach. This approach is particularly useful when a system has multiple actors. With this approach, the architect identifies the major actions a user can perform in the system (such as placing an order). The system itself is always an actor, too, performing automated functions such as billing and replenishing stock.

In our order-entry system example, an architect might identify three actors: the *customer*, the *order packer* (the person who packs the box and sends it off for shipping), and the *system*. The architect then identifies the major actions each of these actors takes and assigns components to those actions:

Customer actor

- Search for items → **Item Search**
- View the details about an item → **Item Details**
- Place an order → **Order Placement**
- Cancel an order → **Order Cancel**
- Register as a new customer → **Customer Registration**
- Update customer information → **Customer Profile**

Order packer actor

- Select the box size → **Order Fulfillment**
- Mark the order as ready for shipment → **Order Fulfillment**
- Ship the order to the customer → **Order Shipment**

System actor

- Adjust inventory → **Inventory Management**
- Order more stock from the supplier → **Supplier Ordering**
- Apply payment → **Order Payment**

Like with the Workflow approach, not every action necessarily has its own component. For example, selecting the box size for the order and marking it ready for shipping are both done by the **Order Fulfillment** component.

Generally, the Actor/Action approach generates more components than the Workflow approach, depending on how many major workflows the architect chooses to model. However, in both approaches, the architect can identify the initial core components and how they communicate even before receiving detailed requirements or specifications.

The Entity Trap

It's all too tempting for an architect to start identifying components by focusing on the entities involved with the system, then deriving components from those entities. For example, in a typical order entry system, you may identify **Customer**, **Item**, and **Order** as the primary entities in the system, and correspondingly create a **Customer Manager** component, an **Item Manager** component, and an **Order Manager** component. We strongly advise against this approach for the following reasons.

First, the names of the logical components are ambiguous and don't describe the role of the component. For example, asking what the **Order Manager** component does just by looking at the name yields the useless answer "it manages orders," indicating nothing about its specific role or responsibility in the system. Compare that to a

component name like `Validate Order`, and the importance of a good, descriptive component name becomes clear. If a component name includes a suffix like `Manager`, `Supervisor`, `Controller`, `Handler`, `Engine`, or `Processor`, that's a good indicator that the architect might be caught in the Entity Trap antipattern.

Second, components become dumping grounds for domain-related functionalities. Consider an entity-based component name like `Order Manager`, as shown in [Figure 8-8](#). Every bit of order functionality would go into that single component: order validation, order placement, order history, fulfilling the order, shipping the order, tracking the order, and so on. Essentially, it becomes like those “kitchen sink” utility classes every developer has written at least once in their career, with dozens of methods that perform string manipulation, data manipulation, time calculations, and whatever else the developer can put in there.

Third, when components become too coarse-grained, they do too much and lose their purpose. Rather than being fine-grained and single purpose (like the `Validate Order` component example), components can become too big. Such components are hard to maintain, test, and deploy, and hence not very reliable.

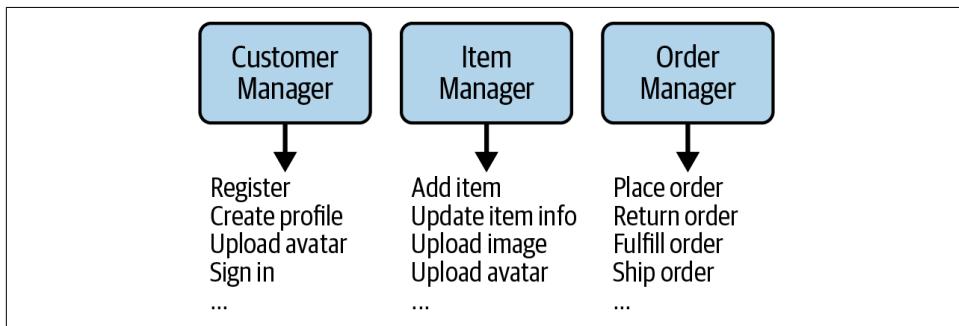


Figure 8-8. Using the Entity Trap antipattern produces components with too much responsibility

If an architect is building a system that truly is entity based and simply performs CRUD-based operations (create, read, update, and delete) against those entities, then the system doesn't need an architecture, but rather a CRUD-based framework, tool, or no-code/low-code environment that allows the developers to generate most of the source code that acts on those entities.

Assigning User Stories to Components

The next step in creating a logical architecture is to assign user stories or requirements to the logical components. This is an iterative process, since most user stories or requirements are not completely known up front; they evolve as the system evolves. This step is meant to start filling those empty buckets, giving the components specific roles and responsibilities, as illustrated in [Figure 8-9](#).

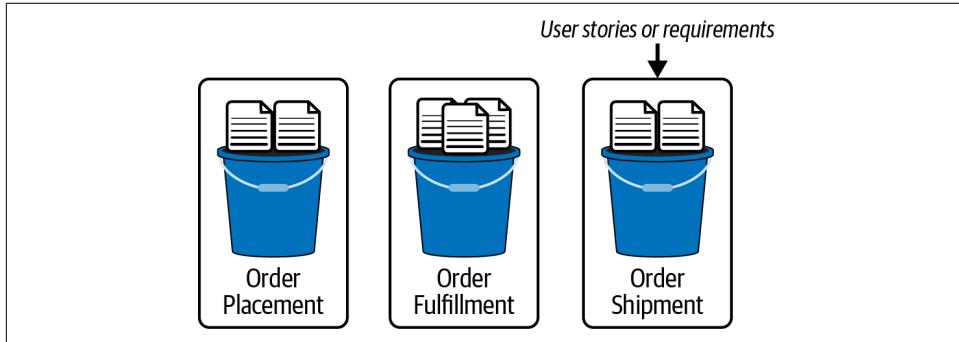


Figure 8-9. Filling up the empty buckets (components) with user stories or requirements

To see how logical components evolve, consider the following user stories:

Customer #1

As a customer, I would like to have my order validated to make sure I have entered everything completely and correctly.

Order preparer

As the person preparing the order, I would like to know what size box I should use for the order, so I can pack it in the most efficient way possible.

Customer #2

As a customer, I would like to receive an email each time the order status changes, so I always know the state of my order.

Assume that the architect has identified the following logical components so far:

- Order Placement
- Order Fulfillment
- Order Shipment
- Inventory Management

It makes sense to assign the first user story to the Order Placement component, since that's the component the user is interacting with to place the order:

Validate the order (Customer #1 user story) → Order Placement

Determining the box size should probably be handled by the `Order Fulfillment` component, since it's responsible for all the system logic needed to prepare and pack the order in a box:

Determine box size (Order preparer user story) → `Order Fulfillment`

But what about the third user story? Which of the four components listed should send emails to the customer when the order has been placed, is ready for shipment, and has shipped? The answer might be the `Order Placement`, `Order Fulfillment`, and `Order Shipment` components. But keep in mind that the user story is implemented through source code, which needs to reside in a specific directory or namespace. Since it wouldn't be a good idea to replicate the code across three components, the architect needs to define a new component to handle this user story:

Email customer (Customer #2 user story) → `Customer Notification` (new)

The `Order Placement`, `Order Fulfillment`, and `Order Shipment` components need to communicate with the new component to let it know to send an email. With this addition, the logical architecture now looks like [Figure 8-10](#).

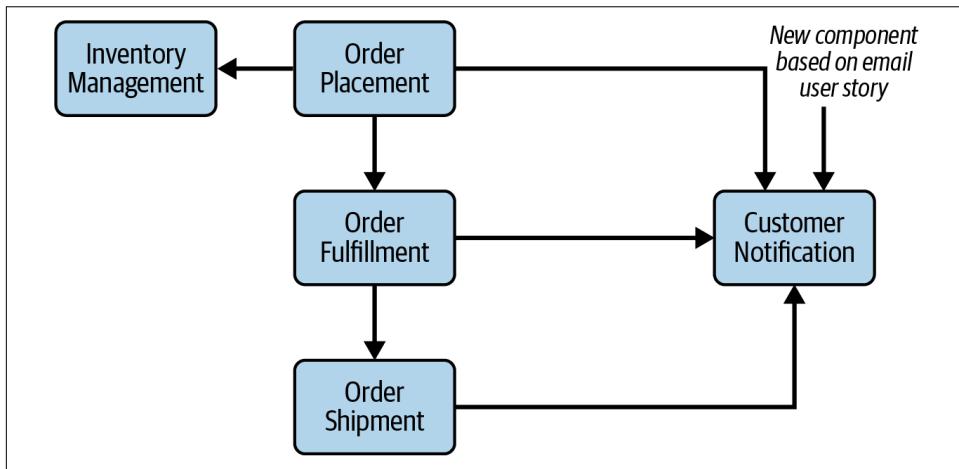


Figure 8-10. Evolving components based on new user stories

Analyzing Roles and Responsibilities

The next step in refining logical components is to analyze the roles and responsibilities of each component. This is how the architect ensures that the requirements or user stories assigned to those components belong there and that the components are not doing too much. What the architect is concerned about in this step is *cohesion*: how, and how much, a component's operations interrelate. Over time, components can get too big, even though their operations all interrelate.

To illustrate how this step works, assume the architect has assigned the following requirements to the `Order Placement` component:

- Validate the order to ensure all fields have been entered and are correct.
- Display the shopping cart with the item descriptions, quantities, and prices.
- Determine the correct shipping address.
- Collect payment information.
- Generate a unique order ID.
- Apply payment for the order.
- Adjust inventory counts for the items ordered.
- Email the customer an order summary.

If the architect were to write a role and responsibility statement for this component, it would read as follows:

This component is responsible for validating the order and displaying the valid shopping cart, complete with the item picture, description, quantity, and price. This component is also responsible for determining the correct shipping address for the order, as well as collecting all the payment information from the customer. In addition, it's also responsible for applying the payment, adjusting inventory, and emailing the customer the order summary.

While all of these operations have to do with placing an order, the `Order Placement` component is clearly taking on too much responsibility, particularly with regard to applying the payment, adjusting the inventory, and emailing the customer. One way to see if a component is taking on too much responsibility is to look for conjunctive phrases such as *and*, *also*, *in addition*, or *as well as*, and excessive use of commas.

Recall from earlier in the chapter that a logical component is represented by a name-space or directory in the code repository. In the case of the `Order Placement` component, *all* the source code representing this component would be in the same directory or namespace, such as `com/app/order/placement` or `com.app.order.placement`. This is a lot of functionality—likely too much code for a single directory. Therefore, it would make sense to separate the class files for payment processing, inventory management, and email communications into separate directories representing those functionalities. That's exactly what separating logical components is all about.

If the architect moves the responsibilities of applying the payment, adjusting the inventory, and sending the customer an email to separate components, they can reduce the responsibility of the single `Order Placement` component, making it easier to maintain, test, and deploy. The resulting components would look like this:

Order Placement

- Validate the order to ensure all fields have been entered and are correct.
- Display the shopping cart with the item descriptions, quantities, and prices.
- Determine the correct shipping address.
- Collect payment information.
- Generate a unique order ID.

Payment Processing

- Apply payment.

Inventory Management

- Adjust the inventory counts for the items ordered.

Customer Notification

- Email the customer an order summary.

Each component now has a clearer, more distinct role and responsibility.

Analyzing Architectural Characteristics

The final analysis step is to consider the architectural characteristics the system will require. Some architectural characteristics, such as scalability, reliability, availability, fault tolerance, elasticity, and agility (the ability to respond quickly to change), may influence the size of a logical component.

For example, breaking up a larger component (one with lots of responsibility) into smaller components makes each of them easier to maintain and test (that's agility), and provides better scalability, elasticity, and fault tolerance. Another good example is where two parts of a system deal with user input: if one part deals with hundreds of concurrent users and the other needs to support only a few at a time, they will need different architecture characteristics. Thus, while a purely functional view of component design might lead an architect to assign a single component to handle user interaction, analyzing the components in terms of architecture characteristics might lead to a subdivision.

Because the architect must know the architectural characteristics before building a logical architecture, it is usually done *after* determining which architectural characteristics are most important to the system.

Restructuring Components

Feedback is critical in software design. Architects must continually iterate on their component designs in collaboration with developers. Designing software provides all kinds of unexpected difficulties. Thus, an iterative approach to component design is key. First, it's virtually impossible to account for all the different discoveries and

edge cases that will arise, any of which could encourage redesign. Second, as the architecture and developers delve more deeply into building the application, they gain a more nuanced understanding of where its behaviors and roles should lie.

Architects should expect to restructure components frequently throughout the lifecycle of a system or product—not only in greenfield systems, but in any that undergo frequent maintenance.

Component Coupling

When components communicate with each other, or when a change to one component might impact other components, the components are said to be *coupled* together. The more coupled a system's components are, the harder it is to maintain and test the system (see [Figure 8-11](#)). Therefore, it's important to pay close attention to coupling.

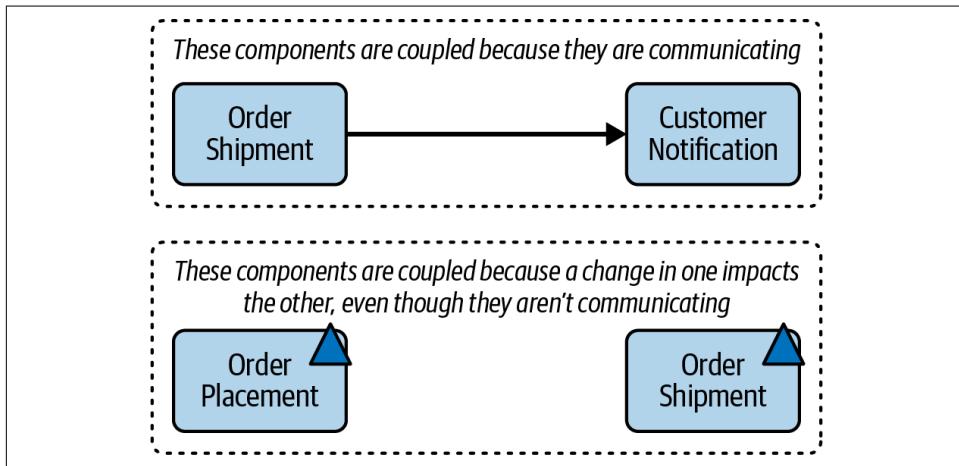


Figure 8-11. Components can be coupled even if they don't communicate directly

Static Coupling

Static coupling occurs when components communicate synchronously with each other. Architects need to be concerned about two types of coupling: afferent and efferent.

Afferent coupling (also known as *incoming* or *fan-in* coupling) is the degree to which other components depend on a target component. For instance, consider the Customer Notification component from our order-entry example in [Figure 8-12](#). To email the customer, both the Order Placement and Order Shipment components need to communicate with the Customer Notification component. This means that the Customer Notification component is said to be *affectionately coupled* to the Order Placement and Order Shipment components and would have an afferent coupling

level of 2 (the number of incoming dependencies). Afferent coupling is usually denoted as CA.

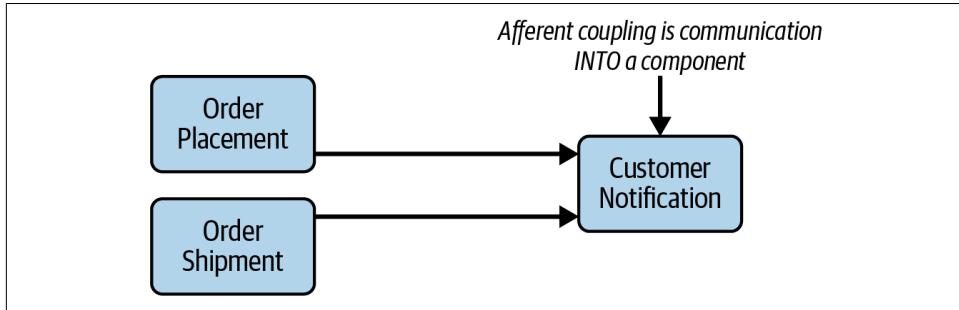


Figure 8-12. The Customer Notification component is afferently coupled to the other components

Efferent coupling (also known as outgoing or fan-out coupling) is the degree to which a target component depends on other components. For example, as illustrated in [Figure 8-13](#), the Order Placement component is dependent on the Order Fulfillment component and, as such, is efferently coupled to it and would have an efferent coupling level of 1 (the number of outgoing dependencies). Efferent coupling is usually denoted as CE.

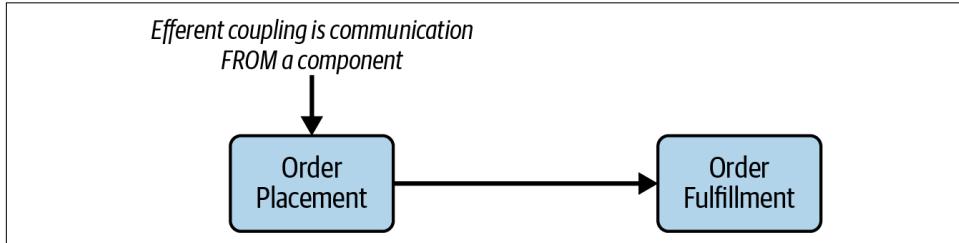


Figure 8-13. The Order Placement component is efferently coupled to the Order Fulfillment component

Temporal Coupling

Temporal coupling describes nonstatic dependencies, usually those based on timing or transactions (single units of work). For example, when the system is processing an order, the functionality in the Order Placement component must be invoked before the functionality in the Order Shipment component. Those components are thus said to be temporally coupled.

The problem with temporal coupling is that it's hard to detect using the tooling currently available on the market. In most cases, this kind of coupling is instead identified through design documents or detected through error conditions.

The Law of Demeter

Most architects are told to strive for loose coupling in system design. Less coupling between components or services makes a system more maintainable, easier to test, and less risky to deploy. It also increases the system's overall reliability, because changes impact fewer components, allowing fewer opportunities for error.

One technique for creating loosely coupled systems is called the *Law of Demeter*, otherwise known as the *Principle of Least Knowledge*. In Greek mythology, the goddess Demeter produced all the grain for the entire world, but she had no idea what people did with it (feeding livestock, making bread, and so on). Demeter was effectively decoupled from the rest of the world.

The Law of Demeter states: *a component or service should have limited knowledge of other components or services*. While this may sound simplistic and obvious, it's not. Let us show you what we mean.

Consider the components and their corresponding communications illustrated in [Figure 8-14](#). Upon accepting a customer's order, the Order Placement component must tell the Inventory Management component to decrement the inventory. If the stock goes down too low, the Order Placement component must do two things: tell the Supplier Ordering component to order more stock from the supplier, and tell the Item Pricing component to adjust the price based on the limited supply available. Finally, once all this is done, the Order Placement component tells the Email Notification component to email the customer with the order details.

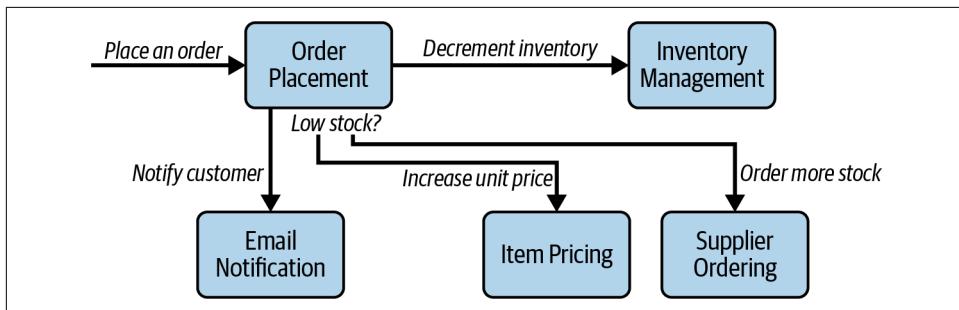


Figure 8-14. The Order Placement component is highly coupled to the rest of the system because it has too much knowledge

Notice that, in this architecture, the Order Placement component is highly coupled to the other components. While it doesn't have the *responsibility* to perform those actions, it does have the *knowledge* that these actions need to occur—and more knowledge means tighter coupling.

The idea behind the Law of Demeter is to limit each component's *knowledge* about the rest of the system. In [Figure 8-14](#), the Order Placement component knows that the inventory must be decremented, more stock may need to be ordered, the item price may need to be adjusted, and an email must be sent to the customer. (That's a lot of knowledge!) But what if that knowledge could be distributed to other components? If so, then the architect can effectively decouple that component from the rest of the system.

To see how the Law of Demeter can be applied to reduce component coupling, refer to the system in [Figure 8-14](#). If the architect added another component between Order Placement and Inventory Management to defer that *knowledge* that inventory must be decremented, the Order Placement component would still have the same efferent (outgoing) coupling level. Therefore, that coupling point needs to remain as is.

However, what about the knowledge that, if the supply goes down too low, the system must order more stock and adjust the item price? Both of those pieces of knowledge *can* be deferred to the Inventory Management component, thereby reducing the coupling level of the Order Placement component. [Figure 8-15](#) illustrates the resulting architecture after applying the Law of Demeter. Removing the *knowledge* that certain functions need to happen makes the Order Placement component less coupled to the rest of the system.

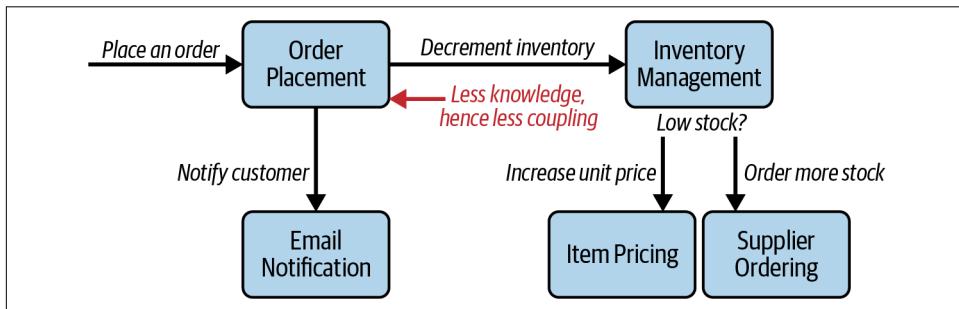


Figure 8-15. The Order Placement component is less coupled to the system when it has less knowledge

The astute reader will observe that, while applying the Law of Demeter reduced the coupling level of the Order Placement component, it also *increased* the coupling level of the Inventory Management component. Applying the Law of Demeter does not necessarily reduce the system-wide level of coupling; rather, it usually redistributes that coupling to different parts of the system.

Case Study: Going, Going, Gone—Discovering Components

If a team has no special constraints and is looking for a good general-purpose component decomposition, the Actor/Actions approach works well as a generic solution.

If the architect applies the Actor/Action approach to Going, Going, Gone (GGG), they'll find that this system has three main roles: the *bidder*, the *auctioneer*, and the *system*—a frequent participant in this modeling technique for internal actions. The roles interact with the system using the following main actions:

Bidder

- View live video stream.
- View live bid stream.
- Place a bid.

Auctioneer

- Enter live bids into system.
- Receive online bids.
- Mark item as sold.

System

- Start auction.
- Make payment.
- Track bidder activity.

Given these actions, the architect can build a set of starter components for GGG and then iterate on it. One such solution appears in [Figure 8-16](#).

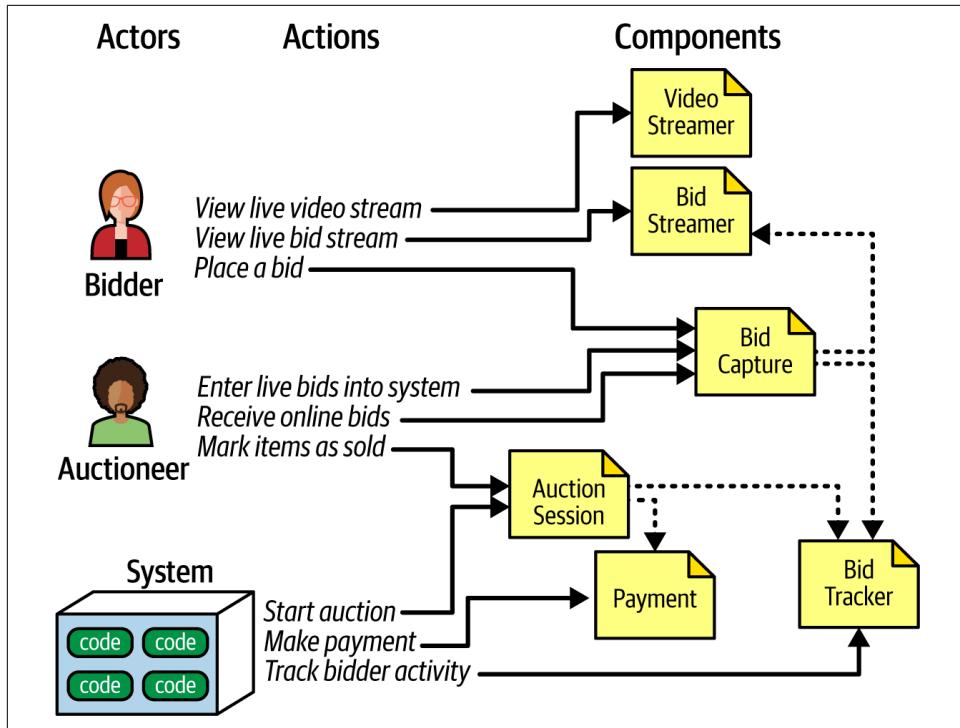


Figure 8-16. Initial set of components for Going, Going, Gone

In Figure 8-16, each role and action maps to a component. The components may need to collaborate to share information. These are the components we identified for this solution:

Video Streamer

Streams a live auction to users.

Bid Streamer

Streams bids to users as they occur. Both Video Streamer and Bid Streamer offer bidders a read-only view of the auction.

Bid Capture

This component captures bids from the auctioneer and bidders.

Bid Tracker

Tracks bids and acts as the system of record.

Auction Session

Starts an auction. When the bidder wins and the auction for that item ends, this component triggers the payment and resolution steps, including notifying bidders of the next item up for bid.

Payment

A third-party payment processor for credit card payments.

After the initial round of component identification (see [Figure 8-6](#)), the architect analyzes the previously identified architecture characteristics to determine if any of them will change the design of the component. For example, the current design features a **Bid Capture** component to capture bids from both bidders and the auctioneer. This makes sense functionally: bids from anyone can be captured and handled the same way. But what previously identified architecture characteristics does bid capture call for? The auctioneer doesn't need the same level of scalability or elasticity as the bidders, who could potentially number in the thousands.

By the same token, the auctioneer might need certain previously identified architecture characteristics more than other parts of the system do—like reliability (ensuring connections don't drop) and availability (ensuring the system stays up). For example, while it would be bad for business if a bidder can't log in to the site or suffers from a dropped connection, it would be disastrous if either of those things were to happen to the auctioneer.

To support the bidders' and auctioneer's different levels of need for the same architecture characteristics, the architect decides to split the **Bid Capture** component into two components: **Bid Capture** and **Auctioneer Capture**. The updated design appears in [Figure 8-17](#).

The architect creates a new component for **Auctioneer Capture**. They also update the information links from the **Auctioneer Capture** to both **Bid Streamer** (to show the online bidders the live bids) and **Bid Tracker** (to manage the bid streams). **Bid Tracker** is now the component that will unify two very different information streams: the auctioneer's single stream of information and the multiple streams from bidders.

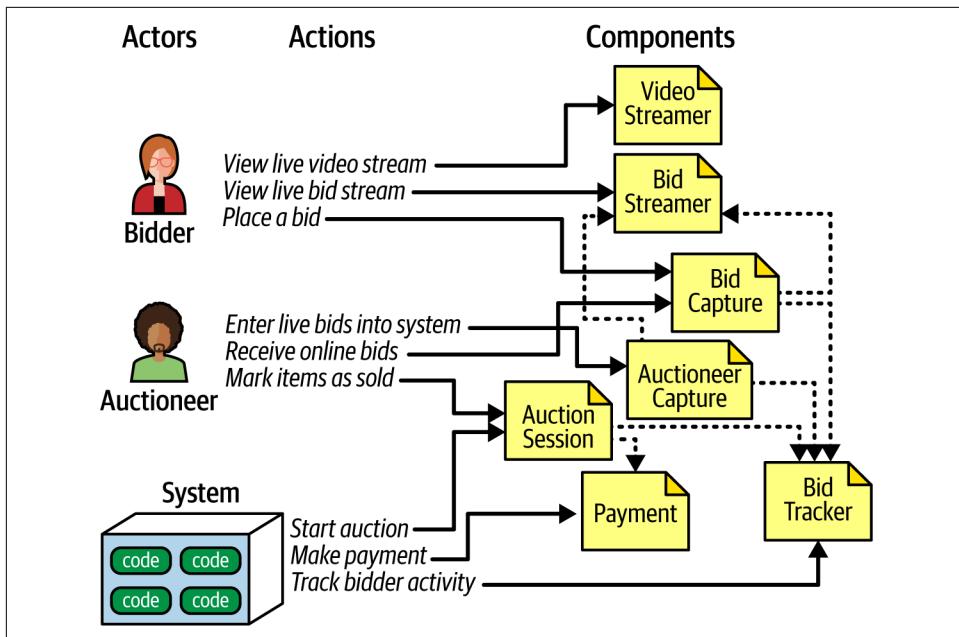


Figure 8-17. Revised components of the GGG case study

The design shown in Figure 8-17 is unlikely to be the final design. More requirements must be uncovered: how people will register new accounts, how payment functions will be administered, and so on. However, this design provides a good starting point for iterating further.

This is one possible set of components to solve the GGG problem—but it's not necessarily the best nor the only one. There are few software systems that can be implemented in only one way. Every design has different sets of trade-offs. As an architect, don't obsess over finding the “one true design,” because there are many that will suffice. Try to assess the trade-offs between different design decisions as objectively as possible, and choose the one that has the “least worst” set of trade-offs.

PART II

Architecture Styles

The difference between an architecture style and an architecture pattern can be confusing; we define it right away, in the upcoming [Chapter 9](#).

Understanding architecture styles occupies much of new architects' time and effort, because styles are important and abundant. Architects must understand the various styles and the trade-offs encapsulated within each to make effective decisions; each architecture style embodies a well-known set of trade-offs that help architects make the right choice for a particular business problem.

CHAPTER 9

Foundations

We know you are anxious to get to the details of architecture styles and patterns, but first we must cover some fundamental and definitional material to set the proper context for later chapters.

Styles Versus Patterns

We first need to distinguish between *architectural styles* and *patterns in architecture*, which are easily confused.

An architecture's style describes several different characteristics of that architecture, including its:

Component topology

An architectural style defines how components and their dependencies are organized. For example, a layered architecture organizes component layers by their technical capabilities, whereas a modular monolith organizes its components around domains (more about this difference in “[Architecture Partitioning](#)” on page 137).

Physical architecture

Often, the style dictates the type of physical architecture: either monolithic or distributed. For example, a modular monolith is generally a monolithic architecture with a single database, whereas an event-driven architecture is always distributed.

Deployment

A system's granularity and its deployment frequency are often associated with its architectural style. Teams generally deploy monolithic architectures as a single deployment along with a single relational database. Conversely, highly agile

distributed architectures, such as microservices, feature automated integration, automated provisioning, and sometimes automated deployments; they are generally deployed in pieces, with a much faster cadence.

Communication style

The architectural style also dictates how components communicate with each other. Monolithic architectures can make method calls within the monolith, whereas distributed architectures communicate via network protocols, like REST or message queues.

Data topology

Just like component topology, a system's data topology is often dictated by its architectural style. Monolithic architectures tend to have a monolithic database, whereas distributed architectures sometimes separate data—depending on the philosophy of the architecture style.

Naming a style provides a concise way to describe this complex set of factors. Each name captures a wealth of understood detail—that's one of the purposes of design patterns. However, whereas a pattern captures a contextualized solution, a style is more specific to architecture, describing the aspects listed above. An architecture style describes the architecture's topology and its assumed and default characteristics, both beneficial and detrimental. We cover a few common modern architecture patterns in [Chapter 20](#). Architects should be familiar with several fundamental styles that are the common building blocks of systems.

Where Do Architectural Styles Come From?

Contrary to popular opinion, there's no official architectural cabal that meets in an ivory tower to decide what new architectural styles come next. Rather, new styles emerge from the constantly evolving ecosystems within which architects work.

For example, say a clever architect notices that a new capability that just appeared in the ecosystem solves a particular nagging problem. They decide to combine it with several other things, new and old. Other architects see this clever solution and copy it. It becomes common enough that giving it a name makes it easier to discuss.

The microservices architecture style is a great example of this phenomenon. The rise of new DevOps capabilities, reliable open source operating systems, and the domain-driven design philosophy allowed architects to build systems in new ways to solve issues like scalability. The name *microservices* came about as a reaction to the common architecture styles at the time, which featured large services and extensive orchestration. Microservices is a label, not a description. It is not a commandment for teams to build the smallest services they can but rather a way to refer to the architecture style.

Fundamental Patterns

Several fundamental patterns appear again and again throughout the history of software architecture, generally because they provide a useful perspective on organizing code, deployments, or other aspects of architecture. For example, the concept of layers separating different concerns based on functionality is as old as software itself. Yet layers (in both styles and patterns) continue to manifest in different guises, including the modern variants we discuss in [Chapter 10](#).

Alas, there's another common antipattern that stems from the absence of architecture: the Big Ball of Mud.

Big Ball of Mud

Architects refer to the absence of any discernible architecture structure as a *Big Ball of Mud*, named after the antipattern Brian Foote and Joseph Yoder defined in a 1997 paper presented at the 1997 conference on Patterns Languages of Programs.

A *Big Ball of Mud* is a haphazardly structured, sprawling, sloppy, duct-tape-and-baling-wire, spaghetti-code jungle. These systems show unmistakable signs of unregulated growth, and repeated, expedient repair. Information is shared promiscuously among distant elements of the system, often to the point where nearly all the important information becomes global or duplicated.

The overall structure of the system may never have been well defined.

If it was, it may have eroded beyond recognition. Programmers with a shred of architectural sensibility shun these quagmires. Only those who are unconcerned about architecture, and, perhaps, are comfortable with the inertia of the day-to-day chore of patching the holes in these failing dikes, are content to work on such systems.

Today, *Big Ball of Mud* might describe a simple scripting application with no real internal structure that has its event handlers wired directly to database calls. Many trivial applications start like this, then become unwieldy as they grow.

In general, avoid this type of architecture at all costs. Its lack of structure makes change increasingly difficult. Such architectures also suffer from problems with deployment, testability, scalability, and performance. The larger these systems become, the worse the pain caused by their lack of architecture becomes.

Unfortunately, this antipattern occurs quite often in the real world. Few architects intend to create a mess, but many projects inadvertently manage to, usually because of a lack of governance around code quality and structure. For example, Neal worked on one client project whose structure appears in [Figure 9-1](#).

The client (whose name is withheld for obvious reasons) created a Java-based web application as quickly as possible over several years. The technical visualization in [Figure 9-1](#) shows its coupling: each dot on the perimeter of the circle represents a

class, and each line represents a connection between the classes, with bolder lines indicating stronger connections. In this code base, changing a class in any way makes it difficult to predict the effect on other classes, making change a terrifying affair.

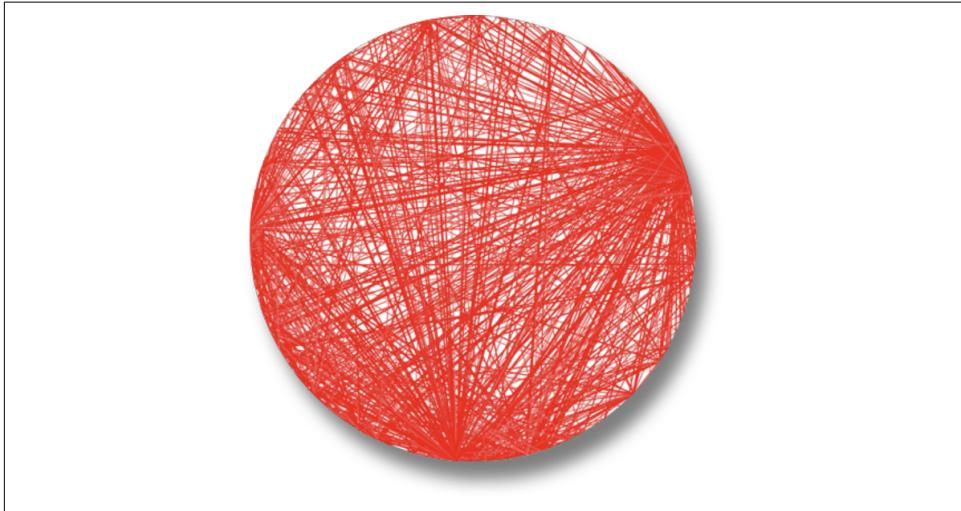


Figure 9-1. A Big Ball of Mud architecture visualized from a real code base. (Made with a now-retired tool called XRay, an Eclipse plug-in.)

The problem with Big Ball of Mud architectures isn't just their lack of structure. Because everything is coupled to everything else, changes tend to have hard-to-predict rippling side effects. This problem can reach a critical point where developers spend all their time chasing bugs and their side effects, rather than working on new features.

Unitary Architecture

In the beginning, there was only the computer, and software ran on it. The two started as a single entity, then split as they evolved and the need grew for more sophisticated capabilities. For example, mainframe computers started as singular systems, then gradually separated data into its own kind of system. Similarly, when personal computers (PCs) were first commercially developed, the focus was largely on single machines. As networking PCs became common, distributed systems (such as client/server) appeared.

Client/Server

Few unitary architectures exist outside embedded systems and other highly constrained environments. Generally, software systems tend to add functionality over time, and separating concerns becomes necessary to maintain their operational architecture characteristics, such as performance and scale.

Many architecture styles deal with how to separate parts of the system efficiently. One fundamental style in architecture separates technical functionality between frontend and backend: this is called a *two-tier*, or *client/server*, architecture. It comes in different flavors, depending on the era and the system's computing capabilities.

Desktop and database server

One early PC architecture encouraged developers to write rich desktop applications in user interfaces like Windows, separating data into a separate database server. This architecture coincided with the appearance of standalone database servers that could connect through standard network protocols. This allowed presentation logic to reside on the desktop, while the more computationally intense action (both in volume and complexity) occurred on more robust database servers.

Browser and web server

Once modern web development arrived, it became common to split architectures into a web browser connected to a web server (which, in turn, was connected to a database server). This separation of responsibilities was similar to the desktop variant but with even thinner clients as browsers, allowing a wider distribution both inside and outside firewalls. Even though the database is separate from the web server, many architects still consider this a two-tier architecture, because the web and database servers run on one class of machine within the operations center, while the UI runs on the user's browser.

Single-page JavaScript applications

As web responsiveness improved, so did JavaScript implementations in browsers. A family of client/server style applications emerged that resembles the original desktop variation, but with the rich client written in JavaScript in the browser, rather than as a desktop application.

As this section illustrates, there will always be layers to separate different parts of architectures, depending on the needs of the application and the capabilities of the platform.

Three-tier

Three-tier architecture, which provided even more layers of separation, became popular during the late 1990s. As tools like application servers became prominent in Java and .NET, companies started building even more layers into their topologies. One system might have a database tier using an industrial-strength database server; an application tier managed by an application server; and a frontend coded in generated HTML and, increasingly, JavaScript (as its capabilities expanded).

The three-tier architecture corresponded with network-level protocols such as [Common Object Request Broker Architecture \(CORBA\)](#) and [Distributed Component Object Model \(DCOM\)](#) to facilitate building distributed architectures.

Just as developers today don't worry about how network protocols like TCP/IP work (because they just work), most architects don't have to worry about this level of plumbing in distributed architectures. The capabilities offered by that era's tools exist today either as tools (like message queues) or as architecture patterns (such as event-driven architecture, covered in [Chapter 15](#)).

Three-Tier Architectures, Language Design, and Long-Term Implications

During the 1990s, as the Java language was designed, three-tier computing was all the rage. People assumed that, in the future, all systems would have three-tier architectures. One of the common headaches with the existing languages at the time, such as C++, was how cumbersome it was to move objects over the network in a consistent way between systems. So Java's designers decided to build this capability into its core, using a mechanism called *serialization*.

Every Java object implements an interface that requires it to support serialization. The designers figured that since three-tiered architecture would be around forever, baking it into the language would offer great convenience. Of course, that architectural style came and went—yet the leftovers appear in Java to this day. Even though virtually no one still uses serialization, new Java features must support it for backward compatibility, greatly frustrating language designers.

Understanding the long-term implications of design decisions has always eluded us, in software as in other engineering disciplines. The perpetual advice to favor simple designs is in many ways a future-proofing strategy.

Architecture Partitioning

The First Law of Software Architecture states that everything in software is a trade-off, and that includes how architects partition components in an architecture. Because components represent a general containment mechanism, architects can partition them any way they want. There are several common styles, with different sets of trade-offs. One particular type of component arrangement has an outsized impact: *top-level partitioning*.

Of the two architecture styles depicted in [Figure 9-2](#), one will be familiar to many: the *layered monolith* (discussed in detail in [Chapter 10](#)). The other, called *modular monolith*, is an architecture style popularized by [Simon Brown](#) that consists of a single deployment unit, associated with a database and partitioned around domains rather than technical capabilities (discussed in [Chapter 11](#)). These two styles represent different methods of top-level partitioning. Note that in both variations, each top-level layer or component likely has other components embedded within it. Top-level partitioning is of particular interest to architects because it defines a fundamental architecture style and way of partitioning code.

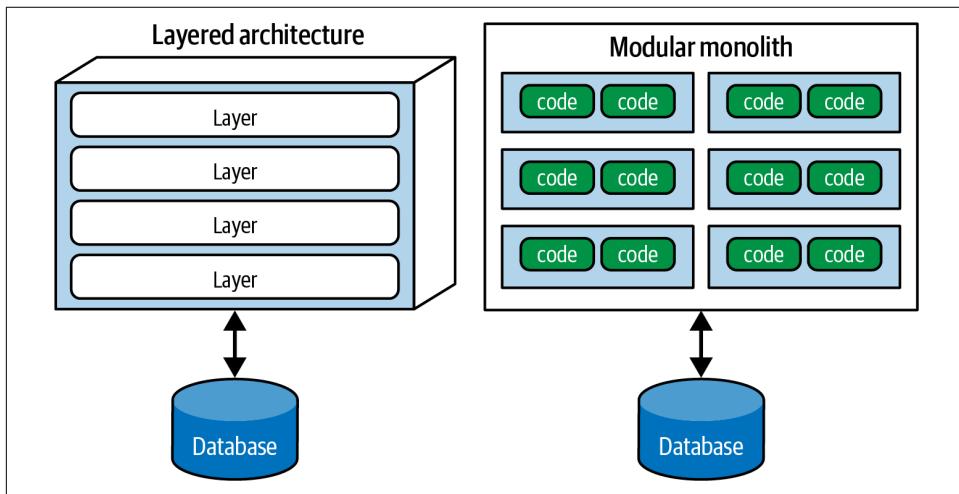


Figure 9-2. Two types of top-level partitioning: technical (such as the layered architecture) and domain (such as the modular monolith)

Organizing architecture based on its technical capabilities, like the layered monolith style does, represents *technical top-level partitioning*.

A common version of this appears in [Figure 9-3](#), where the architect has partitioned the system's functionality into *technical* capabilities: presentation, business rules, services, persistence, and so on. This way of organizing a code base certainly makes sense. All the persistence code resides in one layer, making it easy for developers

to find persistence-related code. Even though the basic concept of layered architecture predates it by decades, the Model-View-Controller design pattern (one of the fundamental patterns in *Head First Design Patterns* by Eric Freeman and Elisabeth Robson (O'Reilly, 2020)), matches with this architectural pattern, making it easy for developers to understand. Thus, in many organizations, it is the default architecture.

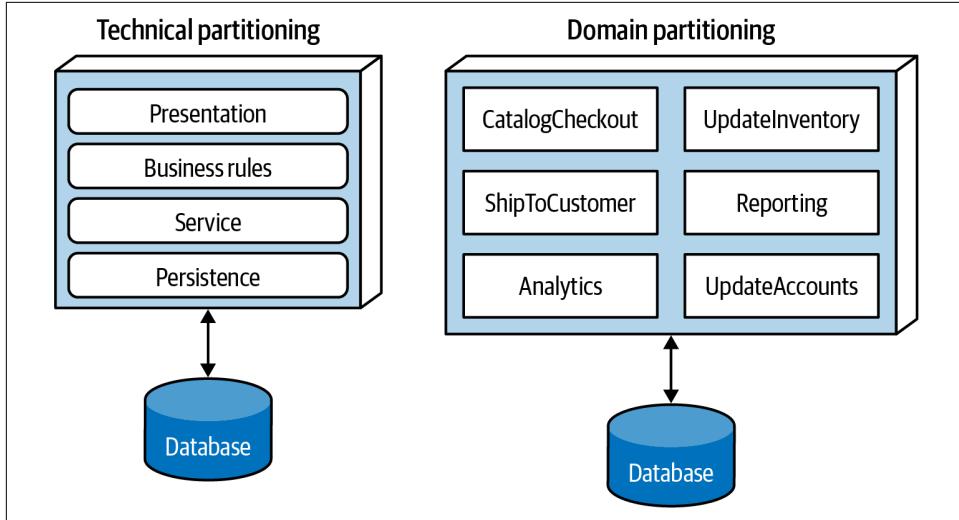


Figure 9-3. Two types of top-level partitioning in architecture

An interesting side effect of layered architecture's predominance relates to how companies often organized the seating in their physical offices according to different project roles. Because of Conway's Law, when using a layered architecture, it makes some sense to have all the backend developers sit together in one department, the DBAs in another, the presentation team in another, and so on.

The other architectural variation in Figure 9-3 is *domain partitioning*, a modeling technique for decomposing complex software systems that organizes components by domain rather than technical capabilities. Domain partitioning was inspired by Eric Evans's book *Domain-Driven Design*. In DDD, the architect identifies domains or workflows, independent and decoupled from each other. The microservices architecture style is based on this philosophy.

A modular monolith architecture is partitioned around domains or workflows, rather than technical capabilities. Because components often nest within one another, each component in the domain-partitioned architecture shown in Figure 9-3 (for example, CatalogCheckout) may use a persistence library and have a separate layer for business rules, but the top-level partitioning will still revolve around domains.

Conway's Law

Back in the late 1960s, [Melvin Conway](#) made an observation that has become known as *Conway's Law*:

Organizations which design systems...are constrained to produce designs which are copies of the communication structures of these organizations.

Paraphrased, this law suggests that when a group of people designs some technical artifact, the structures of their design will replicate how those people communicate. People at all levels of organizations see this law in action, and they sometimes base decisions on it. For example, while it's common for organizations to partition workers based on technical capabilities, this is an artificial separation of common concerns that can hamper collaboration.

A related observation, coined by Jonny Leroy of Thoughtworks, is the [*Inverse Conway Maneuver*](#), which suggests evolving the structures of teams and organizations together to promote the desired architecture. This consideration has since become universally known as *team topologies*.

Organizations have begun to realize that team topologies can have a significant impact on many important facets of their business, including software architecture. In the upcoming style-focused chapters, we discuss each architecture style's effect on these team types.

One of the fundamental distinctions between architecture patterns is what type of top-level partitioning each supports. In the style-specific chapters that follow, we cover this distinction for each individual pattern. Top-level partitioning also has a huge impact on whether the architect decides to initially identify components technically or by domain.

Architects using technical partitioning organize the components of the system by their technical capabilities: presentation, business rules, persistence, and so on. One of the organizing principles of the layered architecture is *separation of technical concerns*, which creates useful levels of decoupling. For example, if the Service layer is only connected to the Persistence layer below and the Business Rules layer above, then changes in persistence will potentially affect only those layers. This decoupling reduces the potential for rippling side effects on dependent components.

It's certainly logical to organize systems using technical partitioning, but, like all things in software architecture, there are some trade-offs. The separation enforced by technical partitioning enables developers to find certain categories of the code base quickly, since it is organized by capabilities, but most realistic software systems require workflows that cut across technical capabilities.

In the technically partitioned architecture shown in [Figure 9-4](#), consider the common business workflow of CatalogCheckout. The code to handle CatalogCheckout in the technically partitioned architecture appears in all the layers. In other words, the domain is smeared across the technical layers.

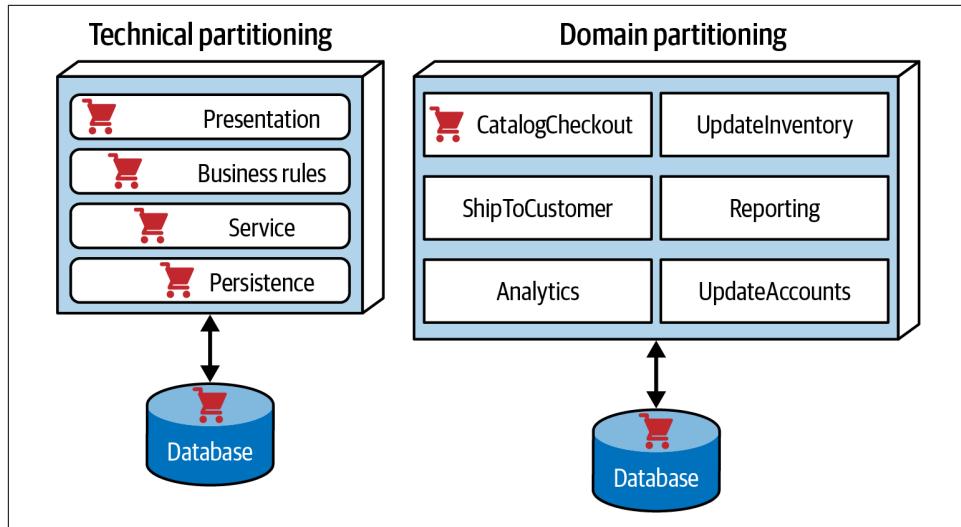


Figure 9-4. Where domains/workflows appear in technically partitioned and domain-partitioned architectures

Contrast this with the domain-partitioned architecture shown in [Figure 9-4](#), in which the architects have built top-level components around workflows and domains. Each component may have subcomponents, including layers, but the top-level partitioning focuses on domains, which better reflects the kinds of changes that most often occur on projects.

Neither of these styles is more correct than the other. (Refer to the First Law of Software Architecture.) That said, we have observed a decided industry trend over the last few years toward domain partitioning for both monolithic and distributed (for example, microservices) architectures. As we've noted, this is one of the first decisions an architect must make.

Kata: Silicon Sandwiches—Partitioning

Consider the case of one of our example katas, “[Kata: Silicon Sandwiches](#)” on page 71. Let’s start by considering the first of two possibilities for Silicon Sandwiches: domain partitioning, shown in [Figure 9-5](#).

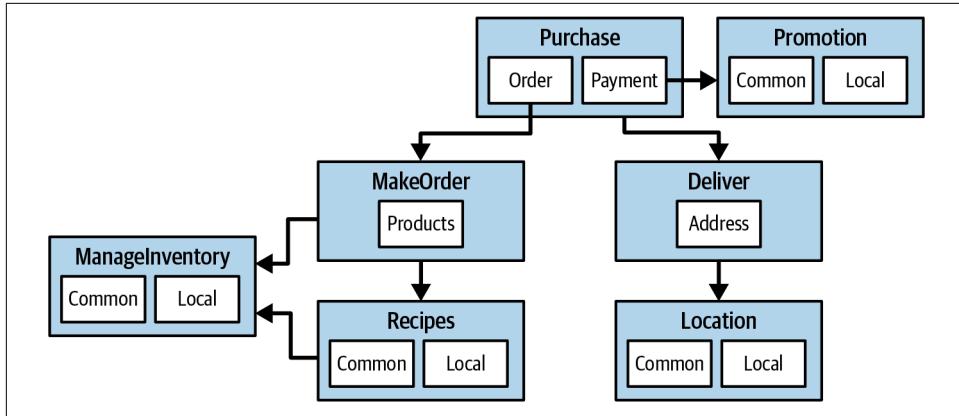


Figure 9-5. A domain-partitioned design for Silicon Sandwiches

In [Figure 9-5](#), the architect has designed around domains (workflows), creating discrete components for Purchase, Promotion, MakeOrder, ManageInventory, Recipes, Delivery, and Location. Within many of these components reside subcomponents to handle both common and local variations types of customization.

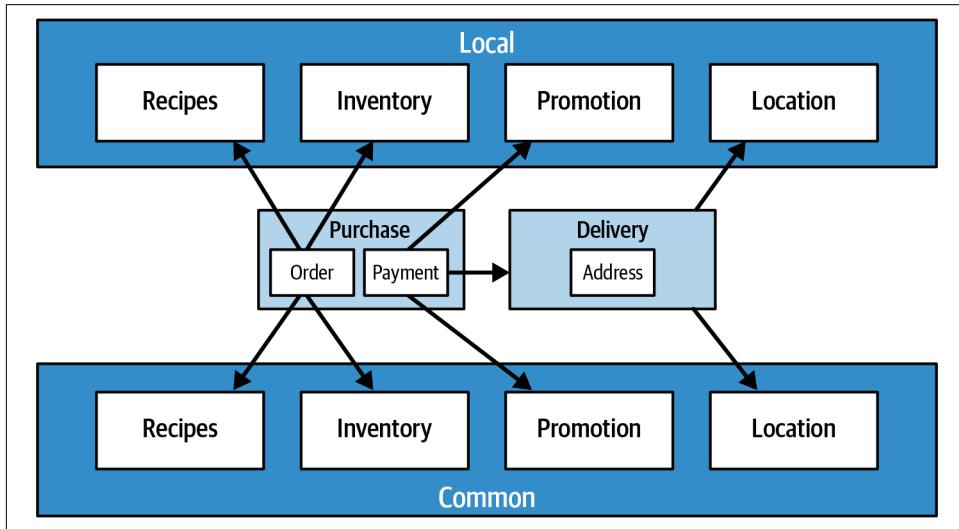


Figure 9-6. A technically partitioned design for Silicon Sandwiches

An alternative design isolates the common and local parts into their own partitions, as illustrated in [Figure 9-6](#). Common and Local represent top-level components, while Purchase and Delivery remain to handle the workflow.

Which is better? It depends! Each kind of partitioning offers different advantages and drawbacks.

Domain partitioning

Domain-partitioned architectures separate top-level components by workflows and/or domains.

Advantages

- Modeled more closely on how the business functions rather than on an implementation detail
- Easier to build cross-functional teams around domains
- Aligns more closely to the modular monolith and microservices architecture styles
- Message flow matches the problem domain
- Easy to migrate data and components to a distributed architecture

Disadvantage

- Customization code appears in multiple places

Technical partitioning

Technically partitioned architectures separate top-level components based on technical capabilities rather than discrete workflows. This may manifest as layers inspired by Model-View-Controller separation or some other ad hoc technical partitioning. The architecture pictured in [Figure 9-6](#) separates its components based on customization.

Advantages

- Clearly separates customization code.
- Aligns more closely to the layered architecture pattern.

Disadvantages

- Higher degree of global coupling. Changes to the Common or Local component will likely affect all the other components.
- Developers may have to duplicate domain concepts in both the Common and Local layers.
- Typically, higher coupling at the data level. In a system like this, the application architects and the data architects would likely collaborate to create a single database, including customization and domains. That in turn would create difficulties in untangling the data relationships later, if the architects eventually want to migrate this architecture to a distributed system. Many other factors contribute to choosing an architecture style, as we cover in [Part II](#).

Monolithic Versus Distributed Architectures

As you learned in Part I, architecture styles can be classified into two main types: *monolithic* (single deployment unit of all code) and *distributed* (multiple deployment units connected through remote access protocols). While no classification scheme is perfect, distributed architectures all share a common set of challenges and issues not found in the monolithic architecture styles, making this classification scheme a good separation between the various architecture styles.

In Part II of this book, we describe the following architecture styles in detail:

Monolithic

- Layered architecture ([Chapter 10](#))
- Pipeline architecture ([Chapter 12](#))
- Microkernel architecture ([Chapter 13](#))

Distributed

- Service-based architecture ([Chapter 14](#))
- Event-driven architecture ([Chapter 15](#))
- Space-based architecture ([Chapter 16](#))
- Service-oriented architecture ([Chapter 17](#))
- Microservices architecture ([Chapter 18](#))

Distributed architecture styles, although much more powerful in terms of performance, scalability, and availability than monolithic architecture styles, have significant trade-offs. The first group of issues facing all distributed architectures are described in the “[fallacies of distributed computing](#)”, first listed by L. Peter Deutsch and other colleagues from Sun Microsystems in 1994. A *fallacy* is something false that someone believes or assumes to be true. All eight of the fallacies of distributed computing apply to distributed architectures today. The following sections describe each fallacy.

Fallacy #1: The Network Is Reliable

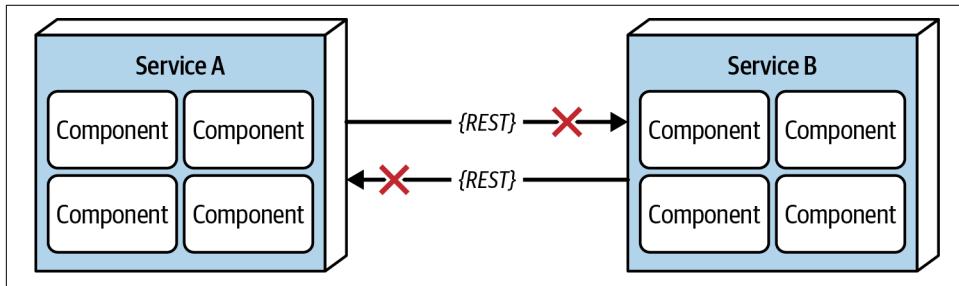


Figure 9-7. The network is not reliable

Developers and architects alike assume that the network is reliable, but it is not. While networks have become more reliable over time, the fact of the matter is that networks still remain generally unreliable. This is significant for all distributed architecture styles, because they rely on the network for communicating to, from, and between services. As illustrated in Figure 9-7, Service B may be totally healthy, but Service A cannot reach it due to a network problem. Even worse, Service A might request that Service B process some data, but receive no response because of a network issue. This is why things like timeouts and circuit breakers exist between services. The more a system relies on the network (as microservices architectures notably do), the more potential it has to become unreliable.

Fallacy #2: Latency Is Zero

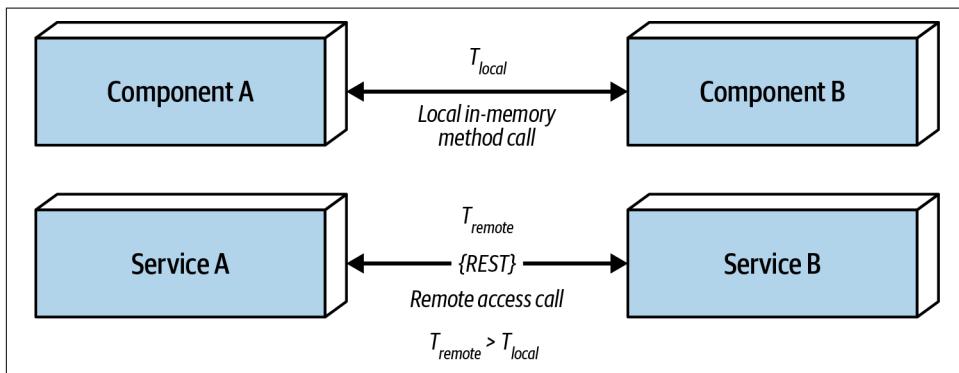


Figure 9-8. Latency is not zero

As Figure 9-8 shows, when a local call is made to another component via a method or function call, the time to access that component (t_{local}) is measured in nanoseconds or microseconds. However, when that same call is made through a remote access protocol (such as REST, messaging, or RPC), the time (t_{remote}) is measured

in milliseconds and thus will always be greater than t_{local} . Latency, in any distributed architecture, is not zero—yet most architects ignore this fallacy, insisting that they have fast networks. Ask yourself: do you know what the average round-trip latency is for a RESTful call in your production environment? Is it 60 milliseconds? Is it 500 milliseconds?

When considering using any distributed architecture, particularly microservices, architects must know this latency average. It is the only way of determining whether a distributed architecture is feasible, due to the fine-grained nature of the services and the amount of communication between them.

For example, say we assume an average of 100 ms of latency per request. Chaining service calls together to perform a particular business function adds 1,000 ms to the request! Knowing the average latency is important, but knowing the 95th to 99th percentile latency is even more important. While the system's average latency might be only 60 ms (which is good), the 95th percentile might be 400 ms! It's usually this “long tail” latency that will kill performance in a distributed architecture. In most cases, a network administrator can provide latency values (see “[Fallacy #6: There Is Only One Administrator](#)” on page 148).

Fallacy #3: Bandwidth Is Infinite

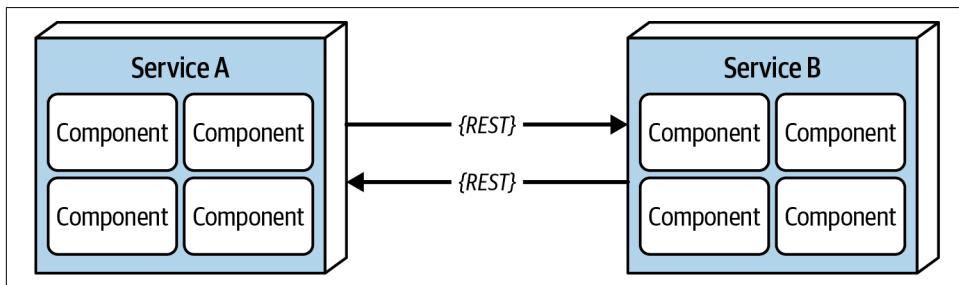


Figure 9-9. Bandwidth is not infinite

Bandwidth is usually not a concern in monolithic architectures, because once a business request goes into a monolith, little or no bandwidth is required to process it. However, as shown in [Figure 9-9](#), once a system is broken apart into smaller deployment units (services) in a distributed architecture, such as microservices, communication to and between these services uses significant bandwidth. This slows the network, impacting latency (fallacy #2) and reliability (fallacy #1).

To illustrate the importance of this fallacy, consider the two services shown in [Figure 9-9](#). Let's say Service A manages the wish list items for the website, and Service B manages the customer profiles. Whenever a request for a wish list comes into Service A, Service A needs the customer name for the response contract for the wish list. To get the name, it must make an interservice call to Service B. Service B returns 45 attributes, totaling 500 KB, to Service A, which only needs the name (200 bytes). This may not sound significant, but requests for the wish list items happen about 2,000 times a second. This means that Service A is calling Service B 2,000 times a second. At 500 KB for each request, *each* interservice call takes 1 GBps of bandwidth!

This form of coupling, called *stamp coupling*, consumes significant amounts of bandwidth in distributed architectures. If Service B were to pass back *only* the 200 bytes of data that Service A needs, it would use only 400 Kbps in total.

Stamp coupling can be resolved by:

- Creating private RESTful API endpoints
- Using field selectors in contracts
- Using [GraphQL](#) to decouple contracts
- Using value-driven contracts with consumer-driven contracts
- Using internal messaging endpoints

Regardless of the technique used, the best way to address this fallacy in a distributed architecture is to ensure that services or systems transmit only the necessary data.

Fallacy #4: The Network Is Secure

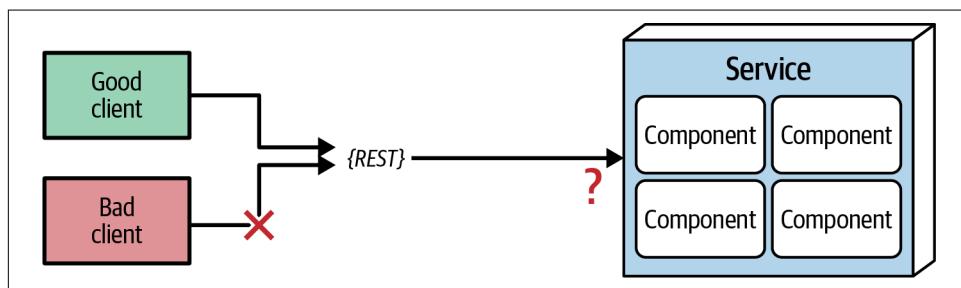


Figure 9-10. The network is not secure

Most architects and developers get so comfortable using virtual private networks (VPNs), trusted networks, and firewalls that they tend to forget about this fallacy of distributed computing—but *the network is not secure*. As shown in [Figure 9-10](#), each and every endpoint to each distributed deployment unit must be secured against

unknown or bad requests. The surface area for threats and attacks increases by magnitudes when moving from a monolithic to a distributed architecture, making security much more challenging. Securing every endpoint, even in interservice communication, is another reason performance tends to be slower in synchronous, highly distributed architecture styles, such as microservices and service-based architectures.

Fallacy #5: The Topology Never Changes

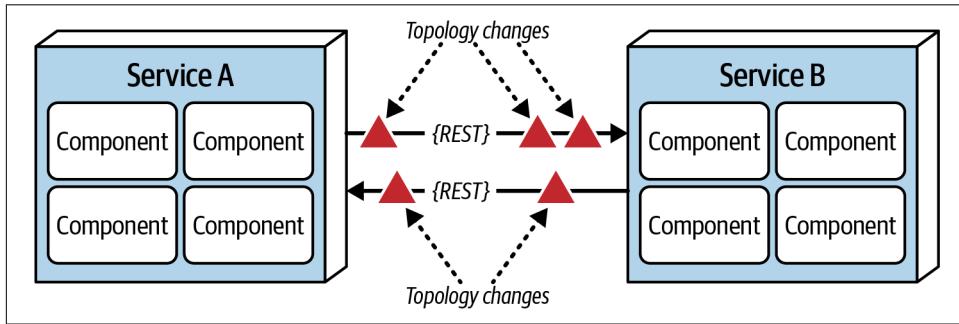


Figure 9-11. The network topology always changes

Fallacy #5, as shown in [Figure 9-11](#), refers to the overall network topology, including all of its routers, hubs, switches, firewalls, networks, and appliances. Architects assume that the topology is fixed and never changes. *Of course it changes.* It changes all the time. What is the significance of this fallacy?

Suppose you come into work on a Monday morning and everyone is running around like crazy, because services keep timing out in production. You work with the teams, frantically trying to figure out why this is happening. No new services were deployed over the weekend. What could it be? After several hours, you discover that a supposedly “minor” network upgrade at two o’clock that morning invalidated all of the system’s latency assumptions, triggering timeouts and circuit breakers.

Architects must be in constant communication with operations and network administrators about what is changing and when so that they can make adjustments to avoid such surprises. This may seem obvious and easy, but it is neither. As a matter of fact, this fallacy leads directly to the next fallacy.

Fallacy #6: There Is Only One Administrator



Figure 9-12. There are many network administrators, not just one

Architects fall into this fallacy all the time: assuming they only need to collaborate and communicate with one administrator. As Figure 9-12 shows, there are dozens of network administrators in a typical large company. With whom should the architect talk about latency or topology changes? This fallacy points to the complexity of distributed architecture and the amount of coordination that must happen to get everything working correctly. A monolithic application, with its single deployment unit, doesn't require this level of communication and collaboration.

Fallacy #7: Transport Cost Is Zero

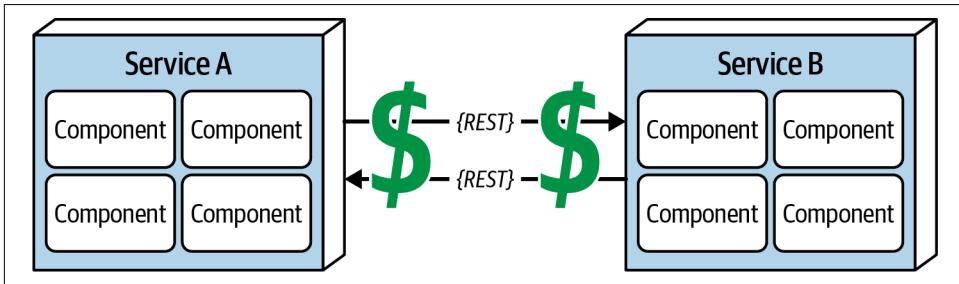


Figure 9-13. Remote access costs money

Many software architects confuse this fallacy, shown in Figure 9-13, with fallacy #2 (latency is zero). *Transport cost* here refers not to latency, but to the actual *monetary cost* of making a “simple RESTful call.” Architects incorrectly assume that the necessary and sufficient infrastructure is already in place for making a simple RESTful call or breaking apart a monolithic application. *It is usually not.* Distributed architectures cost significantly more than monolithic architectures, primarily due to increased needs for hardware, servers, gateways, firewalls, new subnets, proxies, and so on.

We encourage architects embarking on a distributed architecture to analyze their current server and network topology with regard to capacity, bandwidth, latency, and security zones, to avoid getting surprised by this fallacy.

Fallacy #8: The Network Is Homogeneous

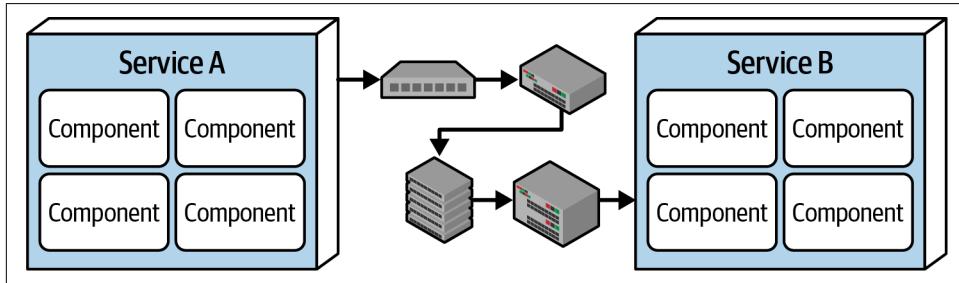


Figure 9-14. The network is not homogeneous

Most architects and developers assume that a network is homogeneous, as shown in Figure 9-14—that it's made up of network hardware from only one vendor. Nothing could be further from the truth. Most companies' infrastructures have multiple network-hardware vendors.

So what? The significance of this fallacy is that not all of those heterogeneous hardware vendors play together well. Does Juniper Networks hardware integrate seamlessly with Cisco Systems hardware? Most of it works, and networking standards have evolved over the years, making this less of an issue. However, not all situations, loads, and circumstances have been fully tested, so network packets do occasionally get lost. This in turn affects network reliability and assumptions and assertions about latency and bandwidth. In other words, this fallacy ties back into all of the other fallacies, forming an endless loop of confusion and frustration when dealing with networks (which is inescapable when using distributed architectures).

The Other Fallacies

The eight fallacies previously listed form a famous set of observations—every architect learns them either from Deutsch's list or the hard way, one by one, over the course of their career. The authors have also learned some painful, near-universal lessons that we offer as an extension to that famous list.

Fallacy #9. Versioning is easy

When two services need to communicate, they pass information in a *contract*, which includes information required for the communication. Often, a service's internal implementation evolves over time, changing fields that the service accepts and passes to other services. One way to solve this problem is to use versioning for the

contract—that is, create different versions for the old and new contracts including different sets of information. However, this seemingly simple decision leads to a host of trade-offs:

- Should the team version at the individual service level or for the whole system?
- How far should the versioning reach? What portion of the architecture will need to support it?
- How many versions should the team support at any given time? (Some teams accidentally find themselves honoring dozens of different versions for different purposes.)
- Should the team deprecate older versions at the system level or service by service?

While versioning is a reasonable approach for evolving communication between services, it has a host of trade-offs that architects should anticipate.

Fallacy #10. Compensating updates always work

Compensating updates is an architectural pattern in which some mechanism (like an *Orchestrator* service) makes sure that several related services all update jointly. If they don't, the orchestrator reverses the update. The *compensating update* is from the orchestrator that issues a reversing operation to put the state back to what it was before.

This is a common pattern that most architects blithely assume always works... but it doesn't. What happens if the compensating update fails? When architects demonstrate how complex interactions in distributed architectures like microservices work, they must also show how compensating updates work. Thus, architects designing transactional workflows in microservices should accommodate the "normal" compensation workflow, but must also consider how to recover if the update and the compensating update (or a portion of it) *both* fail.

Fallacy #11. Observability is optional (for distributed architectures)

A common architectural characteristic for architects to prioritize in distributed architectures is *observability*: the ability to observe each service's interactions with other services and the ecosystem, as captured through monitors or logs. While logging is useful in monolithic architectures, it is *critical* in distributed architectures, which offer many communication failure modes that are hard to debug without comprehensive interaction logs.

Team Topologies and Architecture

Architects and teams have done lots of research on the intersection of architecture and team topologies, above and beyond the implications of architecture partitioning, which we previously discussed. The very influential book *Team Topologies* by Matthew Skelton and Manuel Pais (IT Revolution Press, 2019) defines several team types that intersect with software architecture:

Stream-aligned teams

In team topologies terminology, a *stream* is a stream of work scoped to a particular business domain or capability. *Stream-aligned teams* focus narrowly on a single line of work, such as a product, service, or specific set of features.

The goal of stream-aligned teams is to move as quickly as possible because they are delivering discrete value to the organization. Consequently, the other team types are designed to reduce any friction that could impede the stream-aligned teams.

Enabling teams

An *enabling team* bridges a gap in some capability, providing a place for necessary research, learning, and other tasks that are important but not urgent. They supply knowledge and resources from specialized domains to support stream-aligned teams. Good enabling teams are highly collaborative and proactive.

Complicated-subsystem teams

Many systems include highly specialized systems or parts that require similarly specialized skills. Members of *complicated-subsystem teams* fully understand a complex subsystem or domain and can help a stream-aligned team apply it. Their goal is to reduce other teams' cognitive load.

Platform teams

A *platform team* provides internal services and building blocks for solutions. As defined by [Evan Botcher](#), a platform is

a foundation of self-service APIs, tools, services, knowledge and support which are arranged as a compelling internal product. Autonomous delivery teams can make use of the platform to deliver product features at a higher pace, with reduced coordination.

Platform teams support the other teams, attempting to remove needless friction while providing necessary governance around concerns such as quality and security.

On to Specific Styles

Architects need to understand a number of different architecture styles before they can perform trade-off analysis. Each style supports a different set of architectural characteristics—each one has a “sweet spot” that it handles best. By learning the different styles and their underlying philosophies, an architect better understands when each one works best (or least worst, anyway).

Layered Architecture Style

The *layered* architecture style, also known as *n-tiered*, is one of the most common architecture styles. It's the de facto standard for many legacy applications because of its simplicity, familiarity, and low cost.

The layered architecture style can fall into several architectural antipatterns, including the *Architecture by Implication* and the *Accidental Architecture* antipatterns. When developers or architects “just start coding,” unsure which architecture style they are using, chances are good that they’re implementing the layered architecture style.

Topology

Components within the layered architecture style are organized into logical horizontal layers, as shown in [Figure 10-1](#), with each layer performing a specific role within the application (such as presentation logic or business logic). Although there are no specific restrictions in terms of the number and types of layers that must exist, most layered architectures consist of four standard layers: Presentation, Business, Persistence, and Database. Some architectures combine the Business and Persistence layers, particularly when the persistence logic (such as SQL or HSQL) is embedded within the Business layer components. Smaller applications may have only three layers, whereas larger and more complex business applications may contain five or more.

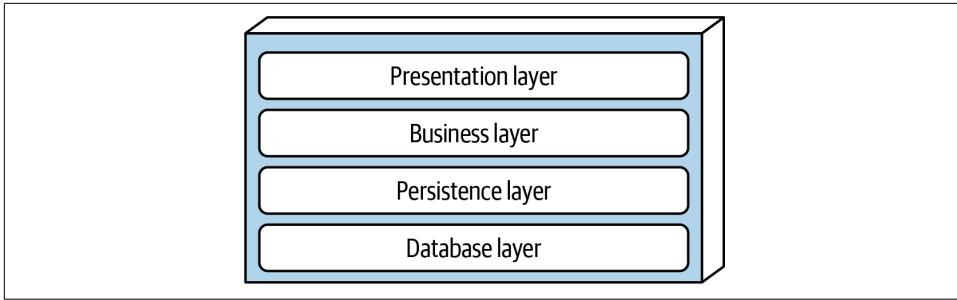


Figure 10-1. Standard logical layers within the layered architecture style

Figure 10-2 illustrates topology variants from a physical layering (deployment) perspective. The first variant combines the Presentation, Business, and Persistence layers into a single deployment unit, with the Database layer typically represented as a separate external physical database (or filesystem). The second variant physically separates the Presentation layer into its own deployment unit, with the Business and Persistence layers combined into a second deployment unit. Again, with this variant, the Database layer is usually physically separated through an external database or filesystem. A third variant combines all four standard layers into a single deployment, including the Database layer. This variant might be useful for smaller applications with an internally embedded database or an in-memory database, such as mobile device applications. Many on-premises (on-prem) products are built and delivered to customers using this third variant.

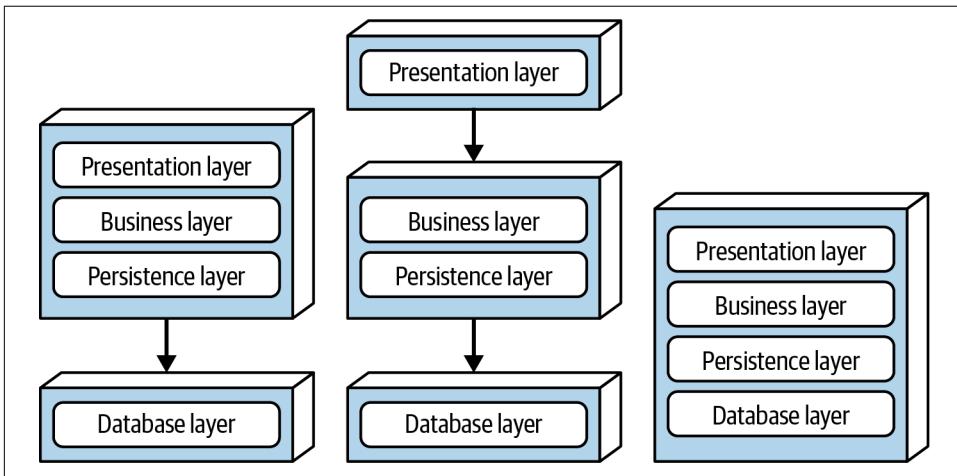


Figure 10-2. Physical topology (deployment) variants

Each layer has a specific role and responsibility, and forms an abstraction around the work that needs to be done to satisfy a particular business request. For example, the Presentation layer is responsible for handling all UI and browser communication logic, whereas the Business layer is responsible for executing specific business rules associated with the request. The Presentation layer doesn't need to know or worry about how to get customer data; it only needs to display that information on a screen in a particular format. Similarly, the Business layer doesn't need to be concerned about how to format customer data for display on a screen or even where that data is coming from; it only needs to get the data from the Persistence layer, perform business logic against it (such as calculating values or aggregating data), and pass that information up to the Presentation layer.

This separation of concerns within the layered architecture style makes it easy to build effective role and responsibility models. Components within a specific layer are limited in scope, dealing only with the logic that pertains to that layer. For example, components in the Presentation layer only handle presentation logic, whereas components residing in the Business layer only handle business logic. This allows developers to leverage their particular technical expertise to focus on the technical aspects of the domain (such as presentation logic or persistence logic). The trade-off of this benefit, however, is a lack of overall *holistic agility* (the entire system's ability to respond quickly to change).

The layered architecture is a *technically partitioned* architecture (as opposed to *domain-partitioned* architecture). This means, as you learned in [Chapter 9](#), that components are separated by their technical role in the architecture (such as presentation or business) rather than by domain (such as customer). As a result, any particular business domain is spread throughout all of the layers of the architecture. For example, the domain of “customer” is contained in the Presentation layer, Business layer, Rules layer, Services layer, and Database layer, making it difficult to apply changes to that domain. As a result, a DDD approach does not fit particularly well with the layered architecture style.

Style Specifics

Layers in this architectural style encapsulate specific areas of technical responsibility, but the layers themselves may exhibit other characteristics.

Layers of Isolation

Each layer can be either closed or open. If a layer is *closed*, then as a request moves from the top layer down to the bottom layer, the request cannot skip any layers. It must go through the layer immediately beneath to get to the next layer (see [Figure 10-3](#)). For example, in an architecture where all the layers are closed, a request

originating from the Presentation layer must first go through the Business layer and then to the Persistence layer before finally making it to the Database layer.

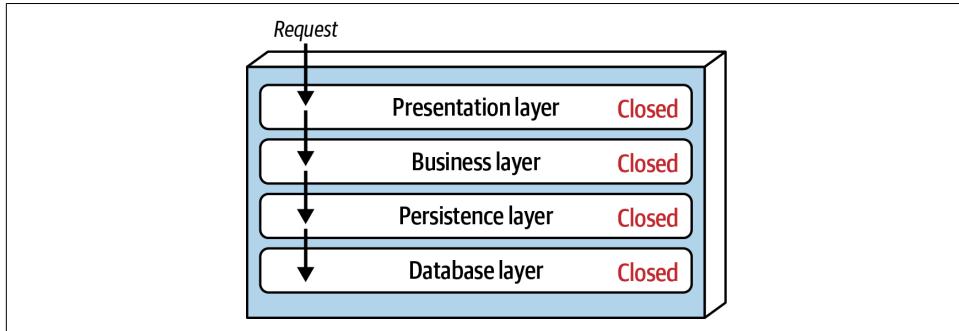


Figure 10-3. Closed layers within a layered architecture

Notice that in [Figure 10-3](#) it would be much faster and easier for the Presentation layer to access the database directly for simple retrieval requests, bypassing any unnecessary layers (what used to be known in the early 2000s as the *Fast-Lane Reader* pattern). For this to happen, the Business and Persistence layers would have to be *open*, allowing requests to bypass other layers. Which is better—open layers or closed layers? The answer to this question lies in a key concept known as *layers of isolation*.

The *layers of isolation* concept means that changes made in one layer of the architecture generally don't impact or affect components in other layers, provided the contracts between those layers remain unchanged. Each layer is independent of the other layers, with little or no knowledge of their inner workings. However, to support layers of isolation, layers involved with the major flow of a request have to be closed. If the Presentation layer can access the Persistence layer directly, then changes made to the Persistence layer would impact both the Business layer *and* the Presentation layer, producing a very tightly coupled application with layer interdependencies between components. This makes a layered architecture very brittle, as well as difficult and expensive to change.

Using layers of isolation also allows any layer in the architecture to be replaced without impacting any other layer (again, assuming well-defined contracts and the use of the *Business Delegate* pattern).¹ For example, you can leverage layers of isolation to replace your older UI framework with a newer one, all within the Presentation layer.

¹ *Business Delegate* is a pattern designed to reduce coupling between business services and the user interface. Business delegates act as adapters to invoke business objects from the presentation tier.

Adding Layers

While closed layers facilitate layers of isolation and therefore help isolate change, there are times when it makes sense for certain layers to be open. For example, suppose your layered architecture's Business layer has shared objects that contain common functionality for business components (such as date and string utility classes, auditing classes, logging classes, and so on). You make an architecture decision restricting the Presentation layer from using these shared business objects. This constraint is illustrated in [Figure 10-4](#), with the dotted line going from a presentation component to a shared business object in the Business layer. This scenario is difficult to govern and control because, *architecturally*, the Presentation layer has access to the Business layer, and hence access to the shared objects within that layer.

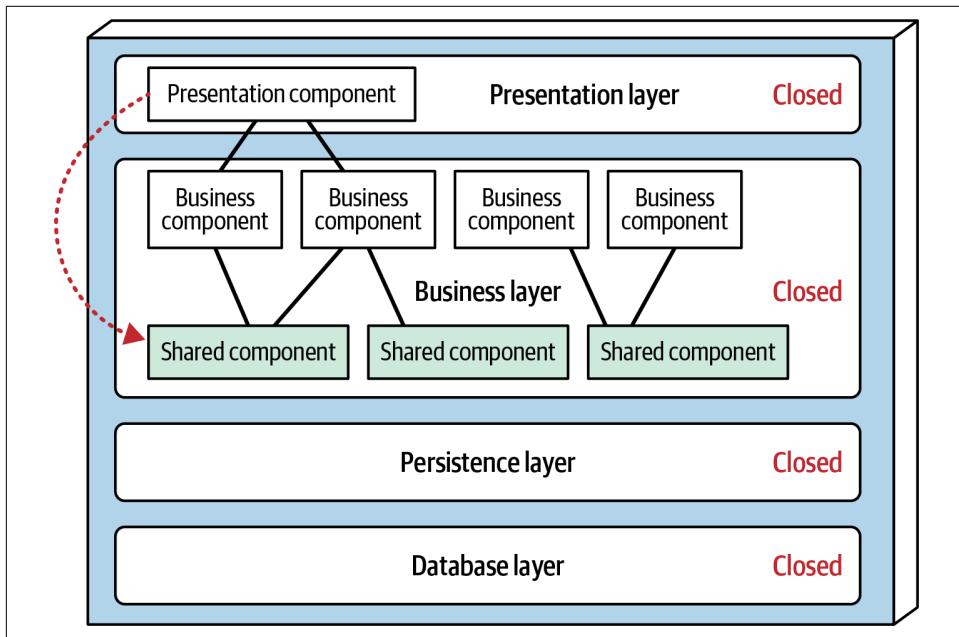


Figure 10-4. Shared objects within the Business layer

One way to mandate this restriction architecturally would be to add a new Services layer containing all of the shared business objects (see [Figure 10-5](#)). Adding this new layer would architecturally restrict the Presentation layer from accessing the shared business objects because the Business layer is closed. However, you must mark the new Services layer as *open*; otherwise, the Business layer would be forced to go through the Services layer to access the Persistence layer. Marking the Services layer as open allows the Business layer to either access that layer (as indicated by the solid arrow) or bypass it and go to the next one down (as indicated by the dotted arrow).

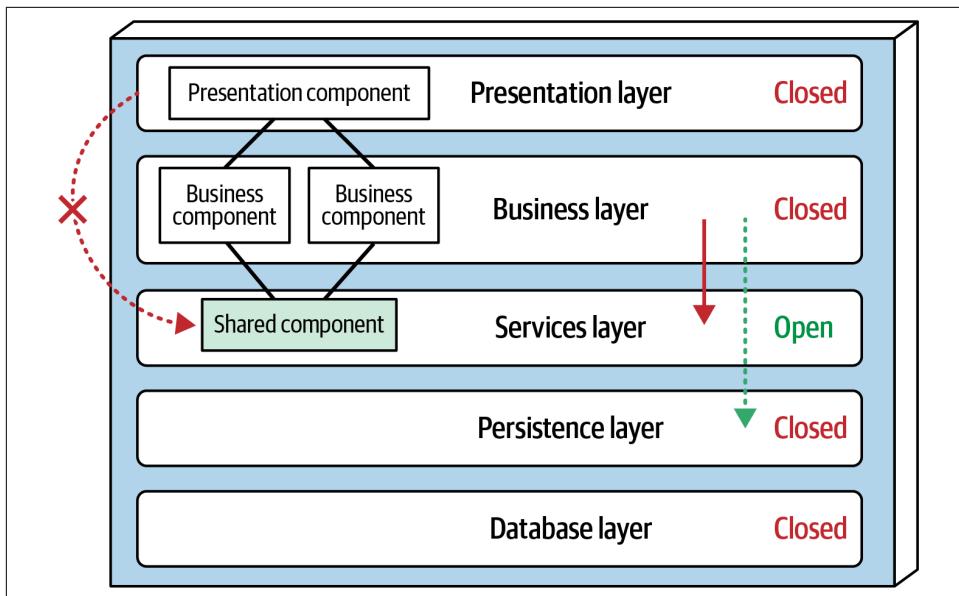


Figure 10-5. Adding a new Services layer to the architecture

Leveraging the concept of open and closed layers helps define the relationship between architecture layers and request flows. It also provides developers with the necessary information and guidance to understand layer access restrictions within the architecture. Failure to document or properly communicate which layers in the architecture are open and closed (and why) usually results in tightly coupled, brittle architectures that are very difficult to test, maintain, and deploy.

Every layered architecture will have at least some scenarios that fall into the *Architecture Sinkhole* antipattern. This antipattern occurs when requests are simply passed through from layer to layer, with no business logic performed. For example, suppose the Presentation layer responds to a user's simple request to retrieve basic customer data (such as name and address). The Presentation layer passes the request to the Business layer, which does nothing but pass the request on to the Rules layer, which in turn does nothing but pass it on to the Persistence layer, which then makes a simple SQL call to the Database layer to retrieve the customer data. The data is then passed all the way back up the stack with no additional processing or logic to aggregate, calculate, apply rules to, or transform any of it. This results in unnecessary object instantiation and processing, draining both memory consumption and performance.

The key to determining whether this antipattern is at play is to analyze the percentage of requests that fall into this category. The 80-20 rule is usually a good practice to follow. For example, it is acceptable if only 20% of the requests are sinkholes;

however, if it's 80%, that's a good indicator that the layered architecture is not the correct architecture style for the problem domain. Another approach to solving the Architecture Sinkhole antipattern is to make all the layers in the architecture open—realizing, of course, that the trade-off is increased difficulty in managing change.

Data Topologies

Traditionally, layered architectures form a monolithic system alongside a single, monolithic database. The common Persistence layer is often used to map object hierarchies between favored object-oriented languages and the set-based realm of relational databases.

Cloud Considerations

Because layered architectures are typically monolithic and partitioned into layers, the cloud options are limited to deploying one or more layers via a cloud provider. The technical partitioning inherent in this architecture is a good fit for separated deployments via the cloud. However, communication latency between on-premises servers and the cloud may create issues, because workflows typically go through most of the layers in this architecture.

Common Risks

Layered architectures don't support fault tolerance, due to their monolithic deployments and lack of architectural modularity. If one small part of a layered architecture causes an out-of-memory condition to occur, the entire application unit crashes. Overall availability is also impacted due to most monolithic applications' high mean time to recover (MTTR): startup times can range from 2 minutes for smaller applications, to 15 minutes or more for most large applications.

Governance

The news is excellent for this architecture style's governance: because it is so common, the architects who built some of the original structural testing tools did so with this architecture in mind. In fact, the example fitness function in [Figure 6-4](#) was created for the layered architecture (see [Example 10-1](#)).

Example 10-1. ArchUnit fitness function to govern layers

```
LayeredArchitecture()  
    .layer("Controller").definedBy("..controller..")  
    .layer("Service").definedBy("..service..")  
    .layer("Persistence").definedBy("..persistence..")
```

```
.whereLayer("Controller").mayNotBeAccessedByAnyLayer()  
.whereLayer("Service").mayOnlyBeAccessedByLayers("Controller")  
.whereLayer("Persistence").mayOnlyBeAccessedByLayers("Service")
```

In [Example 10-1](#), the architect defines the layers in the architecture, providing convenient names such as `Controller` for the component represented by the package name. (In ArchUnit syntax, two periods (..) on either side of a package name indicate its ownership.) Then the architect defines the communication permitted between layers, governing their open and closed layers.

Fitness function libraries support the layered architecture style extremely well, allowing architects to automate governance of the relationships they design between layers during implementation.

Team Topology Considerations

Unlike some of the architectural styles described in this book, the layered architecture style is generally independent of team topologies and will work with any team configuration:

Stream-aligned teams

Because the layered architecture is typically small and self-contained and represents a single journey or flow through the system, it works well with stream-aligned teams. With this team topology, teams generally own the flow through the system through each layer from beginning to end, creating workflows as part of their solutions.

Enabling teams

Because the layered architecture is highly modular and separated by technical concerns, it pairs well with enabling team topologies. Specialists and cross-cutting team members can interface with one or more layers to make suggestions and perform experiments, without affecting the rest of the flow. For example, the team could experiment with a new UI library by adding new behaviors to the Presentation layer, while isolating the other layers from the changes.

Complicated-subsystem teams

Because each layer performs a very specific task, this style works well with the complicated-subsystem team topology. For example, the Persistence layer provides the perfect hook for a team that requires access to operational data for analytical purposes. If allowed access at the Persistence layer, the complicated-subsystem team can work without affecting any other layers that the stream-aligned teams still owns.

Platform teams

Platform teams working on a layered architecture can leverage its high degree of modularity by utilizing the many tools available for it.

The biggest challenge with layered architectures for most platform teams tracks the overall issue with monoliths in general: as they grow, they become increasingly unwieldy. If teams keep adding features to a monolith, no matter how well partitioned and governed, it will eventually start straining at some constraints: database connections, memory, performance, concurrent users, or a host of other impending problems. Keeping the system operational will require the platform team to do increasingly difficult work.

Style Characteristics

A one-star rating in the characteristics ratings table (shown in Figure 10-6) means the specific architecture characteristic isn't well supported in the architecture, whereas a five-star rating means the architecture characteristic is one of the strongest features in the architecture style. The definition for each characteristic identified in the scorecard can be found in [Chapter 4](#).

| | Architectural characteristic | Star rating |
|-------------|------------------------------|-------------|
| | Overall cost | \$ |
| Structural | Partitioning type | Technical |
| | Number of quanta | 1 |
| | Simplicity | ★★★★★ |
| | Modularity | ★ |
| Engineering | Maintainability | ★ |
| | Testability | ★★ |
| | Deployability | ★ |
| | Evolvability | ★ |
| | Responsiveness | ★★★ |
| Operational | Scalability | ★ |
| | Elasticity | ★ |
| | Fault tolerance | ★ |
| | | |

Figure 10-6. Layered architecture characteristics ratings

Overall cost and simplicity are the primary strengths of the layered architecture style. Being monolithic, layered architectures aren't as complex as distributed architecture styles; they're simpler, easier to understand, and relatively low cost to build and maintain. Use caution, though, because these ratings diminish quickly as the monolithic layered architecture gets bigger and consequently more complex.

Neither deployability nor testability rate well for this architecture style. Deployability is low because deployments are high risk, infrequent, and involve a lot of ceremony and effort. For example, to make a simple three-line change to a class file, the entire deployment unit must be redeployed—introducing the potential for changes to the database, configuration, or other aspects of the code to sneak in alongside the original change. Furthermore, this simple three-line change is usually bundled with dozens of other changes, each of which further increases the risk and frequency of deployments. The low testability rating also reflects this scenario; with a simple three-line change, most developers are not going to spend hours executing the entire regression test suite for a simple three-line change (assuming they even have such a test suite). We give testability a two-star rating (rather than one star) because this style offers the ability to mock or stub components or even an entire layer, which eases the overall testing effort.

The engineering characteristics of the layered architecture style reflect the dynamic mentioned above: they all start well, but degrade as the size of the code base grows.

Elasticity and scalability rate very low (one star) for the layered architecture, primarily due to its monolithic deployments and lack of architectural modularity. Although it is possible to make certain functions within a monolith scale more than other functions, this effort usually requires very complex design techniques for which this architecture isn't well suited, such as multithreading, internal messaging, and other parallel processing practices. However, because a layered system's architecture quantum is always 1 (due to its monolithic UI and database and its backend processing), applications can only scale to a certain point.

Architects can achieve high responsiveness in a layered architecture with careful design, and increase it further through techniques such as caching and multithreading. We give this style three stars overall, because it does suffer from a lack of inherent parallel processing, as well as from closed layering and the Architecture Sinkhole antipattern.

When to Use

The layered architecture style is a good choice for small, simple applications or websites. It is also a good starting point for situations with very tight budget and time constraints. Because of its simplicity and its familiarity to developers and architects, this is perhaps one of the lowest-cost styles, promoting ease of development for smaller applications. The layered architecture style is also a good choice when

architects are still determining whether more complex architectures would be more suitable, but must begin development.

When using this technique, keep code reuse at a minimum and keep object hierarchies (the depth of the inheritance tree) fairly shallow to maintain a good level of modularity. This will facilitate moving to another architecture style later on.

When Not to Use

As we've shown, characteristics like maintainability, agility, testability, and deployability are adversely affected as applications using the layered architecture style grow. For this reason, large applications and systems might be better off using other, more modular architecture styles.

Examples and Use Cases

The layered architecture is one of the most common architecture styles and shows up in numerous contexts.

Designers of operating systems (like Linux or Windows) tend to use layers for the same reasons application architects do—separation of concerns. Common layers in operating systems include:

Hardware layer

Includes physical hardware like CPU, memory, and I/O devices

Kernel layer

Provides hardware abstraction, memory management, and process scheduling

System Call Interface layer

Interacts with the kernel to provide system services

User layer

Includes applications and utilities that users interact with

The *networking Open Systems Interconnection* (OSI) model conceives of how networks should delineate responsibility. For example, the base protocol of the internet, TCP/IP, includes the following layers:

Physical layer

Physically transmits data

Data Link layer

Utilizes error detection and frame synchronization

Network layer

Handles routing (such as IP)

Transport layer

Ensures reliable data transmission (such as TCP)

Application layer

Provides services like email (SMTP), file transfer (FTP), and web browsing (HTTP)

Layered architecture promotes separation of concerns, improves maintainability, and allows for independent development of each layer, so any architecture that values these features will benefit from it.

Layered architecture is also extremely common for teams trying to achieve the architectural characteristics of *feasibility*: can we actually deliver the stated scope in the allotted time with the resources available? For example, if an organization is fueled by investors and needs to deliver something as quickly as possible, the simplicity of layered architecture often makes it a good choice, even if parts need to be rewritten later to achieve different capabilities.

The Modular Monolith Architecture Style

Thanks to the widespread adoption of [domain-driven design](#) (DDD), as well as increased focus on domain partitioning, the *modular monolith* architectural style has gained so much popularity since we wrote the first edition of this book in 2020 that we decided to add a chapter to the second edition describing (and rating) it.

Topology

As the name suggests, the modular monolith architecture style is a *monolithic* architecture. As such, it's deployed as a single unit of software: a web archive (WAR) file, a single assembly in .NET, an enterprise archive (EAR) file in the Java platform, and so on. Because modular monolith is considered a *domain-partitioned* architecture (one organized by business domains rather than technical capabilities), its isomorphic shape is defined as *a single deployment unit with functionality grouped by domain area*. [Figure 11-1](#) illustrates the typical topology for modular monolith.

To get a sense of the nature of modular monolith's domain focus, consider the traditional layered architecture (described in [Chapter 10](#)). Its components are defined and organized by their *technical* capabilities: Presentation, Business, and Persistence layers, and so on. For example, the presentation logic for maintaining customer profile information might be represented by a component with the namespace `com.app.presentation.customer.profile`. The third node in the namespace represents the layer's *technical* concern (in this case, the Presentation layer).

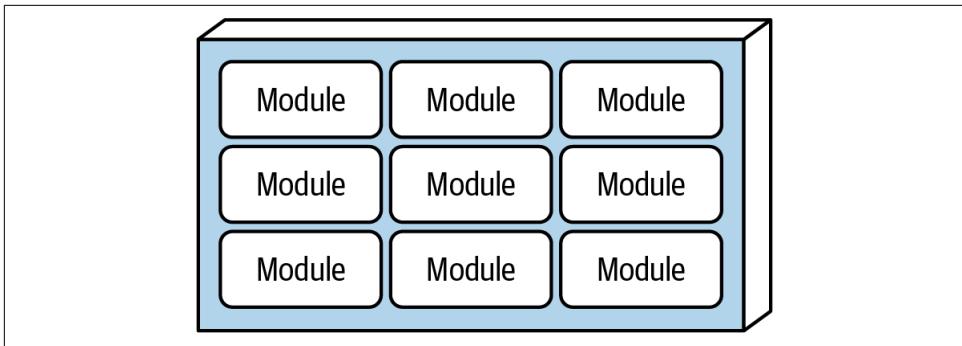


Figure 11-1. With the modular monolith architecture style, functionality is grouped by domain area

Conversely, modular monolith components are primarily organized by *domain*. As such, in a modular monolith architecture, the same customer profile maintenance component would be represented by the namespace `com.app.customer.profile`. Here, the third node of the namespace refers to a *domain* concern rather than a technical one. Depending on the complexity of the component, the namespace might further be divided by technical concern *after* the domain concern, such as `com.app.customer.profile.presentation` or `com.app.customer.profile.business`.

Style Specifics

Domains (or subdomains, in some cases) are called *modules* in this architectural style. Modules can be organized in one of two ways. The simplest architecture is a *monolithic structure*, where all of the modules and corresponding logical components are contained within the same code base, delineated by the namespace or directory structure they are contained in. A slightly more complex option is the *modular structure*, where each module is represented as an independent, self-contained artifact (such as a JAR or DLL file), and the modules are combined into one monolithic unit of software during deployment.

As with everything in software architecture, the choice between these two structural options depends on many factors and trade-offs. The following sections outline both and discuss their corresponding trade-offs.

Monolithic Structure

In the monolithic structure, all of the modules representing the system are contained in a single source-code repository. All the code associated with each module is deployed as a single unit when delivering or releasing the software. This structural option is illustrated in [Figure 11-2](#). Each module is represented by a separate high-level directory containing the components and any subdomains that make up that module.

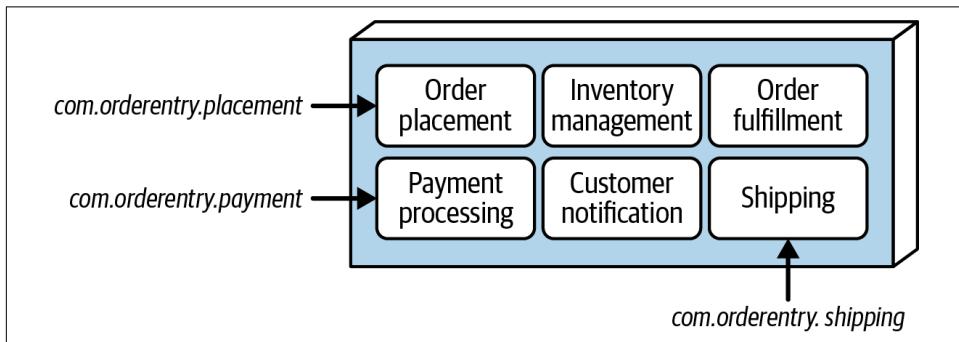


Figure 11-2. An example of the monolithic structure option

The following namespaces illustrate what a modular monolith might look like for the architecture shown in [Figure 11-2](#):

```
com.orderentry.orderplacement  
com.orderentry.inventorymanagement  
com.orderentry.paymentprocessing  
com.orderentry.notification  
com.orderentry.fulfillment  
com.orderentry.shipping
```

This is the simplest option for the modular monolith: all of the system's source code is located in one place and is thus more easily maintained, tested, and deployed. However, strict governance is needed (see [“Governance” on page 172](#)) to maintain the boundaries of each module. Although this structural option is simple, developers have a tendency to reuse too much code across modules, as well as allowing too much communication between modules (see [“Module Communication” on page 168](#)). These practices can turn a well-architected modular monolith into an unstructured Big Ball of Mud.

Modular Structure

With the modular structure, modules are represented as self-contained artifacts (such as JAR and DLL files), then put together into a single deployment unit during deployment. [Figure 11-3](#) illustrates this option using JAR files in the Java platform.

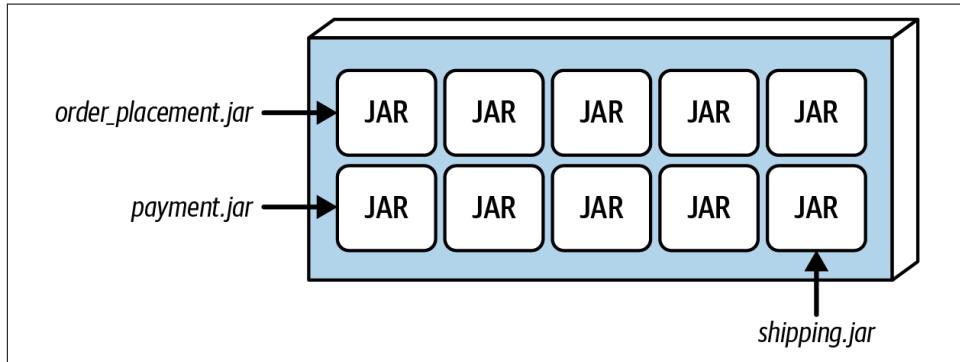


Figure 11-3. An example of the modular structure option using JAR files

The advantage of this structure is that each module is self-contained, allowing teams to work on separate modules (see “[Team Topology Considerations](#)” on page 174), many times even within the team’s own dedicated source code repository for those modules. This option works well when the modules are largely independent of other modules. It also works well for larger, more complex systems where each module requires a different kind of expertise or business knowledge. With the modular structure option, developers are less apt to reuse code too much or to have modules communicate too much with one another (see “[Module Communication](#)” on page 168). This option also tends to produce cleaner boundaries between modules and better overall separation of concerns.

However, this structural option loses its effectiveness when modules that are dependent on one other need to communicate. Where this is the case, the monolithic structure approach is more effective.

Module Communication

Communication between modules is never a good thing in this architectural style, but we do acknowledge that in many cases it’s necessary. For instance, in the architecture shown in [Figure 11-2](#), the OrderPlacement module must communicate with the InventoryManagement module to have it adjust the inventory for the item ordered and perform any additional processing (for example, to order more stock if the inventory is too low). It also has to communicate with the PaymentProcessing module to apply payment for an order. Two primary options exist for communicating between modules, which we describe in the following sections.

Peer-to-peer approach

The most straightforward solution is simple peer-to-peer communication between modules. With this approach, a class file in one module instantiates a class in another module and invokes the necessary method(s) in that class to perform the operation (see [Figure 11-4](#)).

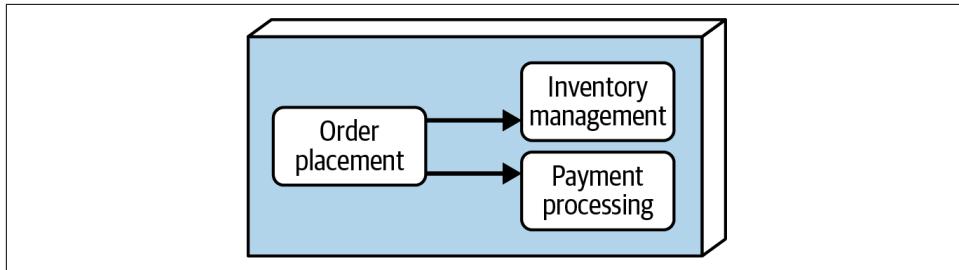


Figure 11-4. Peer-to-peer communication between modules

One issue with the monolithic approach is that it's *too* convenient for developers to instantiate any class contained within another module. This makes it easy to go from a well-structured architecture to the Big Ball of Mud antipattern (see [Figure 9-1](#)).

With the modular structure, however, the classes contained in another module may be located in separate, external artifacts (JARs or DLL files) rather than a separate directory in the source code repository. A module that communicates with other modules won't compile unless it has the class references, which means the developer has to form a *compile-time* dependency between those modules. The usual response to this issue is to create a shared interface class between those modules (in a separately shared JAR or DLL file) so that each module can compile independently of other modules. Either way, too much communication between modules using the modular structure approach results in the [**DLL Hell**](#) antipattern (or, in the Java platform, the [**JAR Hell**](#) antipattern).

Mediator approach

The *mediator* approach decouples modules by using a mediator component to form an abstraction layer between modules. The mediator acts as an orchestrator, accepting requests and delivering them to the appropriate modules. [Figure 11-5](#) illustrates this approach.

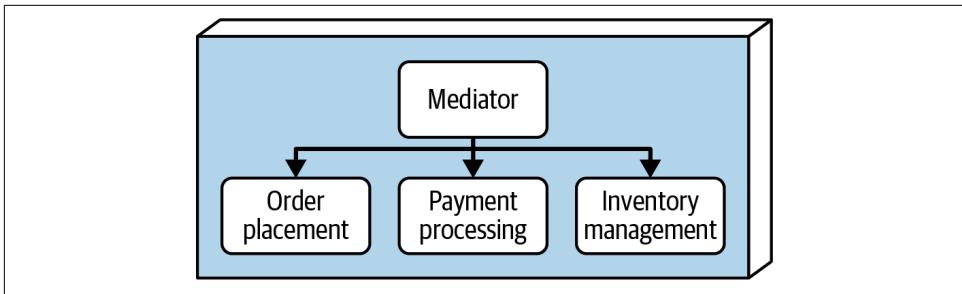


Figure 11-5. The mediator decouples modules, so they don't have to communicate with each other

The astute reader will observe that, while the mediator approach decouples modules, each module is effectively coupled to the mediator. This approach doesn't remove *all* coupling and dependencies, but it does simplify the architecture and keep the modules independent from each other. Note that it is the *mediator*, not the dependent modules, that needs some sort of API or interface to invoke the functionality in other modules.

Data Topologies

Because the modular monolith architecture is usually deployed as a single unit of software, it typically relies on a monolithic database topology. Using a single database helps reduce communication between modules, since the data is shared. However, if the modules are independent from each other and perform specific functions, they can also have their own databases containing specific contextual data, even though the architecture itself is monolithic. **Figure 11-6** illustrates these two database topology options.

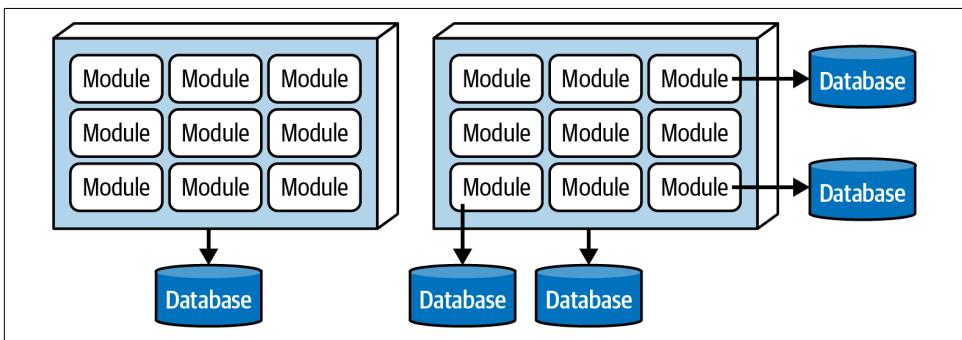


Figure 11-6. Data can be monolithic or modules can have their own databases

Cloud Considerations

Although modular monolithic architectures can be deployed in cloud environments (particularly if it's a small system), they're not generally well suited for cloud deployment: their monolithic nature renders them less likely to be able to take advantage of the on-demand provisioning that cloud environments offer. That said, smaller systems implemented in this architectural style can still leverage many cloud services, such as file storage, database, and messaging.

Common Risks

As with any monolithic system, the primary risk with the modular monolith architectural style is that it can get too big to properly maintain, test, and deploy. Monolithic architectures in and of themselves aren't bad; it's when they get too big that problems start to occur. What "too big" means varies from system to system, but here are some of the warning signs that the system might be too big:

- Changes take too long to make.
- When one area of the system is changed, other areas unexpectedly break.
- Team members get in each other's way when applying changes.
- It takes too long for the system to start up.

Another risk is going overboard with code reuse. Code reuse and sharing are a necessary part of software development, but in this architecture style, too much code reuse blurs the module boundaries, leading the architecture into the risky territory of the *unstructured monolith*: a monolithic architecture with such highly interdependent code that it cannot be unraveled.

Too much intermodule communication is another risk in this architectural style. Ideally, modules should be independent and self-contained. As we've noted, it's normal (and sometimes necessary) for some modules to communicate with others, particularly within a complex workflow. However, if there's too much intercommunication between modules, it's a good indication that the domains may have been ill-defined in the first place. In such cases, it's worth putting additional thought into redefining the domains to accommodate complex workflows and interdependencies.

Governance

The primary artifact in the modular monolith style is a *module*, which represents a particular domain or subdomain and is usually represented through the directory structure or namespace (or package structure in the Java platform). Therefore, one of the first forms of automated governance architects can apply is defining and ensuring compliance with the modules used in the architecture.

To write automated governance checks, architects use a host of tools, including **ArchUnit** for the Java platform, **ArchUnitNet** and **NetArchTest** for the .NET platform, **PyTestArch** for Python, and **TSArch** for TypeScript and JavaScript. The pseudocode in [Example 11-1](#) ensures that all of the source code represented in the architecture example shown in [Figure 11-2](#) falls under one of the listed namespaces that represent each defined module in the system.

Example 11-1. Pseudocode for ensuring the code follows the system's defined modules

```
# The following namespaces represent the modules in the system
LIST module_list = {
    com.orderentry.orderplacement,
    com.orderentry.inventorymanagement,
    com.orderentry.paymentprocessing,
    com.orderentry.notification,
    com.orderentry.fulfillment,
    com.orderentry.shipping
}

# Get the list of namespaces in the system
LIST namespace_list = get_all_namespaces(root_directory)

# Make sure all the namespaces start with one of the listed modules
FOREACH namespace IN namespace_list {
    IF NOT namespace.starts_with(module_list) {
        send_alert(namespace)
    }
}
```

If a developer creates any additional high-level namespaces or directories outside the defined modules and their corresponding namespaces (or directories), they will receive an alert indicating the source code is not in compliance with the architecture.

This form of governance works well with the monolithic structure option of this architecture style (see “[Monolithic Structure](#)” on page 167), but is challenging with the modular structure option (see “[Modular Structure](#)” on page 168) because the code might not be contained in the same monolithic source code repository. With the modular structure option, each module must be tested separately, as shown in [Example 11-2](#).

Example 11-2. Pseudocode for validating the InventoryManagement module

```
# Get the list of namespaces in the system
LIST namespace_list = get_all_namespaces(root_directory)

# Make sure all the namespaces start with com.orderentry.inventorymanagement
FOREACH namespace IN namespace_list {
    IF NOT namespace.starts_with("com.orderentry.inventorymanagement") {
        send_alert(namespace)
    }
}
```

Another way to govern a modular monolith architecture is to control the amount of communication between modules. Defining what is “too much” communication is highly subjective and varies from system to system, but for the most part, architects should try to minimize the number of interdependencies between modules. [Example 11-3](#) shows pseudocode for making sure the maximum total interdependency doesn’t exceed a limit of five communication (or coupling) points.

Example 11-3. Pseudocode for limiting any given module’s total number of dependencies

```
# Walk the directory structure, gathering modules and the source code files
# contained within those modules
LIST module_list = {
    com.orderentry.orderplacement,
    com.orderentry.inventorymanagement,
    com.orderentry.paymentprocessing,
    com.orderentry.notification,
    com.orderentry.fulfillment,
    com.orderentry.shipping
}

MAP module_source_file_map
FOREACH module IN module_list {
    LIST source_file_list = get_source_files(module)
    ADD module, source_file_list TO module_source_file_map
}

# Determine how many references exist for each source file and send an alert if
# the system's total dependency count is greater than 5
FOREACH module, source_file_list IN module_source_file_map {
    FOREACH source_file IN source_file_list {
        incoming_count = used_by_other_module(source_file, module_source_file_map) {
        outgoing_count = uses_other_module(source_file) {
            total_count = incoming_count + outgoing_count
        }
        IF total_count > 5 {
            send_alert(module, total_count)
        }
    }
}
```

A final form of automated governance is to ensure that modules stay independent of one another by restricting one specific module from talking to another module. For example, in [Figure 11-2](#), the OrderPlacement module should not communicate with the Shipping module. [Example 11-4](#) shows the ArchUnit code in Java to govern this dependency.

Example 11-4. ArchUnit code for governing dependency restrictions between specific modules

```
public void order_placement_cannot_access_shipping() {  
    noClasses().that()  
        .resideInAPackage("..com.orderentry.orderplacement..")  
        .should().accessClassesThat()  
            .resideInAPackage("..com.orderentry.shipping..")  
            .check(myClasses);  
}
```

Team Topology Considerations

Because the modular monolith is considered a domain-partitioned architecture, it works best when teams are also aligned by domain area (such as cross-functional teams with specialization). When a domain-based requirement comes along, a domain-focused, cross-functional team can work together on that feature, from the presentation logic all the way to the database. Conversely, teams organized by technical categories (such as UI teams, backend teams, database teams, and so on) do not work well with this architectural style, primarily due to its domain partitioning. Assigning domain-based requirements to technically organized teams requires a lot of communication and collaboration, which often proves difficult.

Here are some considerations for aligning the specific team topologies outlined in “[Team Topologies and Architecture](#)” on page 151 with the modular monolith style:

Stream-aligned teams

Stream-aligned teams generally own the flow through the system from beginning to end, nicely matching the monolithic and generally self-contained shape of the modular monolith.

Enabling teams

Due to this style’s high level of modularity and separation of concerns, enabling team topologies also works well. Specialists and cross-cutting team members can make suggestions and perform experiments by introducing additional modules to the system, with minimal impact to other existing modules.

Complicated-subsystem teams

Each module in a modular monolith architecture generally performs a specific role based on its domain or subdomain (such as `PaymentProcessing`). This works well with the complicated-subsystem team topology, because different team members can focus on complicated domain or subdomain processing independent of other team members (and modules).

Platform teams

Developers can leverage the benefits of the platform-teams topology by utilizing common tools, services, APIs, and tasks, primarily due to the high degree of modularity found in this architectural style.

Style Characteristics

A one-star rating in the characteristics ratings table [Figure 11-7](#) means the specific architecture characteristic isn't well supported in the architecture, whereas a five-star rating means the architecture characteristic is one of the strongest features in the architecture style. The characteristics contained in the scorecard are described and defined in [Chapter 4](#).

The modular monolith architecture style is a *domain-partitioned* architecture because its application logic is partitioned into modules. Because it is usually implemented as a monolithic deployment, its architectural quantum is typically 1.

Overall cost, simplicity, and modularity are the primary strengths of the modular monolith architecture style. Being monolithic in nature, these architectures don't have the complexities associated with distributed architecture styles. They're simpler and easier to understand, and relatively low cost to build and maintain. Architectural modularity is achieved through the separation of concerns between the various modules, representing domains and subdomains.

Deployability and testability, while only two stars, rate slightly higher in modular monolith than the layered architecture due to its level of modularity. That said, this architecture style is still a monolith: as such, ceremony, risk, frequency of deployment, and completeness of testing negatively impact these scores.

Modular monolith architecture elasticity and scalability rate very low (one star), primarily due to monolithic deployments. Although it is possible to make certain functions within a monolith scale more than others, this effort usually requires very complex design techniques (such as multithreading, internal messaging, and other parallel-processing practices) for which this architecture isn't well suited.

| | Architectural characteristic | Star rating |
|-------------|------------------------------|---|
| Structural | Overall cost | \$ |
| | Partitioning type | Domain |
| | Number of quanta | 1 |
| | Simplicity |  |
| | Modularity |  |
| | Maintainability |  |
| | Testability |  |
| | Deployability |  |
| | Evolvability |  |
| | Responsiveness |  |
| Engineering | Scalability |  |
| | Elasticity |  |
| | Fault tolerance |  |
| | | |
| Operational | | |
| | | |

Figure 11-7. Architectural characteristics star ratings for the modular monolith

Modular monolith architecture monolithic deployments don't support fault tolerance. If one small part of the architecture causes an out-of-memory condition, the entire application unit crashes. Furthermore, as in most monolithic applications, overall availability is affected by the high mean time to recovery (MTTR), with startup times usually measured in minutes.

When to Use

Because of its simplicity and low cost, the modular monolith architecture style is a good choice when faced with tight budget and time constraints. It's also a good choice for starting out with a new system. If the system's architectural direction is still unclear, it's often more effective to begin with a modular monolith and later move to a more complicated and expensive distributed architecture style, such as service based (see [Chapter 14](#)) or microservices (see [Chapter 18](#)), than to jump straight into the distributed architecture.

Modular monolith is also a good choice for domain-focused teams, such as cross-functional teams with specialization. This allows each team to focus on a specific module within the architecture from end to end, with minimal coordination with other domain teams. This architectural style is also well suited for situations where a majority of changes to the system are domain based (such as adding expiration dates to items in a customer's wishlist).

Lastly, because the modular monolith is a domain-partitioned architecture, it's well suited to teams engaging in [DDD](#).

When Not to Use

The primary reason not to use this architectural style is when systems or products require high levels of certain operational characteristics, such as scalability, elasticity, availability, fault tolerance, responsiveness, and performance. Like most monolithic architectures, modular monolith is ill-suited for these architectural concerns.

Avoid using modular monolith when a majority of the changes are technically oriented, such as continuously replacing the user interface or database technology. Because this architecture is domain partitioned, such changes impact every module and usually require significant communication and coordination between domain teams. In these situations, the layered architecture style (see [Chapter 10](#)) is a much better choice.

Examples and Use Cases

EasyMeals is a new delivery-based neighborhood restaurant, catering to working people who don't always have time to cook meals after getting home from a busy day. Hungry customers can order a nice dinner online and have it delivered to their doorstep within an hour.

As a small, local restaurant, they don't have high scalability or responsiveness needs. And since their budget is limited, they don't want to spend a lot of money on an elaborate software system. The shape of this business problem makes the modular monolith a good choice for EasyMeals.

[Figure 11-8](#) shows what EasyMeals' simple restaurant management system might look like using the modular monolith architectural style.

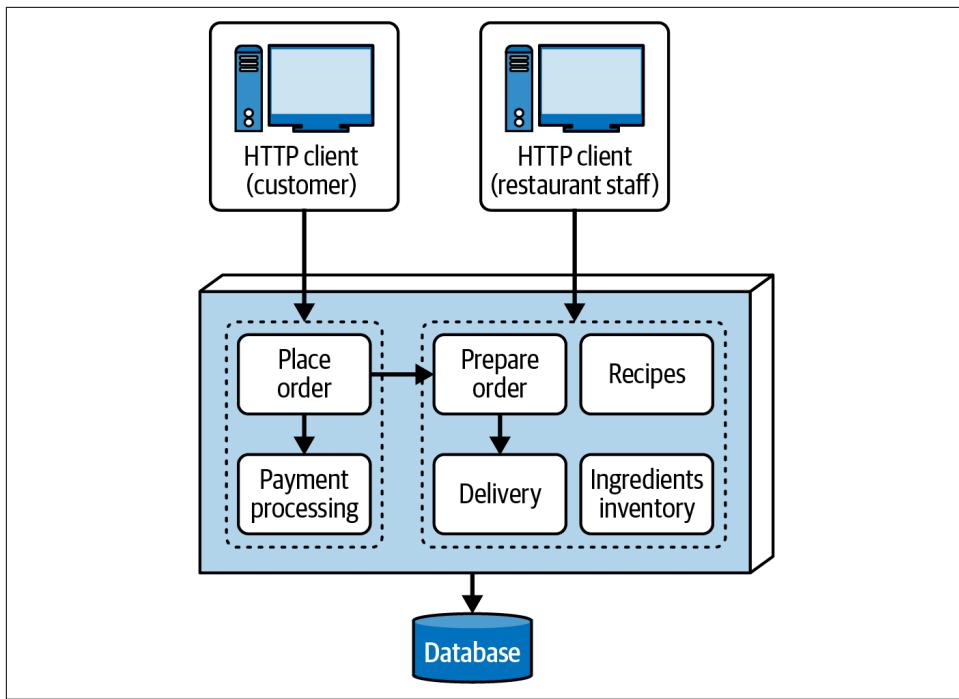


Figure 11-8. A small restaurant ordering and management system using the modular monolith style

Customers access the `PlaceOrder` and `PaymentProcessing` modules through a dedicated user interface through a different UI. The following namespaces represent each of the modules for this system:

```
com.easymeals.placeorder
com.easymeals.payment
com.easymeals.prepareorder
com.easymeals.delivery
com.easymeals.recipes
com.easymeals.inventory
```

The `PlaceOrder` module allows each customer to view the menu; select items; add their name, address, and payment information; and submit the order. The components for this module could be represented through the following namespaces, with source code implementing each of these main functions:

```
com.easymeals.placeorder.menu
com.easymeals.placeorder.shoppingcart
com.easymeals.placeorder.customerdata
com.easymeals.placeorder.paymentdata
com.easymeals.placeorder.checkout
```

This example illustrates that a module in the modular monolith is made up of one to many *components* (see Chapter 8).

The `PaymentProcessing` module is responsible for applying payment. EasyMeals accepts credit cards, debit cards, and PayPal; the modularity of this architecture makes it easy to add an additional payment type (such as loyalty points). Customers enter this information in the `PlaceOrder` module, which passes it to the `Payment Processing` module. The components for this module could be represented through the following namespaces:

```
com.easymeals.payment.creditcard  
com.easymeals.payment.debitcard  
com.easymeals.payment.paypal
```

Once the order is paid for, the `PlaceOrder` module communicates with the `Prepare Order` module, which displays the entire order to the kitchen staff. After cooking, the kitchen staff marks the order as ready for delivery and it is then sent to the `Delivery` module. The following namespaces represent the components within the `PrepareOrder` module:

```
com.easymeals.prepareorder.displayorder  
com.easymeals.prepareorder.ready
```

The `Delivery` module assigns a delivery person to the order and indicates the delivery address. It allows the delivery person to mark the order as delivered, ending the lifecycle for that particular order, and to record any issues (such as an aggressive dog at the front gate or a customer who isn't home). The following namespaces represent the components within the `Delivery` module:

```
com.easymeals.delivery.assign  
com.easymeals.delivery.issues  
com.easymeals.delivery.complete
```

The `Recipes` module allows the cooks and management staff to add items to the menu and maintain the list of ingredients and measurements for each menu item. The following namespaces represent the components within the `Recipes` module:

```
com.easymeals.recipes.view  
com.easymeals.recipes.maintenance
```

Finally, the `IngredientsInventory` module makes sure that there are enough ingredients on hand for the recipes on the menu. This module is a bit more complex than the others: it has a sophisticated AI component that forecasts sales volume to automate the process of procuring ingredients for the week.

The following namespaces represent the components within the `IngredientsInventory` module:

```
com.easymeals.inventory.maintenance  
com.easymeals.inventory.forecasting  
com.easymeals.inventory.ordering  
com.easymeals.inventory.suppliers  
com.easymeals.inventory.invoices
```

And that's it! Modular monolith's simplicity and level of modularity makes it relatively easy to locate and maintain code to fix a bug or add a new feature. This illustrates the power of this simple and straightforward architectural style.

Pipeline Architecture Style

One of the fundamental styles in software architecture is the *pipeline* architecture (also known as the *pipes and filters* architecture). As soon as developers and architects decided to split functionality into discrete parts, this style of architecture followed. Most developers know this architecture as the underlying principle behind Unix terminal shell languages, such as [Bash](#) and [Zsh](#).

Developers in many functional programming languages will see parallels between language constructs and elements of this architecture. In fact, many tools that use the [MapReduce](#) programming model follow this basic topology. While the examples in this chapter show low-level implementations of the pipeline architecture style, it can also be used for higher-level business applications.

Topology

The topology of the pipeline architecture consists of two main component types, pipes and filters. *Filters* contain the system functionality and perform a specific business function, and *pipes* transfer data to the next filter (or filters) in the chain. They coordinate in a specific fashion, with pipes forming one-way, usually point-to-point communication between filters, as illustrated in [Figure 12-1](#).

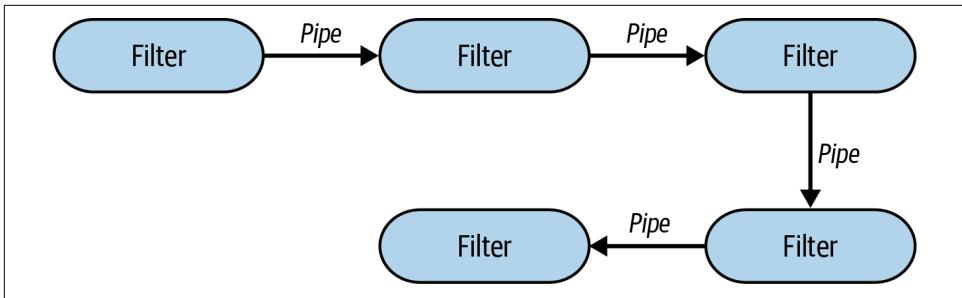


Figure 12-1. Basic topology for pipeline architecture

The isomorphic “shape” of the pipeline architecture is thus a *single deployment unit*, with functionality contained within filters connected by unidirectional pipes.

Style Specifics

While most implementations of the pipeline architecture are monolithic, it is possible to deploy each filter (or a set of filters) as a service, creating a distributed architecture with synchronous or asynchronous remote calls to each service. Whatever its deployment topology, the architecture consists of only two architectural components, filters and pipes, which we describe in detail in the following sections.

Filters

Filters are self-contained pieces of functionality that are independent from other filters. They are generally stateless, and should perform one task only. Composite tasks are typically handled by a sequence of filters rather than a single one.

Because filters can be implemented by more than one class file, they are considered *components* of the architecture (see [Chapter 8](#)). Even if a filter is simple and only implemented as a single class file, it’s still a component.

There are four types of filters in the pipeline architecture style:

Producer

The starting point of a process, producer filters are outbound only. They are sometimes called the *source*. A user interface and an external request to the system are both examples of producer filters.

Transformer

Transformer filters accept input, optionally perform a transformation on some or all of the data, then forward the data to the outbound pipe. Functional-programming advocates will recognize this feature as *map*. Transformer filters might, for example, enhance data, transform data, or perform some sort of calculation.

Tester

Tester filters accept input, test it according to one or more criteria, and then optionally produce output based on the test. Functional programmers will recognize this as similar to *reduce*. A tester filter might check that all data is valid and entered correctly, or act as a switch to determine if processing should move forward (for example, “don’t forward the data to the next filter if the order amount is less than five dollars”).

Consumer

The termination point for the pipeline flow, consumer filters sometimes persist the final result of the pipeline process to a database or display the final results on a UI screen.

The unidirectional nature and simplicity of pipes and filters encourage compositional reuse. Many developers have discovered this ability using shells. A famous story from the blog “[More Shell, Less Egg](#)” illustrates just how powerful these abstractions are. Donald Knuth was asked to write a program to solve this text-handling problem: read a file of text, determine the n most frequently used words, and print a list of those words sorted by frequency. He wrote a program consisting of more than 10 pages of Pascal, designing (and documenting) a new algorithm along the way. Then Doug McIlroy demonstrated a shell script small enough to easily fit within a social media post that solved the problem elegantly:

```
tr -cs A-Za-z '\n' |  
tr A-Z a-z |  
sort |  
uniq -c |  
sort -rn |  
sed ${1}q
```

Even the designers of Unix shells are often surprised at the inventive uses developers find for their simple but powerful composite abstractions.

Pipes

Pipes, in this architecture, form the communication channel between filters. Each pipe is typically unidirectional and point-to-point, accepting input from one source and directing output to another. Their payload can be any data format, but architects typically favor smaller amounts of data to enable high performance.

If a filter (or group of filters) is deployed as a separate service in a distributed fashion, the pipes issue a unidirectional remote call using REST, messaging, streaming, or some other remote communication protocol. Whether their deployment topology is monolithic or distributed, pipes can be either synchronous or asynchronous. In monolithic deployments, architects use threads or embedded messaging for asynchronous communication to a filter.

Data Topologies

Because most pipeline architectures are deployed as monoliths, they are associated with a single monolithic database. However, this is not always the case. Database topology can vary significantly with this architectural style, from a single database per filter.

The example in [Figure 12-2](#) shows a pipeline architecture for a typical continuous fitness function (architectural test) running in a production environment that analyzes a specific operational characteristic (such as responsiveness or scalability). Notice that the Capture Raw Data filter loads a separate database containing raw data. This filter sends the raw data through a pipe to the Time Series Selector filter, which reads configuration information (such as the period of time being analyzed) from a separate database. The transformed data is then sent through another pipe to the Trend Analyzer filter, which analyzes the data and stores those analytics in a separate database. This filter then sends the analytics data through a final pipe to the Graphing Tool filter, which ends the pipeline by generating a graphical report of the data.

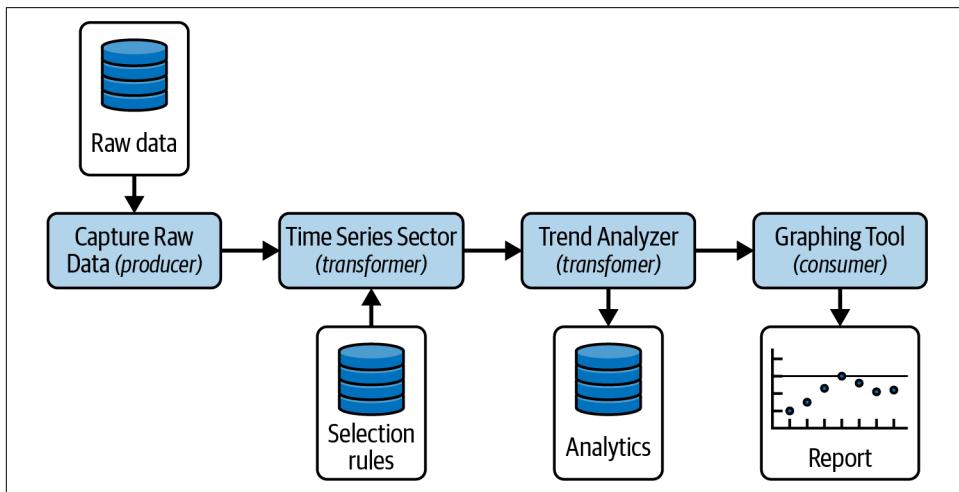


Figure 12-2. Pipeline architectures can have a single database or many

Cloud Considerations

The pipeline architecture style is well suited for cloud-based deployments due to its high level of modularity and separate filter types. Because most pipeline architectures are not overly complex and extensive, they work well being deployed as monolithic architectures, where all filters are deployed in the same deployment unit.

However, filters also work well in cloud environments as distributed functions. In AWS, the pipeline architecture can be deployed as [AWS Step Functions](#), where each

filter is deployed as a separate lambda in the workflow. AWS Step Functions offer two workflows: *Standard*, in which each step is executed exactly once, and *Express*, where each step can execute more than once. The pipeline architecture style works with both. The following code illustrates the continuous fitness function example in [Figure 12-2](#) as a typical cloud deployment for the pipeline architecture, implemented as an AWS Step Function:

```
{  
    "Comment": "Measure and analyze scalability trends.",  
    "StartAt": "Capture Raw Data",  
    "States": {  
        "Capture Raw Data": {  
            "Type": "Task",  
            "Resource": "arn:aws:lambda:region:account_id:function:raw_data_capture",  
            "Next": "Time Series Selector"  
        },  
        "Time Series Selector": {  
            "Type": "Task",  
            "Resource": "arn:aws:lambda:region:account_id:function:time_series_selector",  
            "Next": "Trend Analyzer"  
        },  
        "Trend Analyzer": {  
            "Type": "Task",  
            "Resource": "arn:aws:lambda:region:account_id:function:trend_analyzer",  
            "Next": "Graphing Tool"  
        },  
        "Graphing Tool": {  
            "Type": "Task",  
            "Resource": "arn:aws:lambda:region:account_id:function:graphing_tool",  
            "End": true  
        }  
    }  
}
```

This example isn't the only way to use the pipeline architecture in cloud environments. Filters can also be deployed as serverless functions, as containerized functions, or even as a single service containing all four filter components in a monolithic deployment.

Common Risks

The primary goal behind the pipeline architecture is to separate functionality into single-purpose filters, where each filter performs *one* specific action on the data and then hands it off to another filter for further processing. As such, one of its most common risks is overloading filters with too much responsibility. Good governance, which we cover in the next section, helps teams mitigate this risk by identifying the purpose behind each filter component.

Another common risk with this architecture style is introducing bidirectional communication between filters. Pipes are meant to be *unidirectional only*, providing a clear separation of concerns between filters to avoid collaboration between them. If bidirectional communication turns out to be necessary, this is a good indication that the pipeline architecture might not be the right style to use, or that the filters are too complex; functionality isn't demarcated correctly.

Handling error conditions is another complication that can introduce significant risk within this architectural style. If an error occurs within a pipeline, it is often difficult to determine how to properly exit the pipeline and recover once the pipeline is started. For this reason, it's important for architects to determine any possible fatal error conditions within the pipeline before defining the architecture.

The last risk area is managing the contracts between filters. Each pipe has a contract representing the data (and possibly the corresponding types) that it sends to the next filter. Changing a contract between filters requires strict governance and testing to ensure that other filters receiving the contract don't break.

Governance

Governance for general operational characteristics—such as responsiveness, scalability, availability, and so on—is specific to each individual use case and can vary greatly. However, from a *structural* standpoint, architects use the role and responsibility of each filter type to govern pipeline architectures.

Each of the four basic filter types (producer, transformer, tester, and consumer) performs a specific role. However, it's all too easy for developers to overload filters with too much responsibility, turning the pipeline architecture into an unstructured monolith.

It is difficult to write an automated fitness function to govern whether a producer filter is actually the starting point of the pipeline, or whether a tester filter is actually performing a conditional check to determine whether to continue with the flow or terminate it. Governance techniques help architects guide the development team in adhering to each filter type's role and the responsibility of a specific filter type.

One such technique is to leverage *tags*. (In the Java language, tags are implemented as *annotations*, and in C# as *custom attributes*.) Tags don't actually perform any function; they programmatically provide metadata about the component or service. Utilizing tags in the starting class of the filter component tells developers about the type of filter they are creating or modifying, helping prevent them from giving that filter too much responsibility or performing a function not associated with its type.

To illustrate this technique, consider the following source code that defines tags for the four basic filter types. Here is the Java tag annotation definition:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Filter {
    public FilterType[] value();

    public enum FilterType {
        PRODUCER,
        TESTER,
        TRANSFORMER,
        CONSUMER
    }
}
```

And here is the C# tag custom attribute definition:

```
[System.AttributeUsage(System.AttributeTargets.Class)]
class Filter : System.Attribute {

    public FilterType[] filterType;

    public enum FilterType {
        PRODUCER,
        TESTER,
        TRANSFORMER,
        CONSUMER
    };
}
```

Since filters are essentially an architectural component, they can be implemented through multiple class files. For example, in our continuous fitness function example in [Figure 12-2](#), the Trend Analyzer filter is fairly complex and could have multiple class files. Because of this, we'll define one additional tag to identify the class that represents the filter's entry point, so we can attach other tags to it.

Here is the Java entry point tag definition:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface FilterEntryPoint {}
```

And the C# entry point tag definition:

```
[System.AttributeUsage(System.AttributeTargets.Class)]
class FilterEntryPoint : System.Attribute {}
```

Now the developers can add the `FilterType` tag to the `EntryPoint` class, identifying its type and its corresponding role in the architecture. For example, the following annotations identify the type and role of the Trend Analyzer transformation filter. In Java:

```
@FilterEntrypoint  
@Filter(FilterType.TRANSFORMER)  
public class TrendAnalyzerFilter {  
    ...  
}
```

And in C#:

```
[FilterEntrypoint]  
[Filter(FilterType.TRANSFORMER)]  
class TrendAnalyzerFilter {  
    ...  
}
```

While this technique might not stop every developer from making a transformer filter type perform testing logic (which should be done by a tester filter), at least it provides additional context.

Team Topology Considerations

Unlike some of the architectural styles described in this book, the pipeline architecture style is generally independent of team topologies and will work with any team configuration:

Stream-aligned teams

Because the pipeline architecture is typically small and self-contained and represents a single journey or flow through the system, it works well with stream-aligned teams. With this team topology, teams generally own the flow through the system from beginning to end, nicely matching the shape of the pipeline architecture.

Enabling teams

Because the pipeline architecture is highly modular and separated by technical concerns, it pairs well with enabling team topologies. Specialists and cross-cutting team members can make suggestions and perform experiments by introducing additional filters within the pipeline, without affecting the rest of the flow. For example, in our continuous fitness function example, an enabling team might add a transformation filter after the Time Series Selector filter to do some alternative trend analysis. This new filter would use the same data as the existing Trend Analyzer filter without disrupting the normal pipeline flow.

Complicated-subsystem teams

Because each filter performs a very specific task, the pipeline architecture works well with the complicated-subsystem team topology. Different team members can focus on complicated filter processing independently from one another (and from other filters). The unidirectional handoffs involved in this architectural

style allow members of complicated-subsystem teams to focus narrowly on the complexity contained within their specific filter processing.

Platform teams

Platform teams working on a pipeline architecture can leverage its high degree of modularity by utilizing common tools, services, APIs, and tasks.

Style Characteristics

A one-star rating in the characteristics ratings table (Figure 12-3) means the specific architecture characteristic isn't well supported in the architecture, whereas a five-star rating means the architecture characteristic is one of the strongest features in the architecture style. The characteristics contained in the scorecard are described and defined in Chapter 4.

The pipeline architecture style is a *technically partitioned* architecture because its application logic is separated into filter types. Also, because the pipeline architecture is usually implemented as a monolithic deployment, its architectural quantum is always 1.

| | Architectural characteristic | Star rating |
|-------------|------------------------------|-------------|
| Structural | Overall cost | \$ |
| | Partitioning type | Technical |
| | Number of quanta | 1 |
| | Simplicity | ★★★★★ |
| | Modularity | ★★ |
| Engineering | Maintainability | ★★ |
| | Testability | ★★★★ |
| | Deployability | ★★ |
| | Evolvability | ★★★★ |
| Operational | Responsiveness | ★★★★ |
| | Scalability | ★ |
| | Elasticity | ★ |
| | Fault tolerance | ★ |

Figure 12-3. Pipeline architecture characteristics ratings

Overall cost, simplicity, and modularity are the primary strengths of the pipeline architecture style. Being monolithic, pipeline architectures don't have the complexities associated with distributed architecture styles—they're simple and easy to understand, and are relatively low-cost to build and maintain. Architectural modularity is achieved through separating concerns between the various filter types and transformers: any filter can be modified or replaced without affecting the other filters. For instance, in the Kafka example illustrated in [Figure 12-4](#), the Duration Calculator can be modified to change the duration calculation without changing any other filter.

Deployability and testability, while only average, rate slightly higher than in the layered architecture due to the level of modularity filters can achieve. However, pipeline architectures are still *typically* monolithic, so their downsides include ceremony, risk, frequency of deployment, and completeness of testing.

Elasticity and scalability rate very low (one star) in the pipeline architecture, primarily due to monolithic deployments. Implementing this architecture style as a distributed architecture using asynchronous communication can significantly improve these characteristics, but the trade-off is a blow to overall cost and simplicity.

Pipeline architectures don't support fault tolerance because they are typically deployed as monolithic systems. If one small part of a pipeline architecture causes an out-of-memory condition to occur, the entire application unit is impacted and crashes. Furthermore, overall availability is affected by the high mean time to recovery (MTTR) common in most monolithic applications, with startup times measured in minutes. As with elasticity and scalability, implementing this architecture style as a distributed architecture using asynchronous communication can significantly improve fault tolerance, again paying the price with cost and complexity.

As we've mentioned, most of the low-scoring operational characteristics can be raised by making this a distributed architecture with asynchronous communication, where each filter is a separate deployment unit and the pipes are remote calls. However, doing so will negatively affect other attributes such as simplicity and cost, illustrating one of the classic trade-offs of software architecture.

When to Use

The pipeline architecture is suitable for systems of any complexity with distinct, ordered, and deterministic one-way processing steps. This style's simplicity also makes it well suited for situations involving tight time frames and budget constraints.

When Not to Use

Because of the monolithic nature of this architectural style, it's not a good fit for systems that need high scalability, elasticity, and fault tolerance. However, using a distributed architecture approach will help mitigate these concerns.

The pipeline architecture style is also ill-suited for scenarios involving back-and-forth communication between filters, since the pipes are meant to be unidirectional only. While it's possible to use this architecture style for nondeterministic workflows by leveraging lots of tester filters throughout the pipeline, we don't recommend doing so. It overcomplicates this relatively simple architecture style and can negatively affect maintainability, testability, deployability, and consequently overall reliability. Event-driven architecture (see [Chapter 15](#)) is much better suited for situations involving nondeterministic workflows.

The pipeline architecture style appears in a variety of applications, especially with tasks that facilitate simple, one-way processing. For example, many electronic data interchange (EDI) tools use this pattern, building transformations from one document type to another using pipes and filters. Database extract, transform, and load (ETL) tools leverage pipeline architectures to modify data and flow it from one database or data source to another. Orchestrators and mediators such as [Apache Camel](#) use the pipeline architecture to pass information from one step in a business process to another.

Examples and Use Cases

To illustrate how the pipeline architecture can be used, consider the following example, where service telemetry information is sent from services via streaming to [Apache Kafka](#) (Figure 12-4).

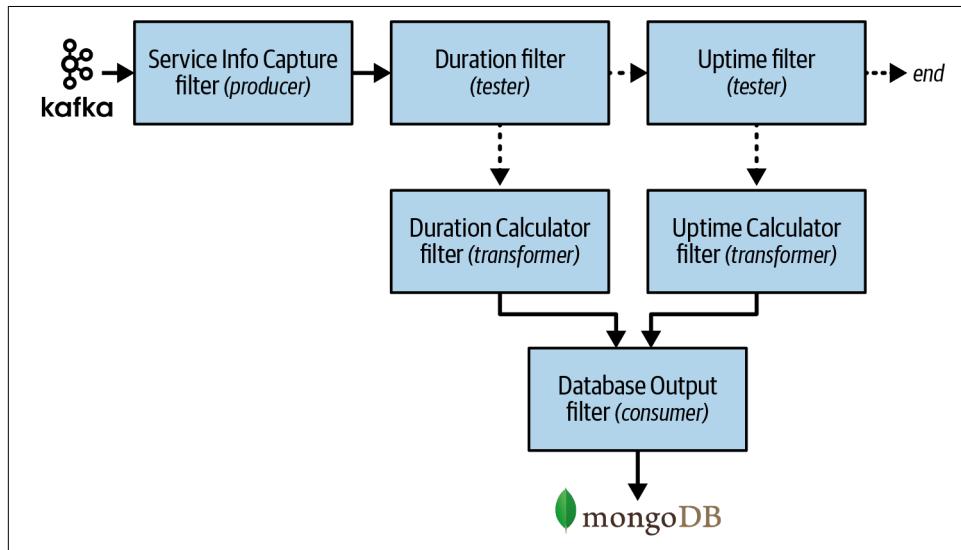


Figure 12-4. Pipeline architecture example

The system depicted in [Figure 12-4](#) uses the pipeline architecture style to process different kinds of data streamed to Kafka. The Service Info Capture filter (producer filter) subscribes to the Kafka topic and receives service information. It then sends this captured data to a tester filter called the Duration filter to determine whether it's related to the duration (in milliseconds) of the service request.

Notice the separation of concerns between the filters; the Service Info Capture filter is only concerned with how to connect to a Kafka topic and receive streaming data, whereas the Duration filter is only concerned with qualifying the data and determining whether to route it to the next pipe. If the data is related to the service request duration, it is passed to the Duration Calculator transformer filter; otherwise, the data is passed to the Uptime tester filter to check if it's related to uptime metrics. If not, then the pipeline ends—the data is of no interest to this particular processing flow. If it is relevant to uptime metrics, the Uptime tester filter passes it to the [Uptime Calculator](#), which uses it to calculate the uptime metrics. The [Uptime Calculator](#) then passes the modified data to the Database Output consumer, which persists it in a [MongoDB](#) database.

This example shows how extensible the pipeline architecture is. For example, in [Figure 12-4](#), a new tester filter could easily be added after the Uptime filter to send the data to be used in a new metric, such as for database connection wait time.

The pipeline architectural style, while typically a monolithic architecture, nevertheless demonstrates a good level of modularity thanks to the use of technically partitioned filters. It's a good fit for situations involving a workflow-based, step-by-step approach to processing data within a system.

Before we move on to more complicated distributed architectures, we have one more monolithic architecture to describe in the next chapter that also supports a good level of modularity through the use of plug-in components: the *microkernel architecture*.

Microkernel Architecture Style

The *microkernel* architecture style (also referred to as the *plug-in* architecture) was invented several decades ago and is still widely used today. This architecture style is a natural fit for *product-based applications*: that is, applications packaged and made available for download and installation as a single, monolithic deployment, typically installed on the customer's site as a third-party product. However, it is also widely used in nonproduct custom business applications, especially problem domains that require customization. For example, an insurance company in the US that has unique rules for each state, or an international shipping company that must adhere to various legal and logistical variations, would both benefit from this style.

Topology

The microkernel style is a relatively simple monolithic architecture consisting of two components: a core system and plug-ins. Application logic is divided between independent plug-in components and the basic core system, which isolates application features and provides extensibility, adaptability, and custom processing logic. [Figure 13-1](#) illustrates the basic topology of the microkernel architecture style.

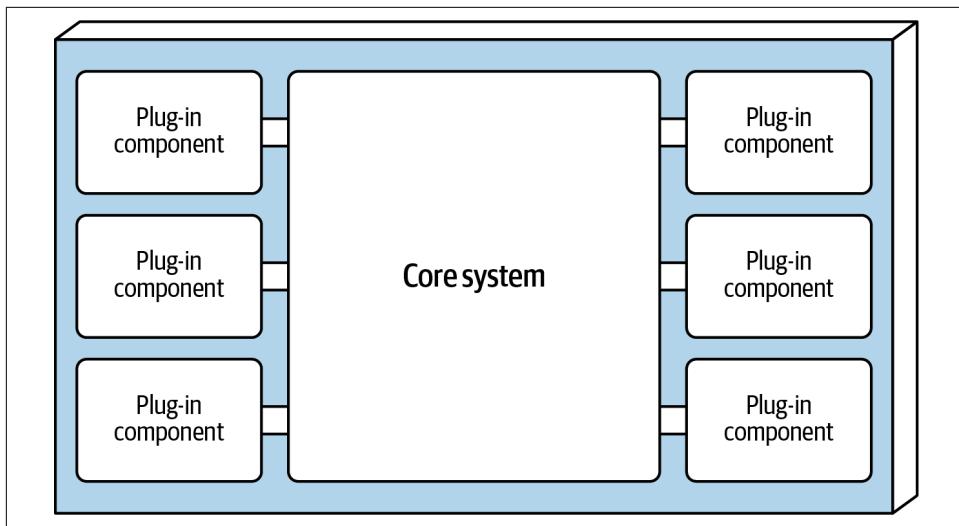


Figure 13-1. Basic components of the microkernel architecture style

Style Specifics

The essence of the microkernel architecture consists of two types of components: the *core system* and *plug-ins*.

Core System

The *core system* is formally defined as the minimal functionality required to run the system. Let's look at the Eclipse IDE for a good example of this. Eclipse's core system is just a basic text editor: open a file, change some text, and save the file. It's not until you add plug-ins that Eclipse starts becoming a usable product.

However, another definition of the core system is the *happy path*: a general processing flow through the application that involves little or no custom processing. A microkernel architecture takes the cyclomatic complexity of the application, removes it from the core system, and places it into separate plug-in components. This allows for better extensibility and maintainability, as well as increased testability.

To dive deeper, we'll return to Going Green, the electronic device recycling application we introduced in [Chapter 7](#). Let's say you're working on Going Green's application, which must assess each electronic device it receives according to specific, custom assessment rules. The Java code for this sort of processing might look as follows:

```

public void assessDevice(String deviceID) {
    if (deviceID.equals("iPhone6s")) {
        assessiPhone6s();
    } else if (deviceID.equals("iPad1")) {
        assessiPad1();
    } else if (deviceID.equals("Galaxy5")) {
        assessGalaxy5();
    } else ...
    ...
}

```

Rather than placing all this client-specific customization—which has lots of cyclomatic complexity—in the core system, you could create a separate plug-in component for each electronic device being assessed. Not only do specific client plug-in components isolate independent device logic from the rest of the processing flow, they also allow for expansion: adding a new device to assess is simply a matter of adding a new plug-in component and updating the registry. With the microkernel architecture style, assessing an electronic device only requires the core system to locate and invoke the corresponding device plug-ins, as illustrated in this revised source code:

```

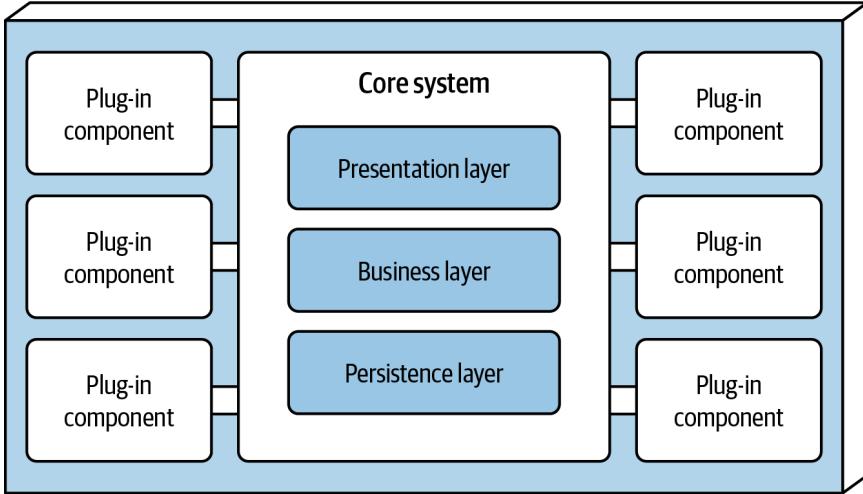
public void assessDevice(String deviceID) {
    String plugin = pluginRegistry.get(deviceID);
    Class<?> theClass = Class.forName(plugin);
    Constructor<?> constructor = theClass.getConstructor();
    DevicePlugin devicePlugin =
        (DevicePlugin)constructor.newInstance();
    devicePlugin.assess();
}

```

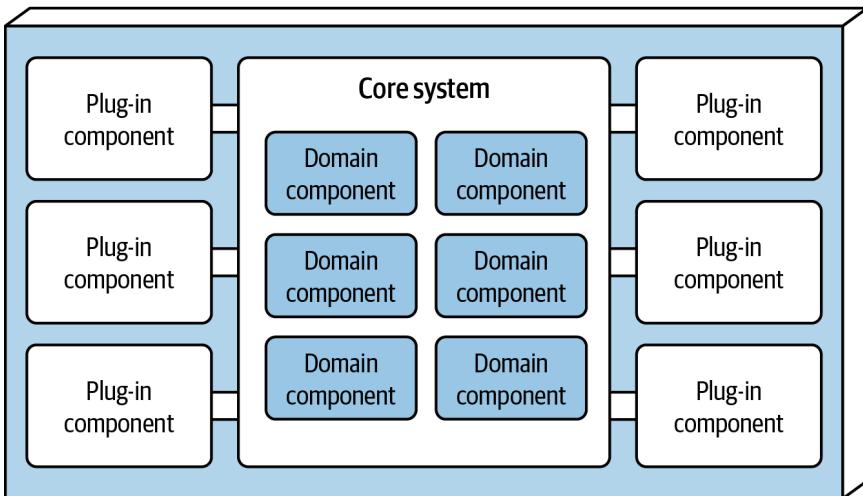
In this example, all of the complex rules and instructions for assessing a particular electronic device are self-contained in a standalone, independent plug-in component that can be generically executed from the core system.

Depending on its size and complexity, you could implement the core system as a layered architecture or as a modular monolith (as illustrated in [Figure 13-2](#)). In some cases, you might even split the core system into separately deployed domain services, with each domain service containing plug-in components specific to that domain. For the sake of this example, we'll assume that you implement it as a layered architecture for Going Green.

Suppose `Payment Processing` is the domain service representing the core system. Each payment method (credit card, PayPal, store credit, gift card, and purchase order) would have a separate plug-in component specific to that payment domain.



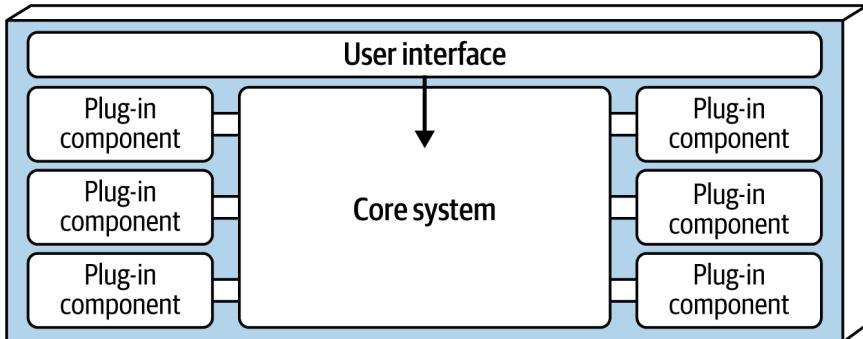
Layered core system (technically partitioned)



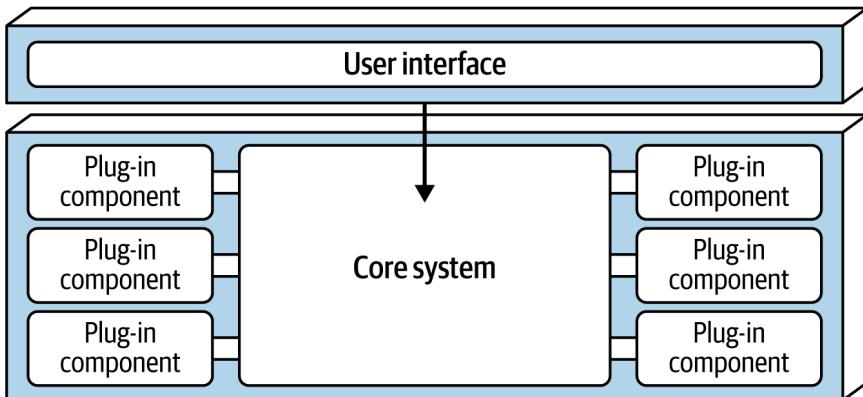
Modular core system (domain partitioned)

Figure 13-2. Variations of the microkernel architecture core system

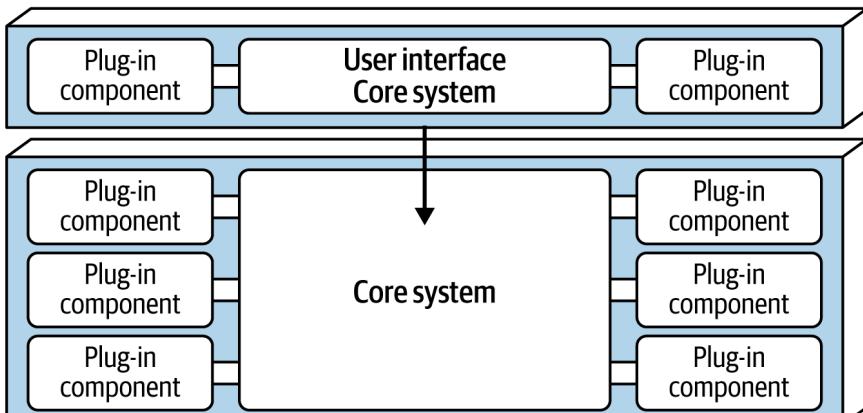
The core system's Presentation layer can be embedded within the core system or implemented as a separate user interface, with the core system providing backend services. As a matter of fact, you could also implement a separate UI using a microkernel architecture style. [Figure 13-3](#) illustrates these Presentation layer variants in relation to the core system.



Embedded user interface (single deployment)



Separate user interface (multiple deployment units)



Separate user interface (multiple deployment units, both microkernel)

Figure 13-3. User interface variants

Plug-In Components

Plug-in components are standalone, independent components that contain specialized processing, additional features, and custom code meant to enhance or extend the core system. Additionally, they isolate highly volatile code, creating better maintainability and testability within the application. Ideally, plug-in components should have no dependencies between them.

Communication between the plug-in components and the core system is generally *point-to-point*, meaning the “pipe” that connects the plug-in to the core system is usually a method invocation or function call to the entry-point class of the plug-in component. In addition, the plug-in component can be either compile based or runtime based. *Runtime* plug-in components can be added or removed at runtime without redeploying the core system or other plug-ins, and are usually managed through frameworks such as [Open Service Gateway Initiative \(OSGi\) for Java](#), [Penrose \(Java\)](#), [Jigsaw \(Java\)](#), or [Prism \(.NET\)](#). *Compile-based* plug-in components are much simpler to manage, but modifying, removing, or adding them requires the entire monolithic application to be redeployed.

Point-to-point plug-in components can be implemented as shared libraries (such as a JAR, DLL, or Gem), package names in Java, or namespaces in C#. In our Going Green application, each electronic device plug-in can be written and implemented as a JAR, DLL, or Ruby Gem (or any other shared library), with the name of the device matching the name of the independent shared library, as illustrated in [Figure 13-4](#).

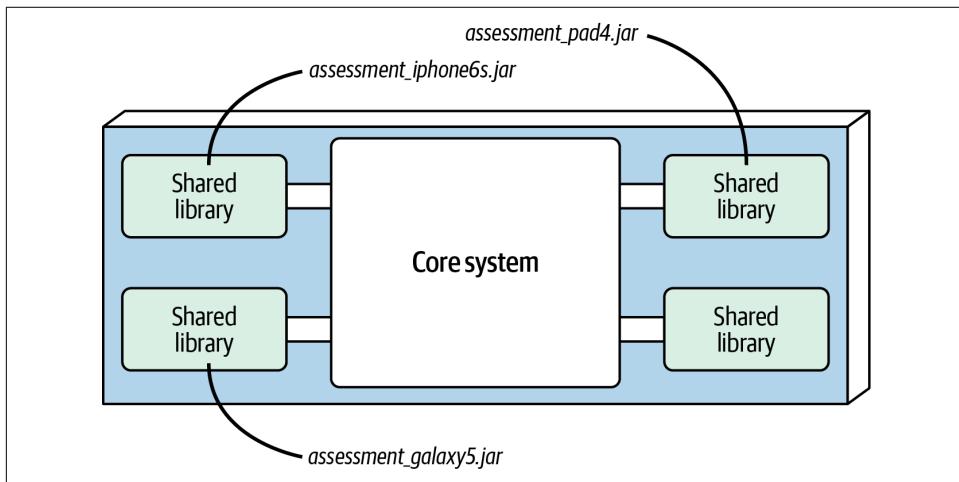


Figure 13-4. Shared library plug-in implementation

An easier approach, shown in [Figure 13-5](#), is to implement each plug-in component as a separate namespace or package name within the same code base or IDE project. When creating the namespace, we recommend the following semantics: `app.plugin.<domain>.<context>`. For example, consider the namespace `app.plugin.assessment.iphone6s`. The second node (`plugin`) makes it clear that this component is a plug-in and therefore should strictly adhere to the basic rules (they must be self-contained and separate from other plug-ins). The third node describes the domain (in this case, `assessment`), allowing plug-in components to be organized and grouped by a common purpose. The fourth node (`iphone6s`) describes the specific context for the plug-in, making it easy to locate the device plug-in for modification or testing.

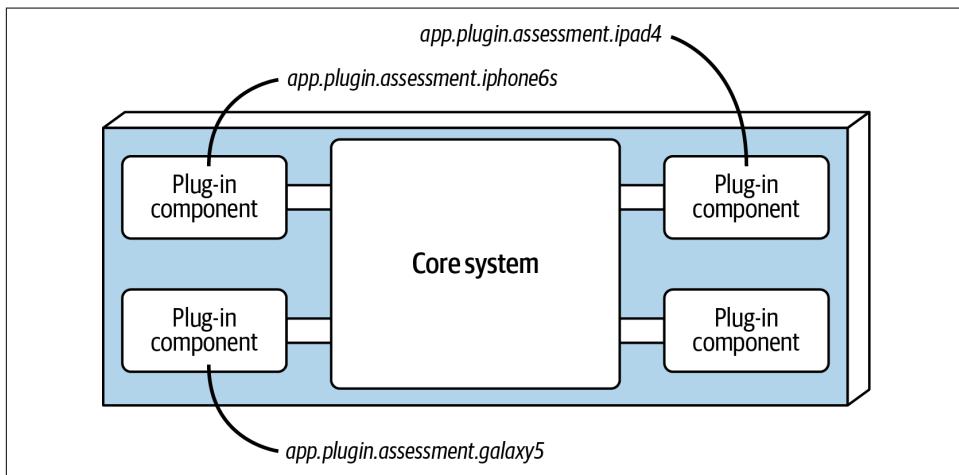


Figure 13-5. Package or namespace plug-in implementation

Plug-in components do not always have to use point-to-point communication with the core system. Alternatives include using REST or messaging to invoke plug-in functionality, with each plug-in being a standalone service (or maybe even a micro-service, implemented using a container). Although this may sound like a good way to increase overall scalability, note that this topology (illustrated in [Figure 13-6](#)) is still only a single architecture quantum, due to the monolithic core system. Every request must first go through the core system to get to the plug-in service.

The benefits of the remote-access approach to plug-in components implemented as individual services is that it allows for better overall component decoupling, better scalability and throughput, and runtime changes without any special frameworks (like OSGi, Jigsaw, or Prism). It also allows for asynchronous communications to plug-ins which, depending on the scenario, could significantly improve overall user responsiveness. Using the Going Green example, rather than having to wait for the electronic device assessment to run, the core system could make an asynchronous request to kick off an assessment for a particular device. When the assessment

completes, the plug-in can notify the core system through another asynchronous messaging channel, which in turn notifies the user that the assessment is complete.

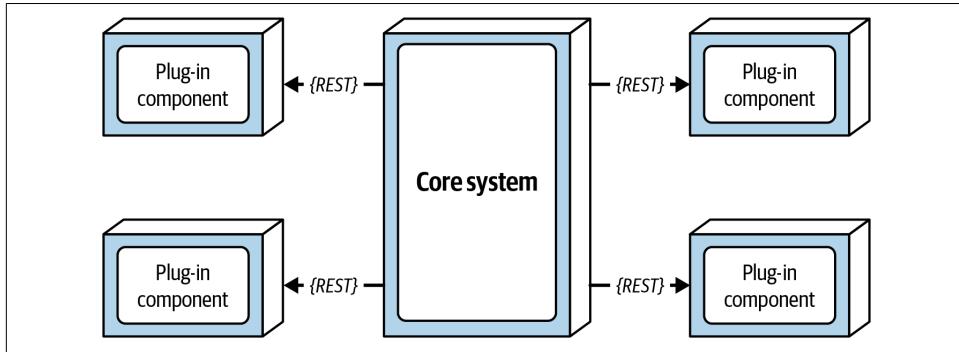


Figure 13-6. Remote plug-in access using REST

With these benefits come trade-offs. Remote plug-in access turns the microkernel architecture into a distributed architecture rather than a monolithic one, making it difficult to implement and deploy for most third-party on-prem products. Furthermore, it creates more overall complexity and cost, and complicates the overall deployment topology. If a plug-in becomes unresponsive or stops running, particularly if the system is using REST, the request cannot be completed. This would not be the case with a monolithic deployment. The choice between point-to-point and remote communication should be based on the project's specific requirements and a careful trade-off analysis.

The Spectrum of “Microkern-ality”

Not all systems that support plug-ins are microkernels, but all microkernels support plug-ins. A system’s degree of “microkern-ality,” as we call it, depends on how much standalone functionality exists in the core system. This is reflected in the spectrum illustrated in Figure 13-7.



Figure 13-7. The spectrum of “microkern-ality”

In Figure 13-7, “pure” microkernel architectures (like the aforementioned Eclipse IDE or linter tools) have very little core functionality. For example, a linter parses source code and delivers the abstract syntax tree so that a developer can write rules about language use. The core parses the code, but until someone writes a plug-in to take advantage of it, it’s of little use. Contrast that with a web browser, which supports

plug-ins but is perfectly functional without them; browsers fall on the righthand side of the spectrum.

Determining the volatility of the core goes a long way toward helping architects decide between a system that just supports plug-ins or a more “pure” microkernel.

Registry

The core system needs to know which plug-in modules are available and how to get to them. One common way of implementing this is through a *plug-in registry*. This registry contains information about each plug-in module, including things like its name, data contract, and remote access protocol details (depending on how the plug-in is connected to the core system). For example, a plug-in for tax software that flags high-risk tax-audit items might have a registry entry that contains the name of the service (AuditChecker), the data contract (input data and output data), and the contract format (XML).

The registry can be as simple as an internal map structure owned by the core system containing a key and the plug-in component reference, or it can be as complex as a registry and discovery tool embedded within the core system or deployed externally (such as [Apache ZooKeeper](#) or [Consul](#)). Using the electronics recycling example, the following Java code implements a simple registry within the core system, showing examples of a point-to-point entry, a messaging entry, and a RESTful entry for assessing an iPhone 6S device:

```
Map<String, String> registry = new HashMap<String, String>();
static {
    //point-to-point access example
    registry.put("iPhone6s", "Iphone6sPlugin");

    //messaging example
    registry.put("iPhone6s", "iphone6s.queue");

    //restful example
    registry.put("iPhone6s", "https://atlas:443/assess/iphone6s");
}
```

Contracts

The contracts between plug-in components and the core system are usually standard across a domain of plug-in components and include behavior, input data, and output data returned from the plug-in component. Custom contracts are typically used in situations where the plug-in components are developed by a third party, so the architect has no control over the contract the plug-in uses. In such cases, it is common to create an adapter between the plug-in contract and your standard contract so that the core system doesn’t need specialized code for each plug-in.

Plug-in contracts can be implemented in XML, in JSON, or even in objects passed back and forth between the plug-in and the core system. In the electronics recycling application, the following contract (implemented as a standard Java interface named `AssessmentPlugin`) defines the overall behavior (`assess()`, `register()`, and `deregister()`), along with the corresponding output data expected from the plug-in component (`AssessmentOutput`):

```
public interface AssessmentPlugin {
    public AssessmentOutput assess();
    public String register();
    public String deregister();
}

public class AssessmentOutput {
    public String assessmentReport;
    public Boolean resell;
    public Double value;
    public Double resellPrice;
}
```

In this contract example, the device assessment plug-in is expected to return the assessment report with:

- A formatted string
- A resell flag (true or false) indicating whether this device can be resold on a third-party market or safely disposed of
- If the item can be resold, its calculated value and recommended resell price

Notice the roles and responsibility model between the core system and the plug-in component in this example, specifically with the `assessmentReport` field. It is not the responsibility of the core system to format and understand the details of the assessment report, only to either print it out or display it to the user.

Data Topologies

Generally, teams implement microkernel architectures as monolithic architectures, using a single (typically relational) database.

It's uncommon practice for plug-in components to connect directly to a centrally shared database. Instead, the core system takes on this responsibility, passing whatever data is needed to each plug-in. The primary reason for this practice is decoupling. Making a database change should only affect the core system, not the plug-in components. That said, plug-ins can have separate data stores that are only accessible to that plug-in. For example, each device assessment plug-in in the Going Green system could have its own simple database or rules engine containing all of the specific assessment rules for that product. The data store owned by the plug-in

component can be external (as shown in [Figure 13-8](#)), or embedded as part of the plug-in component or monolithic deployment (as in the case of an in-memory or embedded database).

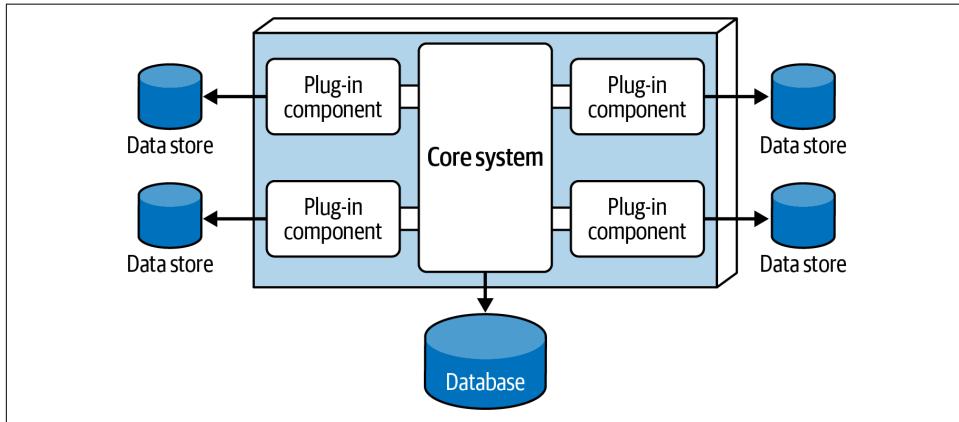


Figure 13-8. A plug-in component can own its own data store

Cloud Considerations

Because microkernel architectures are typically monolithic, the cloud represents a few coarse-grained options. The first option is to deploy the entire application on the cloud, using cloud facilities or containers. The second is to place just the data in the cloud and implement the microkernel as an on-premises system. The third option segregates the core system as on-premises and places the plug-ins in the cloud. While this may seem like a good option from a modularity standpoint, it comes with some challenging implications for responsiveness. Generally, in a microkernel architecture, plug-in calls happen frequently and each call passes a fair amount of information, since teams implement key workflows using plug-ins. The latency incurred by separating the core and plug-ins may lead to undesirable overhead.

Common Risks

The common risks associated with this architecture mainly arise from misapplying it.

Volatile Core

The core in a microkernel architecture is supposed to be as stable as possible after initial development; part of the benefit of this style is that it isolates changes to plug-ins. Building a core that undergoes constant change undermines the philosophy of this architecture, but it's a common mistake. Often this is the result of architects misjudging the core's volatility; they must then address that volatility via refactoring.

Plug-In Dependencies

Microkernels work best when the plug-ins communicate only with the core system, not with each other. Most systems that use plug-ins that are not microkernels use *dependency-free plug-ins*, which are plug-ins that have no dependencies other than the core. In other words, the plug-ins don't communicate with each other and therefore have no shared dependencies that must be resolved by the core. However, complex applications of microkernel architecture, such as the Eclipse IDE, sometimes build dependencies between components, making the core resolve transitive dependency conflicts between them.

While it's thus possible to have dependencies, they create myriad complexities around transitive dependency management. What happens if two plug-ins depend on different versions of the same core library? The core must resolve those dependencies and facilitate communication between the different plug-in versions. Anyone who has struggled with adding plug-ins in an environment where transitive dependencies are in play understands the headache of untangling conflicting versions. It is best to try to avoid dependencies between plug-ins whenever possible.

Governance

Governance in a microkernel architecture revolves around checking how well the architects are honoring its philosophy.

Common governance checks include:

- Volatility checks for the core—which are fitness functions wired into checking churn in version control, rather than a specific code check
- Rate of change in the core
- Contract tests (especially if some plug-ins support different versions than others because of gradual evolution)
- Other structural verifications for the topology

Team Topology Considerations

The obvious split for teams in this architecture is between core and plug-ins, reflecting its topology:

Stream-aligned teams

The core is an obvious sweet spot for stream-aligned teams, which build the core functionality of the system. Plug-ins may also fall to this team, depending on the type of application.

Enabling teams

The microkernel architecture is extremely well suited for enabling teams, since it segregates some behavior in plug-ins to allow for A/B testing and other experiments.

Complicated-subsystem teams

Microkernels are also well suited for complicated-subsystem teams because they defer specialized behavior to plug-ins. For example, specialized processing like analytics might be isolated in plug-ins, allowing the stream-enabled team to work on core behavior and call out to sophisticated plug-ins for specialized behavior.

Platform teams

Platform teams mostly concern themselves with the operational details for this architecture, as with other monolithic architectures.

Architecture Characteristics Ratings

A one-star rating in the characteristics ratings in [Figure 13-9](#) means the specific architecture characteristic isn't well supported in the architecture, whereas a five-star rating means the architecture characteristic is one of the strongest features in the architecture style. The definition for each characteristic identified in the scorecard can be found in [Chapter 4](#).

In the microkernel architecture style, similar to layered, simplicity and overall cost are the main strengths, and scalability, fault tolerance, and elasticity the main weaknesses. These weaknesses are due to the typical monolithic deployments found with the microkernel architecture. Also like the layered architecture style, the architecture quanta is always singular (1) because all requests must go through the core system to get to the independent plug-in components. That's where the similarities end.

Microkernel is unique in that it is the only architecture style that can be both domain partitioned *and* technically partitioned. While most microkernel architectures are technically partitioned, the domain-partitioning aspect comes about mostly through a strong domain-to-architecture isomorphism. For example, problems that require different configurations for each location or client match extremely well with this architecture style. So do products or applications that emphasize user customization and feature extensibility (such as Jira or an IDE like Eclipse).

| | Architectural characteristic | Star rating |
|-------------|------------------------------|----------------------|
| Structural | Overall cost | \$ |
| | Partitioning type | Domain and technical |
| | Number of quanta | 1 |
| | Simplicity | ★★★★★ |
| | Modularity | ★★★★ |
| | Maintainability | ★★★★ |
| | Testability | ★★★★ |
| | Deployability | ★★★★ |
| | Evolvability | ★★★★ |
| | Responsiveness | ★★★★ |
| Engineering | Scalability | ★ |
| | Elasticity | ★ |
| | Fault tolerance | ★ |
| | | |
| Operational | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |

Figure 13-9. Microkernel architecture characteristics ratings

Testability, deployability, and reliability rate a little above average (three stars), primarily because functionality can be isolated to independent plug-in components. If done right, this reduces the overall testing scope of changes as well as the overall risk of deployment, particularly if plug-in component deployment is runtime based.

Modularity and evolvability also rate a little above average (three stars). With the microkernel architecture style, additional functionality can be added, removed, and changed through independent, self-contained plug-in components, making it relatively easy to extend and enhance applications, and allowing teams to respond to changes much faster. Consider the tax-preparation software example from the previous section. If the US tax law changes (which it does all the time), requiring a new tax form, that new tax form can be created as a plug-in component and added to the application without much effort. Similarly, if a tax form or worksheet is no longer needed, that plug-in can simply be removed from the application.

Responsiveness is always an interesting characteristic to rate with the microkernel architecture style. We gave it three stars (a little above average), mostly because microkernel applications are generally small and don't grow as big as most layered architectures. Also, they don't suffer as much from the Architecture Sinkhole

antipattern we discussed in [Chapter 10](#). Finally, microkernel architectures can be streamlined by unplugging unneeded functionality, making the application run faster. A good example of this is [WildFly](#) (previously the JBoss Application Server). Unplugging unnecessary functionality like clustering, caching, and messaging makes the application server perform much faster than it does with these features in place.

Examples and Use Cases

Most tools for developing and releasing software are implemented using the microkernel architecture. Some examples include the [Eclipse IDE](#), [PMD](#), [Jira](#), and [Jenkins](#), to name a few. Internet web browsers, such as Chrome and Firefox, also commonly use the microkernel architecture: viewers and other plug-ins add additional capabilities not otherwise found in the basic browser (the core system). We could point to endless examples of product-based software, but what about large business applications? The microkernel architecture applies to these situations as well.

To illustrate this point, consider our earlier example of tax-preparation software. The US tax agency, the Internal Revenue Service, has a basic two-page tax form called the 1040 form that contains a summary of all the information needed to calculate a person's tax liability. Each line in the 1040 tax form has a single number, such as gross income, and arriving at each of those numbers requires many other forms and worksheets. Each additional form and worksheet can be implemented as a plug-in component, with the 1040 summary tax form being the core system (the driver). This way, changes to tax law can be isolated to an independent plug-in component, making changes easier and less risky.

Another example of a large, complex business application that can leverage the microkernel architecture is insurance claims processing. Claims processing is a very complicated process. Each jurisdiction has different rules and regulations for what is and isn't allowed in an insurance claim. For example, some jurisdictions (such as US states) allow insurance companies to provide free windshield replacement if your windshield is damaged by a rock; others do not. This creates an almost infinite set of conditions for a standard claims process.

Most insurance claims applications leverage large, complex rules engines to handle much of this complexity. A *rules engine* is a framework or library that allows developers (or end users) to define a series of rules or steps to define a workflow declaratively, either using visual tools or a domain-specific language. However, these rules engines can grow into the Big Ball of Mud antipattern, where a simple rule change requires an army of analysts, developers, and testers to make sure nothing breaks. Using the microkernel architecture pattern can solve many of these issues.

The claims rules for each jurisdiction can be contained in separate, standalone plug-in components, implemented as source code or as a specific rules-engine instance

accessed by the plug-in component. This way, rules can be added, removed, or changed for a particular jurisdiction without affecting any other part of the system. Furthermore, new jurisdictions can be added and removed without affecting other parts of the system. The core system, in this example, would be the standard process for filing and processing a claim—something that doesn't change often.

The microkernel architecture style is extremely common; once you've seen it, you start noticing it everywhere. It's a case where an architecture structure (core + plugins) matches the common domain problem of customization, and it turns out that customization comes up a lot in software.

Service-Based Architecture Style

Service-based architecture is a hybrid variant of the microservices architectural style and is considered one of the most pragmatic styles available, mostly due to its flexibility. Although service-based architecture is distributed, it doesn't have the same level of complexity and cost as other distributed architectures (such as microservices or event-driven architecture), making it a popular choice for business-related applications.

Topology

The basic topology of service-based architecture follows a distributed macro-layered structure consisting of a separately deployed user interface; separately deployed, remote, coarse-grained services; and, optionally, a monolithic database.

Although [Figure 14-1](#) illustrates the *basic* topology of this style, it can vary greatly, including using separate UIs and separate databases. This chapter will describe these topology variants in detail.

Services within this architectural style typically represent a specific domain or subdomain with a system, so they are called *domain services*. Domain services are generally coarse-grained and represent some portion of the system's functionality (such as order fulfillment or order shipping). They are typically independent of each other and separately deployed. Services are typically deployed much like any monolithic application would be, so they do not require containerization (although deploying a domain service in a container such as Docker or Kubernetes is certainly an option). When using a single monolithic database, an architect should try to minimize the number of domain services (we recommend no more than 12) to avoid change control, scalability, and fault tolerance issues.

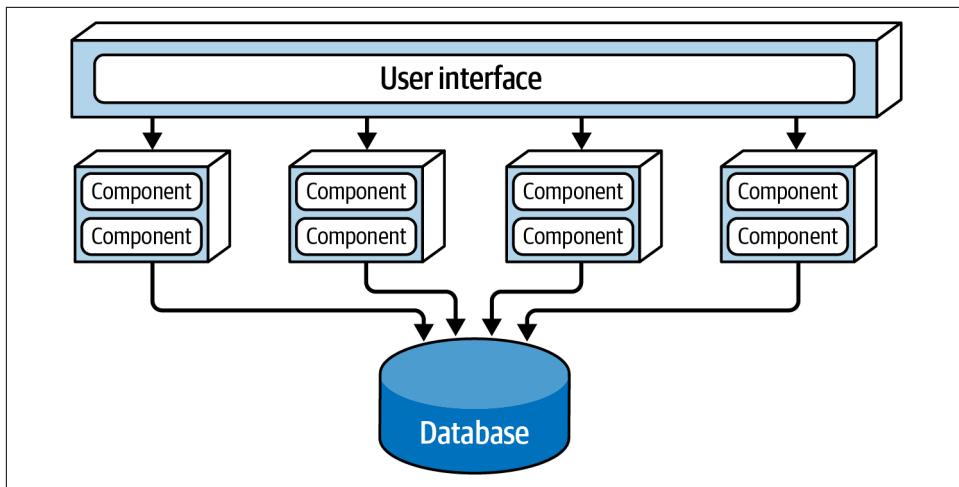


Figure 14-1. Basic topology of the service-based architectural style

Typically, each domain service is deployed as a single instance. However, based on the system's scalability, fault tolerance, and throughput needs, architects sometimes create multiple instances of a domain service. Doing so usually requires some sort of load-balancing capability between the UI and the domain service so that the UI can be directed to a healthy and available service instance.

Services are accessed remotely from a UI using a remote access protocol, typically REST. Other possibilities include messaging, remote procedure call (RPC), an API layer with a proxy or gateway, or even SOAP. However, in most cases, the UI has an embedded **service locator pattern** so it can access the services directly; the service locator pattern can also be embedded within an API Gateway or proxy.

The service-based architecture typically uses a centrally shared monolithic database. This allows services to leverage SQL queries and joins just as a traditional monolithic layered architecture would. Because of the small number of domain services typically found in this architectural style, exhausting the available database connections is rarely an issue. Managing database changes, however, can present a challenge. “[Data Topologies](#)” on page 215 describes techniques for addressing and managing database changes within a service-based architecture.

Style Specifics

Because domain services in a service-based architecture are generally coarse-grained, each domain service is typically designed using a layered architectural style consisting of an API Facade layer, a Business layer, and a Persistence layer. Another popular design approach is to partition each domain service into subdomains, similar to the modular monolith architectural style (see [Chapter 11](#)). Both approaches are illustrated in [Figure 14-2](#).

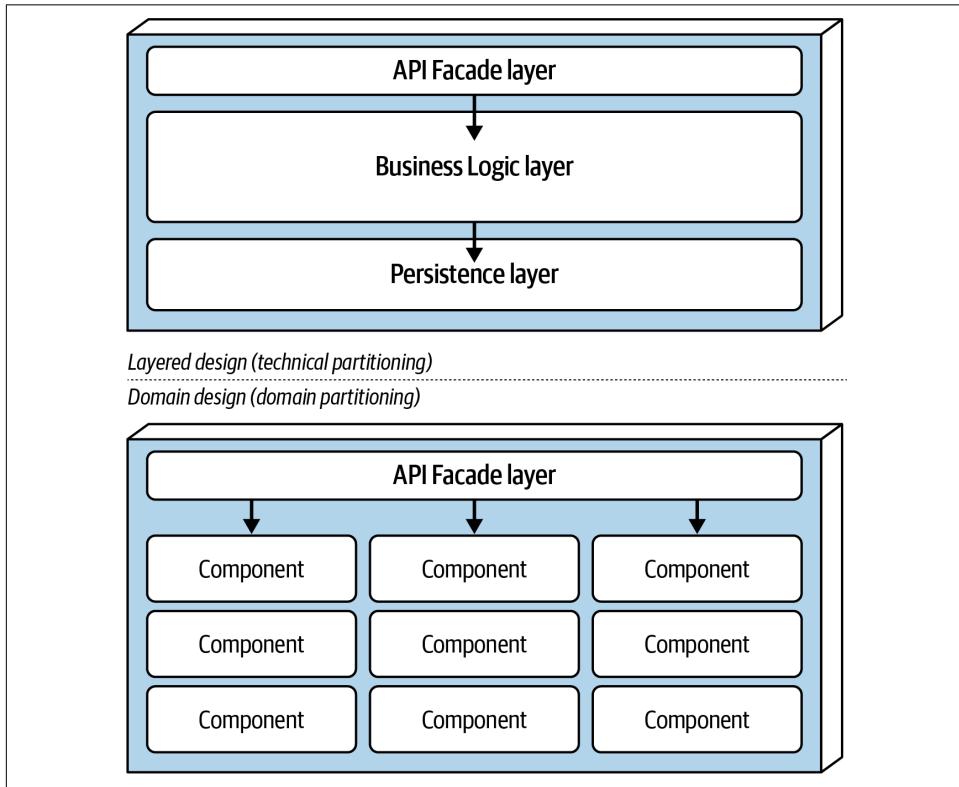


Figure 14-2. Domain service design variants

Regardless of its design, a domain service must contain some sort of API access facade that the UI interacts with to execute some sort of business functionality. The API access facade typically takes on the responsibility of orchestrating the business request from the UI.

Let's consider an example from an ecommerce site with a service-based architecture. A business request comes from the UI: a customer is ordering some items. This single request is received by the API access facade within the `OrderService` domain service. The API access facade internally orchestrates everything that has to happen to fulfill this request: placing the order, generating an order ID, applying the payment, and updating the inventory for each product ordered. In a microservices architecture, completing this request would likely involve orchestrating many separately deployed, remote, single-purpose services. This difference in granularity—between internal class-level orchestration in a service-based architecture and external service orchestration in microservices—points to one of the many significant differences between the two styles.

Because domain services are coarse-grained, regular ACID (atomicity, consistency, isolation, durability) database transactions with standard database commits and rollbacks can ensure database integrity within a single domain service. Compare this to the fine-grained single-purpose services of highly distributed architectures like microservices, which use a distributed transaction technique known as BASE transactions (basic availability, soft state, eventual consistency). Such fine-grained services don't support the same level of database integrity that ACID transactions in a service-based architecture can support.

To illustrate this point, consider the order checkout process within our example service-based ecommerce site. Suppose the customer places an order, but the credit card they use for payment has expired. Since this payment is an atomic transaction within the same service, the service can remove everything added to the database so far using a standard transaction rollback. The service notifies the customer that the payment cannot be applied.

Now consider this same process in a microservices architecture, with its smaller, fine-grained services. First, the `OrderPlacement` service accepts the request, creates the order, generates an order ID, and inserts the order into the order tables. Once this is done, the `OrderPlacement` service makes a remote call to the `PaymentService`, which tries to apply the payment. If the payment cannot be applied because the credit card is expired, the order cannot be placed. Now the data is in an inconsistent state: the order information has already been inserted, but has not been approved. A separate action known as a *compensating update* (discussed in [Chapter 9](#)) would need to be applied to the `OrderPlacement` service to bring the data to a consistent state.

Service Design and Granularity

Because domain services are coarse-grained, they allow for better data integrity and consistency, but there's a big trade-off. In a service-based architecture, changing the order-placement functionality in an `OrderService` would require the team to test and redeploy all of the service's functionality, including payment processing. In a microservices architecture, however, the same change would only affect a smaller, fine-grained `OrderPlacement` service, requiring no testing or deployment for the `PaymentService`. Furthermore, because a domain service deploys more functionality, there's more risk that something else might break (including payment processing). With microservices, each service has a single responsibility, so there's less chance of breaking other functionalities when making changes.

User Interface Options

The service-based architecture style includes many UI variants, making it very flexible. For example, an architect could break apart the single monolithic UI illustrated in [Figure 14-1](#) into separate UIs, even to the point of matching each domain service. This would increase the system's overall scalability, fault tolerance, and agility. These UI variants are illustrated in [Figure 14-3](#).

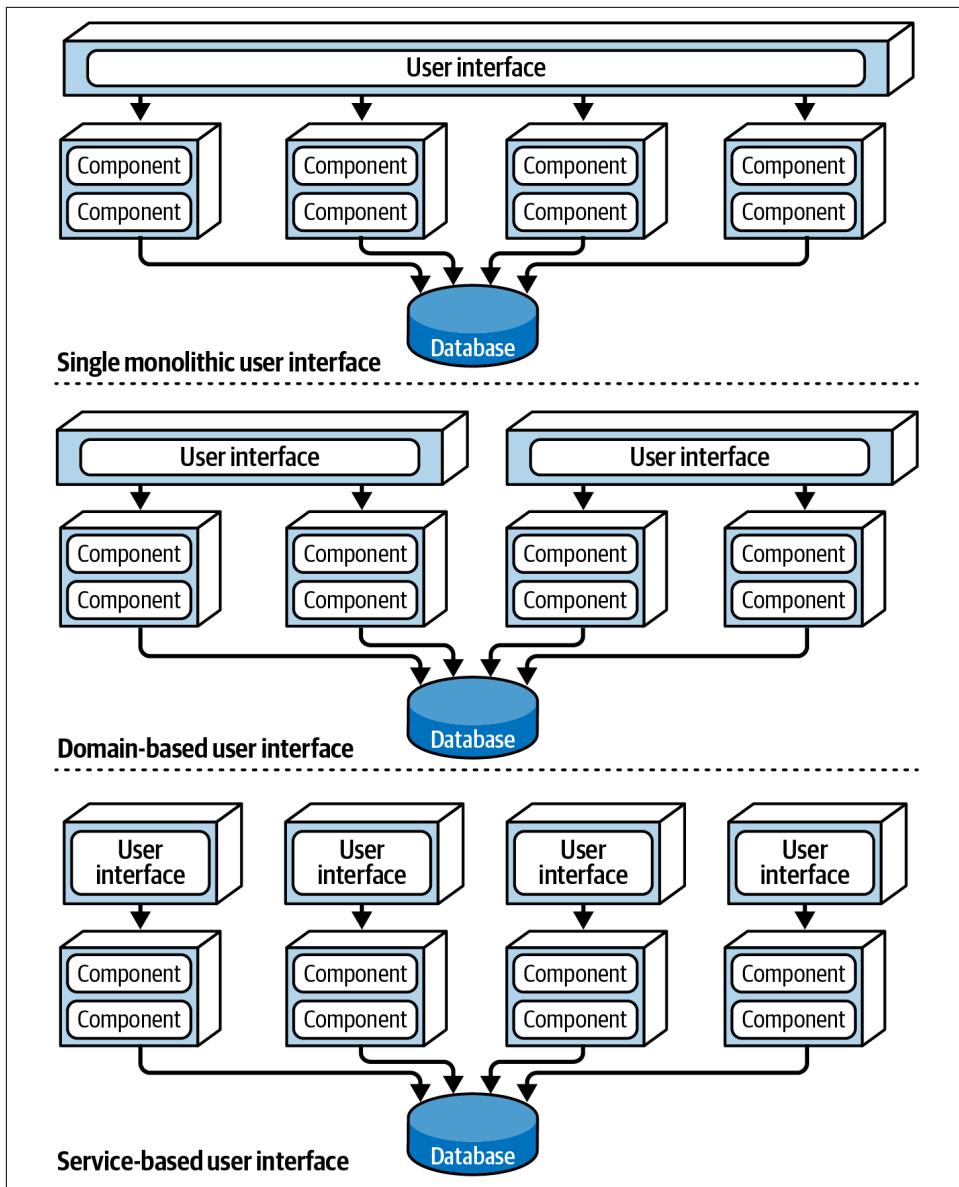


Figure 14-3. Three user-interface variants in a service-based architecture

For example, a typical ordering system could have a UI for customers to place orders, and separate, internal UIs for the order packers to view the items that need to be packed and for customer support.

API Gateway Options

Thanks to the flexibility of this architectural style, it's possible to add an API layer consisting of a reverse proxy or API Gateway between the UI and services, as shown in [Figure 14-4](#). This is useful for exposing domain service functionality to external systems, for consolidating shared cross-cutting concerns (such as metrics, security, auditing requirements, and service discovery) and moving them into the API Gateway, and for load-balancing domain services that have multiple instances.

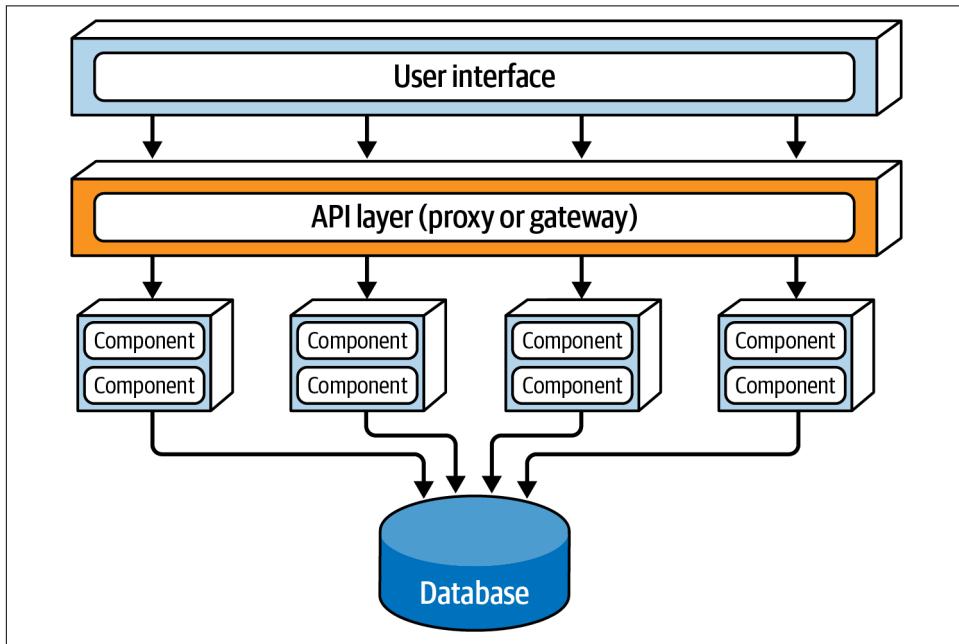


Figure 14-4. Adding an API layer between the UI and the domain services

Data Topologies

Service-based architectures offer architects many choices of database topology, further illustrating their flexibility. This style is unique in being a distributed architecture that can effectively support a monolithic database. However, that single monolithic database could be broken apart into separate databases, even going as far as to create a domain-scoped database to match each domain service (similar to microservices). If using multiple separate databases, the architect must make sure that no other domain service needs the data in each database, which might incur interservice communication between domain services. It's usually preferable to share data rather than call another domain service in this architectural style. These database variants are illustrated in [Figure 14-5](#).

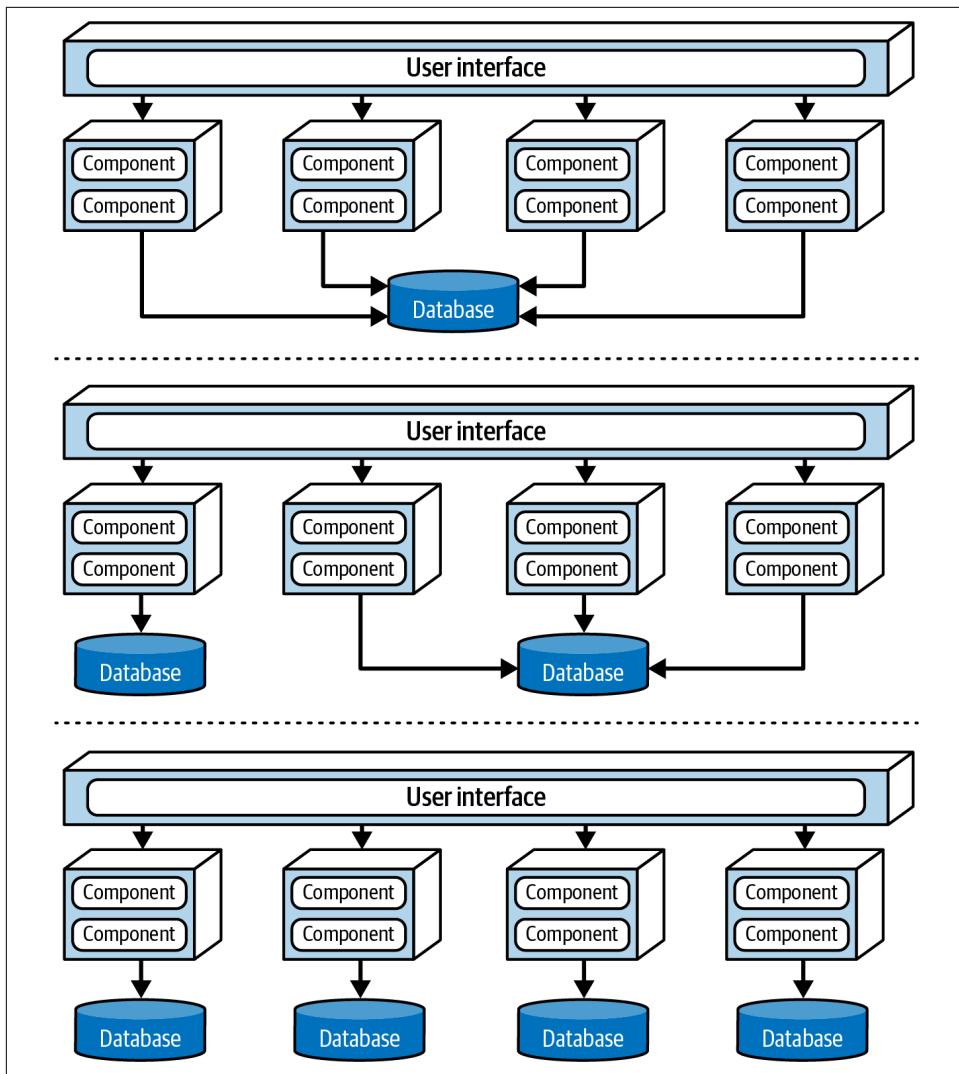


Figure 14-5. Service-based architecture database topologies

Although this architectural style supports a monolithic database, schema changes to the database table can, if not done properly, affect every domain service. This makes changing the database a very costly, risky task that demands a great deal of effort, coordination, and overall reliability.

In a service-based architecture, the shared class files representing the database table schemas (called *entity objects*) usually reside in a custom shared library used by all of the domain services, such as a JAR or DLL file. These shared libraries can also contain SQL code. Creating a single shared library of all of the entity objects is

the *least* effective way to implement a service-based architecture. Any change to the database table structures also requires changing that library of the corresponding entity objects, which means changing and redeploying *every* service, regardless of whether it actually accesses the changed table. Shared library versioning can help address this issue, but it's still difficult to know which services the table change will affect without doing detailed manual analysis. This scenario, which is considered an antipattern in service-based architecture, is illustrated in [Figure 14-6](#).

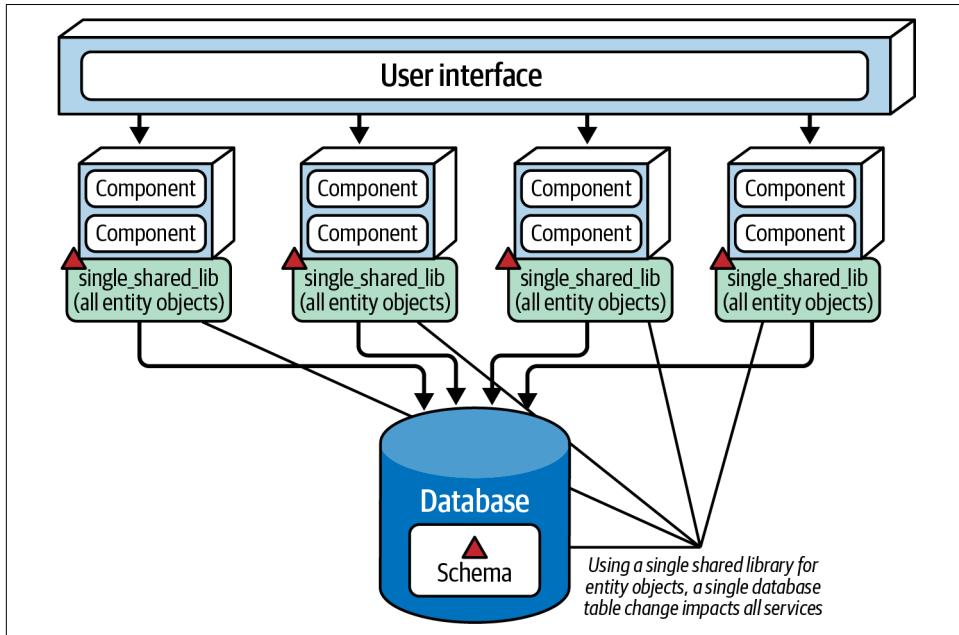


Figure 14-6. Using a single shared library for database entity objects impacts all services when a change occurs and is considered an antipattern in service-based architecture

One way to mitigate the impact and risk of database changes is to logically partition the database and represent that logical partitioning through separate shared libraries. Notice that in [Figure 14-7](#), the database is logically partitioned into five separate domains: common, customer, invoicing, order, and tracking. The domain services use five corresponding shared libraries that match the logical partitions in the database. Using this technique, any changes the architect makes to a table within a particular logical domain (in this case, Invoicing) would match the corresponding shared library containing the entity objects (and possibly SQL as well). The only services affected would be those using that shared library. No other services would be affected by this change, so there would be no need to retest and redeploy them.

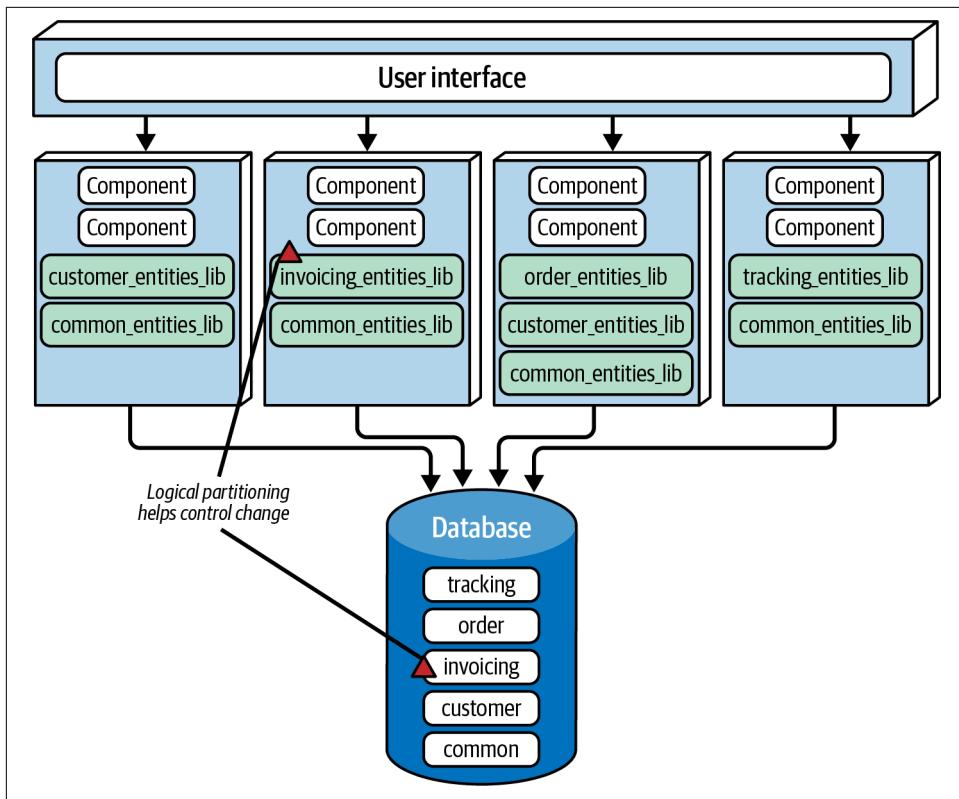


Figure 14-7. Using multiple shared libraries for database entity objects

In [Figure 14-7](#), the database contains a Common domain and a corresponding `common_entities_lib` shared library, which is used by all services. This is relatively common. Since these tables are common to all services, changing them requires coordinating all services that access the shared database. One way to mitigate the potential rippling side effects of changes to these tables (and to their corresponding entity objects and domain services) is to lock the common entity objects in the version control system (if available) and allow only the database team to make changes. This helps control change and emphasizes the significance of making changes to the common tables used by all services.



To better control database changes within a service-based architecture, make the logical partitioning in the database as fine-grained as possible—while still maintaining well-defined data domains.

Cloud Considerations

Being a distributed architecture, the service-based style works well in cloud environments, even though its domain services are typically coarse-grained. Due to their large scope, domain services are typically implemented as containerized services rather than as serverless functions, and they can easily leverage cloud file storage, database, and messaging services.

Common Risks

While interservice communication is typical with microservices, it's something architects try to avoid in the service-based architectural style. Ideally, domains should be as independent as possible, with coupling focused only at the database level. Too much communication between domain services is a good indication that either the architect didn't partition the domains correctly or this isn't the right architectural style for the problem they are solving.

Another common risk is creating *too many* domain services. The practical upper limit is around 12. Any more than that is likely to start causing issues with testing, deployment, monitoring, and database connections and changes.

Governance

In addition to the common structural and operational architectural governance techniques discussed throughout this book, such as cyclomatic complexity, scalability, responsiveness, and so on, architects can apply specific governance tests to ensure the structural integrity of a service-based architecture.

Because domain services should be as independent as possible, the first aspect of governing this style is ensuring that changes don't span across multiple domain services. If they do, that's a good indication that the domain boundaries aren't appropriately defined, or that service-based is not the most appropriate architectural style for the problem.

When interservice communication is unavailable, architects can also govern the amount of communication between domain services. There are certainly situations and workflows where one domain service must communicate with another: for example, an `OrderProcessing` domain might need to communicate with a `CustomerNotification` domain to email order status information to the customer. For the most part, though, domain services should be largely independent from each other, with orchestration taking place at the UI or API Gateway level.

Team Topology Considerations

Since service-based architectures are domain partitioned, they work best when teams are also aligned by domain area (such as cross-functional teams with specialization). When a domain-based requirement comes along, a domain-focused cross-functional team can work together on that feature within a specific domain service, without interfering with other teams or services. Conversely, technically partitioned teams (such as UI teams, backend teams, database teams, and so on) do not work well with this architectural style because of its domain partitioning. Assigning domain-based requirements to a technically organized team requires a level of interteam communication and collaboration that proves difficult in most organizations.

Here are some considerations for architects with regard to aligning a service-based architecture with the specific team topologies outlined in “[Team Topologies and Architecture](#)” on page 151:

Stream-aligned teams

If domain boundaries are properly aligned, stream-aligned teams work well with this architectural style, particularly when their streams are focused on a specific domain. However, service-based architecture becomes more challenging when streams cross the boundaries defined by the domain services. In this case, the architect should analyze the boundaries and granularity of the domain services and either realign them to the streams or choose a different architectural style.

Enabling teams

Service-based architecture is not as effective when paired with the enabling team topology as other distributed architectures are, due to the coarse-grained nature of its domain services. However, an architect can increase this style’s modularity by carefully identifying and creating appropriate components within each domain service. Specialists and cross-cutting team members can then make suggestions and perform experiments based on those components.

Complicated-subsystem teams

Complicated-subsystem teams can leverage this architecture style’s domain-level and subdomain-level modularity to focus on complicated domain or subdomain processing, independent of other team members (and services).

Platform teams

Service-based architecture’s high degree of modularity helps teams leverage the benefits of the platform team topology by utilizing common tools, services, APIs, and tasks.

Style Characteristics

A one-star rating in the characteristics ratings table in [Figure 14-8](#) means the specific architecture characteristic isn't well supported in the architecture, whereas a five-star rating means the architecture characteristic is one of the strongest features in the architectural style. The definition for each characteristic identified in the scorecard can be found in [Chapter 4](#).

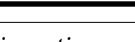
| | Architectural characteristic | Star rating |
|-------------|------------------------------|---|
| | Overall cost | \$\$ |
| Structural | Partitioning type | Domain |
| | Number of quanta | 1 to many |
| | Simplicity |  |
| | Modularity |  |
| Engineering | Maintainability |  |
| | Testability |  |
| | Deployability |  |
| | Evolvability |  |
| Operational | Responsiveness |  |
| | Scalability |  |
| | Elasticity |  |
| | Fault tolerance |  |

Figure 14-8. Service-based architecture characteristics ratings

Service-based architecture is a *domain-partitioned* architecture, meaning that its structure is driven by domains rather than technical considerations (such as presentation logic or persistence logic).

Consider the example of the Going Green electronics recycling application, which we introduced in [Chapter 7](#) and revisited in [Chapter 13](#). For the purposes of this chapter, let's imagine that Going Green uses a service-based architecture style. Each service, being a separately deployed unit of software, is scoped to a specific domain (such as item assessment). Changes made within this domain only impact that specific service and its corresponding UI and database. Nothing else needs to be modified to support a specific assessment change.

In a distributed architecture, the number of quanta can be greater than or equal to one. For example, if Going Green's services all share the same database or UI, the entire system would be only a single quantum. However, as discussed in “[Style Specifics](#)” on page 211, both the UI and database can be federated (broken apart), resulting in multiple quanta within the overall system. In [Figure 14-9](#), Going Green’s system contains two quanta. One is for the customer-facing portion of the application and contains a separate customer UI, database, and set of services (Quoting and Item Status). The other deals with the internal operations of receiving, assessing, and recycling electronic devices. Even though the internal operations quantum contains separately deployed services and two separate UIs, they all share the same database, making the internal operations portion of the application a single quantum.

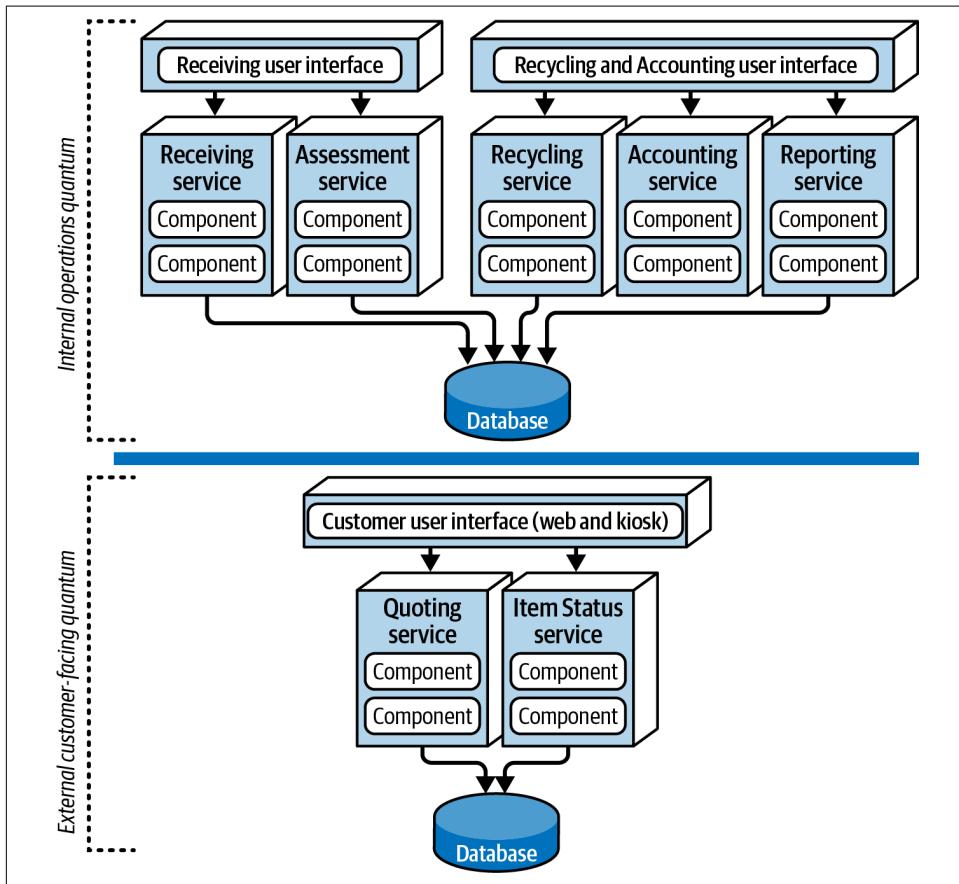


Figure 14-9. Separate quanta in a service-based architecture

Although we didn't give service-based architecture any five-star ratings, it nevertheless rates high (four stars) in many vital areas. Breaking an application apart into separately deployed domain services using this architectural style allows for faster change (agility), better test coverage due to modularity based on domain scoping (testability), and the ability to deploy more frequently with less risk than with a monolithic architecture (deployability). These three characteristics lead to better time to market, allowing organizations to deliver new features and fix bugs relatively quickly.

Fault tolerance and overall application availability also rate high for service-based architecture. Even though domain services tend to be coarse-grained, the four-star rating comes from the fact that services in this architectural style are usually self-contained and, thanks to code and database sharing, do not typically use interservice communication. As a result, if one domain service goes down (for instance, Going Green's Receiving service), it doesn't affect any of the other six services.

Scalability only rates three stars due to the coarse-grained nature of the services; correspondingly, elasticity rates only two stars. Although programmatic scalability and elasticity are certainly possible with this architectural style, it does replicate more functionality than architecture styles with finer-grained services (such as microservices), making it less cost-effective and less efficient in terms of machine resources. Typically, service-based architectures use only a single instance of each service, unless there is a need for better throughput or failover. A good example of this, illustrated in [Figure 14-9](#), is Going Green—only its Quoting and Item Status services need to scale to support high customer volumes. The other operational services only require single instances, making it easier to support things like single in-memory caching and database-connection pooling.

Simplicity and overall cost are two other drivers that differentiate this architectural style from other, more expensive and complex distributed architectures, such as microservices, event-driven architecture, or even space-based architecture. This makes service-based one of the easiest and most cost-effective distributed architectures to implement. While that's an attractive proposition, there is, as always, a trade-off. The higher the cost and complexity, the better these four-star characteristics (such as scalability, elasticity, and fault tolerance) become.

Its flexibility, combined with its many three-star and four-star architecture characteristics, make service-based architecture one of the most pragmatic styles available. While there are much more powerful distributed architectural styles, many companies find that such power comes at too steep a price. Others find that they simply don't *need* that much power. It's like buying a Ferrari but only using it to commute to work in rush-hour traffic—sure, it looks cool, but what a waste of power, speed, and agility!

Service-based architecture is also a natural fit for domain-driven design. Because services are coarse-grained and domain scoped, each domain fits nicely into a separately deployed service encompassing that particular domain. Compartmentalizing that functionality into a single unit of software makes it easier to apply changes to that domain.

Maintaining and coordinating database transactions is always an issue with distributed architectures, which typically rely on *eventual consistency* (meaning that independent database updates will eventually be in sync with each other) rather than traditional *ACID transactions* (meaning database updates are coordinated and performed together in a single unit of work). Service-based architecture leverages ACID transactions better than any other distributed architecture style because its domain services are coarse-grained. This means the transaction scope is set to a particular domain service, allowing for the traditional commit-and-rollback transaction functionality found in most monolithic applications.

Finally, service-based architecture is a good choice for architects who want to achieve a good level of modularity without getting tangled up in the complexities of granularity and service coordination (see “[Choreography and Orchestration](#)” on page 341 in [Chapter 18](#)).

Examples and Use Cases

To illustrate the flexibility and power of the service-based architectural style, we’ll revisit our example of Going Green, a system used to recycle old electronic devices (such as an iPhone or Galaxy cell phone).

Going Green’s processing flow works as follows:

1. The customer asks Going Green (via a website or kiosk) how much money it will pay for an old electronic device (*quoting*).
2. If satisfied with the quote, the customer sends the device to the recycling company (*receiving*).
3. Going Green assesses the device’s condition (*assessment*).
4. If the device is in good working condition, Going Green pays the customer for the device (*accounting*). During this process, the customer can go to the website at any time to check on the status of the item (*item status*).
5. Based on the assessment, Going Green either safely destroys the device and recycles its parts, or resells it on a third-party sales platform, such as Facebook Marketplace or eBay (*recycling*).
6. Going Green periodically runs financial and operational reports on its recycling activity (*reporting*).

Figure 14-10 illustrates this system as implemented using a service-based architecture. Each of the domain areas we just identified is implemented as a separately deployed, independent domain service. The services that need to scale (and hence would require multiple service instances) are those needing higher throughput (in this case, the customer-facing Quoting and ItemStatus services). Since the other services don't need to scale, they only require a single service instance.

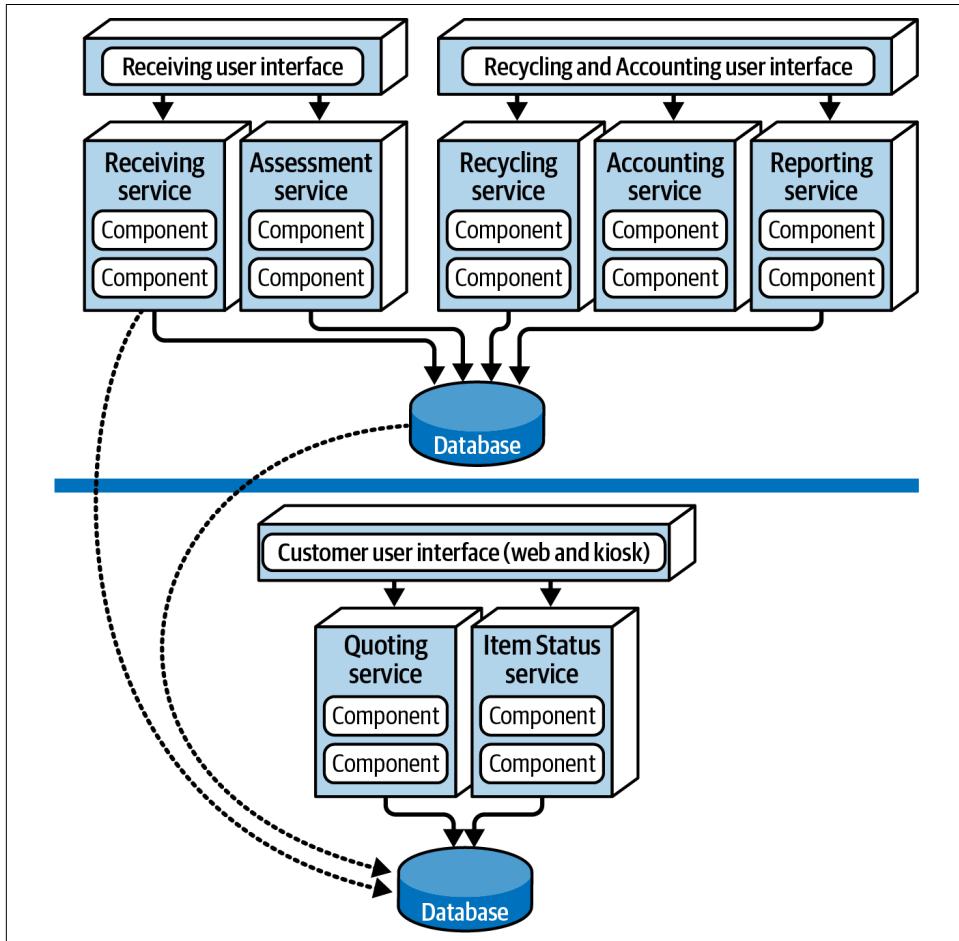


Figure 14-10. Going Green's electronics recycling application using service-based architecture

In this example, the UI applications are separated into domains: *Customer Facing*, *Receiving*, and *Recycling and Accounting*. This separation provides good fault tolerance at the UI level, good scalability, and appropriate security (since external customers have no network path to access internal functionality). Notice, too, that there

are two separate physical databases: one for external customer-facing operations, and one for internal operations. This arrangement allows internal data and operations to reside in a separate network zone from the external operations (denoted by the horizontal line). It also provides much better security-access restrictions and data protection—and constitutes a separate architectural quantum). One-way access through the firewall allows internal services to access and update customer-facing information, but not vice versa. Alternatively, depending on the database, teams could also use internal table mirroring and external table synchronization to synchronize the data between the two databases.

Additionally, the `Assessment` service changes constantly, as new products are received or come on the market. With a service-based architecture, these frequent changes are isolated to a single domain service, providing agility, testability, and deployability.

Service-based architecture is a very flexible architectural style that offers domain partitioning and good levels of scalability, agility, fault tolerance, availability, and responsiveness at a fairly low price, compared to other distributed architectures. These factors make it a popular choice.

Service-based architectures also make good “stepping stone” migration targets for other distributed architectures, whether the organization is migrating to another distributed architecture style or creating a new distributed system from scratch. This drives us to our main point:

Not every portion of an application needs to be microservices.

—Mark Richards

Initially moving to or creating a service-based architecture as a “stepping stone” *before* migrating to the target architectural style allows teams to analyze the domains and decide which portions of the architecture *should* be microservices. For example, in the Going Green example, the `Recycling` and `Accounting` services don’t need to be broken down any further and should probably remain domain services. However, the `Assessment` service changes frequently and requires high levels of agility, so *that* service should be broken down into separate services, one for each type of electronic device. If the Going Green team skipped this step and moved *straight* to a microservices architecture, *every* piece of functionality would likely end up as a microservice, even if it didn’t need to be one.

These are a few of the many reasons that service-based architectures are a favorite among architects. However, it’s only one of many distributed architectural styles, and understanding all of them is helpful for determining the right fit for a particular business problem. To that end, let’s look at some other distributed architectures.

Event-Driven Architecture Style

The *event-driven* architecture (EDA) style is a popular distributed, asynchronous architecture style used to produce highly scalable and high-performance applications. It is also particularly adaptable and can be used for small applications as well as large, complex ones. Event-driven architecture is made up of decoupled event processing components that asynchronously trigger and respond to events. It can be used as a standalone architecture style or embedded within other architecture styles (such as an event-driven microservices architecture).

Many developers and software architects consider EDA more of an architectural pattern than an architectural style. We disagree. Your authors have developed entire systems that rely solely on EDA, which is why we maintain that it's primarily an architectural style. While EDA can be used in other architectural styles—such as microservices and space-based architecture—to form hybrid architectures, it remains at its core a way of designing complex systems.

Most applications follow what is called a *request-based* model, as illustrated in [Figure 15-1](#). When a customer requests, for example, their order history for the past six months, this request is initially received by a *request orchestrator*. The request orchestrator is typically a user interface, but it can also be implemented through an API layer, orchestration services, event hubs, an event bus, or an integration hub. Its role is to direct the request to various *request processors*, deterministically and synchronously. They process the request by retrieving and updating the customer's information in a database. Retrieving order history information is a data-driven, deterministic request made to the system within a specific context, not an event happening that the system must react to, which is why this is a request-based model.

An *event-based* model, on the other hand, reacts to a particular event by taking action. For example, take submitting an online auction bid for a particular item. The bidder submitting the bid is not making a request to the system, but rather initiating

an event that happens after the current asking price is announced. The system must respond to that event by comparing the bid to others received at the same time and determining the current highest bidder.

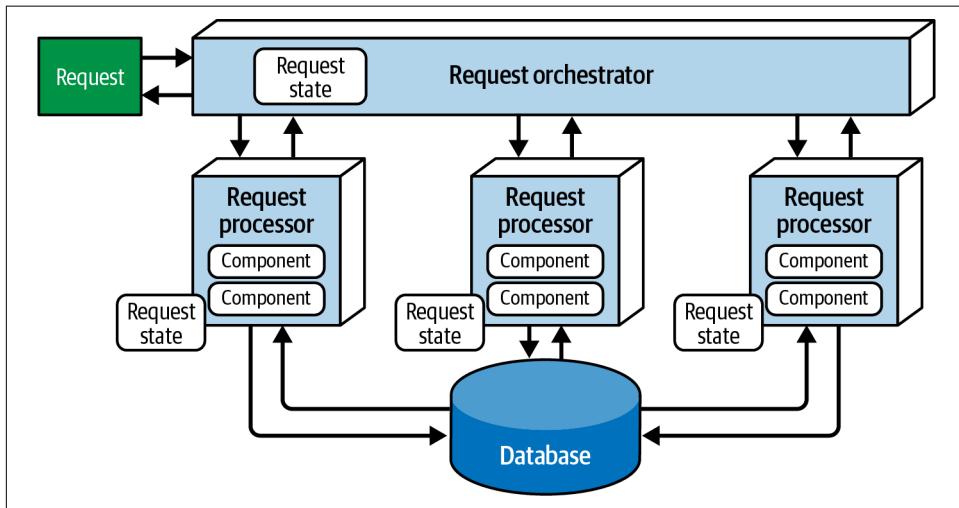


Figure 15-1. Request-based model

Topology

Event-driven architecture leverages asynchronous *fire-and-forget* communication, where services trigger events and other services respond to those events. The four primary architectural components of its topology are an *initiating event*, an *event broker*, an *event processor* (usually just called a *service*), and a *derived event*.

The *initiating event* is the event that starts the entire event flow. This could be a simple event, like placing a bid in an online auction, or more complex events, like making updates to a health-benefits system when an employee gets married. The initiating event is sent to an event channel in the *event broker* for processing. A single *event processor* accepts the initiating event from the event broker and begins processing that event.

The event processor that accepted the initiating event performs a specific task associated with processing that event (such as placing a bid for an auction item), then asynchronously advertises what it did to the rest of the system by triggering what is called a *derived event* to an *event broker*. Other event processors respond to the derived event, perform specific processing based on it, then advertise what they did through new derived events. This process continues until all event processors are idle and all derived events have been processed. [Figure 15-2](#) illustrates this event-processing flow.

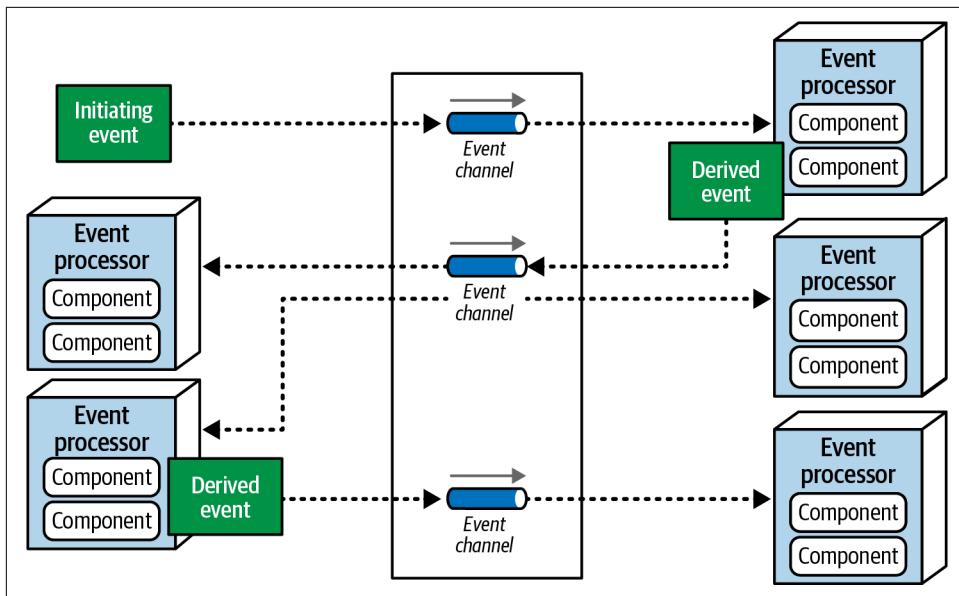


Figure 15-2. Basic topology of an event-driven architecture

The event-broker component is usually *federated* (meaning it has multiple domain-based clustered instances). Each federated broker contains all of the *event channels* (such as queues and topics) used within the *event flow* (the entire workflow for processing the event) for that particular domain. Because of the decoupled, asynchronous, fire-and-forget broadcast nature of this architectural style, the broker topology uses topics, topic exchanges (in the case of the [Advanced Message Queuing Protocol \(AMQP\)](#)), or streams with a publish-and-subscribe messaging model.

To illustrate how EDA processing works overall, consider the workflow of a typical retail-order entry system, as illustrated in [Figure 15-3](#), where customers can place orders for items (say, a book like this one). In this example, the `Order Placement` event processor receives the initiating event (`place order`), inserts the order in a database table, and returns an order ID to the customer. It then advertises to the rest of the system that it created an order through an `order placed` derived event. Notice that three event processors are interested in that derived event: the `Notification` event processor, the `Payment` event processor, and the `Inventory` event processor, all of which perform their tasks in parallel.

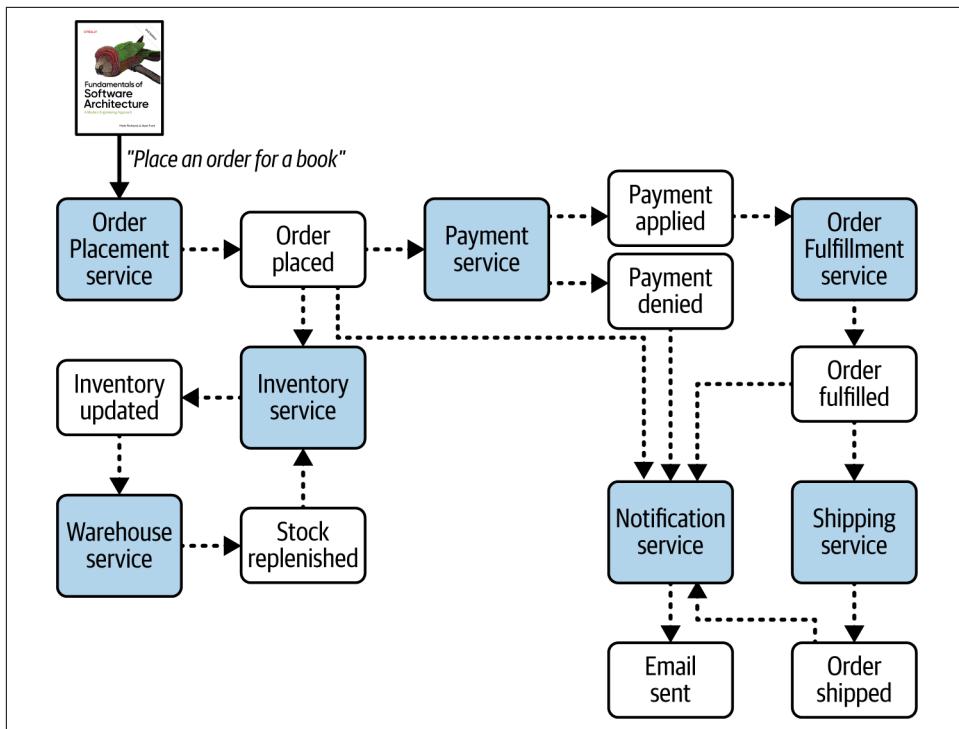


Figure 15-3. Example of the event-driven architecture topology

The **Notification** event processor receives the **order placed** derived event and emails the customer with the order details. Because the **Notification** event processor has performed an action, it then generates an **email sent** derived event. However, notice in Figure 15-3 that no other event processors are listening to that derived event. This is typical within EDA and illustrates this style's *architectural extensibility*—the ability for future event processors to respond to the derived event without any changes to the existing system (see “[Triggering Extensible Events](#)” on page 235).

The **Inventory** event processor also listens for the **order placed** derived event and adjusts the corresponding inventory for that book. It then advertises this action by triggering an **inventory updated** derived event, which in turn triggers a response from the **Warehouse** event processor. This processor responds to and manages the corresponding inventory between warehouses, reordering items if supplies get too low. When stock is replenished, the **Warehouse** event processor triggers a **stock replenished** derived event, to which the **Inventory** event processor responds by adjusting the current inventory.

In this case, the `Inventory` event processor would not trigger a corresponding `inventory adjusted` event. If it did, that would lead to what is called a *poison event*—an event that keeps looping and looping forever.



A *poison event* occurs when a derived event keeps getting triggered and responded to in a continuous loop between services. These can happen frequently when using an event-driven architecture, so be careful to avoid them.

The `Payment` event processor also responds to the `order placed` derived event. It responds by charging the customer's credit card. [Figure 15-3](#) shows that one of two possible derived events will be generated as a result of the `Payment` event processor's action: one to notify the rest of the system that the payment was applied (`payment applied`), and one to notify the system that the payment was denied (`payment denied`). The `Notification` event processor is interested in the `payment denied` derived event, because if this happens, it needs to email the customer informing them that they must update their credit card information or choose a different payment method.

The `Order Fulfillment` event processor listens for the `payment applied` derived event and performs various automated functions within order picking and packing, such as instructing the worker where to find the item and what size box is needed for the order. Once that's completed, it triggers an `order fulfilled` derived event telling the rest of the system that it has fulfilled the order. Both the `Notification` and the `Shipping` event processors listen for this derived event. Concurrently, the `Notification` event processor notifies the customer that the order has been fulfilled and is ready for shipment, and the `Shipping` event processor selects a shipping method, ships the order, and sends out an `order shipped` derived event. The `Notification` event processor also listens for the `order shipped` derived event and notifies the customer that the order is on its way.

All of the event processors are highly decoupled and independent of each other. One way to understand this asynchronous processing workflow is to think about it as a relay race. In a relay race, runners hold a baton (a wooden stick) and run for a certain distance (say, 1.5 kilometers), then hand off the baton to the next runner, who does the same, on down the chain until the last runner crosses the finish line. Once a runner hands off the baton, that runner is done with the race and can move on to other things. This is also true with event processors: once an event processor hands off an event, it is no longer involved with processing that specific event and is available to react to other initiating or derived events. In addition, each event processor can scale independently to handle varying load conditions or backups.

Style Specifics

The following sections describe EDA in more detail, including some considerations, patterns, and hybrids of this complex architectural style.

Events Versus Messages

Event-driven architecture leverages events to pass and process information. But is an *event* really that different from a *message*? Turns out, yes, it really is.

An *event* broadcasts to other event processors that something has already happened: “I just placed an order.” A *message*, however, is more of a command or query, such as “apply the payment for this order” or “give the shipping options for this order.” This is not a subtle difference. When we talk about *event processing*, we mean *reacting* to something that has already happened, whereas a message describes something that *needs to be done*. In our example, it’s clear that “I just placed an order” is an event because it does not describe what processing needs to occur, as a message would. This illustrates EDA’s decoupled nature.

The second main difference between an event and a message is that an event typically does not require a response from the receiver, whereas a message usually does. This reduces back-and-forth communication between event processors, further decoupling them from one another.

Another major difference between events and messages is that an event is typically broadcast to multiple event processors, whereas a message is almost always directed to one event processor. In our simple order-processing example, several event processors are interested in and respond to the `order created` event, whereas only one event processor responds to the message `apply payment`. Events typically use a *publish and subscribe* (one-to-many) form of communication, whereas messages typically use a *point-to-point* (one-to-one) form of communication.

A final difference between events and messages is the physical artifact that represents the communication channel. Events use a *topic*, *stream*, or notification service so that multiple event processors can subscribe to the channel and listen for the event. Messaging typically uses a *queue* or *messaging service* to guarantee that only one type of event processor will receive that message.

Event-driven architecture mostly uses *events* (hence its name), but it can use messages on occasion, such as requesting data from another event processor (see “[Data Topologies](#)” on page 268 and “[Request-Reply Processing](#)” on page 256). Later in this chapter, we’ll show you *mediated event-driven architecture* (see “[Mediated Event-Driven Architecture](#)” on page 258), which uses messages to control the processing order of events.

Can you pick out which of the following choices are events and which are messages?

- “Adventurous Air Flight 6557, turn left, heading 230 degrees.”
- “In other news, a cold front has moved into the area.”
- “OK, class, turn to page 145 in your workbooks.”
- “Hi, everyone! Sorry I’m late for the meeting.”

Let’s look at these one by one:

“Adventurous Air Flight 6557, turn left, heading 230 degrees.”

This is a *message* because it’s a command (something that needs to be done) and because it’s directed toward one target, the pilot, even though several other pilots may hear the message.

“In other news, a cold front has moved into the area.”

This is an *event*: it’s being broadcast to multiple people, it describes something that has already happened, and the news broadcaster is not expecting a reply. (There are, however, some messages that don’t require a reply either.)

“OK, class, turn to page 145 in your workbooks.”

This one is a little tricky. Turns out this is a *message*, even though it’s being broadcast to multiple students. It’s a *command* to do something, not something that has already happened (which would make it an event). This illustrates an important point about the difference between an event and a message: broadcasting a command (such as turning to page 145 in the workbooks) through a publish-and-subscribe channel doesn’t turn it into an *event*.

“Hi, everyone! Sorry I’m late for the meeting.”

This is an *event*, because this person being late for the meeting has already happened. It’s also being broadcast to multiple people, and no response is expected.

Derived Events

Derived events are a critical and necessary part of EDA. They are created and triggered by event processors *after* the initiating event is received. An event processor can trigger more than one derived event, based on its processing.

Consider the derived events triggered when the `Payment` event processor charges a customer’s credit card for a purchase in [Figure 15-3](#). As illustrated in [Figure 15-4](#), charging a credit card involves checking for possible fraud (processed by the `Fraud Detection` event processor) and checking the credit card balance (processed by the `Credit Limit` event processor). EDA can leverage a single event (`creditcard charged`) to do both activities at the same time.

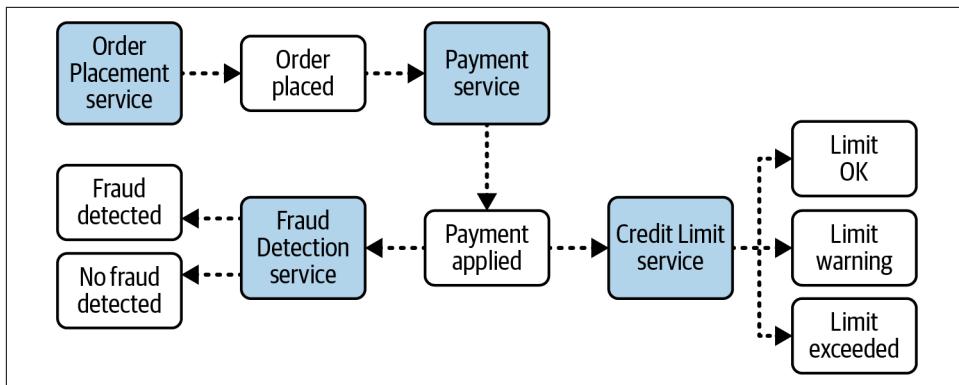


Figure 15-4. Derived events are generated in response to the initiating event

Notice how many derived events this single action generates. In the **Fraud Detection** event processor, it triggers two possible derived events: one if the processor detects fraud, another if no fraud is detected. Both of these derived events are necessary, because different event processors might take further actions depending on the outcome of the fraud detection.

Now look at the derived events from the **Credit Limit** event processor. First, a **limit okay** derived event indicates to the rest of the system there is no risk for this purchase and that the customer has plenty of credit available. As a matter of fact, this particular event could also store in its event payload the amount of credit the customer has left, which might be of interest to downstream event processors. Second, the **limit warning** derived event, warning that the card's balance is close to the credit limit, might be of interest to other downstream event processors—for instance, the **Notification** event processor, which could notify the customer that they are close to their credit limit. Last, the fatal **limit exceeded** derived event is of interest to several event processors, including **Notification**, **Decline Purchase**, and perhaps a marketing-based **Extend Credit Limit** event processor that automatically extends the customer's credit limit to allow the purchase.

This illustrates that more than one derived event can be triggered from an event processor. However, be careful not to get trapped in the *Swarm of Gnats* antipattern, where a processing unit sends out too many fine-grained events (see “[The Swarm of Gnats Antipattern](#)” on page 246).

Triggering Extensible Events

In EDA, it's usually a good practice for each event processor to advertise what it has done to the rest of the system, regardless of whether or not any other event processor cares about what that action was.

When no event processors care about or respond to the event, we call it an *extensible derived event* because it nevertheless provides support for *architectural extensibility* by providing a built-in “hook” in case processing that event requires additional functionality. For example, suppose that, as part of a complex event process (as illustrated in [Figure 15-5](#)), the **Notification** event processor generates an email that it sends to a customer, notifying them of a particular action. It then advertises that it has sent the email to the rest of the system through a new derived event (**email sent**). Since no other event processors are currently listening or responding to that event, the message simply disappears (or is ignored, in the case of event streaming). That might seem like a waste of resources, but it's not. Suppose the business decides to analyze all emails sent to customers. The team can add a new **Email Analyzer** event processor to the overall system with minimal effort and with no changes to other event processors, because the email information is already available via the **email sent** derived event.

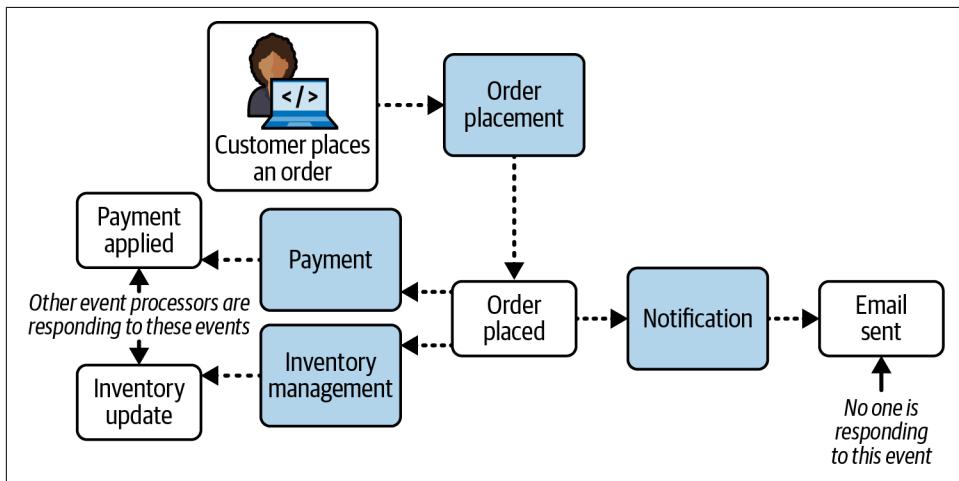


Figure 15-5. Notification event is sent, but ignored and not used

Asynchronous Capabilities

The event-driven architectural style has a unique characteristic in that it relies primarily on asynchronous communication for both *fire-and-forget* processing (no response required) and *request/reply* processing (when a response is required from the event consumer, see “[Request-Reply Processing](#)” on page 256). Asynchronous

communication can be a powerful technique for increasing a system's overall responsiveness.

In [Figure 15-6](#), a user is posting a product review on a website. The comment service, in this example, takes 3,000 milliseconds to validate and post that comment. The comment has to go through several parsing engines: a check for unacceptable words, a check to make sure that the comment is not indicating abusive text (such as “slow thinker” or “unable to think clearly”), and finally a context check to make sure the comment is about the product (and not, say, just a political rant).

The top path in [Figure 15-6](#) posts the comment using a synchronous RESTful call. That means 50 ms in network latency for the service to receive the post, 3,000 ms to validate and post the comment, and 50 ms of latency to tell the user that the comment was posted. The total time to post a comment, from the user’s point of view, is thus 3,100 milliseconds. Now look at the bottom path, which uses asynchronous messaging. Here, the user’s total posting time is only 25 ms, as opposed to 3,100 ms. It still takes the system 25 ms to receive the comment and 3,000 ms to post it, for a total of 3,025 ms, but from the end user’s perspective, only 25 ms go by before the system responds that it has accepted the comment (although it hasn’t actually been posted yet).

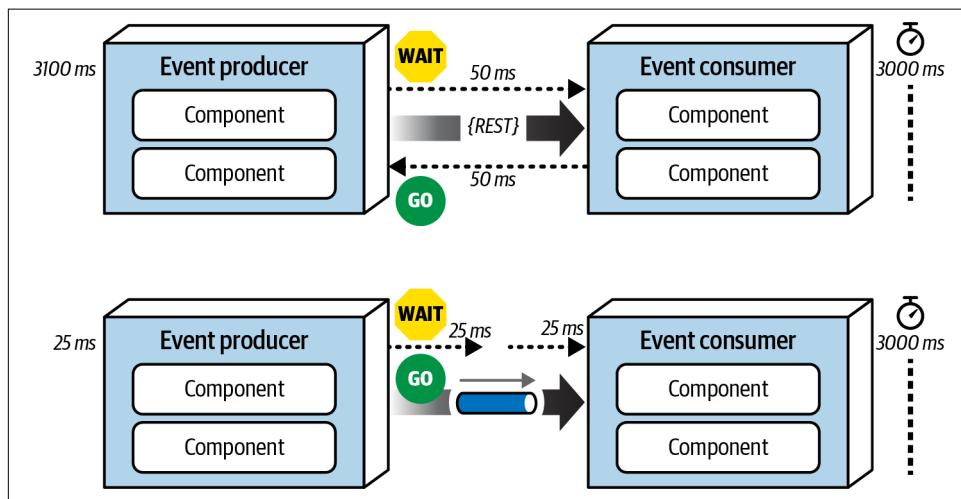


Figure 15-6. Synchronous versus asynchronous communication

The difference in response time between 3,100 ms and 25 ms is staggering. There is, however, one caveat: on the synchronous top path, the end user gets a *guarantee* that their comment has been posted. However, on the asynchronous bottom path, the post is only acknowledged, with a future promise that *eventually* it will be posted. What would happen if the user’s comment includes profanity and the system rejects it? There is no way for it to notify the end user—or is there? If the user must register

with the website to post a comment, the system could send them a message indicating a problem with the comment and suggesting how to repair it.

This example illustrates the difference between *responsiveness* (the time it takes to get information back to the user) and *performance* (the time it takes to insert the comment into the database). When the user doesn't need any information back (other than an acknowledgment or a thank-you message), why make them wait? Responsiveness is all about notifying the user that the action has been accepted and will be processed momentarily, whereas performance is about making the end-to-end process faster. On the asynchronous bottom path, the architect did nothing to optimize the way the comment service processes the comment (that's addressing *responsiveness*). If the architect took the time to optimize the comment service, perhaps by executing all of the text and grammar parsing engines in parallel, leveraging caching and other similar techniques but still using synchronous communication, they would be instead addressing overall *performance*.

That was a simple example. What about a more complicated one? This time, let's look at an online *stock trade*, where a user is purchasing some stock asynchronously. What if there is no way to notify the user of an error?

While asynchronous communication significantly improves responsiveness, error handling is a big issue. The difficulty of addressing error conditions adds to the complexity of this architecture style. “[Error Handling](#)” on page 249 illustrates a pattern of reactive architecture for addressing error-handling challenges: the *Workflow Event* pattern.

In addition to its responsiveness, asynchronous communication also provides a good level of dynamic decoupling and avoids an antipattern, *Dynamic Quantum Entanglement*, which occurs when two architectural quanta communicate through synchronous communication. (You probably recall from [Chapter 7](#) that an *architectural quantum* is a part of the system that can be deployed independently from the rest of the system and is bound through synchronous dynamic coupling, and that architectural characteristics live at the quantum level.) Because these two architectural quanta are now dependent on each other, they essentially become *entangled*. The dependency turns them into a single architectural quantum. Asynchronous communication can help detangle architectural quanta because it removes that dynamic dependency.

To illustrate this important point, consider the two systems in [Figure 15-7](#). In this example, the **Portfolio Management** system creates a trade order to buy some stock. It synchronously sends this trade order to the **Trade Order** system, which will perform compliance checks and create the trade order. Because the communication between these two systems is synchronous, the **Portfolio Management** system must necessarily block and wait for a trade confirmation number from the **Trade Order** system. These two systems have become entangled and now form a single architectural quantum.

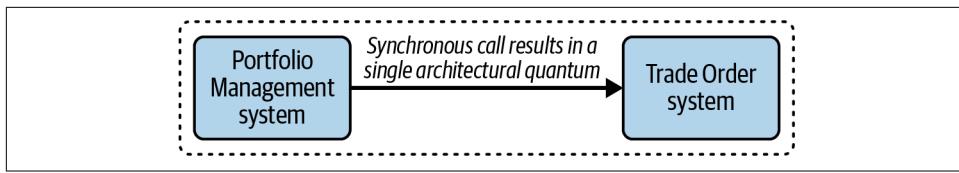


Figure 15-7. These systems form a single architectural quantum due to synchronous dynamic coupling

The significance of this entanglement is that the architectural characteristics now live *between* these two systems. If the **Trade Order** system becomes unavailable or unresponsive, the **Portfolio Management** system cannot submit the trade order. This also degrades responsiveness: if the **Trade Order** system is slow, the **Portfolio Management** system will be slow too. Scalability also suffers because if the **Portfolio Management** system needs to scale, so does the **Trade Order** system. If it doesn't or can't, then the **Portfolio Management** system can't scale as needed.

An architect can detangle these architectural quanta by replacing the synchronous call with an asynchronous call between the two systems, as illustrated in [Figure 15-8](#).

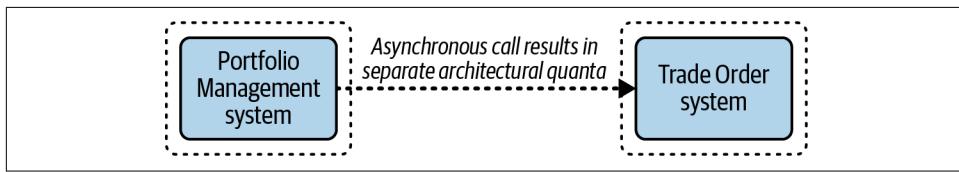


Figure 15-8. These systems form separate architectural quanta due to their asynchronous dynamic coupling

By using asynchronous communication, the **Portfolio Management** system can send the trade order through a queue or some other asynchronous means, so it doesn't have to wait for the **Trade Order** system to create the trade order. Once the **Trade Order** system performs its compliance checks and creates the trade order, it can send the confirmation number to the **Portfolio Management** system through a separate, asynchronous channel. Removing this dependency from the two systems' dynamic coupling allows them to form two separate architectural quanta. If the **Trade Order** system is unavailable or unresponsive, the **Portfolio Management** system can still issue trade orders, knowing that at some point they will be created and confirmation numbers sent back.

Broadcast Capabilities

Another of EDA's unique characteristics is its ability to broadcast events without knowing what other processing units (if any) are receiving those events or what processing they will perform in response. As [Figure 15-9](#) shows, this dynamically decouples the event processors from each other.

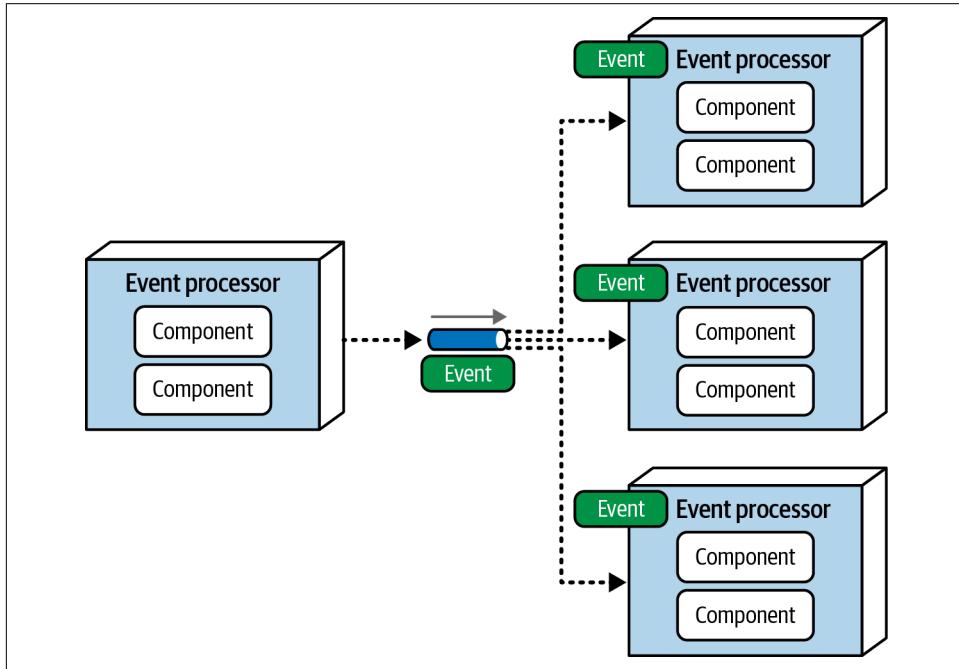


Figure 15-9. Broadcasting events to other event processors

Broadcast capabilities are an essential part of many patterns, including eventual consistency and complex event processing (CEP). For example, prices for instruments traded on the stock market change frequently. Every time a new ticker price (the current price of a particular stock) is published, many event processors might respond to the new price (such as trade analytics, or buying or selling the stock). However, the event processor that publishes the latest price simply broadcasts it, with no knowledge of how that information will be used. This is known as *semantic decoupling* in that one event processor has no knowledge of (or dependency on) the actions of other event processors.

Event Payload

The information contained in an event is known as its *payload*. Payloads can vary significantly: an event payload could be a simple key-value pair, or all the information necessary for downstream processing. The two basic types are *data-based* and a *key-based* event payloads. Architects must do careful trade-off analysis to determine which of these options suits each type of event triggered in the system. In this section, we describe these two payload types and their corresponding trade-offs.

Data-based event payloads

A *data-based event payload* is an event payload that sends all necessary information for processing. In the example illustrated in [Figure 15-10](#), a customer places an order. First, the Order Placement event processor inserts the complete order into the database (the system of record). It then broadcasts an event called `order_placed`, which contains all of the order details (in this case, 45 attributes, totalling 500 KB of memory). The Payment event processor responds to this event by pulling information from the event payload—specifically the order ID, the customer's information, and the total cost of the order—and using it to apply the payment. Simultaneously, the Inventory Management event processor uses the item ID and item quantity from the event payload to adjust the current inventory for the item purchased.

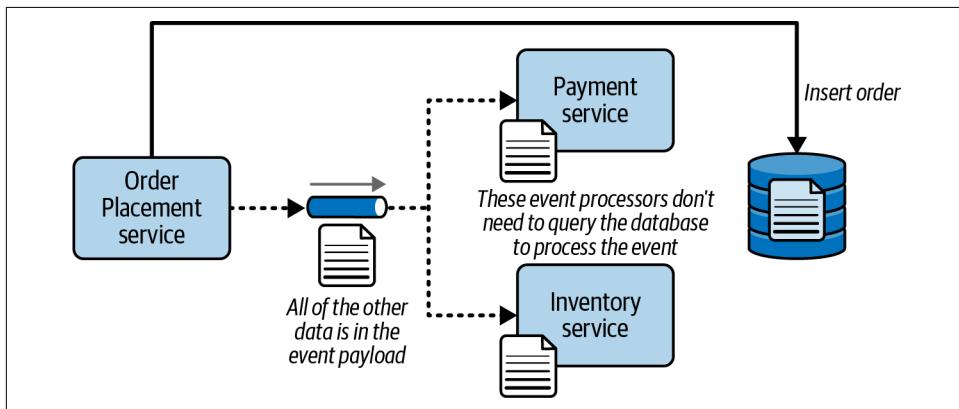


Figure 15-10. Data-based event payloads contain all the necessary data for processing

The Payment and Inventory Management event processors didn't have to query the database to get the order information, because the data was already contained in the event payload. This is one of the biggest advantages of using data-based event payloads. The less an event processor queries the database, the better its performance, responsiveness, and scalability will be. Furthermore, given the highly dynamic decoupled nature of EDA, the Order Placement event processor might not know which other event processors are responding to the event or what data they might need.

for processing. Sending all of the information in the event payload guarantees that each responding event processor will have the information it needs to perform its processing. Event processors might not even have access to the database that contains the order information, particularly in data topologies with strict bounded contexts, domain based or database-per-service (see “[Data Topologies](#)” on page 268).

While these advantages demonstrate ways of creating more responsive, scalable, and flexible systems, data-based payloads do have several disadvantages. The first is that it’s harder to maintain data consistency and data integrity when you have *multiple systems of record*. Because all of the order information is contained in the database *and* in the events being triggered in the system, the order information can easily get out of sync—particularly if the order is updated during processing.

For example, suppose a customer places an order for a hundred items, but only meant to order one, and realizes their mistake immediately after submitting the order. Or perhaps the customer realizes just after ordering that they’ve used the incorrect shipping address (this has happened to your authors many times). In either of these situations, the customer immediately updates the order with the correct information. The database, which is the single system of record, contains the corrected values, but some of the events containing the older values might not be processed immediately. This means that any of the older, incorrect values still being processed will be used instead of the newer, correct ones. To further complicate this scenario, it is very difficult in EDA to control the timing of events, so it’s possible that the newer values might be processed *before* the older values. This means that if other event processors use the older, incorrect values, those could be overlaid on the correct, newer values.

A second major disadvantage of the data-based event payload is about contract management and versioning. We know that an order in this system has 45 attributes. Because all of that information is contained in the event payload, the event needs some sort of *contract*—a way of structuring the data being sent. The architect is now faced with myriad decisions: should the payload type be a JSON object? An XML object? Should the contract be strict or loose? (A *strict* contract is one that uses some sort of schema or object definition, such as a JSON schema, GraphQL spec, or class definition; whereas a *loose* contract might use simple JSON name-value pairs.) Each of these decisions carries with it many trade-offs, and each forms a tight static coupling between event processors.

And then there’s versioning. For strict contracts, an architect or developer might use a [vendor MIME type](#) in the event headers to specify the version number. This helps make the system more agile and provides backward compatibility (to avoid breaking other event processors). However, all event processors must leverage the same versioning logic, which requires strong governance. If an event processor ignores a contract version when responding to a strict contract’s payload, changing

that schema is likely to cause that event processor to fail. In addition, it's very difficult to implement version communication and deprecation strategies in a highly decoupled, asynchronous architecture like EDA. All this makes data-based event payloads somewhat fragile.

Data-based event payloads can also suffer from *stamp coupling*: a form of static coupling where several modules (in this case, event processors) all share a common data structure, but only use parts of it (and in many cases different parts of it). When this situation occurs, changing the common data structure can require changing other event processors, even ones that don't care about the data.

Figure 15-11 illustrates how stamp coupling works and its negative impact on the architecture. In this example, the Order Placement event processor sends an order_placed event consisting of 45 attributes, containing all the information about the order, at a size of 500 KB. The Inventory event processor responds to the order_created event, but only requires two attributes, the item_id and the quantity, totaling only 30 bytes. In this example, changing to the payload, for example by removing an address-line attribute, would affect the Inventory event processor even though it doesn't care about that field.

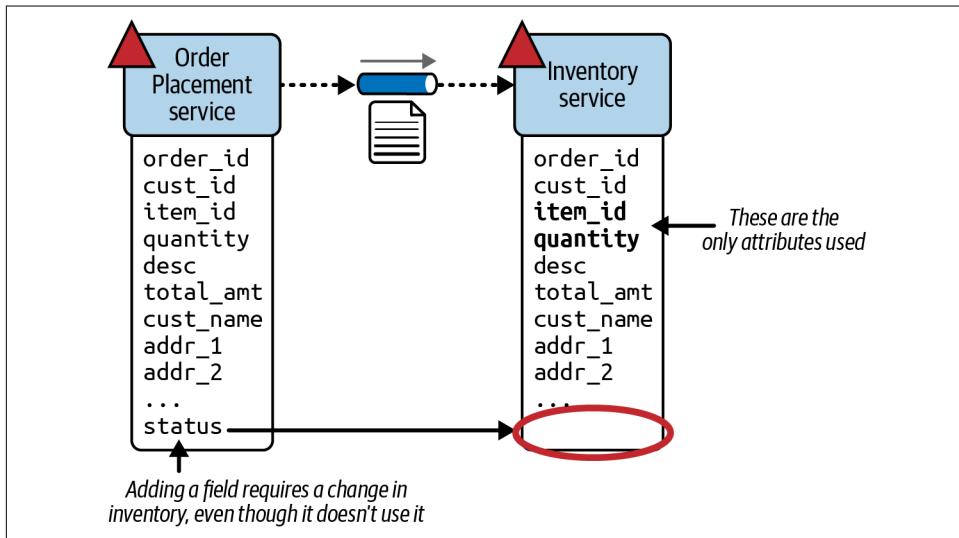


Figure 15-11. An example of stamp coupling, where another service only needs part of the data sent

In this example, using contract versioning with strict contracts helps mitigate the risk of the Inventory event processor breaking, but at some point—when the contract version is deprecated or a breaking contract change occurs—a developer will have to retest and redeploy it.

One commonly overlooked problem with stamp coupling is *bandwidth utilization*. The third fallacy of **distributed computing** is that “bandwidth is infinite.” It’s not, of course. As a matter of fact, in most cloud-based environments, bandwidth is what costs so much. Going back to our example in [Figure 15-11](#), with a data-based payload, if customers are placing 500 orders per second, sending a single 500-KB event to the **Inventory** event processor will utilize 250,000 KB per second of bandwidth. However, if the system sends only the 30 bytes of data actually *needed*, the event will only utilize 15 KB per second of bandwidth. This is a staggering difference, one worth investigating when using data-based event payloads.

One reason architects sometimes limit stamp coupling is to leverage **consumer-driven contracts**, where each consumer of a message has its own contract containing only the data it needs for processing. However, because of EDA’s broadcast capabilities and because the system can’t always know which event processors will respond to an event, it’s difficult to leverage consumer-driven contracts with events in event-driven architecture. For this reason (and to address the other disadvantages of data-based event payloads), many architects turn to key-based event payloads.

Key-based event payload

A *key-based event payload* is an event payload that contains only a key identifying the context for the event (such as an order ID or customer ID). With key-based event payloads, the event processors responding to the event must query a database to retrieve the information they need to process the event.

When a customer places an order, the `Order Placement` event processor inserts the order into the database and triggers a key-based event called `order_placed`. This event contains a single key value containing the order ID in simple JSON:

```
{  
  "order_id": "123"  
}
```

One of the main disadvantages of key-based event payloads is that each event processor responding to the event must query the database to get the information it needs to process the order. For instance, when the `Payment` event processor responds to the event, it must query the database for the order information it needs to process payment. The `Inventory` event processor also responds simultaneously to the event, so it must also query the database to retrieve the item ID and quantity. This can detract from responsiveness, performance, and scalability, and can overwhelm a database, particularly in a highly parallel and asynchronous architecture such as event-driven architecture. (See [“Data Topologies” on page 268](#) for ways to mitigate this risk.) Key-based event payloads also present a challenge if the required data is not easily accessible (such as if it lies within the bounded context of another event processor). [Figure 15-12](#) illustrates this technique.

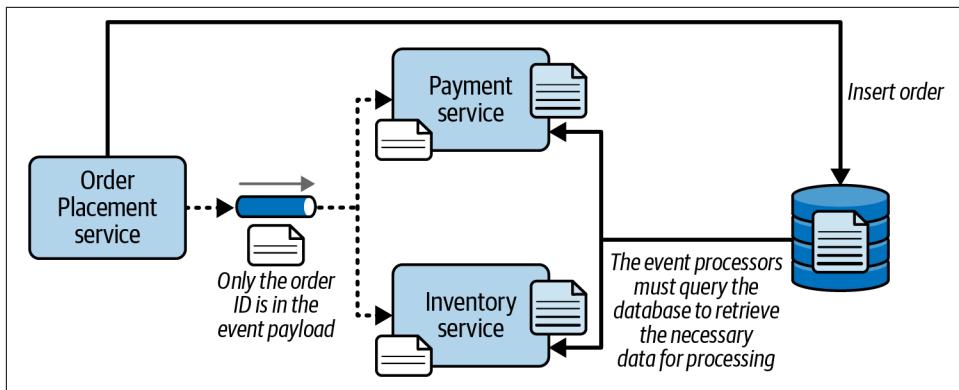


Figure 15-12. With key-based event payloads, only the context key is contained in the event

However, using key-based event payloads carries many advantages, some of which may outweigh the performance and scalability issues. The first main advantage is better overall data consistency and data integrity, thanks to having a *single system of record*. Because data about the event is located in only one place (the database), key-based event payloads can handle changes to the data during event processing much more easily than data-based event payloads.

The second main advantage is that because the contract in key-based event payloads is so simple and rarely changes, architects typically implement it through loose, schema-less JSON or XML. Therefore, key-based event payloads don't have the same issues with contract change management, versioning, and communication and depreciation strategies that data-based event payloads tend to have.

Another advantage of key-based event payloads: they don't have the same stamp coupling and bandwidth issues as data-based event payloads. Because there's no opaque data associated with the event, contracts are simple, small, and utilize minimal bandwidth. Thus, they tend to perform faster, from a network and message broker perspective, than data-based event payloads.

Trade-off summary

Choosing between a data-based event payload versus a key-based data payload requires careful trade-off analysis. Remember, it's not an all-or-nothing proposition: each type of event can use a different payload type. **Table 15-1** summarizes the trade-offs associated with data-based and key-based event payloads.

Table 15-1. Data-based versus key-based event payloads

| Criteria | Data-based payloads | Key-based payloads |
|-----------------------------|---------------------|--------------------|
| Performance and scalability | Good | Bad |
| Contract management | Bad | Good |
| Stamp coupling | Bad | Good |
| Bandwidth utilization | Bad | Good |
| Restricted database access | Good | Bad |
| Overall system fragility | Bad | Good |

Notice that the overall trade-off between these two options boils down to scalability and performance versus contract management and bandwidth utilization. Ask which one is more important for each particular event. Some event processing requires extreme levels of scale and performance, in which case a data-based event payload would be a better choice; some event processing data will undergo frequent change, in which case a key-based event payload might be more appropriate.

As with most things in software architecture, architects' choices lie on a spectrum, not a simple binary. That's why it's important to be careful to avoid triggering what are known as anemic events.

Anemic events

An *anemic event* is a derived event with a payload that doesn't contain enough information to help the event processor make decisions, and lacks the necessary context for further downstream processing.

Figure 15-13 illustrates an anemic derived event. In this example, a customer has updated some information in their user profile. Once that information is updated in the database, the `Customer Profile` event processor triggers a `profile_updated` event, using a key-based event payload that passes only the customer ID as its key-based data.

The three services responding to this event receive only the customer ID and the context that the customer's profile was changed. The first service (`Service 1`) has no idea what data was changed in the profile: name, address, some other critical information? Unfortunately, querying the database cannot answer this question, so `Service 1` has no idea how to respond or what action to take. `Service 2` responds to the `profile_updated` event, but going only by the key, does not know if it needs to perform any additional processing. Finally, `Service 3` responds to the same event, but has no idea what the prior values were and therefore cannot perform its processing. All three of these event processors need to respond in some way to the customer updating their profile but can't, due to a lack of information. These are

anemic events: events that do not include additional information to further process the event.

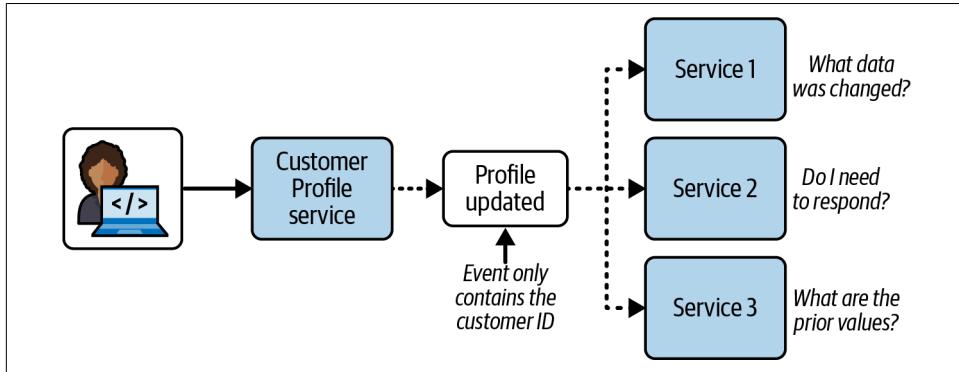


Figure 15-13. An anemic event lacks enough context to process the event

To avoid anemic events such as this one, include the updated customer information *as well as the prior values*, since most databases typically don't reflect that information.

This is an example of the *spectrum* of event payload granularity. On the extreme left side of the spectrum sit key-based event payloads, where only the key is contained within the event. While this has merit when creating or deleting an order, it doesn't work well when a customer updates an order. On the far right side of the spectrum is the data-based event payload, where *all* the data is included, whether it's needed or not. This is where stamp coupling rears its ugly head. The customer-profile-update scenario fits somewhere between these extremes because it provides the right level of information, thus avoiding the anemic derived event problem.

The Swarm of Gnats Antipattern

Related to anemic events is an antipattern known as the *Swarm of Gnats*. You probably know gnats as very small, annoying flying insects that buzz around your head, bothering you enough to send you back indoors on a beautiful sunny day. Whereas anemic events are concerned about the granularity of an event *payload*, the Swarm of Gnats antipattern is concerned about the granularity of the triggered events *themselves* and with how many derived events are triggered from an event processor. If an architect triggers too many derived events from a single event processor, they risk getting caught up in the Swarm of Gnats antipattern.

Consider the credit card payment example shown in Figure 15-14, where a customer places an order and their credit card is charged to pay for it. When the credit card is charged, the Payment event processor triggers a payment applied event, and (fortunately) the Fraud Detection event processor listens for it. This event processor

analyzes each charge to determine whether it's legitimate or fraudulent. Whatever the outcome, the `Fraud Detection` event processor triggers a `fraud_checked` derived event with the outcome of the fraud check contained in its event payload.

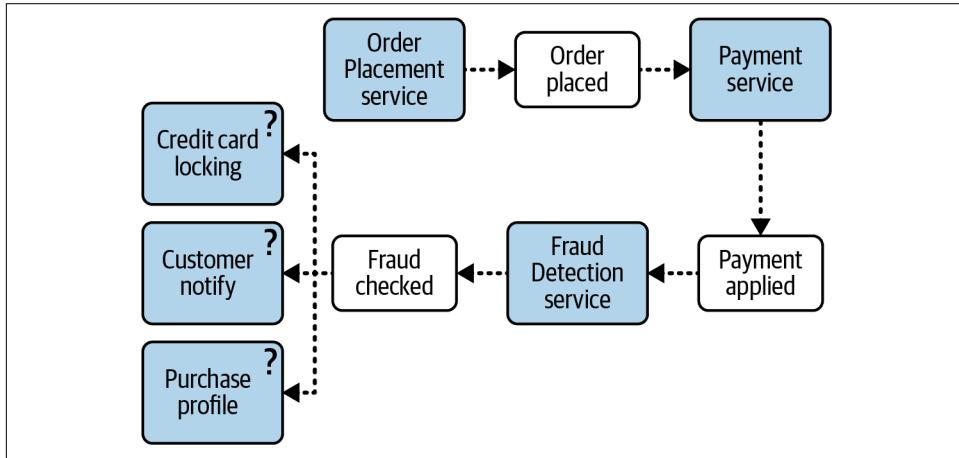


Figure 15-14. An example of an event that is too coarse-grained

Three event processors are interested in the outcome of the credit card fraud check:

- If fraud is detected, the `Credit Card Locking` event processor locks the customer's credit card to prevent further charges.
- The `Customer Notify` event processor notifies the customer of possible fraud.
- If fraud is *not* detected, the `Purchase Profile` event processor updates its algorithms.

Unfortunately, when a single `fraud_checked` derived event is triggered, *all* of these event processors must respond to the event, check the payload for the outcome, and decide whether to take action. Because this derived event is too coarse-grained, all of the event processors must perform additional processing: analyzing the payload of the single derived event to decide whether to take action. If no fraud has been detected, this is a waste of bandwidth and processing power, since only the `Purchase Profile` event processor needed to take any action.

A much more efficient approach would be to trigger *two* separate derived events (`fraud_detected` and `no_fraud_detected`), as shown in Figure 15-15. Here, the derived events triggered by the `Fraud Detection` event processor provide context *outside* of the event payload, allowing each event processor to decide whether to respond without having to analyze the event's internal payload.

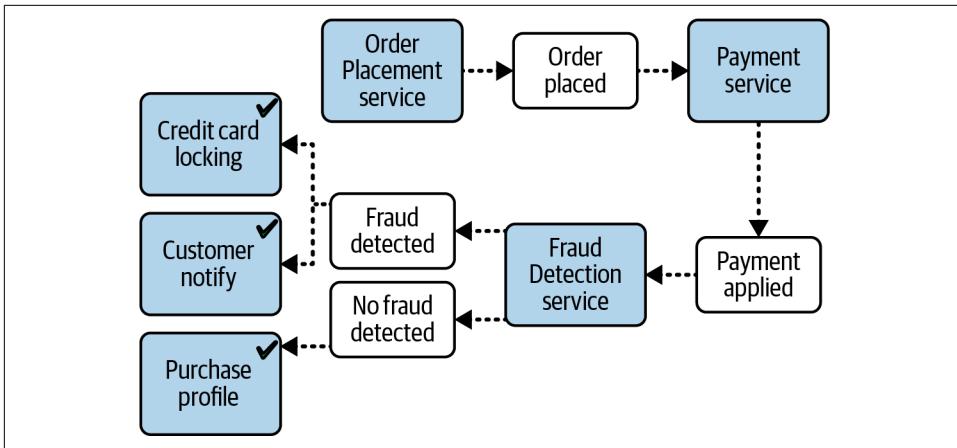


Figure 15-15. Triggering multiple events allows for more efficient processing and decision making

In this example, triggering multiple derived events for each outcome allows for better event flow, less churn, and more efficient processing. However, triggering *too many* derived events results in the *Swarm of Gnats* antipattern.

The scenario shown in [Figure 15-16](#) illustrates how this antipattern can occur. A customer has recently moved and needs to change their user profile on the website to update their credit card's bill-to address, ship-to address (where orders will be shipped), and phone number from their old landline to their cell phone. When the customer hits the Submit button for these profile changes, the `Customer Profile` event processor receives the update request, updates the database, and triggers a separate event for each update that contains the necessary information to do any further processing.

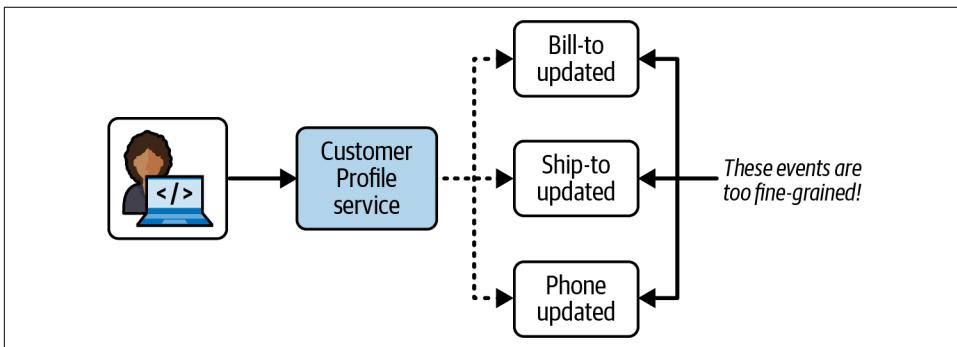


Figure 15-16. Triggering too many fine-grained derived events is known as the *Swarm of Gnats* antipattern

The problem with triggering too many fine-grained, detailed events is that it can saturate and overwhelm the system with derived events all related to the same thing: the customer updated their user profile. This antipattern also tends to proliferate numerous small derived events from other event processors, eventually making it hard for anyone to understand the system's overall event flows.

To avoid this antipattern, the architect could bundle each individual profile update into a single `profile_updated` derived event for the complete action, containing the before and after data of all updated fields. This more efficient approach is illustrated in [Figure 15-17](#).

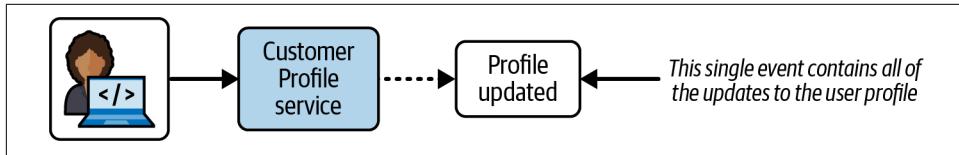


Figure 15-17. Combining individual state changes into a single derived event avoids the Swarm of Gnats antipattern

Determining the right level of granularity for derived events can be quite challenging. We recommend focusing on the *outcome* of the processing or state change to avoid the Swarm of Gnats antipattern and help simplify event flows.

Error Handling

The Workflow Event pattern of reactive architecture is one way of addressing error handling in an asynchronous workflow. This pattern addresses both resiliency and responsiveness, since it allows the system to handle asynchronous errors without affecting its responsiveness.

The Workflow Event pattern leverages delegation, containment, and repair by using a *workflow delegate*, as illustrated in [Figure 15-18](#). In this pattern, an event processor asynchronously passes data through a message channel to the event consumer. If the event consumer experiences an error while processing the data, it immediately delegates that error to the Workflow Processor service and moves on to the next message in the event queue. This way, the next message is immediately processed, so overall responsiveness remains the same. If the event consumer were to spend time trying to figure out the error, then it would not be processing the next message in the queue—delaying not only the next message, but all other messages waiting in the processing queue.

When the Workflow Processor service receives an error, it tries to figure out what's wrong with the message. Perhaps there's a static, deterministic error? It could analyze the message using some machine-learning or AI algorithms to look for some anomaly in the data. Either way, the workflow processor *programmatically* (that is, without

human intervention) makes changes to the original data to try to repair it, then sends it back to the originating queue. The event consumer sees this updated message as a new one and tries to process it again, hopefully this time with more success.

Of course, the workflow processor can't always determine what's wrong with the message. In these cases, it sends the message off to another queue, which is then received by a dashboard on the desktop of a knowledgeable person. This person looks at the message, applies manual fixes, and then resubmits it to the original queue (usually through a reply-to message header variable).

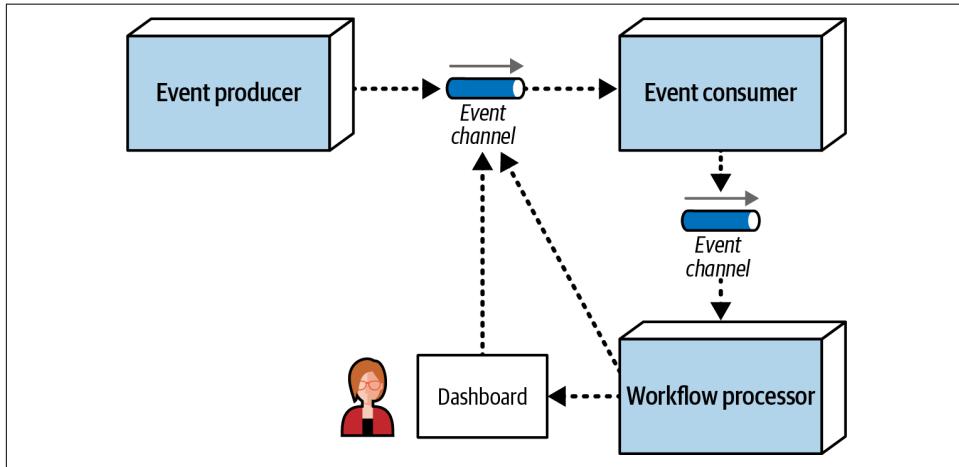


Figure 15-18. The Workflow Event pattern of reactive architecture

Suppose a trading advisor in one part of the country accepts *trade orders* (instructions on what stock to buy and for how many shares) on behalf of a large trading firm in another part of the country. The advisor batches up the trade orders in what is usually called a *basket* and asynchronously sends them to a broker across the country, who then purchases the stock. To simplify the example, suppose the contract for the trade instructions must adhere to the following:

```
ACCOUNT(String),SIDE(String),SYMBOL(String),SHARES(Long)
```

Suppose the large trading firm receives the following basket of Apple (AAPL) trade orders from the trading advisor:

```
12654A87FR4,BUY,AAPL,1254
87R54E3068U,BUY,AAPL,3122
6R4NB7609JJ,BUY,AAPL,5433
2WE35HF6DHF,BUY,AAPL,8756 SHARES
764980974R2,BUY,AAPL,1211
1533G658HD8,BUY,AAPL,2654
```

The fourth trade instruction (2WE35HF6DHF,BUY,AAPL,8756 SHARES) has the word SHARES after the number of shares for the trade. When the primary firm processes these asynchronous trade orders without any error handling capabilities, the following error occurs within the TradePlacement service:

```
Exception in thread "main" java.lang.NumberFormatException:  
  For input string: "8756 SHARES"  
  at java.lang.NumberFormatException.forInputString  
(NumberFormatException.java:65)  
  at java.lang.Long.parseLong(Long.java:589)  
  at java.lang.Long.<init>(Long.java:965)  
  at trading.TradePlacement.execute(TradePlacement.java:23)  
  at trading.TradePlacement.main(TradePlacement.java:29)
```

When this exception occurs, because this was an asynchronous request, there is no user to synchronously respond to and fix the error. The TradePlacement service can't do anything, except possibly log the error condition.

Applying the Workflow Event pattern can fix this error programmatically. Because the primary firm has no control over the trading advisor or the trade-order data it sends, it must react to fix the error itself (see [Figure 15-19](#)). When the same error occurs (2WE35HF6DHF,BUY,AAPL,8756 SHARES), the TradePlacement service immediately delegates the error via asynchronous messaging to the Trade Placement Error service for error handling, passing it along with the error information about the exception:

```
Trade Placed: 12654A87FR4,BUY,AAPL,1254  
Trade Placed: 87R54E3068U,BUY,AAPL,3122  
Trade Placed: 6R4NB7609JJ,BUY,AAPL,5433  
Error Placing Trade: "2WE35HF6DHF,BUY,AAPL,8756 SHARES"  
Sending to trade error processor <-- delegate the error fixing and move on  
Trade Placed: 764980974R2,BUY,AAPL,1211  
...
```

The Trade Placement Error service, acting as the workflow delegate, receives the error and inspects the exception. Seeing that the issue is with the word SHARES in the Number of Shares field, the Trade Placement Error service strips off the word SHARES and resubmits the trade for reprocessing:

```
Received Trade Order Error: 2WE35HF6DHF,BUY,AAPL,8756 SHARES  
Trade fixed: 2WE35HF6DHF,BUY,AAPL,8756  
Resubmitting Trade For Re-Processing
```

The TradePlacement service can now process the fixed trade successfully:

```
...
trade placed: 1533G658HD8,BUY,AAPL,2654
trade placed: 2WE35HF6DHF,BUY,AAPL,8756 <-- this was the original trade in error
```

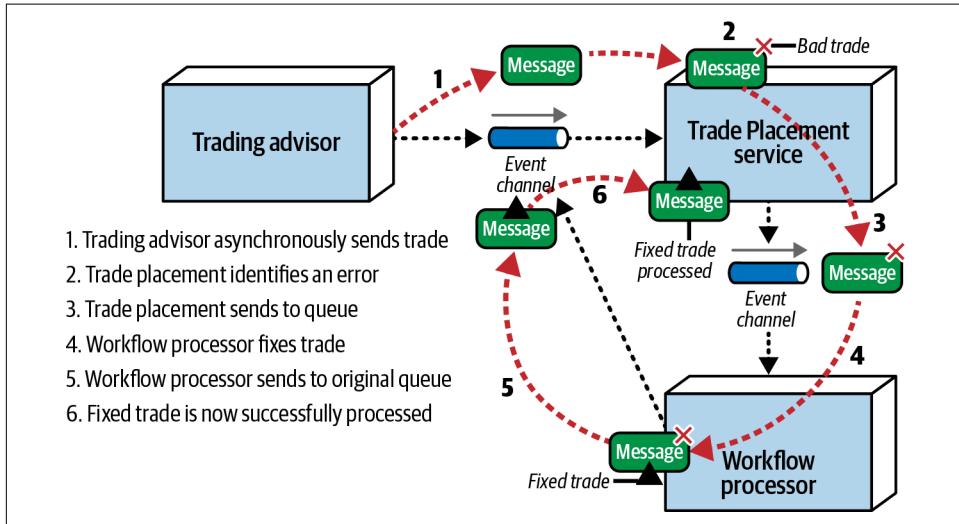


Figure 15-19. Error handling with the Workflow Event pattern

One consequence of using the Workflow Event pattern is that messages that are sent to a workflow processor and then resubmitted are processed out of sequence. In our trading example, the order of the messages matters, because all trades within a given account must be processed in order (for example, a SELL for IBM must occur before a BUY for AAPL within the same brokerage account). It would be complex, although not impossible, to maintain the message order within a given context (in this case, the brokerage account number). One way to address this is for the TradePlacement service to queue and store the brokerage account number of the erroneous trade. Any trade with that same brokerage account number would be stored in a temporary queue for later processing (in first-in, first-out, or FIFO, order). Once the erroneous trade is fixed and processed, the TradePlacement service de-queues the remaining trades for that same account and processes them in order.

Preventing Data Loss

Architects dealing with asynchronous communications are always concerned with *data loss*: when an event or message gets dropped or never makes it to its final destination. Fortunately, there are basic out-of-the-box techniques to prevent data loss.

Architects can implement event channels in a variety of ways. Most event-driven architectures use the [Advanced Message Queuing Protocol \(AMQP\)](#) for triggering and responding to events. Examples of AMQP brokers include [Amazon SNS \(Simple Notification Service\)](#), [RabbitMQ](#), [Solace](#), and [Azure Event Hubs](#). With AMQP, events are published to an exchange. The exchange uses the binding rules set by the consuming event processors to forward the event to a queue for each event processor that subscribes to that event. AMQP brokers can also leverage what is known as the *Event Forwarding* pattern to prevent data loss, a technique we describe in this section.

Another event channel implementation is the [Jakarta Messaging API](#) (formerly [Java Message Service or JMS](#)), which uses *topics* rather than the two-step forwarding process queues use. Nevertheless, Jakarta Messaging can still leverage the *Event Forwarding* pattern to prevent data loss, provided that the event processors responding to an event are configured as durable subscribers. A *durable subscriber* is one that is guaranteed to receive an event. If the event processor is down or otherwise unavailable, the JMS topic stores the event until the subscribing event processor becomes available.

Another possible event-channel implementation is event streaming using [Kafka](#) as an *event broker* (the software product containing the queues and topics). The techniques for preventing data loss within event streaming are very different from those used for the Event Forwarding pattern described in this section. Refer to the [Kafka website](#) for more information about preventing data loss when using this kind of streaming event broker.

Consider a typical scenario in which event processor A asynchronously publishes an event to a message broker, which eventually goes to an AMQP queue or JMS topic. Event processor B responds to the event and inserts the data from the payload into a database. As illustrated in [Figure 15-20](#), there are three ways data loss can occur in this scenario:

1. While event processor A is publishing the event, it crashes before an acknowledgment from the event broker can be sent; alternately, the event broker sends an acknowledgment to event processor A, but then crashes before the event is accepted by another event processor.
2. Event processor B accepts the event from the queue but crashes before it can process the event.
3. Event processor B is unable to persist the message to the database due to a data error.

Each of these areas of data loss can be mitigated through the *Event Forwarding* pattern.

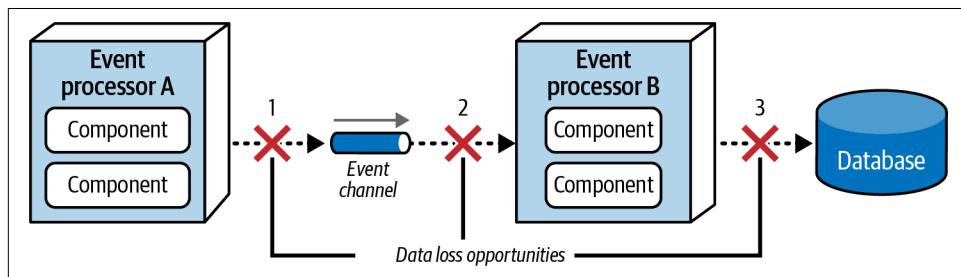


Figure 15-20. Places where data loss can happen within an event-driven architecture

With the first issue, the event never makes it to the queue or the broker fails before the event is read. To address this, use persistent message queues along with synchronous send. Persisted message queues support *guaranteed delivery*: when the event broker receives the event, not only does it store the event in memory for fast retrieval, it also persists the event in some sort of physical data store (such as a filesystem or database). If the event broker goes down, the event is physically stored on disk, so it will still be available for processing when the event broker comes back up. *Synchronous send* does a blocking wait in the event processor, stopping it from triggering the event until the broker acknowledges that it has persisted the event to the database. These two basic techniques prevent data loss between the event producer and the queue, because the event is either still with the event producer or persisted within the queue.

A basic messaging technique called *client acknowledge mode* can address the second issue, where event processor B de-queues the next available event and crashes before it can process the event. By default, when an event is read from a queue, it is immediately removed from that queue (this is called *auto acknowledge mode*). Client acknowledge mode keeps the event in the queue and attaches the client ID to it so that no other consumers can read or process the event. With this mode, if event processor B crashes, the event is still preserved in the queue, preventing message loss.

The third issue, in which event processor B is unable to persist the event to the database due to some data error, can be addressed with ACID transactions via a database commit. Once the event processor issues a database commit, the data is guaranteed to be persisted in the database. *Last participant support* (LPS) removes the event from the persisted queue by acknowledging that all processing has been completed and that the event has been persisted. This guarantees that the event has not been lost in transit from event processor A to the database. These techniques are illustrated in Figure 15-21.

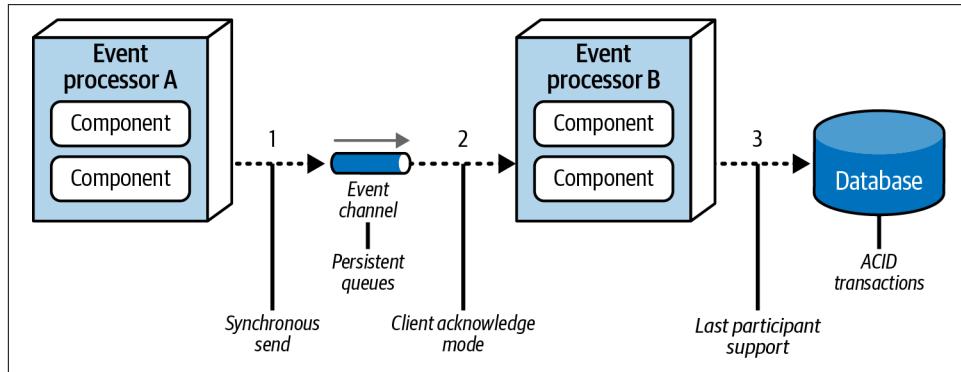


Figure 15-21. Preventing data loss within an event-driven architecture

Request-Reply Processing

So far in this chapter, we've dealt with asynchronous requests that don't need an immediate response from the event consumer. But what about event processors that need information back immediately from another event processor—for example, waiting for some sort of confirmation ID or acknowledgment before triggering an event? This scenario requires synchronous communication to complete the request.

In EDA, synchronous communication is typically accomplished through *request-reply* messaging (sometimes referred to as *pseudosynchronous communications*). Each event channel within request-reply messaging consists of two queues: a *request queue* and a *reply queue*. The message producer making the initial request for information asynchronously sends data to the request queue, and then returns control to the message producer. The message producer then does additional processing, and eventually waits on the reply queue for the response. The message consumer receives and processes the message, then sends the response to the reply queue. The event producer receives the message with the response data. This basic flow is illustrated in [Figure 15-22](#).

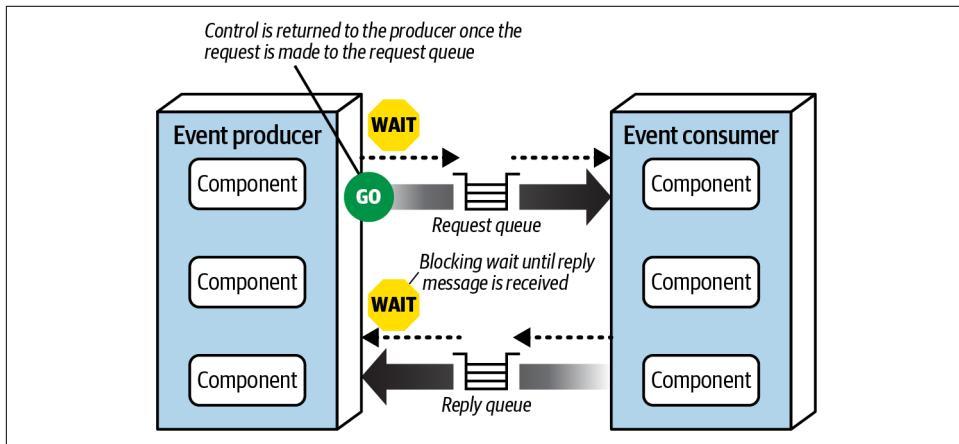


Figure 15-22. Request-reply message processing

There are two primary ways to implement request-reply messaging. The first (and most common) technique is to put a *correlation ID* (CID) field in the message header of the reply message, usually set to the message ID of the original request message (simply called ID in Figure 15-23). It works like this:

1. The event producer sends a message to the request queue and records the unique message ID (ID 124). Notice that the CID in this case is **null**.
2. The event producer does a blocking wait on the reply queue with a message filter (also called a *message selector*), where the CID in the message header equals the original message ID (124). There are two messages in the reply queue: ID 855 with CID 120, and ID 856 with CID 122. Neither of these messages will be picked up, because neither correlation ID matches what the event consumer is looking for (CID 124).
3. The event consumer receives the message (ID 124) and processes the request.
4. The event consumer creates the reply message containing the response and sets the CID in the message header to the original message ID (124).
5. The event consumer sends the new message ID (857) to the reply queue.
6. The event producer receives the message, because the CID (124) matches the message selector from step 2.

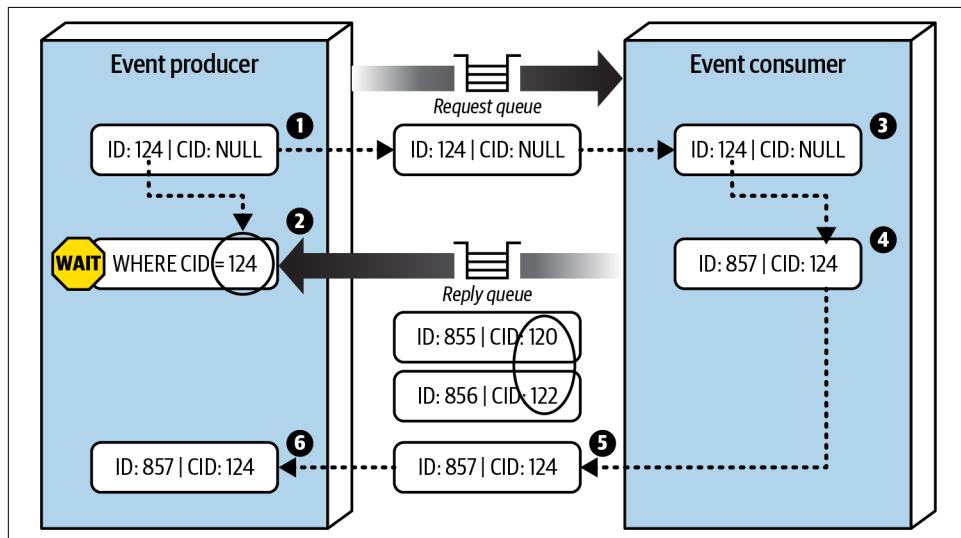


Figure 15-23. Request-reply message processing using a correlation ID

The other way to implement request-reply messaging is to use a *temporary queue* for the reply queue. A temporary queue is dedicated to a specific request, created when the request is made, and deleted when the request ends. This technique, as illustrated

in [Figure 15-24](#), does not require a correlation ID, because the temporary queue is a dedicated queue only known to the event producer for that specific request. The temporary queue technique works as follows:

1. The event producer creates a temporary queue (or one is automatically created, depending on the message broker) and sends a message to the request queue, passing the name of the temporary queue in the reply-to header (or some other agreed-upon custom attribute in the message header).
2. The event producer does a blocking wait on the temporary reply queue. No message selector is needed because any message sent to this queue belongs solely to the event producer that sent the original message.
3. The event consumer receives the message, processes the request, and sends a response message to the reply queue named in the reply-to header.
4. The event processor receives the message and deletes the temporary queue.

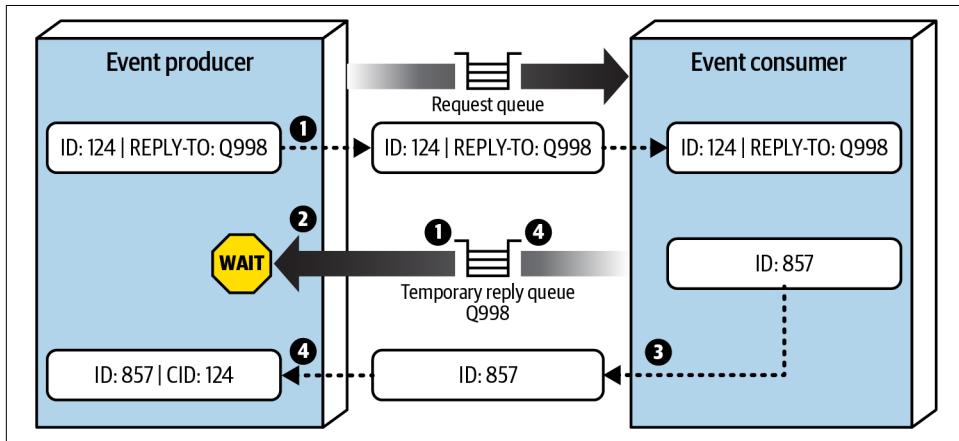


Figure 15-24. Request-reply message processing using a temporary queue

While the temporary queue technique is much simpler, the message broker must create a temporary queue for each request and then delete it immediately. This can significantly slow the broker down and can affect overall performance and responsiveness, particularly for large message volumes and high concurrency. For this reason, we usually recommend using the correlation ID technique.

Mediated Event-Driven Architecture

So far in this chapter we've focused on *choreographed* EDA, where event processors trigger events through broadcast capabilities, and multiple event processors respond to the event. However, there may be times when an architect wants more control over

the processing of an event. In this case, the architect can use an *orchestrated* form of EDA known as the mediator topology.

The *mediator topology* addresses some of the shortcomings of the standard choreographed EDA topology we've described so far in this chapter. It's centered on an *event mediator*, which manages and controls the workflow for initiating events that require coordination between multiple event processors. The architecture components that make up the mediator topology are: an initiating event, an event queue, an event mediator, event channels, and event processors.

Importantly, the mediated topology typically uses *messages* rather than *events* (see “[Events Versus Messages](#)” on page 232). They are generally commands (such as `ship_order`) rather than events that have happened (such as `order_shipped`).

Like in the choreographed topology, the initiating event is what starts the whole process. However, in the mediator topology ([Figure 15-25](#)), an event mediator accepts the initiating event. It only knows the steps involved in processing the event, so it generates corresponding derived messages and sends them to dedicated message channels (usually queues) in a point-to-point fashion. Event processors then listen to the dedicated event channels, process messages, and (usually) respond to the mediator when they have completed their work. Event processors within the mediator topology do not advertise what they've done to the rest of the system through additional derived messages.

In most implementations of the mediator topology, there are multiple mediators, each usually associated with a particular domain or grouping of events. This avoids having a single point of failure, which can be an issue with this topology, and increases overall throughput and performance. For example, a customer mediator might handle all customer-related events (such as new customer registrations and profile updates), while an order mediator handles order-related activities (such as adding an item to a shopping cart and checking out).

How architects choose to implement the event mediator usually depends on the nature and complexity of the messages the event mediator is processing. For example, for events requiring simple error handling and orchestration, mediators such as [Apache Camel](#), [Mule ESB](#), or [Spring Integration](#) will usually suffice. Message flows and message routes within these types of mediators are typically custom written in programming code (such as Java or C#) to control the event-processing workflow.

However, if the event workflow requires lots of conditional processing and multiple dynamic paths with complex error handling directives, then a mediator such as [Apache ODE](#) or the [Oracle BPEL Process Manager](#) would be a more appropriate choice. These mediators are based on the [Business Process Execution Language \(BPEL\)](#), an XML-like structure that describes the steps involved in processing an event. BPEL artifacts also contain structured elements used for error handling,

redirection, multicasting, and so on. BPEL is a powerful but relatively complex language to learn, so architects usually create mediators using the GUI tools in BPEL's engine suite.

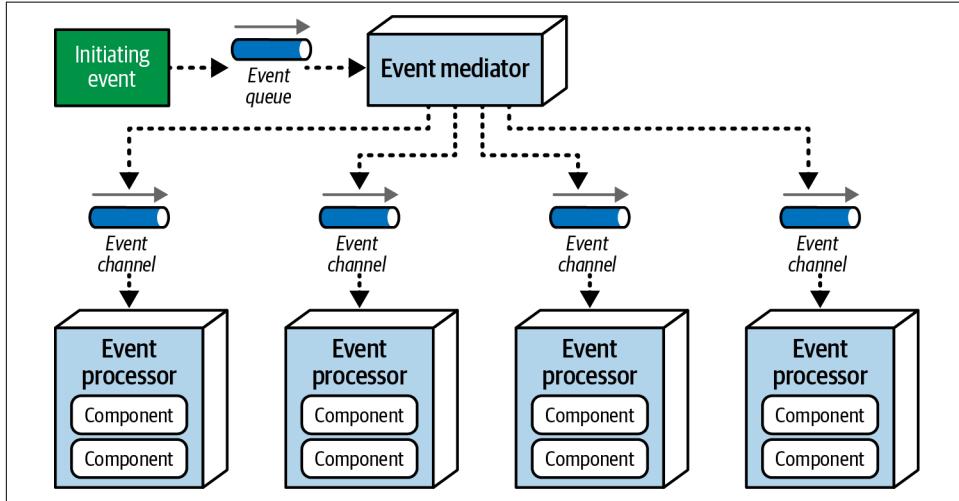


Figure 15-25. Mediator topology

BPEL is good for complex, dynamic workflows, but it does not work well for event workflows with long-running transactions that involve human intervention throughout the event process. For example, suppose a trade is being placed through a `place_trade` initiating event. The event mediator accepts this event, but during the processing, finds that manual approval is required because the trade is over a certain number of shares. The event mediator now has to stop the event processing, notify a senior trader to get the manual approval, and wait for that approval. In these cases, a Business Process Management (BPM) engine, such as [jBPM](#), would be more appropriate than using an event mediator.

Before choosing what kind of event mediator to implement, it's important to know the types of events it will process. For complex, long-running events involving human interaction, Apache Camel would be extremely difficult to use and maintain. By the same token, using a BPM engine for simple event flows would waste months of effort on something Apache Camel could accomplish in a matter of days.

Of course, it's rare to have all events fit neatly into one class of complexity. We recommend classifying events as simple, hard, or complex, and sending every event through a simple mediator, such as Apache Camel or Mule. The simple mediator can handle the event itself or forward it to another, more complex event mediator based on that complexity classification. This mediator delegation model, illustrated in [Figure 15-26](#), ensures that all types of events are handled by the type of mediator that will process them most effectively.

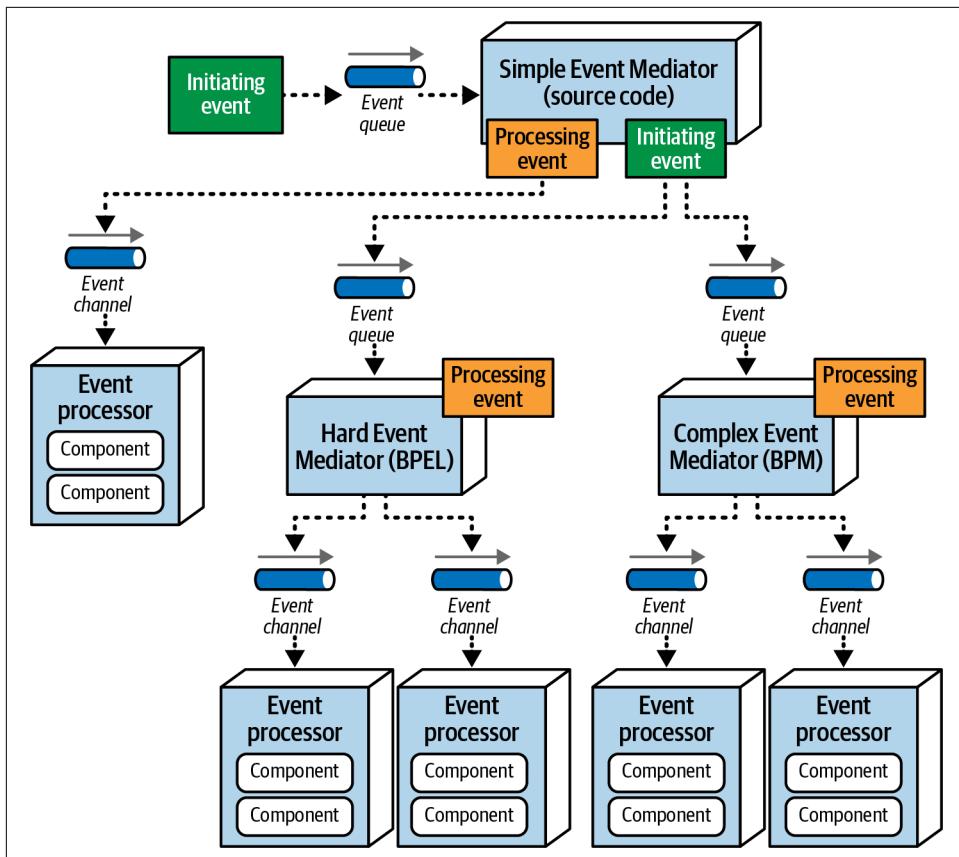


Figure 15-26. Delegating the event to the appropriate type of event mediator

Notice in [Figure 15-26](#) that the **Simple Event Mediator** generates and sends a derived message when the event workflow is simple enough to be handled entirely by the simple mediator. However, when the initiating event is classified as hard or complex, the **Simple Event Mediator** forwards the original initiating event to the corresponding mediators (BPEL or BPM). The **Simple Event Mediator**, having intercepted the original event, may still be responsible for knowing when that event is complete, or it could simply delegate the entire workflow (including client notification) to the other mediators.

To examine how the mediator topology works, let's consider the same retail order entry system that we described in the section on choreographed topology, but this time using the mediator topology. The mediator knows the steps required to process this particular event. The mediator component's internal event flow is illustrated in [Figure 15-27](#).

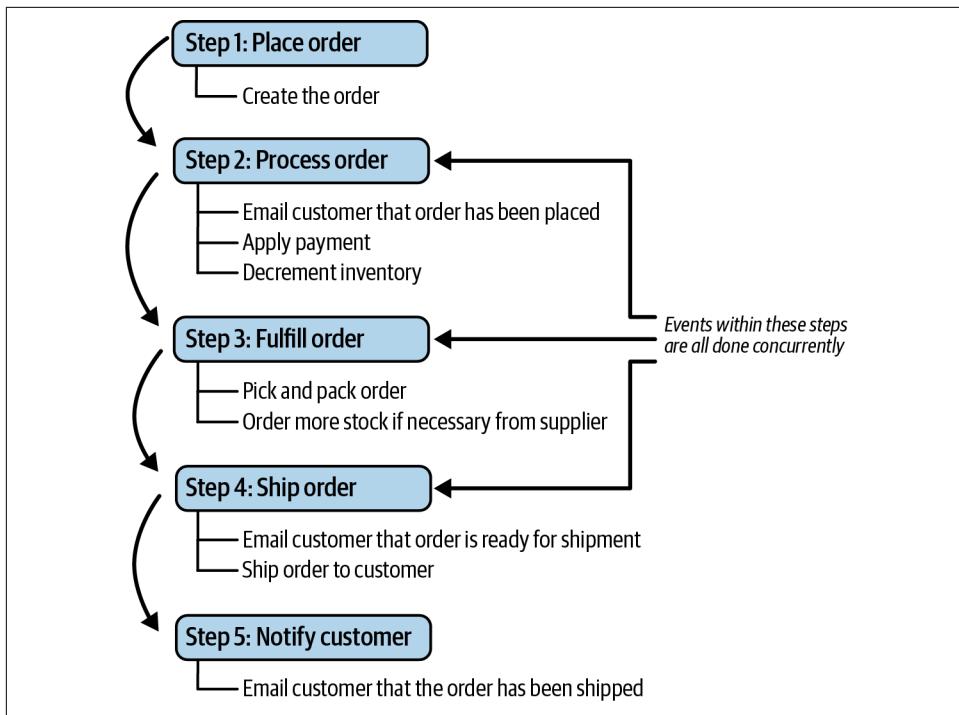


Figure 15-27. Mediator steps for placing an order

In keeping with the prior example, the same initiating event (`place_order`) is sent to the event mediator via a dedicated queue for processing. The Customer mediator picks up this initiating event and begins generating derived messages, based on the flow in Figure 15-27. The events shown in steps 2, 3, and 4 are all done concurrently and serially, between steps. In other words, step 3 (fulfill order) must be completed and acknowledged before the customer can be notified that the order is ready to be shipped in step 4 (ship order).

Once it receives the initiating event, the Customer mediator generates a `create_order` derived message, which it sends to the `order_placement` queue (see Figure 15-28). The Order Placement event processor accepts the message and validates and creates the order, and sends the mediator an acknowledgment and the order ID in return. At this point, the mediator might send that order ID to the customer, indicating that the order was placed, or it might have to continue until all the steps are complete (this would depend on specific business rules about order placement).

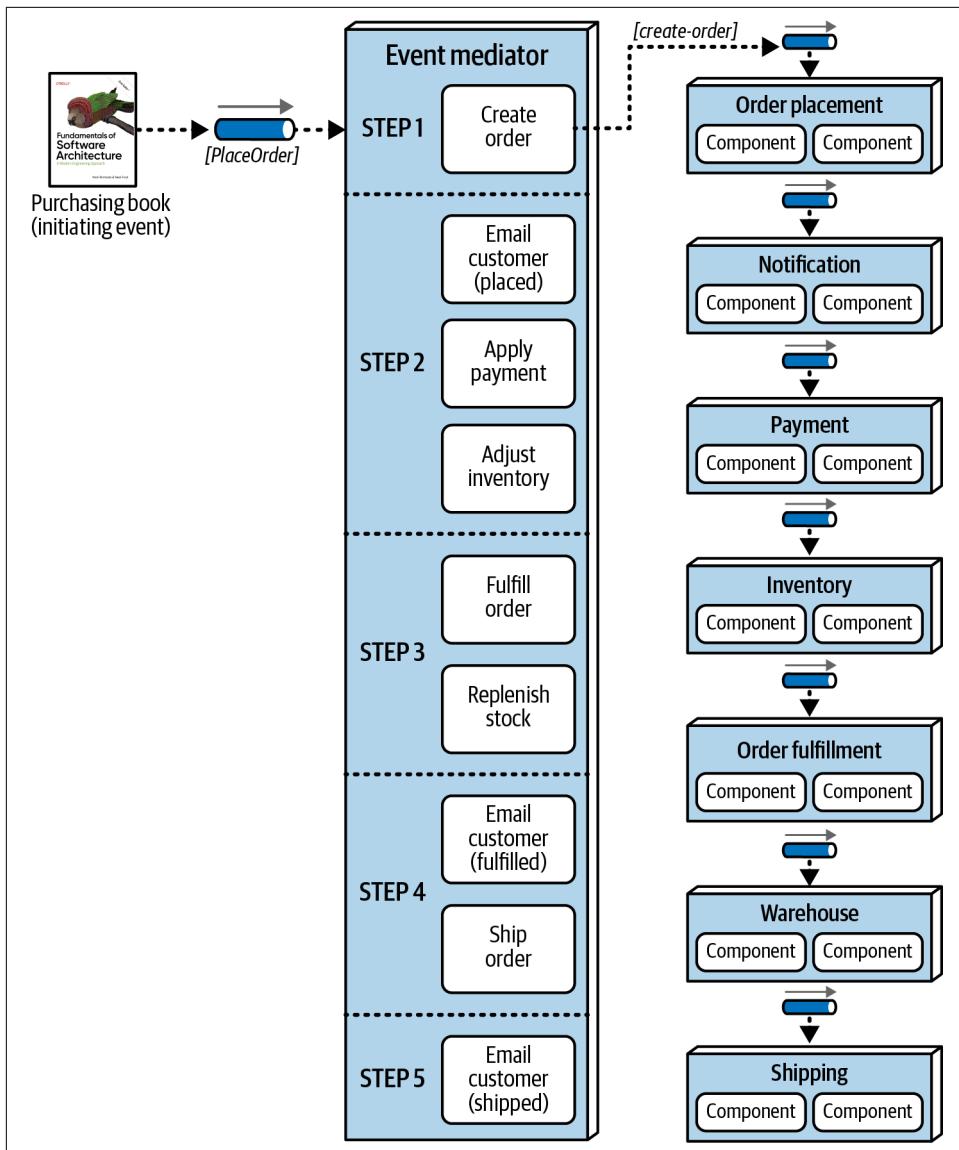


Figure 15-28. Step 1 of the mediator example

Now that step 1 is complete, the mediator moves to step 2 (see [Figure 15-29](#)) and generates three derived messages at the same time: `email customer`, `apply payment`, and `adjust inventory`. It sends all three to their respective queues. All three event processors receive these messages, perform their respective tasks, and notify the mediator that their processing is complete. The mediator must wait until it receives acknowledgment from all three parallel processes before moving on to step 3. If an

error occurs in one of the parallel event processors, the mediator can take corrective action (more about that later in this section).

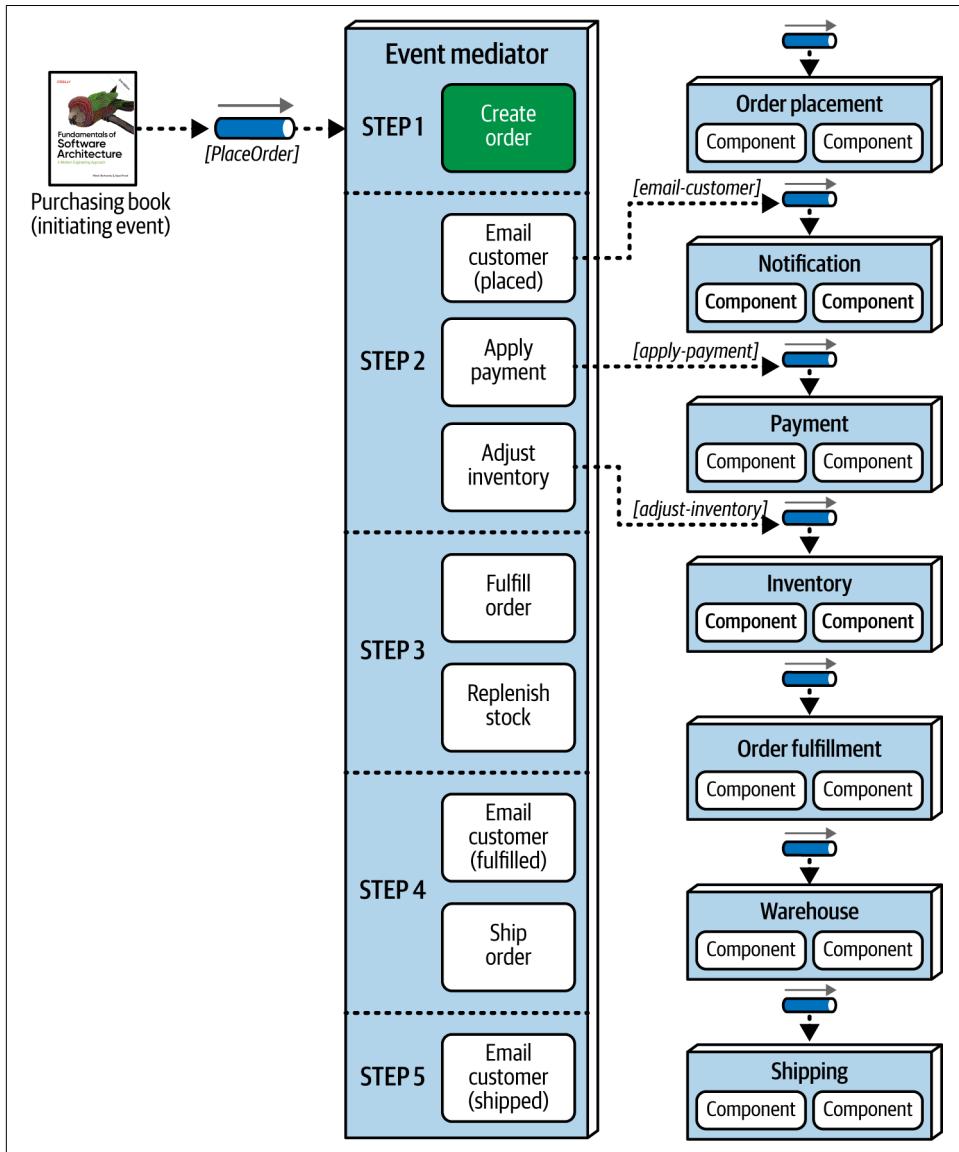


Figure 15-29. Step 2 of the mediator example

Once the mediator gets a successful acknowledgment from all of the event processors in step 2, it can move to step 3 to fulfill the order (see [Figure 15-30](#)). Once again, both of these messages (`fulfill order` and `order stock`) can occur simultaneously. The `Order Fulfillment` and `Warehouse` event processors accept the messages, perform their work, and return an acknowledgment to the mediator.

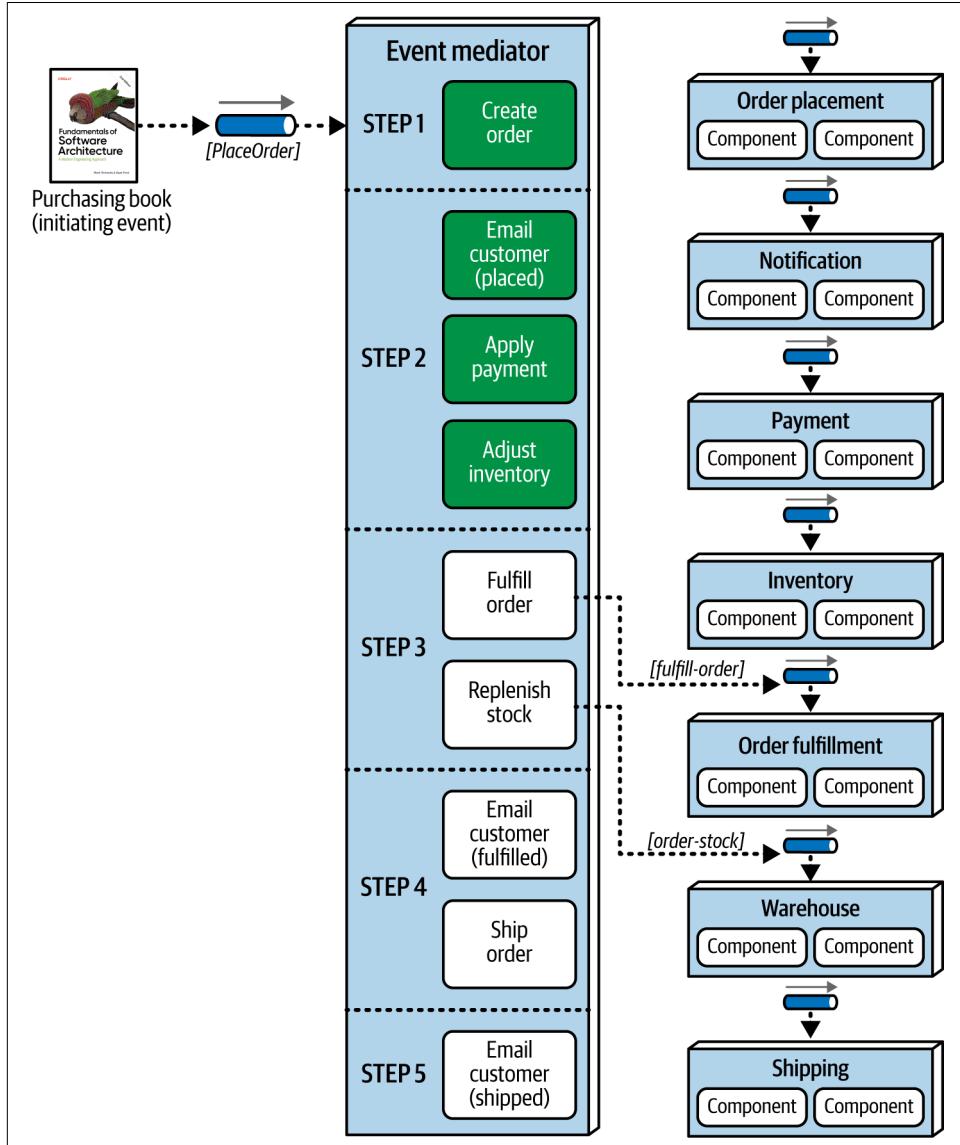


Figure 15-30. Step 3 of the mediator example

The mediator moves on to step 4 (see [Figure 15-31](#)) to ship the order. This step generates two derived messages: a `ship order` message, and another `email customer` message with specific information about what to do (notify the customer that the order is ready to be shipped).

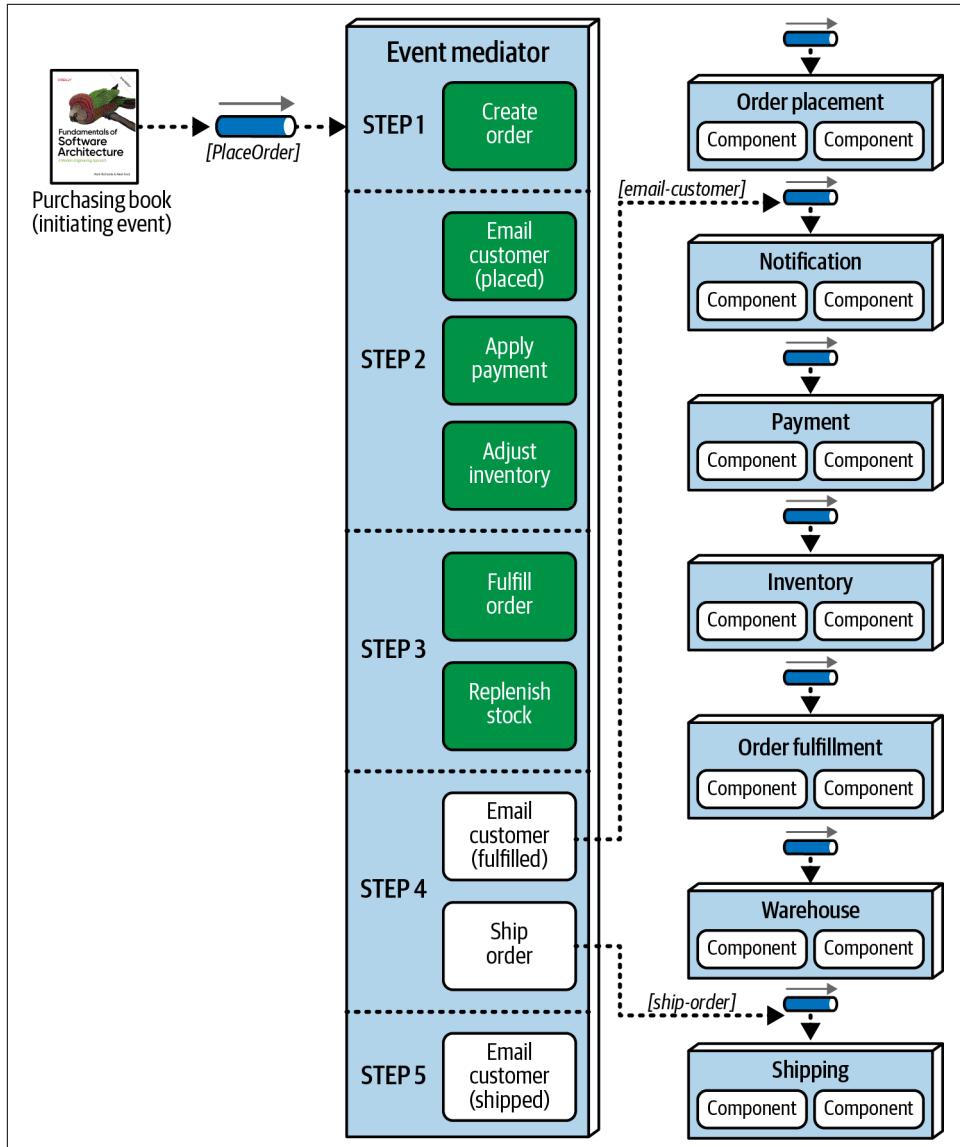


Figure 15-31. Step 4 of the mediator example

Finally, the mediator moves to step 5 (see [Figure 15-32](#)) and generates another contextual `email customer` message to notify the customer that the order has been shipped. This ends the workflow. The mediator marks the initiating event flow complete and removes all state associated with the initiating event.

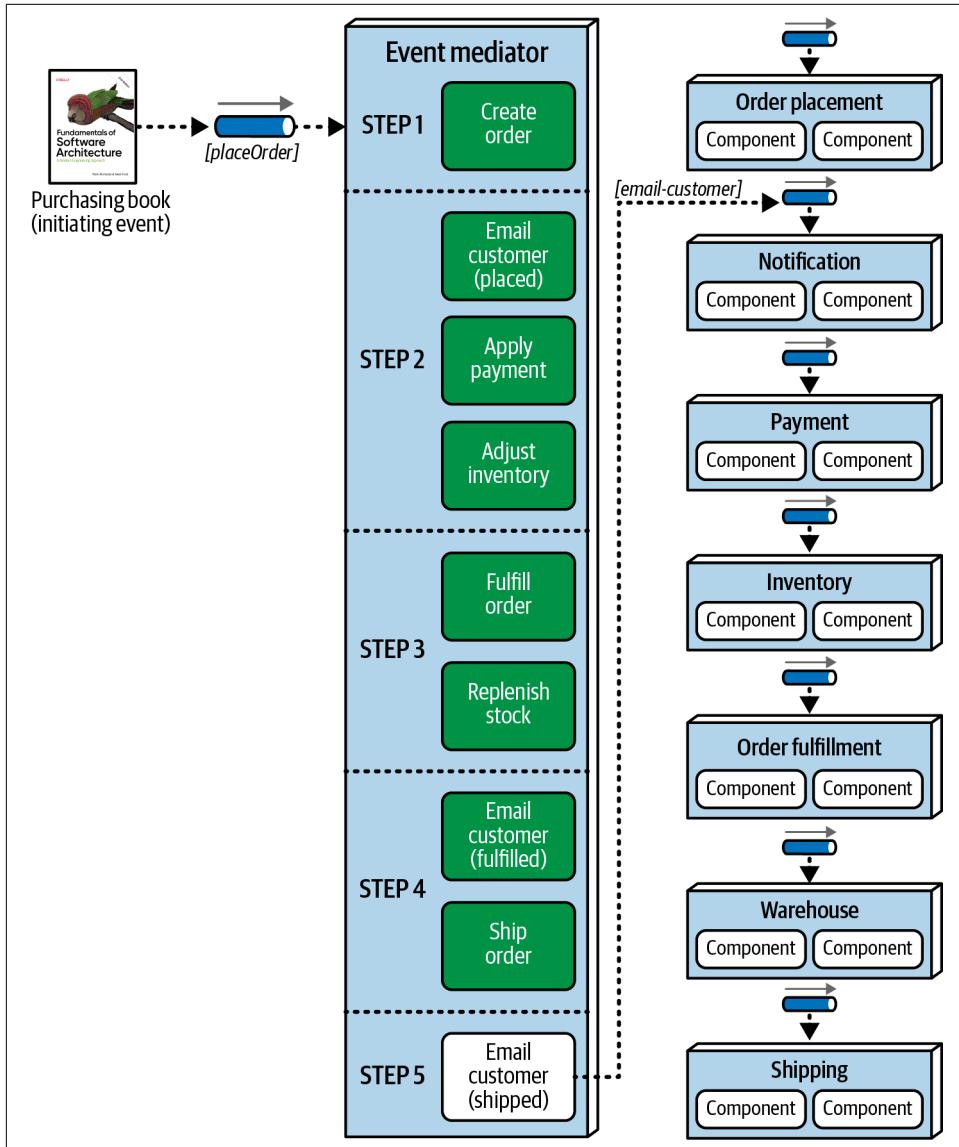


Figure 15-32. Step 5 of the mediator example

In this topology, unlike the choreographed topology, the mediator component has knowledge of and control over the workflow. It can maintain event state and manage error handling, recoverability, and restart capabilities. For example, suppose in our example that the payment was not applied because the credit card has expired. When the mediator receives this error condition, it knows that the order cannot be fulfilled (step 3) until payment is applied, so it stops the workflow and records the state of the request in its own persistent datastore. Once payment is eventually applied, the workflow can be restarted from where it left off (in this case, the beginning of step 3).

While the mediator topology addresses the issues associated with the choreographed topology, it has its own disadvantages. First of all, it is very difficult to declaratively model the dynamic processing that occurs within a complex event flow. As a result, many workflows within the mediator topology only handle general processing, but use a hybrid model that combines the mediator and choreographed topologies to address the dynamic nature of complex event processing, such as out-of-stock conditions or other nontypical errors. Furthermore, although the event processors can easily scale in the same manner as the choreographed topology, the mediator must scale as well—something that occasionally produces a bottleneck in the overall event-processing flow. Event processors are not as highly decoupled in the mediator topology as they are in the choreographed topology. Finally, performance is not as good in this topology, because the mediator controls event processing.

The trade-off between the choreographed and mediator topologies essentially comes down to weighing workflow control and error handling capability against high performance and scalability. Although performance and scalability are still good within the mediator topology, they are not as high as with the choreographed topology.

Data Topologies

With all this talk of events and event processing, it's easy to forget about the data side of EDA. Database topologies are a unique and interesting aspect of this architectural style. It offers many options, each with significant trade-offs that can have a big impact on the overall architecture. To describe the different database topologies within EDA, we'll use a simplified version of the example we showed in [Figure 15-3](#) (see [Figure 15-33](#)).

When a customer places an order, the `Order Placement` event processor creates the order, then triggers an `order placed` event, to which both the `Payment` and `Inventory` event processors respond. Once the payment has been applied, the `Order Fulfillment` event processor helps the order packer prepare the order, then triggers an `order fulfilled` event. The `Shipping` event processor responds to the `order fulfilled` event by shipping the order to the customer, completing the processing and fulfilling the customer's request.

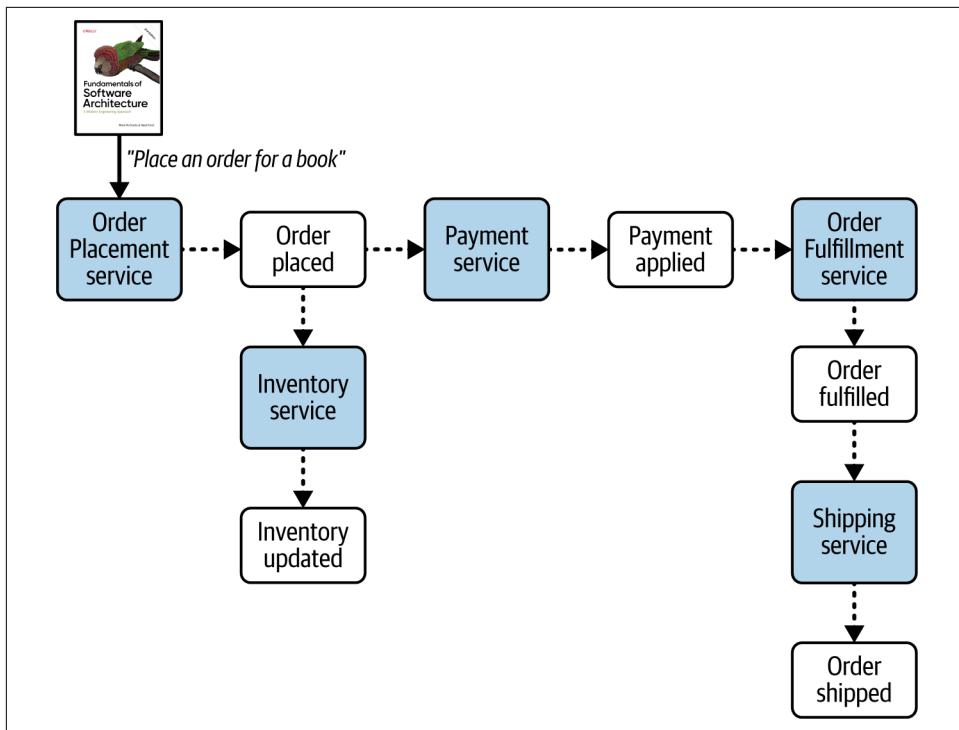


Figure 15-33. A simplified example of the example order-entry system using EDA

One complication of EDA is that the `Order Placement` event processor needs to know two pieces of information: how many items are currently in stock, and what shipping options are available based on the customer's location. How it gets this information will depend on the type of database topology the architecture uses. Let's look at each database topology option to see its significant trade-offs.

Monolithic Database Topology

The first, and perhaps most common, database topology used within EDA is the *single monolithic database* topology. With this topology, all data is available to all event processors through a central database.

The main benefit of the monolithic database topology is that any event processor can query the data it needs directly from the database, without having to synchronously communicate with any other event processors. This is a significant advantage, since event-driven architectures rely on highly decoupled event processors communicating through asynchronous communications. In Figure 15-34, the `Order Placement` event processor can simply query the central monolithic database to retrieve the number of items currently in stock and the customer's shipping options.

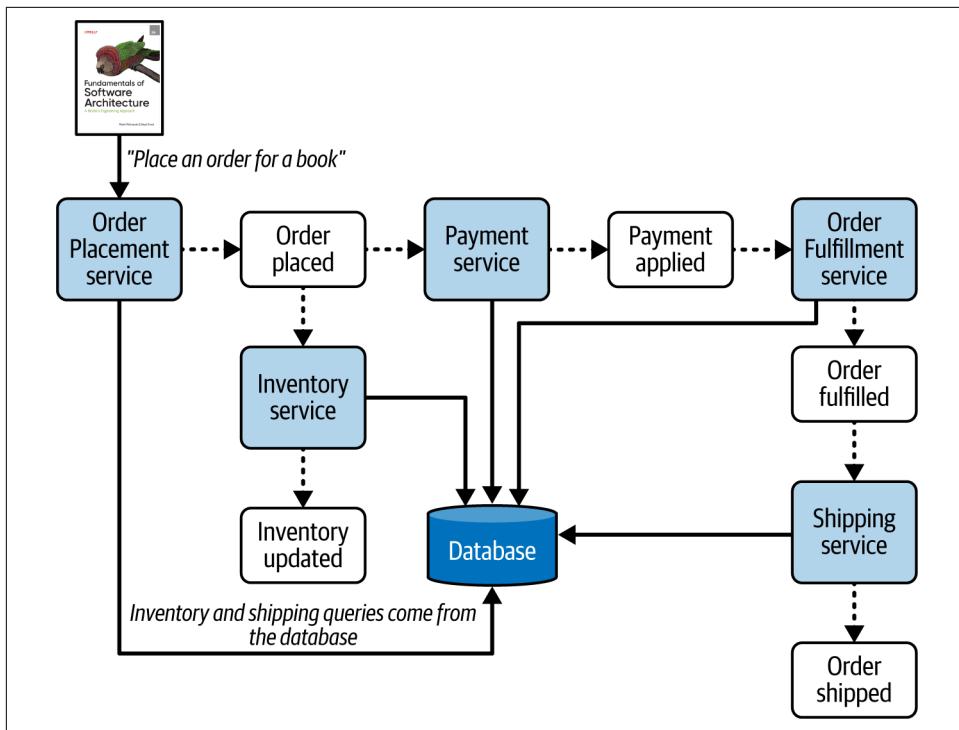


Figure 15-34. With the monolithic database topology, data is available directly from the database

While the monolithic database topology supports decoupling and limits communication between event processors, it carries with it some messy disadvantages, the first of which is fault tolerance. If the central monolithic database crashes or goes down for maintenance, all event processors become unavailable.

The second issue is scalability. Because event-driven architectures use asynchronous communication, each event processor can scale independently of the others. The event channel essentially acts as a backpressure point so that individual event processors can scale as necessary, regardless of whether other event processors scale as well. However, if all the event processors are simultaneously querying and writing to the same database, the *database* must scale to meet these demands. Many databases are unable to achieve this at high concurrency loads.

The third issue is change control. When the structure of the database changes (such as dropping a column or attribute), multiple event processors are affected and must coordinate, even for a single database change.

Finally, the monolithic database topology necessarily creates a single architectural quantum, due to the shared monolithic database.

Domain Database Topology

Another possible database topology within EDA is the *domain database topology*, which groups event processors into various domains, each owning its own database (Figure 15-35).

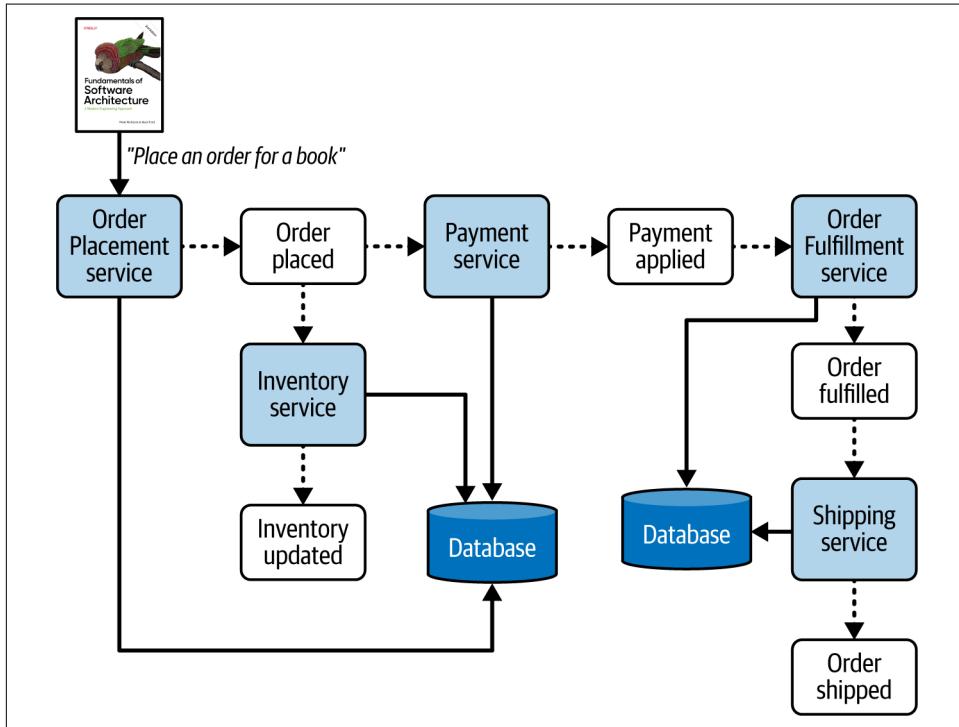


Figure 15-35. The domain database topology uses a separate database for each domain

The main advantages of the domain database topology over the monolithic database topology, which flow from being domain partitioned, are better fault tolerance, scalability, and change control. In the example shown in Figure 15-35, if the order-processing domain database (the one associated with the Order Fulfillment and Order Shipping event processors) crashes or becomes unavailable due to maintenance, the order placement domain is still fully operational and can continue to accept orders. The event channel containing the payment applied derived event acts as a backpressure point, queuing events until the order-processing database becomes available. The same is true with scalability and change control; each domain database only has to worry about scaling based on its domain's specific event processors, and only those domain-scoped event processors need to change if the database structure changes.

However, consider the two pieces of information needed by the `Order Placement` event processor: the number of books currently on hand and the shipping options. With the domain database topology, the `Order Placement` event processor can simply query its domain database to get the book inventory, similar to how it retrieved this information with the monolithic database topology. However, it must make a *synchronous* call to the `Order Shipping` event processor to retrieve the shipping options, thus synchronously coupling these services (see [Figure 15-36](#)).

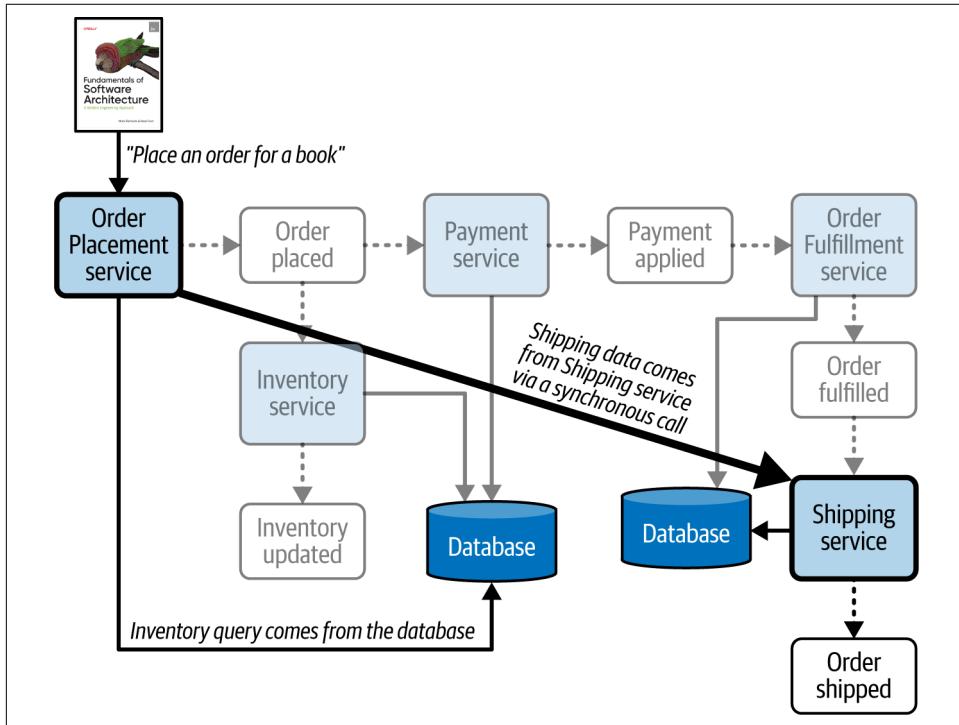


Figure 15-36. Data needed by the Order Placement event processor may require synchronous communication to an event processor in another domain

Architects should try to avoid synchronous coupling in a highly dynamic, decoupled architecture like EDA. Synchronous calls can take a toll on fault tolerance and scalability, negating many of the benefits of using this topology. Always check that the domains remain fairly independent from each other, and minimize synchronous interservice calls as much as possible. If too much synchronous communication between event processors is required, reevaluate the domain boundaries, combine the domains into a single one, or move to a monolithic database topology.

Dedicated Data Topology

Another viable option within EDA is the *dedicated database topology*, commonly referred to in the microservices world as the *database-per-service* pattern. With this database topology, each event processor owns its own dedicated database in a tightly formed bounded context, similar to microservices (see “[Data Topologies](#)” on page 348 in Chapter 18). This topology is illustrated in [Figure 15-37](#).

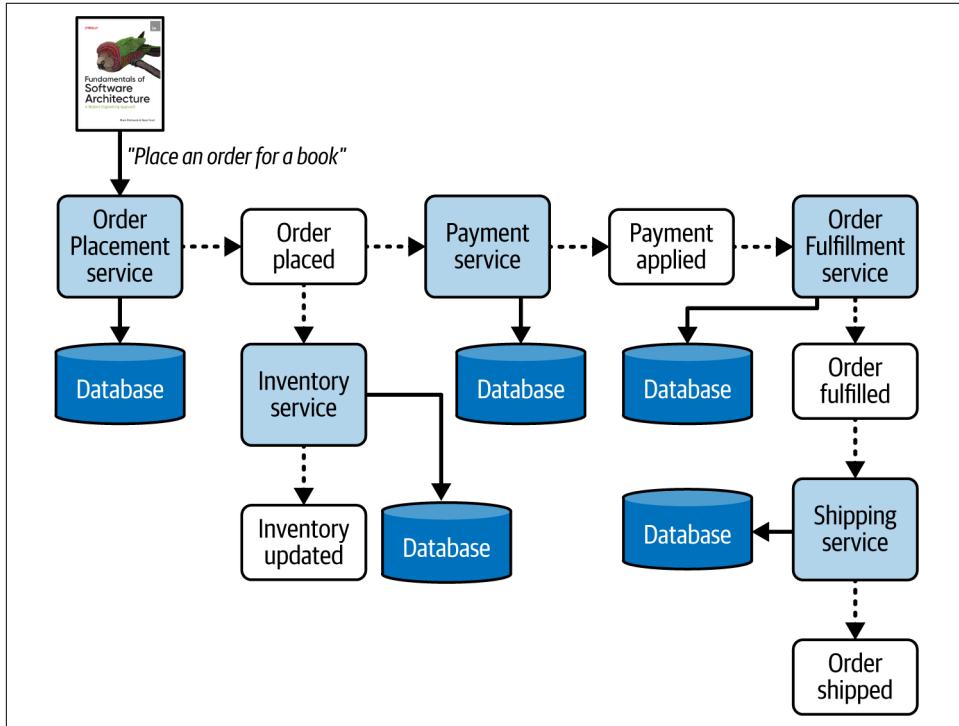


Figure 15-37. The dedicated database topology of EDA uses a separate database for each event processor

Not surprisingly, the dedicated database topology has the highest levels of fault tolerance, scalability, and change control out of all the available topologies. If an event processor or database goes down, the outage is isolated to just that event processor; all other event processors function normally. Databases only need to scale based on the single event processor within its bounded context, making this topology the most scalable of the three discussed in this section. Finally, changes to the database structure only impact the event processor attached to that database.

On the negative side, depending on the database technology stack, this can be a very expensive option. Perhaps the biggest disadvantage, however, is synchronous dynamic coupling between event processors. To illustrate this disadvantage, look

again at the two pieces of information the Order Placement event processor needs for its processing: book inventory and shipping options. In this topology, the Order Placement event processor would need to make synchronous calls to *both* the Inventory event processor and the Order Shipment event processor to get the information it needs, forming tight synchronous coupling points throughout the architecture (as shown in Figure 15-38). As with the domain database topology, identify all data-related requirements in each event processor before selecting this database topology option.

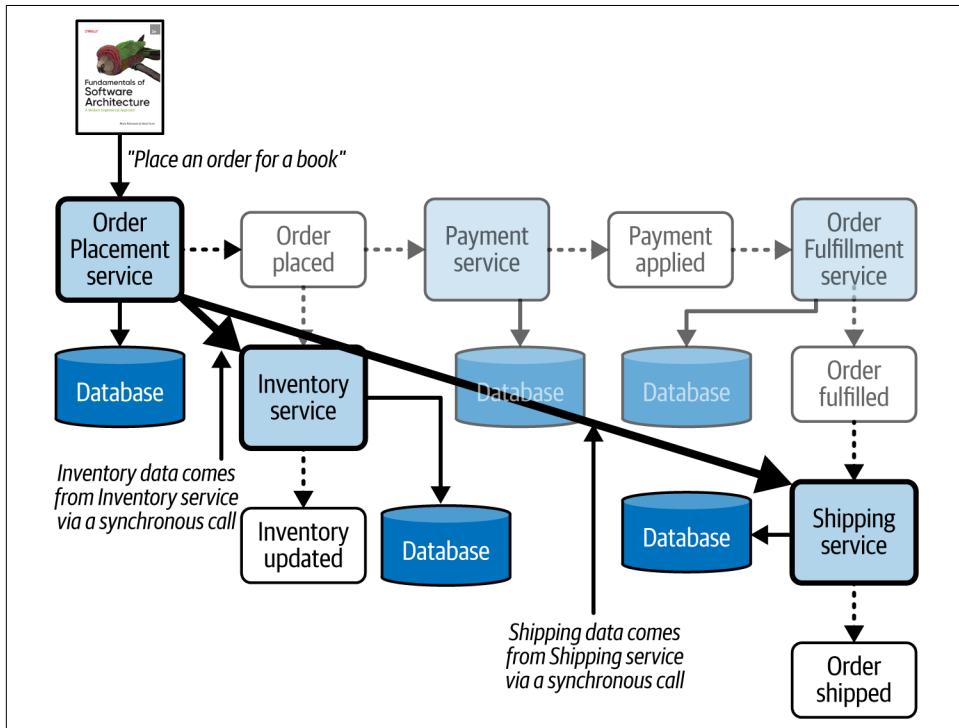


Figure 15-38. Data needed by the Order Placement event processor may require synchronous communication to other event processors

The dedicated database topology is a good choice when event processors are mostly self-contained, only needing the data within its own bounded context and corresponding database. If event processors are communicating too much (see “Governance” on page 276), the architect should consider moving to the domain or even the monolithic database topology to improve overall performance and scalability. However, if the specific situation involves frequent structural changes to the database, that might warrant a trade-off between these operational characteristics to minimize the number of event processors affected.

Cloud Considerations

Event-driven architectures work well with cloud-based environments and implementations, primarily due to their highly decoupled nature. EDA can easily leverage the asynchronous services provided by cloud vendors, and the elastic nature of cloud infrastructure and cloud-based services matches its shape. Essentially, cloud-based environments are a good match for EDA.

Common Risks

One of the main risks associated with EDA is that its nondeterministic event processing can cause side effects: for instance, event processors unexpectedly triggering derived events or not responding to an event when they should. Event workflows can get very complex in an event-driven architecture, and many times it's difficult to know exactly what will happen when an event is triggered.

Another big risk is having too much static coupling (and hence brittleness) within the event-driven architecture. Although EDA is highly *dynamically* coupled, event payload contracts (see “[Event Payload](#)” on page 240) can also introduce tight *static* coupling. Changing a contract can be a daunting task, since architects don't always know which event processors are responding to a particular event. When an event payload contract does change, it may negatively impact several other event processors, adding to the overall brittleness of this architectural style. Key-based event payloads help mitigate this risk, but the risks of this practice include issues with scalability and performance and the possibility of anemic events (see “[Event Payload](#)” on page 240).

Watch out for too much synchronous communication between event processors as well. Event-driven architecture gets its superpowers from its highly dynamically decoupled event processors. However, if event processors keep needing to communicate synchronously, it's a good sign that EDA is not the most appropriate architectural style.

Finally, overall state management is both a risk and a challenge in EDA. It's good to know when the initiating event has been completely processed, but that can be very hard to determine due to EDA's nondeterministic, asynchronous parallel event processing. Occasionally an architect can identify the final processing endpoint and have the event processor that accepted the initiating event subscribe to that “ending” event, but in most cases this is hard to determine. As a result, it's difficult to know when an initiating event has been fully processed, or even its current state.

Governance

A majority of the governance associated with EDA is nonstructural and requires observability in the form of logs as part of an overall governance mesh. Depending on the infrastructure and environment, some governance metrics associated with EDA may need to be manually collected.

The two main areas of governance in this style are static coupling through contract management, and dynamic coupling through synchronous calls. Both of these aspects are considered structural decay in EDA, so it's important to keep an eye on them.

From a static coupling perspective, architects can set up governance around things like the rate of change of event payload contracts and overall stamp coupling. Changing contracts can be very risky in EDA because of its decoupled nature. Changing an event contract, particularly one with no associated schema, can break a downstream event processor. This is a particular risk in EDA because it's so difficult to test nondeterministic end-to-end event flows.

Stamp coupling (see “[Event Payload](#)” on page 240) can be governed by continually recording and observing which fields in an event contract are not being used by event processors responding to the event. Observing these unused fields can help trim down contract size, reduce bandwidth, and help manage stamp coupling and corresponding unnecessary changes to event processors.

From a dynamic coupling standpoint, architects can write automated fitness functions to observe and track synchronous communication between event processors through logs and other observable means (such as source code annotations or use of standard synchronous custom-identifier libraries). *Any* synchronous communication in an event-driven architecture should be tracked and discussed to ensure that it is necessary, particularly if using a domain or dedicated database topology.

Team Topology Considerations

EDA is largely considered a technically partitioned architecture due to the many artifacts that make up each domain: multiple event processors, event channels, message brokers, and possibly databases, depending on the database topology. Nevertheless, it can work well when teams are aligned within domain areas (such as cross-functional teams with specialization). That said, specific types of team topologies (see “[Team Topologies and Architecture](#)” on page 151) might find EDA challenging.

Here are some considerations regarding the alignment between these specific team topologies and EDA:

Stream-aligned teams

Depending on the size of the system, stream-aligned teams might find themselves struggling to implement domain-based changes, due to the decoupled nature of event processors. Because domains and subdomains in EDA are typically implemented with multiple event processors and derived events, stream-aligned teams might find it a challenge to understand all of these moving parts. For example, adding a step to the order-placement workflow might require changing multiple event processors, as well as restructuring how (and when) the existing derived events are triggered. The larger and more complex the event-driven architecture, the less effective stream-aligned teams will be.

Enabling teams

Because of the integration required between event processors based on derived events and their contracts, enabling teams don't work well in event-driven architectures. Experimentation and efficiency by enabling teams *within* a stream can disrupt a stream-aligned team's understanding and management of an overall event flow, and usually requires too much coordination between stream-aligned teams and enabling teams.

Complicated-subsystem teams

Complicated-subsystem teams work well with EDA due to its decoupled, asynchronous nature. Complex processing can be easily isolated through separate event processors, and the complicated-subsystem team can focus on those, leaving less complicated processing to the stream-aligned teams. Because event processors are highly dynamically decoupled, stream-aligned teams only need to coordinate with complicated-subsystem teams for static event-payload contracts and derived events.

Platform teams

In EDA, developers can leverage the benefits of the platform-teams topology by utilizing common tools, services, APIs, and tasks, primarily due to EDA's technical partitioning. This is especially true if teams treat the infrastructure-related parts of EDA (such as the message brokers, event hubs, event buses, and other event-channel artifacts) as platform related.

Style Characteristics

A one-star rating in the characteristics ratings table in [Figure 15-39](#) means the specific architecture characteristic isn't well supported in the architecture, whereas a five-star rating means it's one of that style's strongest features. Definitions for each characteristic identified in the scorecard can be found in [Chapter 4](#).

Event-driven architecture is primarily a technically partitioned architecture, in that any particular domain is spread across multiple event processors and tied together

through brokers, contracts (event payload), and topics. Changes to a particular domain usually impact multiple event processors and other messaging artifacts, so EDA is not generally considered domain partitioned.

| | Architectural characteristic | Star rating |
|-------------|------------------------------|-------------|
| Structural | Overall cost | \$\$\$ |
| | Partitioning type | Technical |
| | Number of quanta | 1 to many |
| | Simplicity | ★★ |
| Engineering | Modularity | ★★★★★ |
| | Maintainability | ★★★★★ |
| | Testability | ★★ |
| | Deployability | ★★★★ |
| Operational | Evolvability | ★★★★★ |
| | Responsiveness | ★★★★★ |
| | Scalability | ★★★★★ |
| | Elasticity | ★★★★ |
| | Fault tolerance | ★★★★★ |

Figure 15-39. EDA characteristics ratings

The number of quanta within EDA can vary from one to many, depending on the database interactions within each event processor and whether the system uses request-reply processing. Even though communication in an EDA relies on asynchronous calls, if multiple event processors share a single database instance, they would all be contained within the same architectural quantum. The same is true for request-reply processing: even though the communication between the event processors is still asynchronous, if a response is needed right away from the event consumer, it ties those event processors together synchronously, forming a single quantum.

For example, imagine that one event processor sends a request to another event processor to place an order. The first event processor must wait for an order ID from the other event processor to continue. If the second event processor (the one that places the order and generates an order ID) is down, the first event processor cannot continue. This means they are part of the same architecture quantum and share the

same architectural characteristics, even though they are both sending and receiving asynchronous messages.

Event-driven architecture rates very high (4 to 5 stars) for performance, scalability, and fault tolerance, its primary strengths. Its high performance is achieved by combining asynchronous communications with highly parallel processing. High scalability is realized through the programmatic load balancing of event processors (also called *competing consumers* and *consumer groups*). As the request load increases, additional event processors can be programmatically added to handle the additional requests. The reason we only gave it four stars (rather than five) is because of the database (see [Chapter 16](#), on space-based architectures, for an example of a five-star rating for these characteristics). EDA achieves fault tolerance through its decoupled, asynchronous event processors, which provide eventual consistency and processing of event workflows. If other downstream processors are unavailable, as long as the user interface or an event processor making a request does not need an immediate response, the system can process the event at a later time.

EDA rates relatively low on overall *simplicity* and *testability*, mostly due to its non-deterministic and dynamic event flows. In request-based models, deterministic flows are relatively easy to test because their paths and outcomes are generally known. Such is not the case with the event-driven model. Sometimes architects just don't know how event processors will react to dynamic events or what messages they might produce. These are known as *nondeterministic workflows*. These systems' "event tree diagrams" can be extremely complex, generating hundreds or even thousands of scenarios, making it very difficult to govern and test.

Finally, EDAs are highly evolutionary, hence the five-star rating. Adding new features through existing or new event processors is relatively straightforward. By providing hooks via triggered derived events, the event and its corresponding data are already available for other processing, so no changes are required in the infrastructure or existing event processors to add that new functionality.

EDA's disadvantages include difficulty controlling the overall workflow associated with an initiating event. Event processing is very dynamic due to changing conditions, and it's difficult to know when the business transaction based on the initiating event has been completed.

Error handling is also a big challenge with EDA. Because there is typically no mediator monitoring or controlling the business transaction (except with the mediator topology), if a failure occurs, other services will be unaware of the crash. The business process based on that initiating event gets stuck and is unable to move without some sort of automated or manual intervention, even as all other processes move along without regard for the error. For example, if the `Payment` event processor in our ordering system crashes and does not complete its assigned task, the `Inventory` event

processor still adjusts the inventory, and all other further event processors react as though everything is fine.

Restarting a business transaction (recoverability) is very difficult to do in EDA. Because other actions have been taken asynchronously through the processing of the initiating event, many times it's simply not feasible to resubmit the initiating event.

Choosing Between Request-Based and Event-Based Models

The request-based and event-based models are both viable approaches for designing software systems. However, choosing the right model is essential to success. We recommend choosing the request-based model for well-structured, data-driven requests (such as retrieving customer profile data), when the priority is certainty and control over the workflow. We recommend choosing the event-based model for flexible, action-based events that require high levels of responsiveness and scale, with complex and dynamic user processing.

Understanding the event-based model's trade-offs also helps in determining the best fit. [Table 15-2](#) lists the advantages and disadvantages of the event-based model of EDA.

Table 15-2. Trade-offs of the event-driven model

| Advantages over request-based | Trade-offs |
|--|---|
| Better response to dynamic user content | Only supports eventual consistency |
| Better scalability and elasticity | Less control over processing flow |
| Better agility and change management | Less certainty over outcome of event flow |
| Better adaptability and extensibility | Difficult to test and debug |
| Better responsiveness and performance | |
| Better real-time decision making | |
| Better reaction to situational awareness | |

Examples and Use Cases

Any business problem focused on responding to things happening in the system (either internal or external) is a good candidate for EDA. The order-entry system example we've used throughout this chapter is a good use case for EDA in that it allows for parallel, decoupled processing of an order. Systems that require high levels of responsiveness, performance, scalability, fault tolerance, and elasticity are also great candidates for EDA.

Another good use case to demonstrate the power and effectiveness of EDA is our ongoing Going, Going, Gone auction-system example, where users bid on items put up for auction until a final bid goes uncontested and the bidder wins that item. In

these systems, the number of bidders is usually unknown, requiring both scalability and elasticity—particularly if the auction is timed and the bidding comes to a close. These systems also need high levels of responsiveness. Perhaps the best reason that online bidding systems and EDA are a good match, though, is that EDA regards placing a bid not as a *request made to the system*, but as an *event that has happened*.

As [Figure 15-40](#) shows, lots of actions take place in the system when a bidder places a bid. These actions can all be asynchronous, done either at the same time or later, as backend processing (such as the `Bidder Tracker` event processor). When a bidder places a bid, the `Bid Capture` event processor receives that initiating event, determines if it's higher than the prior bid, and triggers a `bid placed` event. The `Auctioneer` event processor responds to that event and updates the website with the item's new bid price. Simultaneously, the `Bid Streamer` event processor responds to the same event, streaming the bid out to the website bid history or even to the individual bidders (depending on the UI). Finally, the `Bidder Tracker` event processor responds to the event to persist the bidder and their bid for tracking and auditing purposes.

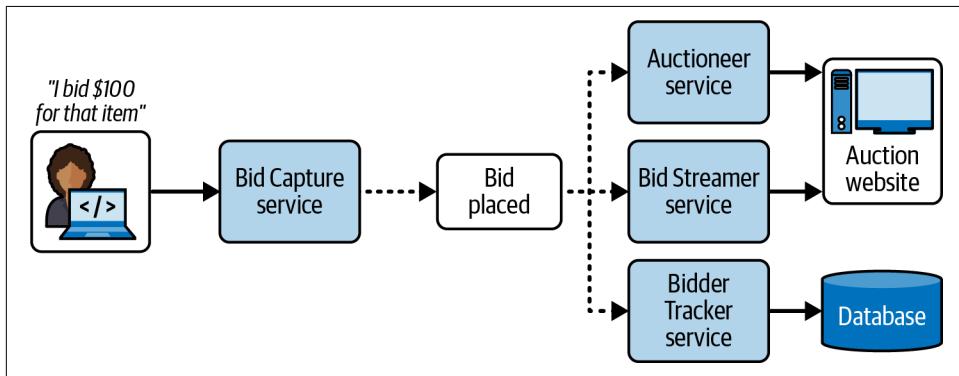


Figure 15-40. An online bidding system example using EDA

Many other event processors are involved in the workflow of this initiating event, and many other derived events are triggered, but this small part of the system demonstrates a good use of the event-driven architectural style, illustrating responsiveness, fault tolerance, scalability, and elasticity.

Event-driven architecture is a very complicated architectural style, but also a very powerful one. Closely analyze the workflows and processing needed for the business problem to determine if dealing with EDA's complexity is worthwhile given its superpowers. If a majority of the processing needed is request based, consider the microservices architectural style (discussed in [Chapter 18](#)) instead.

Space-Based Architecture Style

Most web-based business applications follow the same general request flow: a request from a browser hits the web server, then an application server, then finally the database server. While this typical request flow works for a small set of concurrent users, bottlenecks start appearing as the concurrent user load increases: first at the web-server layer, then at the application-server layer, and finally at the database-server layer.

The usual response when increased user load causes bottlenecks is to scale out the web servers. This is relatively easy and inexpensive, and sometimes it works. However, in most cases of high user load, scaling out the web-server layer just moves the bottleneck down to the application server. Scaling application servers can be more complex and expensive than scaling web servers, and doing so usually just moves the bottleneck down to the database server, which is even more difficult and expensive to scale. Even if you can scale the database, what you eventually end up with is a triangle-shaped topology, with the widest part of the triangle being the web servers (easiest to scale) and the smallest part being the database (hardest to scale), as illustrated in [Figure 16-1](#).

In any high-volume application with a large concurrent user load, the database will usually be the final limiting factor in how many transactions the application can process concurrently. While various caching technologies and database-scaling products can help, scaling out a normal application for extreme loads remains a very difficult proposition.

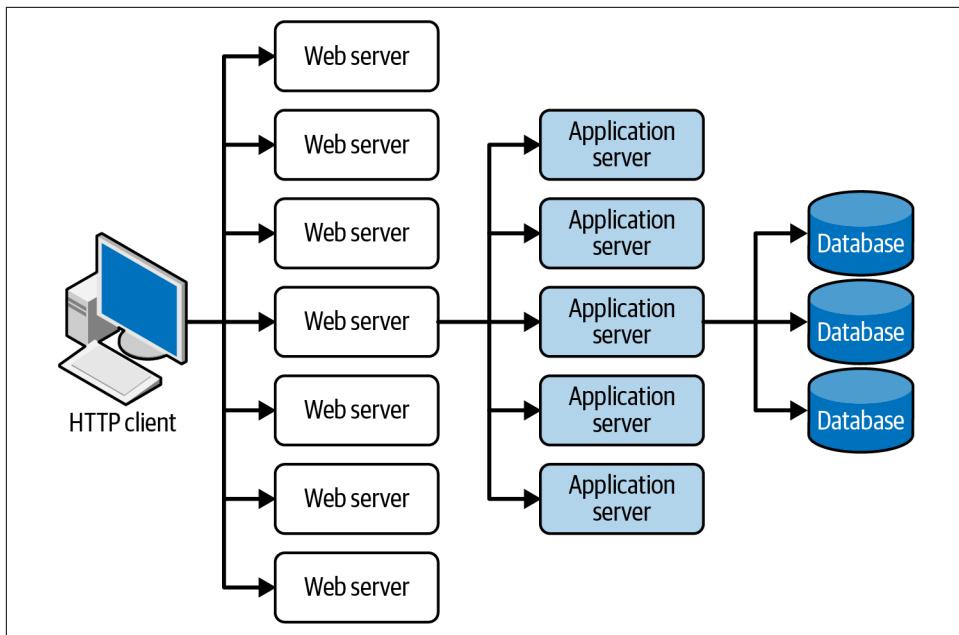


Figure 16-1. Scalability limits within a traditional web-based topology

The *space-based* architecture style is specifically designed to address problems involving high scalability, elasticity, and concurrency. It is also a useful architecture style for applications that have variable and unpredictable concurrent user volumes. It's often better to solve extreme and variable scalability architecturally, rather than try to scale out a database or retrofit caching technologies into an architecture that can't scale as well.

Topology

Space-based architecture gets its name from the concept of *tuple space*, the technique of using multiple parallel processors that communicate through shared memory. Space-based systems achieve high scalability, elasticity, and performance by replacing the central database as a synchronous constraint in the system with replicated in-memory data grids.

Application data is kept in-memory and replicated among all the active processing units. When a processing unit updates data, it asynchronously sends that data to the database through a data pump, usually via messaging with persistent queues. Processing units start up and shut down dynamically as user load increases and decreases, which addresses variable scalability. Because no central database is involved in the application's standard transactional processing, the database bottleneck is removed. This allows for near-infinite scalability within the application.

Space-based architecture is a complicated architectural style consisting of many different artifacts that work together to process a single request. Its primary artifacts are:

Processing units

Contain the application functionality

Virtualized middleware

The collection of infrastructure-related artifacts that are used to manage and coordinate the processing units

Messaging grid

Used to manage input requests and session state

Data grid

Manages the synchronization and replication of data between processing units

Processing grid

Used to manage request orchestration between multiple processing units

Deployment manager

Manages the starting and tearing down of processing unit instances as load increases and decreases

Data pumps

Asynchronously send updated data to the database

Data writers

Perform the updates from the data pumps

Data readers

Read database data and deliver it to processing units upon startup

Figure 16-2 illustrates these primary artifacts.

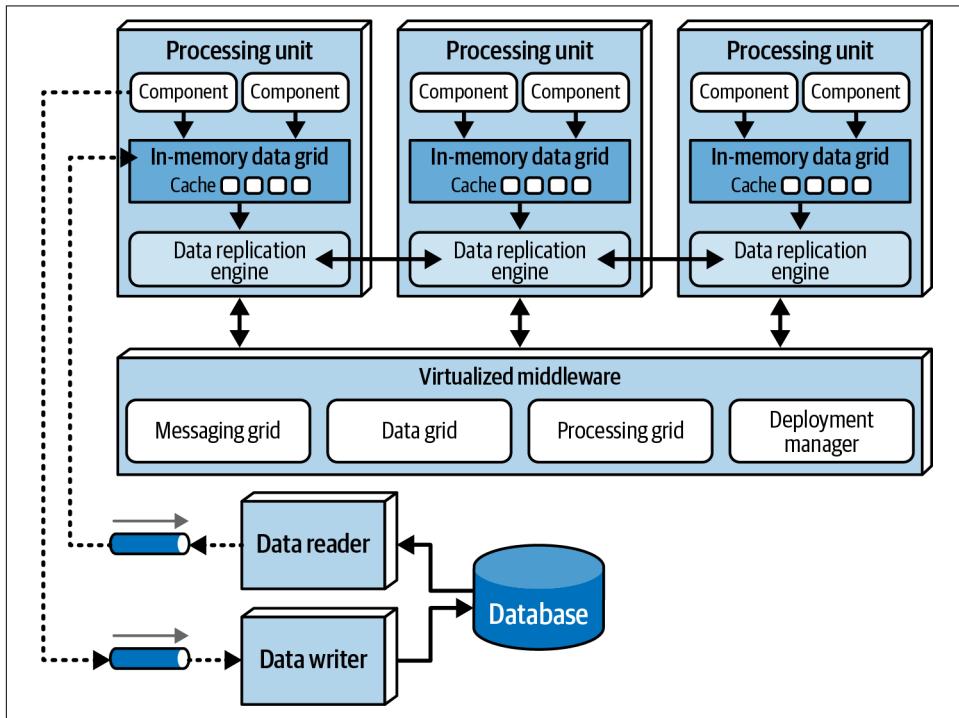


Figure 16-2. Space-based architecture's basic topology

Style Specifics

The following sections describe these primary artifacts and how they work in detail.

Processing Unit

The processing unit (illustrated in [Figure 16-3](#)) contains the application logic (or portions of it), usually including web-based components as well as backend business logic. The contents of the processing unit vary based on the type of application. Smaller web-based applications are usually deployed into a single processing unit, whereas larger applications often split their functionality into multiple processing units, based on the application's functional areas. The processing unit can also contain small, single-purpose services (much like microservices). In addition, the processing unit also contains an in-memory data grid and replication engine, which are usually implemented through such products as [Hazelcast](#), [Apache Ignite](#), and [Oracle Coherence](#) (see “[Data Grid](#)” on page 288).

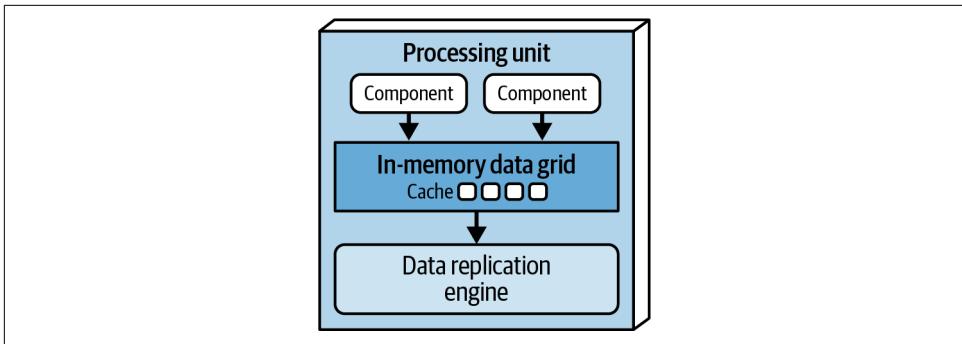


Figure 16-3. The processing unit contains the application's functionality

Virtualized Middleware

The *virtualized middleware*, as shown in [Figure 16-4](#), contains various infrastructure-related artifacts and is used to manage and control the processing units. At a minimum, this middleware artifact contains a *messaging grid* to manage input requests and user session state, a *data grid* to manage data replication and synchronization, and a *deployment manager* to start and tear down processing unit instances as they are needed and not needed. Optionally, the virtualized middleware also contains a *processing grid* in the event two or more processing units need to be orchestrated for a single business request. An architect can add any other infrastructure-related function to the virtualized middleware as needed, such as security functionality, metrics gathering for observability, and so on.

Since no single product can perform all of the functions of the virtualized middleware, it is usually implemented through third-party products such as web servers, caching tools, load balancers, service orchestrators, and deployment managers to manage monitoring, starting, and tearing down the processing units. Each of these middleware artifacts is described in detail in the following sections.

Messaging Grid

The *messaging grid*, shown in [Figure 16-4](#), is part of the virtualized middleware and manages input requests and session state. When a request comes into the virtualized middleware, the messaging grid component determines which active processing units are available to receive it, then forwards the request to one of those processing units.

The complexity of the messaging grid can vary, from a simple round-robin algorithm to a more complex next-available algorithm that keeps track of which processing unit is most available. This component is usually implemented using a typical web server with load-balancing capabilities (such as HA Proxy or Nginx).

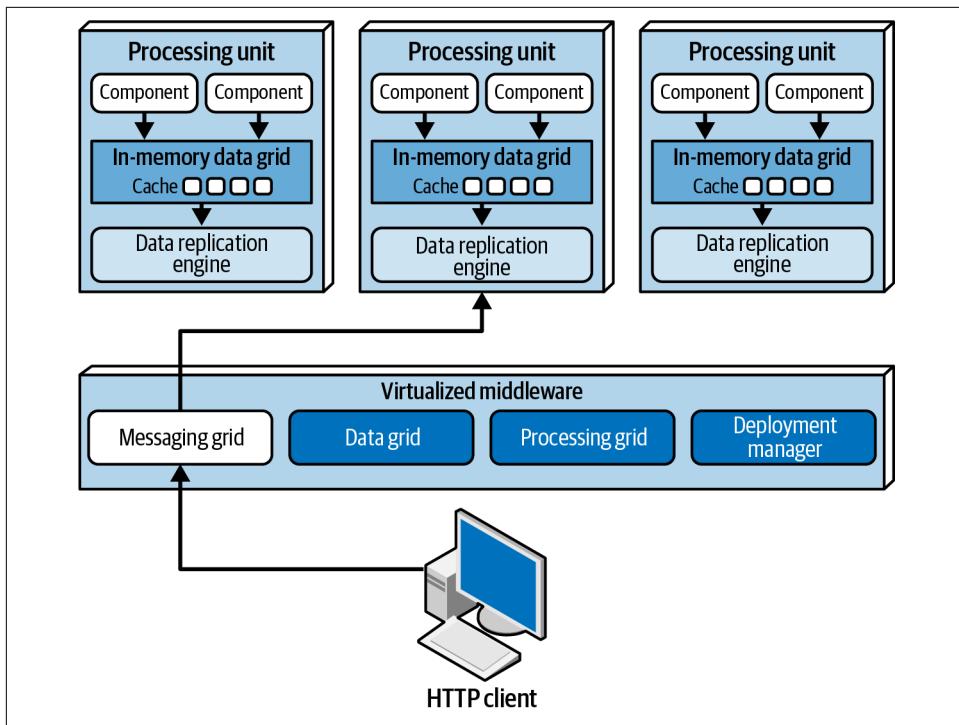


Figure 16-4. The messaging grid handles requests and session state

Data Grid

The *data grid* is a component—perhaps the most crucial component—of the virtualized middleware. In most modern implementations, the data grid is implemented solely within the processing units as an in-memory replicated cache (see “[Replicated and distributed caching](#)” on page 291). However, for replicated caching implementations that require an external controller, or that use a distributed cache, this functionality resides in *both* the processing units and the data grid components of the virtualized middleware.

Since the messaging grid can forward a request to any of the processing units available, it is essential that each processing unit’s in-memory data grid contains *exactly the same data*. As illustrated by the dotted lines in [Figure 16-5](#), data replication is typically done asynchronously between processing units, usually completing data replication in less than 100 ms.

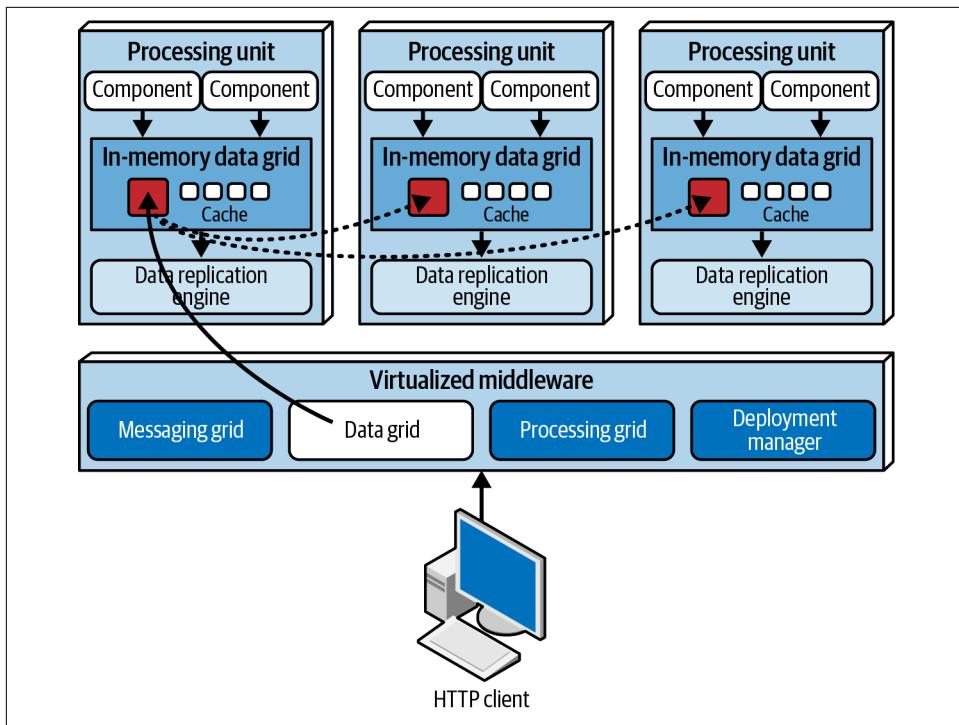


Figure 16-5. The data grid synchronizes the in-memory caches

Data is synchronized between processing units that contain the same named data grid. For example, the following Java code uses Hazelcast to create an internal replicated data grid for processing units that contain customer profile information:

```
HazelcastInstance hz = Hazelcast.newHazelcastInstance();
Map<String, CustomerProfile> profileCache =
    hz.getReplicatedMap("CustomerProfile");
```

All processing units that need access to customer-profile information should contain this code. When any processing unit updates data in the `CustomerProfile` cache, the data grid replicates the update to all other processing units that contain that same named `CustomerProfile` cache. A processing unit can contain as many in-memory replicated caches as it needs to complete its work. Alternatively, one processing unit can make a remote call to another processing unit to ask for data (choreography) or leverage the processing grid (described in the next section) to orchestrate the data request (see “[Orchestration Versus Choreography](#)” on page 375 in Chapter 20 for more information about orchestration and choreography).

Using data replication within the processing units allows additional instances to start without having to read data from the database, as long as at least one instance contains the named replicated cache. When a new processing unit instance starts,

it broadcasts a request through the caching provider (such as Hazelcast) to join other processing units with the same named cache. Once other processing units acknowledge the broadcast request and connect to the new processing unit, one of them (usually the first one to connect to the new processing unit) sends the cache data to the the new instance so it's in sync with all other instances with the same named cache.

Each instance of a processing unit contains a *member list* containing the IP addresses and ports of all other processing-unit instances containing the same named cache. For example, suppose there is a single processing-unit instance containing the customer profile functionality and its in-memory replicated cached data. In this case there is only one instance, so its member list contains only itself. This is illustrated in the following logging statements, generated using Hazelcast:

```
Instance 1:  
Members {size:1, ver:1} [  
    Member [172.19.248.89]:5701 - 04a6f863-dfce-41e5-9d51-9f4e356ef268 this  
]
```

When another processing unit starts up with the same named cache, the member lists of both services are updated to reflect the IP addresses and ports of each processing unit:

```
Instance 1:  
Members {size:2, ver:2} [  
    Member [172.19.248.89]:5701 - 04a6f863-dfce-41e5-9d51-9f4e356ef268 this  
    Member [172.19.248.90]:5702 - ea9e4dd5-5cb3-4b27-8fe8-db5cc62c7316  
]  
  
Instance 2:  
Members {size:2, ver:2} [  
    Member [172.19.248.89]:5701 - 04a6f863-dfce-41e5-9d51-9f4e356ef268  
    Member [172.19.248.90]:5702 - ea9e4dd5-5cb3-4b27-8fe8-db5cc62c7316 this  
]
```

When a third processing unit starts up, the member lists of instance 1 and instance 2 are both updated to reflect the new third instance:

```
Instance 1:  
Members {size:3, ver:3} [  
    Member [172.19.248.89]:5701 - 04a6f863-dfce-41e5-9d51-9f4e356ef268 this  
    Member [172.19.248.90]:5702 - ea9e4dd5-5cb3-4b27-8fe8-db5cc62c7316  
    Member [172.19.248.91]:5703 - 1623eadf-9cfb-4b83-9983-d80520cef753  
]  
  
Instance 2:  
Members {size:3, ver:3} [  
    Member [172.19.248.89]:5701 - 04a6f863-dfce-41e5-9d51-9f4e356ef268  
    Member [172.19.248.90]:5702 - ea9e4dd5-5cb3-4b27-8fe8-db5cc62c7316 this  
    Member [172.19.248.91]:5703 - 1623eadf-9cfb-4b83-9983-d80520cef753  
]
```

```

Instance 3:
Members {size:3, ver:3} [
    Member [172.19.248.89]:5701 - 04a6f863-dfce-41e5-9d51-9f4e356ef268
    Member [172.19.248.90]:5702 - ea9e4dd5-5cb3-4b27-8fe8-db5cc62c7316
    Member [172.19.248.91]:5703 - 1623eadf-9cfb-4b83-9983-d80520cef753 this
]

```

Now all three instances know about each other (including themselves, as denoted by the word `this` at the end of the member line). Suppose instance 1 receives a request from a customer to update their bill-to address. When instance 1 updates the cache with a `cache.put()` or similar cache update method, the data grid (such as Hazelcast) will asynchronously update the other replicated caches with the same update, ensuring that all three customer profile caches contain the new bill-to address, thus always remaining in sync with one another with the same data.

When processing-unit instances go down, all other processing-units' member lists are automatically updated to reflect the change. For example, if instance 2 goes down, the caching product immediately updates the member lists of instance 1 and 3 to remove the lost instance:

```

Instance 1:
Members {size:2, ver:4} [
    Member [172.19.248.89]:5701 - 04a6f863-dfce-41e5-9d51-9f4e356ef268 this
    Member [172.19.248.91]:5703 - 1623eadf-9cfb-4b83-9983-d80520cef753
]

Instance 3:
Members {size:2, ver:4} [
    Member [172.19.248.89]:5701 - 04a6f863-dfce-41e5-9d51-9f4e356ef268
    Member [172.19.248.91]:5703 - 1623eadf-9cfb-4b83-9983-d80520cef753 this
]

```

Replicated and distributed caching

Space-based architecture relies on caching to process transactions in applications. Space-based architecture removes the need for direct reads and writes to a database, which is how it can support high scalability, elasticity, and performance. This architecture style mostly relies on in-memory replicated caching, although it can use distributed caching as well.

With *replicated caching*, as illustrated in [Figure 16-6](#), each processing unit contains its own in-memory data grid that is synchronized between all processing units using that same named cache. When a cache within any of the processing units is updated, the other processing units are automatically updated with the new information.

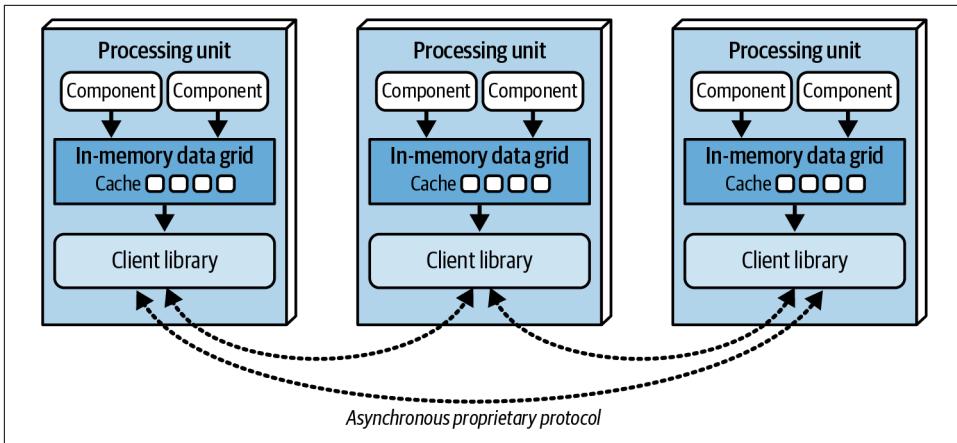


Figure 16-6. Replicated caching synchronizes in-memory caches between processing units

Not only is replicated caching extremely fast, it also supports high levels of fault tolerance. Since there is no central server holding the cache, replicated caching does not have a single point of failure.¹

While replicated caching is the standard caching model for space-based architecture, there are some cases where it can't be used. One such situation is when the system must handle high data volumes. When internal memory caches grow larger than 100 MB, each processing unit must use so much memory that it can cause issues with elasticity and scalability. Processing units are generally deployed within a virtual machine or a container (such as Docker), each of which only has a certain amount of memory available for internal cache usage. This limits the number of processing-unit instances that can be started to process high-throughput situations.

Another situation where replicated data caches don't work well is when the cache data is updated very frequently. As shown in “[Data Collisions](#)” on page 304, if the update rate of the cache data is too high, the data grid might be unable to keep up, which could harm data consistency across all processing-unit instances. When these situations occur, most architects choose to use a distributed cache instead.

¹ There may be exceptions to this rule based on the implementation of the caching product used. Some caching products require an external controller to monitor and control data replication between processing units, but most are moving away from this model.

Distributed caching, as illustrated in [Figure 16-7](#), requires an external server or service dedicated to holding a centralized cache. In this model, the processing units do not store data in internal memory. Instead, they use a proprietary protocol to access the data from the central cache server. Distributed caching supports high levels of data consistency, because the data is all kept in one place and does not need to be replicated. However, this model doesn't perform as well as replicated caching because the cache data must be accessed remotely, adding to the overall latency of the system.

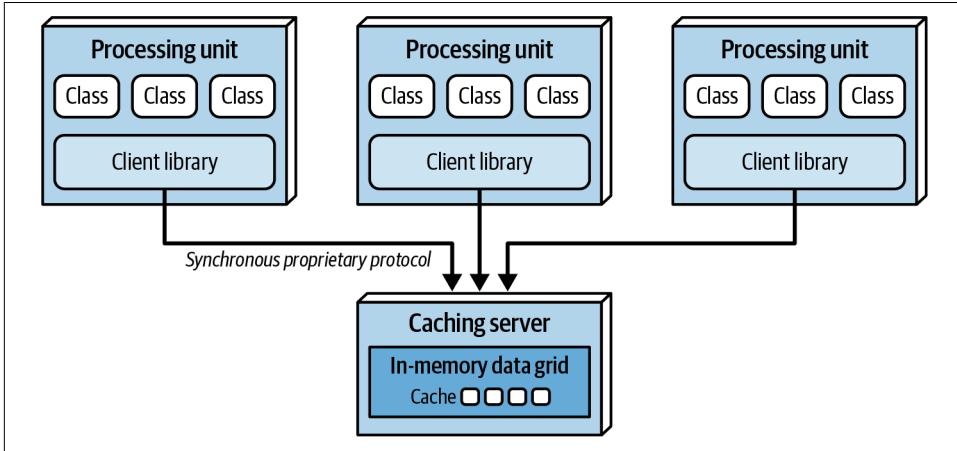


Figure 16-7. Distributed caching creates good data consistency between processing units

Fault tolerance is also an issue with distributed caching. If the cache server containing the data goes down, none of the processing units can access or update data. This single point of failure can be mitigated by mirroring the distributed cache, but this could present consistency issues if the primary cache server goes down unexpectedly and the data does not make it to the mirrored cache server.

When the size of the cache is relatively small (under 100 MB) and the update rate of the cache is low enough that the caching product's replication engine can keep up, the decision between using a replicated cache and a distributed cache becomes a question of prioritizing data consistency versus performance and fault tolerance. A distributed cache will always offer better data consistency over a replicated cache, because the data is in one place, not spread across multiple processing units. However, performance and fault tolerance will always be better with a replicated cache. Many times, the deciding factor ends up being the *type* of data being cached in the processing units. If the system's primary need is for highly consistent data (such as inventory counts of the available products), that usually warrants a distributed cache. If data does not change often (such as reference data like name/value pairs, product codes, and product descriptions), consider using a replicated cache for quick lookup. **Table 16-1** summarizes some criteria for choosing when to use a distributed versus a replicated cache.

Table 16-1. Distributed versus replicated caching

| Decision criteria | Replicated cache | Distributed cache |
|-------------------|-------------------|-------------------|
| Optimization | Performance | Consistency |
| Cache size | Small (<100 MB) | Large (>500 MB) |
| Type of data | Relatively static | Highly dynamic |
| Update frequency | Relatively low | High update rate |
| Fault tolerance | High | Low |

When choosing which caching model to use with space-based architecture, remember that in most cases, *both* models will be applicable. In other words, neither replicated caching nor distributed caching solves every problem. Different processing units can also use different models. Rather than compromising by choosing a single consistent caching model across the application, leverage each model for its strengths. For example, for a processing unit that maintains the current inventory, choose a distributed caching model for data consistency; for a processing unit that maintains the customer profile, choose a replicated cache for performance and fault tolerance.

Near-cache considerations

A *near-cache* is a hybrid caching model bridging in-memory data grids with a distributed cache. In this model (illustrated in [Figure 16-8](#)) the distributed cache is referred to as the *full backing cache*, and each in-memory data grid contained within a processing unit is called a *front cache*. The front cache always contains a smaller subset of the full backing cache, and uses an *eviction policy* to remove older items so that newer ones can be added. The front cache has three eviction policy options: a *most recently used* cache, containing the most recently used items; a *most frequently used* cache, containing the most frequently used items; or a *random replacement* eviction policy, which removes items randomly when space is needed. Random replacement is a good eviction policy when there is no clear analysis of the data providing a reason to keep either the latest used or the most frequently used.

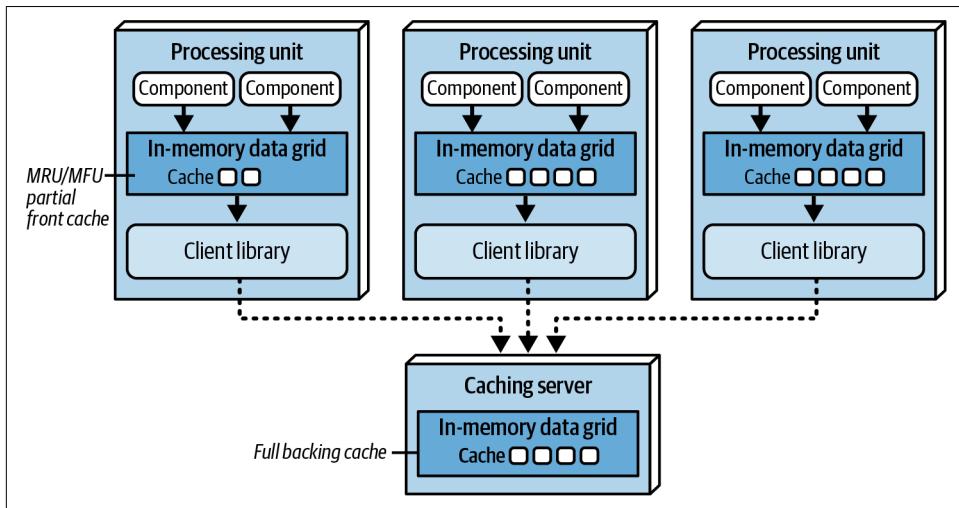


Figure 16-8. The near-cache model uses both a front cache and a backing cache

While the front caches are always kept in sync with the full backing cache, the front caches contained within each processing unit are not synchronized between other processing units that share the same data. This means that multiple processing units that share the same data context (such as a customer profile) will likely all have different data in their front caches. This creates inconsistencies in performance and responsiveness between processing units, so we do not recommend using a near-cache model in a space-based architecture.

Processing Grid

The *processing grid*, illustrated in [Figure 16-9](#), is an optional component within the virtualized middleware that manages orchestrated request processing when multiple processing units are involved in a single business request. If a request requires coordination between more than one type of processing unit (such as an order-processing unit and a payment-processing unit), the processing grid mediates and orchestrates the request between those two processing units.

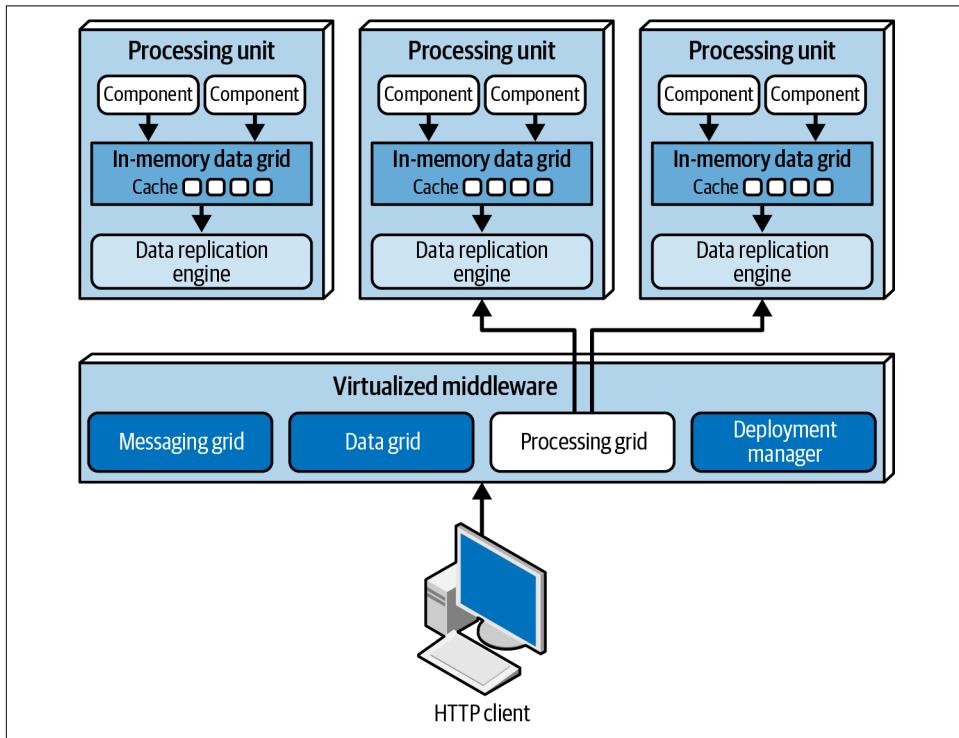


Figure 16-9. The processing grid manages orchestration between processing units

Most modern space-based implementations (particularly those with fine-grained services) implement their processing-grid functionality through separate, fine-grained *orchestration processing units* rather than a single coarse-grained orchestration engine, with each orchestration processing unit handling a single major workflow. For an ecommerce example, let's say that when a customer places an order, three processing units must coordinate: *Order Placement*, *Payment*, and *Inventory Adjustment*. The architect could create an *Order Placement Orchestrator* processing unit to orchestrate these three processing units. They might also create separate orchestration processing units for other major workflows, such as handling order returns and replenishing stock.

Deployment Manager

The Deployment Manager component manages the dynamic startup and shutdown of processing-unit instances based on load conditions. This component continually monitors response times and user loads, starts up new processing units when load increases, and shuts down processing units when the load decreases. It is critical to achieving variable scalability (elasticity) within an application. Most cloud-based infrastructures handle this responsibility, as do service-orchestration products such as [Kubernetes](#).

Data Pumps

A *data pump* is a way of sending data to another processor, which then updates a database. Space-based architectures need data pumps because processing units do not read from and write to databases directly. Data pumps in space-based architecture are always asynchronous, providing eventual consistency between the in-memory cache and the database. When a processing-unit instance receives a request and updates its cache, that processing unit becomes the owner of the update, and therefore responsible for sending it through the data pump so that the database can be updated eventually.

Data pumps are usually implemented in a space-based architecture using messaging, as shown in [Figure 16-10](#).

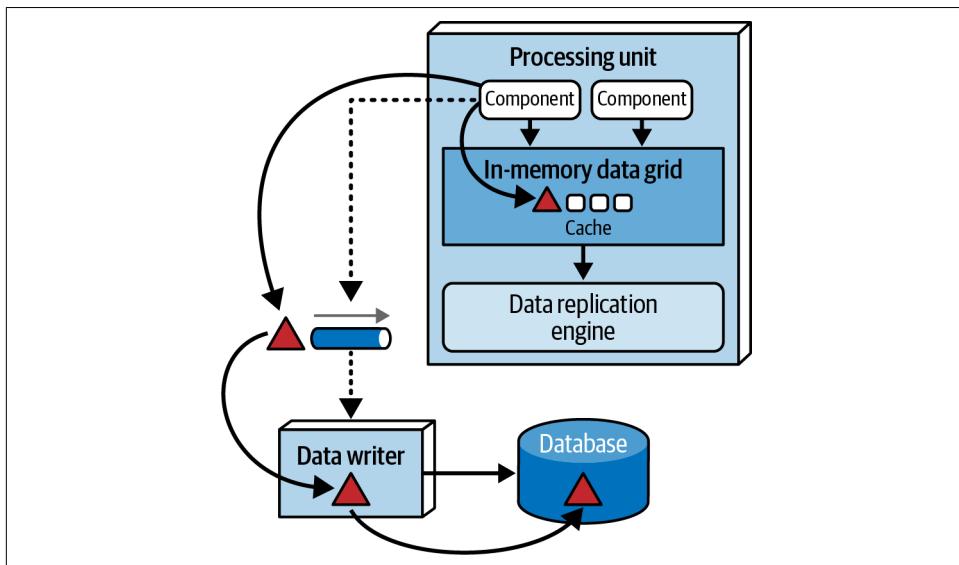


Figure 16-10. Data pumps are used to send data to a database

Messaging supports not only asynchronous communication, but also guaranteed delivery, message persistence, and message order, through first-in, first-out (FIFO) queuing. Furthermore, messaging decouples the processing unit and the data writer so that if the data writer is unavailable, processing within the processing units isn't interrupted.

Most space-based architectures have multiple data pumps. Usually each one is dedicated to a particular domain or subdomain (such as customer or inventory), but they can be dedicated to each type of cache (such as `CustomerProfile`, `CustomerWishlist`, and so on) or to a processing-unit domain (such as `Customer`) that also contains a much larger general cache.

Data pumps usually have contracts, including an action associated with the contract data (add, delete, or update). The contract can be a JSON schema, XML schema, an object, or even a *value-driven message* (a map message containing name-value pairs). For updates, the data pump's message payload usually only contains the new data values. For example, if a customer changed a phone number on their profile, only the new phone number would be sent, along with the customer ID and an action to update the data.

Data Writers

The `Data Writer` component accepts messages from a data pump and updates the database with the information in their payloads (see [Figure 16-10](#)). Data writers can be implemented as services, applications, or data hubs (such as [Ab Initio](#)). Their granularity can vary, based on the scope of the data pumps and processing units.

A *domain-based data writer* contains the necessary database logic to handle all updates within a particular domain (such as order processing), regardless of the number of data pumps it is reading from. The system in [Figure 16-11](#) has four different processing units and four different data pumps to represent the customer domains (`Profile`, `WishList`, `Wallet`, and `Preferences`)—but only one data writer. That single customer data writer listens to all four data pumps and contains the database logic (such as SQL) to update customer-related data in the database.

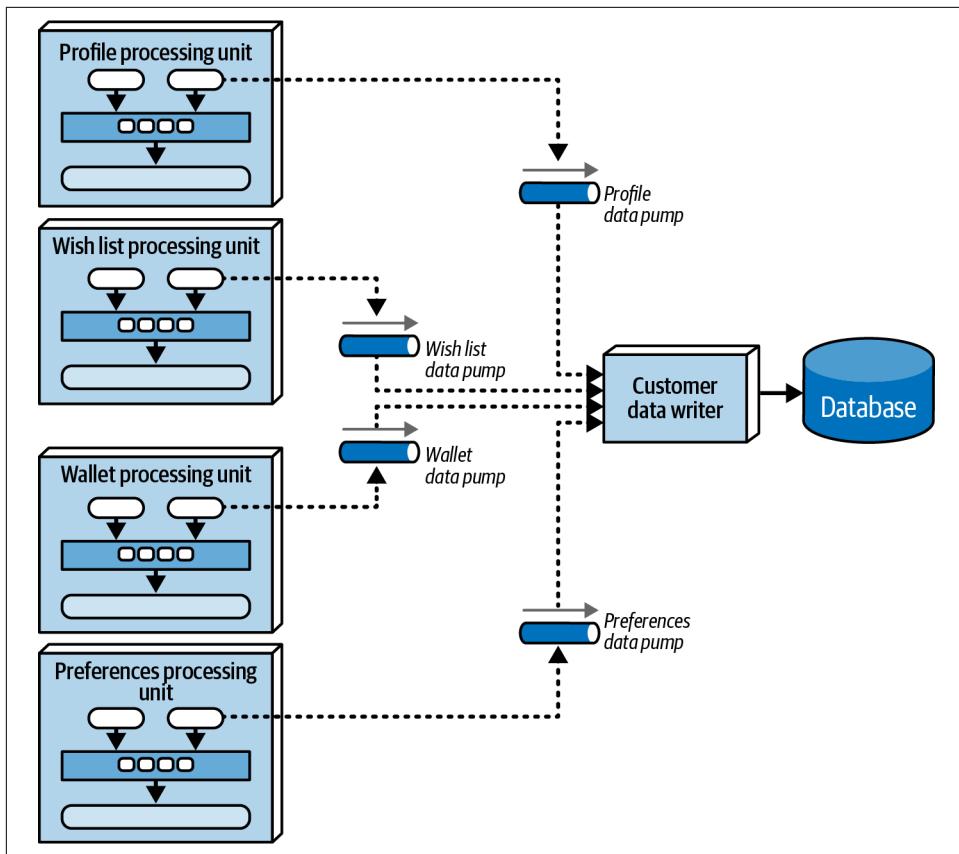


Figure 16-11. Domain-based data writer

Alternatively, each class of processing unit can have its own dedicated Data Writer component, as illustrated in [Figure 16-12](#). In this model, each data writer is dedicated to a corresponding data pump and contains only the database processing logic for that particular processing unit (such as Wallet). While this model tends to produce a lot of data-writer components, it does provide better scalability and agility because it aligns the processing unit, data pump, and data writer.

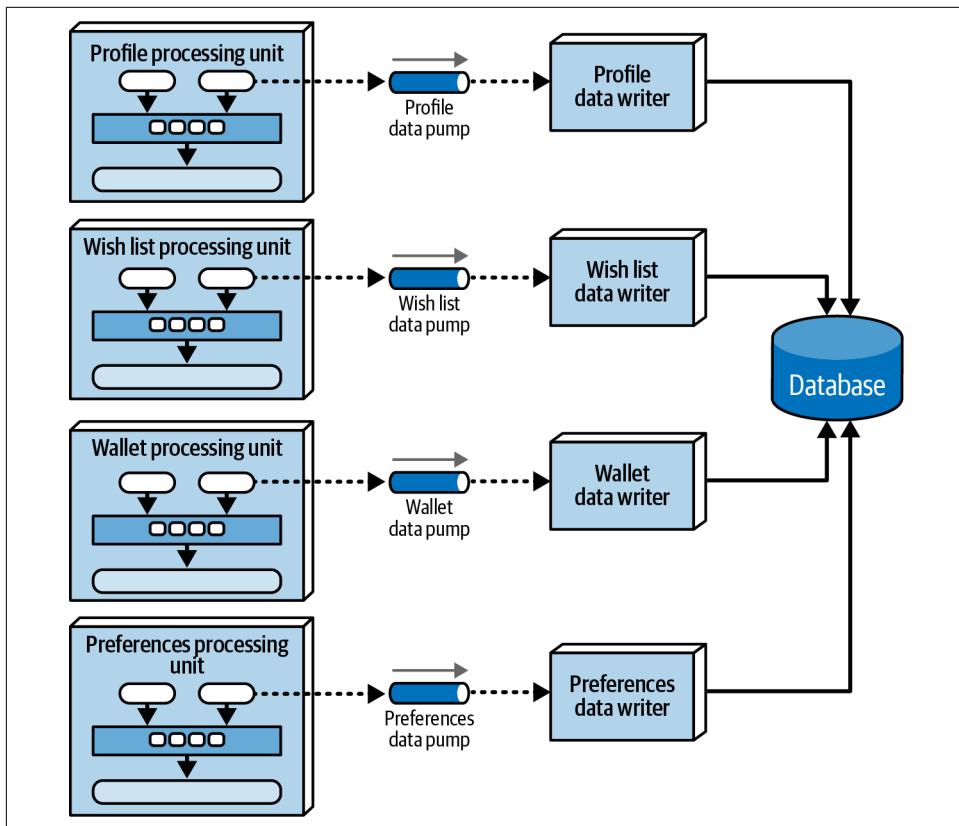


Figure 16-12. Dedicated data writers for each data pump

Data Readers

While data writers are responsible for updating the database, *data readers* read data from the database and send it to the processing units via a *reverse data pump* (which we'll discuss more in a moment). In space-based architectures, data readers are only invoked in one of three situations: all processing unit instances of the same named cache crash, all processing units within the same named cache are redeployed, or archive data not contained in the replicated cache must be retrieved.

If all instances come down due to a system-wide crash or redeployment, data must be loaded in the cache from the database—something we generally try to avoid in space-based architecture. When instances of a class of processing unit start coming back up, each one will try to grab a lock on the cache. The first one to get the lock becomes the temporary cache owner; the others go into a wait state until the lock is released. (This might vary based on the type of cache implementation, but regardless, there is one primary owner of the cache in this scenario.) To load the

cache, the temporary cache owner sends a message to a queue, requesting data. The Data Reader component accepts the read request and then performs the necessary database-query logic to retrieve the needed data. It sends that data to a different queue called a reverse data pump, which sends it to the temporary cache owner processing unit. Once it loads the cache, the temporary owner releases the lock. All other instances are then synchronized, and processing can begin. This processing flow is illustrated in [Figure 16-13](#).

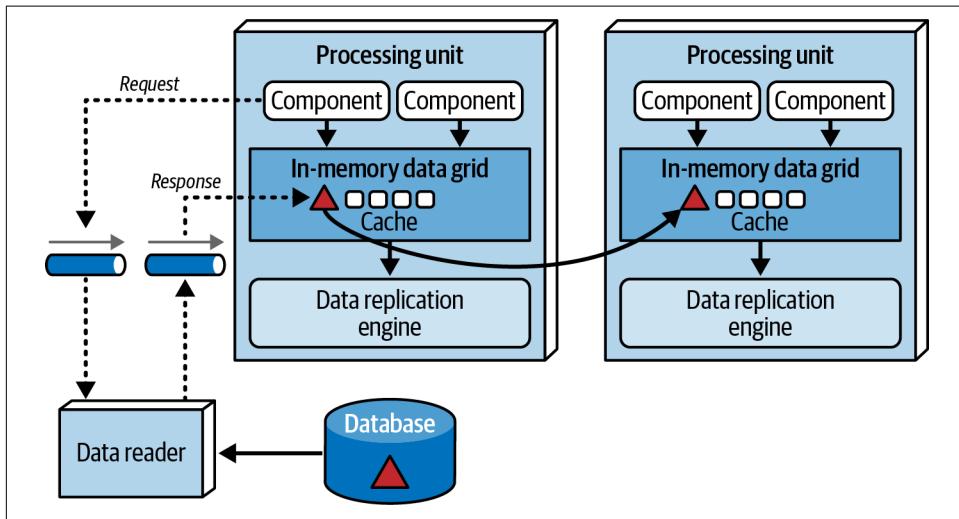


Figure 16-13. Data readers send data to the processing units

Like data writers, data readers can be domain based or dedicated to a specific class of processing unit (the latter is more common). Its implementation is also the same as the data writers—service, application, or data hub.

The data writers and data readers essentially form a *data abstraction layer* (or *data access layer*, in some cases). The difference between the two is in the amount of detailed knowledge the processing units have with regard to the database schema (the structure of the tables). With a data access layer, the processing units are coupled to the underlying data structures in the database, and only access the database indirectly, through the data readers and writers. With a data abstraction layer, on the other hand, the processing unit is decoupled from the underlying schemas through the use of separate contracts.

Space-based architecture generally relies on a data abstraction layer model so that the replicated cache schema in each processing unit can be different from the underlying database schemas. This means that incremental changes to the database don't necessarily affect the processing units. To facilitate this incremental change, the data writers and readers contain transformation logic. If a column type changes or a

column or table is dropped, the data readers and writers can buffer the database change until the necessary changes can be made to the processing-unit caches.

Data Topologies

Since processing units don't interact with the database directly, space-based architecture is tremendously flexible in the database topologies it can use. The use of asynchronous data pumps combined with data readers and writers means request processing (transactional) is largely independent of the database, giving the architect a wide variety of choices regarding the database topology and database type.

The choice of database topology is influenced by a host of factors. In a space-based architecture, the primary deciding factor is how the system will use the backing database. For example, if reporting and data analytics are especially important, a monolithic database topology might be more effective—unless the reporting and data analytics are done through a data mesh, in which case a domain-based database topology might be better.

Throughput and overall domain-based data consistency are also considerations when choosing a database topology. A single monolithic database might prove to be a bottleneck during synchronization, slowing overall synchronization time and detracting from data consistency; a domain-based database topology might offer better overall synchronization time and hence data consistency, if the data can be cleanly domain partitioned. Finally, if downstream systems need to use the database for further processing, a monolithic database topology might be a better fit.

Cloud Considerations

Space-based architecture offers some unique options when it comes to deployment environments. The entire system—including the processing units, virtualized middleware, data pumps, data readers and writers, and the database—can be deployed within cloud-based environments or on-premises (on-prem). However, this architecture style can also be deployed in *both of these environments at once* (as illustrated in [Figure 16-14](#)), a unique, powerful feature not found in other architecture styles. In this hybrid topology, applications are deployed via processing units and virtualized middleware in managed cloud-based environments, while the physical databases and corresponding data are kept on-prem. This supports very effective cloud-based data synchronization, thanks to the asynchronous data pumps and eventual consistency model of this architecture style. Transactional processing can take place in dynamic, elastic cloud-based environments, while physical data management, reporting, and data analytics stay in secure on-prem environments.

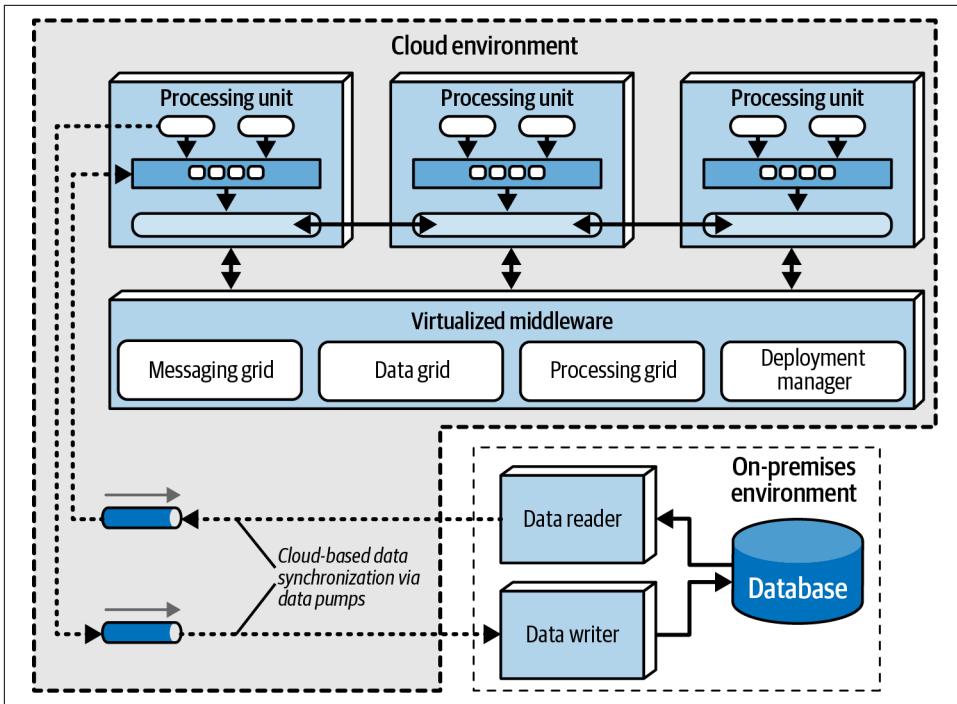


Figure 16-14. Hybrid cloud-based and on-prem topology

The elastic nature of cloud infrastructure and cloud-based services—hybrid or not—matches the shape of this architectural style well, making cloud-based environments a good choice for space-based architectures.

Common Risks

Not surprisingly, most of the risks associated with the space-based architectural style have to do with data, given its use of caching and background data synchronization. The following sections describe some common risks.

Frequent Reads from the Database

Space-based architecture achieves high scalability and concurrency by using caching for all transactional data. This prevents the system from doing excessive database reads and writes, which can lower the system's overall scalability, elasticity, and responsiveness, making it difficult to realize the benefits of this architectural style.

Database reads typically only occur in two scenarios: reading archived data (such as order history or past bank statements), or *cold-starting* a processing unit (the initial start of a processing unit when no other instances are running). If the cached data

volumes are so high that most data needs to be archived and retrieved from the backing database, or if processing units crash or are redeployed frequently, this might not be the right architecture for the problem domain.

Data Synchronization and Consistency

Because data pumps and data writers synchronize data between the in-memory cache and the database, data will always be eventually consistent in a space-based architecture. However, because this style is typically used in situations with a very high concurrent user load, bottlenecks in the data pump are common. These bottlenecks can significantly delay data from making it to the backing database. This can be a significant risk if downstream systems need to have updated data quickly.

Another inherent risk related to data synchronization is data loss within the data pump. This risk is usually mitigated by using *persisted queues* (in which data in a queue is stored on disk as well as in memory) and using client-acknowledgment mode in the data writers when reading from the data pump. Client-acknowledgment mode keeps the message in the queue until the data writer acknowledges to the message broker that it has completed processing. During message processing, the message broker ensures that no other data writer can read the in-process message. While these techniques help prevent data loss in the data pump, they can also slow down overall responsiveness and degrade data consistency.

High Data Volumes

Because all transactional memory is cached in the processing units, data volumes need to remain relatively low, particularly as more instances of a processing unit are added. This makes it critical to pay close attention to the size of the in-memory cache to avoid a processing unit running out of memory and hence crashing.

Data Collisions

A *data collision* occurs when data is updated in one cache instance (cache A), and during replication to another cache instance (cache B), the same data is updated by that cache (cache B). This can happen when using replicated caching in an *active/active state*, meaning that multiple processing units can update the same data at the same time. Collisions typically occur when the update rate of the data in the cache exceeds the *replication latency (RL)*—the time it takes to synchronize each same-named cache. In this scenario, the local update to cache B will be overridden by the old data from cache A, and the same data in cache A will be overridden by the update from cache B. This makes the data in each cache inconsistent.

To demonstrate the data collision problem, let's assume there are two service instances (A and B) of an order placement service, each containing a replicated cache of product inventory (blue widgets). The flow is as follows:

1. The current inventory count is 500 units in each instance A and B.
2. Instance A receives a request from a customer to purchase 10 units, and updates the inventory cache for blue widgets to 490 units.
3. Before instance A's data can be replicated to instance B, instance B receives a purchase request for 5 units, and updates the inventory cache for blue widgets to 495 units.
4. The cache in instance B gets updated to 490 units due to replication from instance A's update.
5. The cache in instance A gets updated to 495 units due to replication from instance B's update.
6. Both caches are incorrect and out of sync: the inventory should be 485 units in each of the instances.

Several factors that influence how many data collisions might occur: the number of processing-unit instances that contain the same cache, the cache's update rate, the cache's size, and the RL of the caching product. We can probabilistically determine how many potential data collisions might occur based on these factors, using the following formula:

$$\text{CollisionRate} = N * \frac{UR^2}{S} * RL$$

Here, N represents the number of service instances using the same named cache. UR represents the update rate in milliseconds (squared), S is the cache size (in number of rows), and RL is the caching product's replication latency (in milliseconds).

This formula is useful for determining the percentage of data collisions that will likely occur based on updates within a given time frame (for example, each hour), and thus how feasible it would be for this system to use replicated caching. For example, consider the values shown in [Table 16-2](#) for the factors involved in this calculation.

Table 16-2. Base values

| | |
|---------------------------|-------------------|
| Update rate (UR): | 20 updates/second |
| Number of instances (N): | 5 |
| Cache size (S): | 50,000 rows |
| Replication latency (RL): | 100 milliseconds |
| Updates: | 72,000 per hour |
| Collision rate: | 14.4 per hour |
| Percentage: | 0.02% |

Applying these factors to the formula yields 72,000 updates an hour, with a high probability that 14 updates to the same data *may* collide. Given the low percentage (0.02%), replication would be a viable option.

Variations in RL can have a significant impact on data consistency. Replication latency depends on many factors, including the type of network and the physical distance between processing units. RL values must be calculated and derived from actual measurements in a production environment, which is why they're rarely published. The value of 100 ms used in the prior example is a good planning number if the actual RL is unavailable. For example, changing the RL from 100 ms to 1 ms yields the same number of updates (72,000 per hour) but produces a much lower probability of collisions occurring (0.1 collisions per hour). This scenario is shown in [Table 16-3](#).

Table 16-3. Impact on replication latency

| | |
|---------------------------|----------------------------------|
| Update rate (UR): | 20 updates/second |
| Number of instances (N): | 5 |
| Cache size (S): | 50,000 rows |
| Replication latency (RL): | 1 millisecond (changed from 100) |
| Updates: | 72,000 per hour |
| Collision rate: | 0.1 per hour |
| Percentage: | 0.0002% |

The number of processing units that contain the same named cache (as represented by N) also has a direct proportional relationship to the number of possible data collisions. For example, reducing the number of processing units from 5 instances to 2 instances yields a data-collision rate of only 6 per hour, out of 72,000 updates per hour, as shown in [Table 16-4](#).

Table 16-4. Impact on the number of processing-unit instances

| | |
|---------------------------|--------------------|
| Update rate (UR): | 20 updates/second |
| Number of instances (N): | 2 (changed from 5) |
| Cache size (S): | 50,000 rows |
| Replication latency (RL): | 100 milliseconds |
| Updates: | 72,000 per hour |
| Collision rate: | 5.8 per hour |
| Percentage: | 0.008% |

The cache size is the only factor that is inversely proportional to the collision rate: as the cache size decreases, collision rates increase. In this example, reducing the cache size from 50,000 rows to 10,000 rows (and keeping everything else the same as in the first example) yields a collision rate of 72 per hour, significantly higher than with 50,000 rows, as shown in [Table 16-5](#).

Table 16-5. Impact on cache size

| | |
|---------------------------|-----------------------------------|
| Update rate (UR): | 20 updates/second |
| Number of instances (N): | 5 |
| Cache size (S): | 10,000 rows (changed from 50,000) |
| Replication latency (RL): | 100 milliseconds |
| Updates: | 72,000 per hour |
| Collision rate: | 72.0 per hour |
| Percentage: | 0.1% |

Under normal circumstances, most systems do not have consistent update rates over a long period of time (such as an eight-hour day). When using this calculation, we recommend understanding your system's maximum update rate during peak usage and calculating minimum, normal, and peak collision rates.

Governance

Space-based architecture, with its many moving parts, is a complicated architectural style to design and implement. Proper governance is critical to ensuring success—especially for controlling memory consumption, given this style's issues with internal memory usage (as we noted earlier, in [“Common Risks” on page 303](#)).

To address these memory issues, we recommend writing a continuous, automated governance fitness function to have each instance of a processing unit make its current memory usage observable periodically. Since all instances of a processing unit have the same replicated cache, the fitness function only needs to report on the name of the processing unit. Use a separate fitness function to record the number of instances for each processing unit, allowing you to calculate the total memory consumption per processing unit. [Figure 16-15](#) shows an example of this continuous fitness function's output.

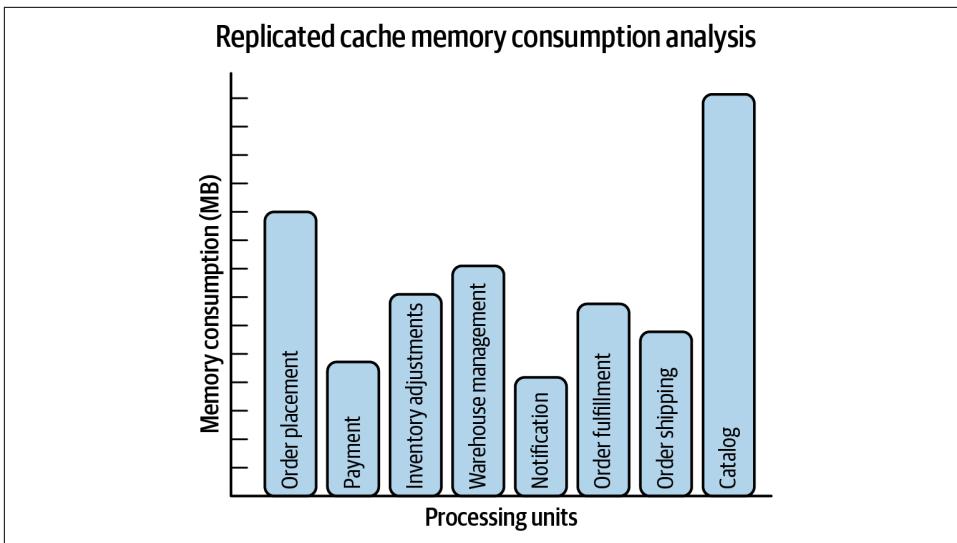
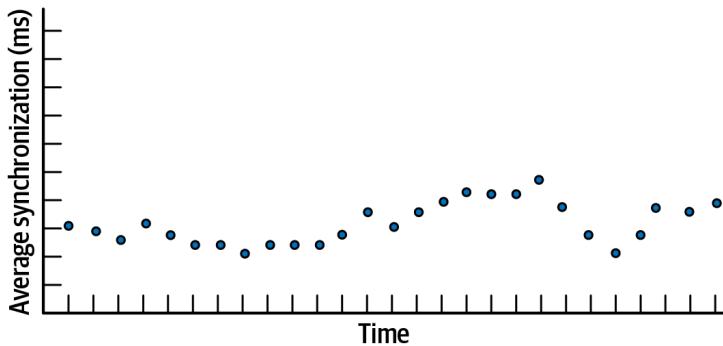


Figure 16-15. An example fitness function to track memory consumption

Another useful governance strategy to maintain overall data consistency is to track and measure synchronization time—specifically, how long it takes a cache update to sync to the corresponding database. A good method is to have each processing unit stream the request ID of an update along with its corresponding timestamp, and have each data writer stream the same request ID along with the timestamp after the database commit. Then, write a fitness function to associate these request IDs and subtract the timestamps to calculate the synchronization time. The fitness function could track this at an atomic level, using times associated with a particular processing unit, or holistically, by averaging the overall synchronization times. Analyzing these trends helps architects track the effects of changes to the architecture, such as whether the changes are making synchronization times better or worse, and whether the architecture is meeting the business's synchronization timing goals. [Figure 16-16](#) illustrates this kind of governance through a continuous fitness function.

Average cache to database synchronization analysis



Other governance fitness functions for space-based architecture could track the frequency of reads to the database (requests to data readers), which can affect scalability, elasticity, and responsiveness. It's also a good idea to use fitness functions to measure scalability, elasticity, and responsiveness: since these architectural characteristics are the main reasons to use a space-based architecture, it makes sense to track and measure them.

Team Topology Considerations

Although space-based architecture is largely considered a technically partitioned architecture due to the many artifacts that make up any particular domain or sub-domain, it's most effective with technically partitioned teams that are aligned with its technical areas (such as functionality, data pumps, data readers and writers, and backend database management). However, it can still work well when teams are aligned by domain area (such as cross-functional teams with specialization). Here are some things to consider about aligning the specific team topologies outlined in “[Team Topologies and Architecture](#)” on page 151 with space-based architecture:

Stream-aligned teams

Depending on the size of the system, stream-aligned teams might find themselves struggling to implement domain-based changes based on the technical partitioning in this architectural style. For example, a stream-based change might impact one or more processing units, data pumps, data readers, data writers, cache contracts, or orchestrators, as well as the backing database. This can be a lot for a single stream-based team to manage, particularly if those artifacts are shared by other teams. The larger and more complex the system, the less effective stream-aligned teams will be with space-based architecture.

Enabling teams

Because some of this style's artifacts (data pumps, data readers, data writers, and virtualized middleware) may be shared or cross-cutting, it's a good fit for enabling teams. Dedicating a team to a particular artifact (such as a data writer and its corresponding data pump) can allow its members to experiment and find ways of making these artifacts more efficient, independent of the team working on the primary functionality in a particular processing unit.

Complicated-subsystem teams

Complicated-system teams can leverage the space-based architecture style's technically partitioned nature to focus on one part of the system (such as the data grid or data pumps). Some of these artifacts can get extremely complex, so they lend themselves well to the complicated-subsystem team topology. Dealing with data collisions (see “[Data Collisions](#)” on page 304) and other sorts of asynchronous data synchronization errors in the data writers is quite complex; this is a

great example of a complicated subsystem that functional domain-based teams working on processing units shouldn't need to concern themselves about.

Platform teams

As with most architectural styles, developers working on a space-based architecture can leverage the benefits of the platform-team topology by utilizing common tools, services, APIs, and tasks, especially if the infrastructure-related parts of the architecture (such as the data pumps and virtualized middleware) are considered platform related.

Style Characteristics

A one-star rating in the characteristics ratings table in [Figure 16-18](#) means the specific architecture characteristic isn't well supported in the architecture, whereas a five-star rating means the architecture characteristic is one of the strongest features in the architecture style. Definitions for each characteristic identified in the scorecard can be found in [Chapter 4](#).

| | Architectural characteristic | Star rating |
|--------------|------------------------------|-------------|
| Overall cost | | \$\$\$\$ |
| Structural | Partitioning type | Technical |
| | Number of quanta | 1 to many |
| | Simplicity | ★ |
| | Modularity | ★★★ |
| Engineering | Maintainability | ★★★ |
| | Testability | ★ |
| | Deployability | ★★★ |
| | Evolvability | ★★★ |
| Operational | Responsiveness | ★★★★★ |
| | Scalability | ★★★★★ |
| | Elasticity | ★★★★★ |
| | Fault tolerance | ★★ |

Figure 16-18. Space-based architecture characteristics ratings

Space-based architecture maximizes elasticity, scalability, and performance—hence we've given all of these characteristics five-star ratings. These are the driving characteristics and main advantages of this architecture style. Leveraging in-memory data caching and removing the database as a constraint can give systems built with this architecture style the high levels of elasticity, scalability, and performance they need to process millions of concurrent users.

The trade-offs for these advantages involve the system's overall simplicity and testability. Space-based architectures are very complicated, due to their use of caching and eventual consistency of the primary data store, which is the ultimate system of record. Given this style's numerous moving parts, architects should take special care to avoid data loss in the event of a crash (see "[Preventing Data Loss](#)" on page 253 in [Chapter 15](#)).

Testability gets a one-star rating due to the complexity of simulating the high levels of scalability and elasticity this style supports. Testing hundreds of thousands of concurrent users at peak load is a very complicated and expensive task, so most high-volume testing occurs within production environments, with actual extreme load, incurring significant risk for normal operations.

Cost is another trade-off. Space-based architecture is relatively expensive, mostly due to its overall complexity, licensing fees for caching products, and the resource utilization needed within cloud and on-prem systems to support its high scalability and elasticity.

While processing units, being separately deployed, might lend themselves to a level of domain partitioning, we have identified space-based as a technically partitioned architecture, since any given domain is represented by different technical components, such as processing units, data pumps, data readers and writers, and the database.

The number of quanta within a space-based architecture can vary, based on how the UI is designed and how processing units communicate. Because the processing units do not communicate synchronously with the database, the database itself is not part of the quantum equation. As a result, quanta within a space-based architecture are typically delineated through the associations between the various UIs and the processing units. Processing units that synchronously communicate (with each other or through the processing grid for orchestration) would all be part of the same architectural quantum.

Examples and Use Cases

Space-based architecture is well suited for applications that experience high spikes in user or request volume and applications that have throughput in excess of 10,000 concurrent users. We'll look at two space-based architecture use cases here: an online concert-ticketing application and an online auction system. Both require high levels of performance, scalability, and elasticity.

Concert Ticketing System

Concert ticketing systems are a unique problem domain, in that concurrent user volume is relatively low until a popular concert is announced. Once tickets for an especially popular entertainer go on sale, user volumes usually spike from several hundred concurrent users to several thousand or even tens of thousands, depending on the concert, all trying to acquire good seats! Tickets usually sell out in a matter of minutes, so these systems really require the characteristics supported by space-based architecture.

There are many challenges associated with this sort of system. First, only a certain number of tickets are available in total, regardless of the seating preferences. Seating availability must be updated continually and as fast as possible, given the high number of concurrent requests. Continually accessing a central database synchronously would be unlikely to work—it would be very difficult for a typical database to handle tens of thousands of concurrent requests through standard transactions at this scale and with this update frequency.

Space-based architecture would be a good fit for a concert ticketing system, due to its high elasticity requirements. The *deployment manager* would immediately recognize a sudden increase in the number of concurrent users and start up lots of processing units to handle the large volume of ticket purchase requests. Optimally, the deployment manager could be configured to start the necessary number of processing units shortly *before* tickets go on sale, so they would be on standby right before the significant increase in user load.

Online Auction System

Online auction systems (sites for bidding on items within an auction, such as eBay) share many characteristics with the online concert-ticketing systems we just described. Both require high levels of performance and elasticity, and both have unpredictable spikes in user and request load. When an auction starts, there is no way to determine how many people join or how many concurrent bids will occur for each asking price.

Space-based architecture is well suited for this problem domain because it allows for multiple processing units to be started as the load increases, then destroyed as

the auction winds down and they're no longer needed. Individual processing units can be devoted to each auction, ensuring consistency in the bidding data. Also, the asynchronous nature of the data pumps means that bidding data can be sent to other processing (such as bid history, bid analytics, and auditing) without much latency, increasing the overall performance of the bidding process.

Space-based architecture is a complicated but very powerful architectural style. It is the only architectural style that maximizes the combination of responsiveness, scalability, and elasticity, primarily because of how it uses caching and its lack of direct database access. As such, it can be considered a specialized architectural style, used in situations that must maximize these particular architectural characteristics.

Orchestration-Driven Service-Oriented Architecture

Architecture styles make sense to architects in the context of the era when they evolved, but lose relevance in later eras, much like art movements. *Orchestration-driven service-oriented architecture* (SOA) exemplifies this tendency. The external forces that often influence architecture decisions, combined with a logical but ultimately disastrous organizational philosophy, doomed this architecture to irrelevance. However, it provides a great example of how a particular organizational idea can make logical sense yet hinder the most important parts of the development process. It illustrates one of the dangers of ignoring our First Law: *everything in software architecture is a trade-off*.

Topology

The topology of orchestration-driven SOA is shown in [Figure 17-1](#).

Not all examples of this style of architecture have these exact layers, but they all follow the same idea of establishing a taxonomy of services within the architecture, each layer with a specific, well-defined responsibility.

Service-oriented architecture is a distributed architecture. The exact demarcation of boundaries isn't shown in [Figure 17-1](#) because it varies based on organization and tools: some of the parts of the taxonomy may exist inside an application server. Orchestration-driven SOA centers on a specific taxonomy of services, with different technical responsibilities within the architecture and roles dedicated to specific layers.

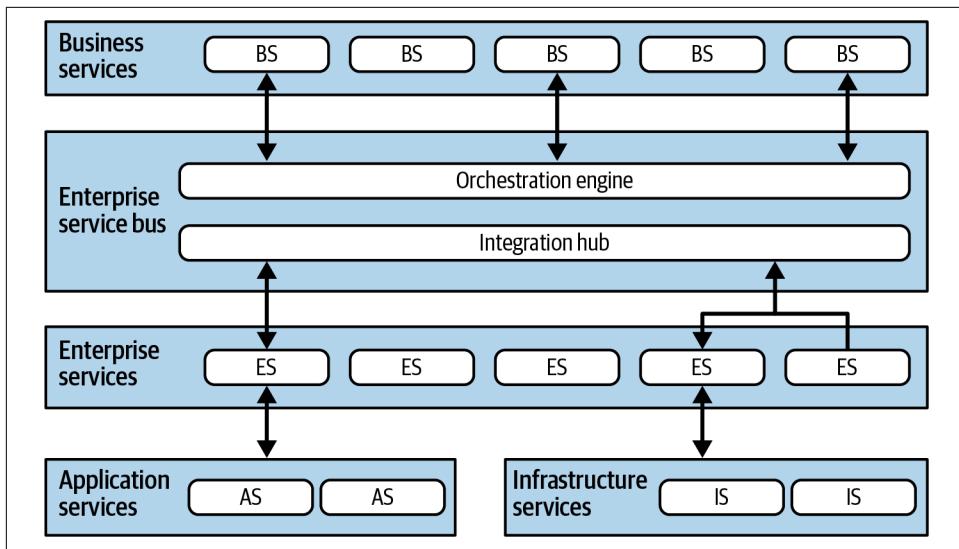


Figure 17-1. Topology of orchestration-driven service-oriented architecture

Style Specifics

Orchestration-driven SOA is mostly of historical interest to current-day architects; the lessons learned from building these architectures are an important part of the field's evolution. However, architects still find parts of it relevant in some integration architecture scenarios.

Service-oriented architecture appeared in the late 1990s, just as small companies of all kinds were growing into enterprises at a breakneck pace, merging with smaller companies, and requiring more sophisticated IT to accommodate this growth. However, computing resources were scarce, precious, and commercial. Distributed computing had just become possible and necessary, and many companies needed its scalability and other beneficial characteristics.

Many external drivers forced architects in this era toward distributed architectures with significant system constraints. Before open source operating systems were thought reliable enough for serious work, operating systems were expensive and licensed per machine. Similarly, commercial database servers came with Byzantine licensing schemes, which sometimes caused application-server vendors (which offered database connection pooling) to react by battling with database vendors. Given that so many resources were expensive at scale, architects adopted a common philosophy of reusing as much as possible.

These technical concerns wedged with organizational concerns about duplication of both information and workflows—because of frequent mergers and growth,

organizations struggled with the variety and inconsistencies between core business entities. This made the goals of SOA attractive. Consequently, architects embraced *reuse* in all forms as the dominant philosophy in this architecture, the side effects of which we cover in “[Reuse...and Coupling](#)” on page 320. This style of architecture also exemplifies how far architects can push the idea of technical partitioning, which can have good motivations but leads to bad consequences when taken to extremes.

Why So Many Service Names?

Architects are often confused by the multitude of things in software architecture called *services*. We cover three different “services” styles even in this book: SOA in this chapter, microservices in [Chapter 18](#), and service-based architectures in [Chapter 14](#). Part of the problem is the inflexibility of language to describe architectures. The other part lies in the constant evolution of the software-development ecosystem. *Service* is a nice generic name for something that provides a, well, service, so architects tend to reuse the name. As styles evolve, we change what we mean by *service*. For example, an *entity service* in an orchestration-driven SOA is different in virtually every way from a service in a microservices architecture (which is distinct from a service-based architecture). As annoying as it is, architects must often parse the context when the word *service* appears in a name; the term itself has suffered from [semantic diffusion](#).

Taxonomy

The driving philosophy behind this architecture is a specific type of abstraction and enterprise-level reuse. Many large companies were annoyed at how much they had to continually rewrite software. They arrived at a strategy that seemed to solve that problem gradually by creating a strict *service taxonomy*, with well-defined layers and corresponding responsibilities. Each layer of the taxonomy supports the dual goals of ultimate abstraction and reuse.

Business services

Business services sit at the top of this SOA and provide the entry point for business processes. For example, services like `ExecuteTrade` or `PlaceOrder` represent the correct scope of behavior for these services. One litmus test common at the time was: can an architect answer “yes” to the question “Are we in the business of...” for each of these services? If so, then the service is at the appropriate level of granularity. However, while a developer might need a method like `CreateCustomer` to carry out a business process like `ExecuteTrade`, `CreateCustomer` is at the wrong level of abstraction for a business service. The company isn’t in the *business* of creating customers, but it needs to create customers in order to execute trades.

These service definitions contain no code—just input, output, and sometimes schema information. Business users and/or analysts define these service signatures, hence the name *business services*.

Enterprise services

The *enterprise services* contain fine-grained shared implementations. Typically, a team of developers builds atomic behavior around particular business domains, such as `CreateCustomer` or `CalculateQuote`, and transactional entities, such as `Customer`, `Order`, and `Lineitem`. These enterprise services are the building blocks that make up the business services, tied together via the orchestration engine.

It is worth noting the abstraction differences between business and enterprise services. While business services are quite coarse-grained, enterprise services are fine-grained and meant to capture different types of abstractions, workflows, and entities. The architect's goal in creating enterprise services is to create perfectly encapsulated building blocks of isolated business functionality that can be freely composed into more complex business workflows.

While that's a laudable goal, architects find that the ideal sweet spot of abstraction between all these forces is elusive at best and likely impossible because of numerous competing trade-offs. Ultimately, like other technically partitioned architectures, this architecture attempts a strict separation of responsibility, which flows from the imperative of reuse. The idea is that if developers can build fine-grained enterprise services at just the correct level of granularity, the business won't have to rewrite that part of the business workflow again. Gradually, the business will build up a collection of reusable assets in the form of reusable enterprise services—in theory, anyway.

Unfortunately, the dynamic nature of reality and the evolutionary impact of the software development ecosystem defy these attempts. Business components aren't like construction materials, where solutions last decades. Markets, technology changes, engineering practices, and a host of other factors confound attempts to impose stability on the software world.

Application services

Not all services in the architecture require the same level of granularity or reuse as the enterprise services. *Application services* are one-off, single-implementation services. For example, perhaps one application needs geolocation, but the organization doesn't want to take the time or effort to make that a reusable service. An application service, typically owned by a single application team, solves that problem.

Infrastructure services

Infrastructure services supply operational concerns, such as monitoring, logging, authentication, authorization, and so on. These services tend to be concrete implementations, owned by a shared infrastructure team that works closely with operations. Architects' philosophy in building this architecture revolves around technical partitioning, so it makes sense that they would build separate infrastructure services.

Orchestration engine and message bus

The *orchestration engine* forms the heart of this distributed architecture, stitching together the business service implementations using orchestration, including features like transactional coordination and message transformation. The orchestration engine defines the relationship between the business and enterprise services, how they map together, and where transaction boundaries lie. It also acts as an integration hub, allowing architects to integrate custom code with package and legacy software systems. This combination of features highlights the modern-day use of tools like enterprise service buses (ESBs). While most architects consider it a bad idea to build an entire architecture around ESBs, they are immensely useful in integration-heavy environments. Where architects must combine an integration hub and orchestration engine, why not use a tool that already includes them? (This points to another important skill to develop as an architect—how to discern the true uses of tools, separate from the hype, both good and bad.)

Because the message bus forms the heart of the architecture, Conway's Law (see "[Conway's Law](#)" on page 139) correctly predicts that the team of integration architects responsible for this engine tends to become a political force within an organization—and eventually a bureaucratic bottleneck.

While this centralized, taxonomized approach might sound appealing, in practice it has mostly been a disaster. Offloading transaction behavior to an orchestration tool sounds good, but architects struggle to find the correct level of granularity. While they can build a few services wrapped in a distributed transaction, the architecture becomes increasingly complex. Developers must figure out where the appropriate transaction boundaries between services lie as entities become involved in numerous workflows. Despite managers predicting and hoping that organizations could successfully build transactional building blocks as enterprise services, it has proved difficult in practice.

Message flow

All requests go through the orchestration engine, where this architecture's logic resides. Thus, message flow goes through the engine even for internal calls, as shown in [Figure 17-2](#).

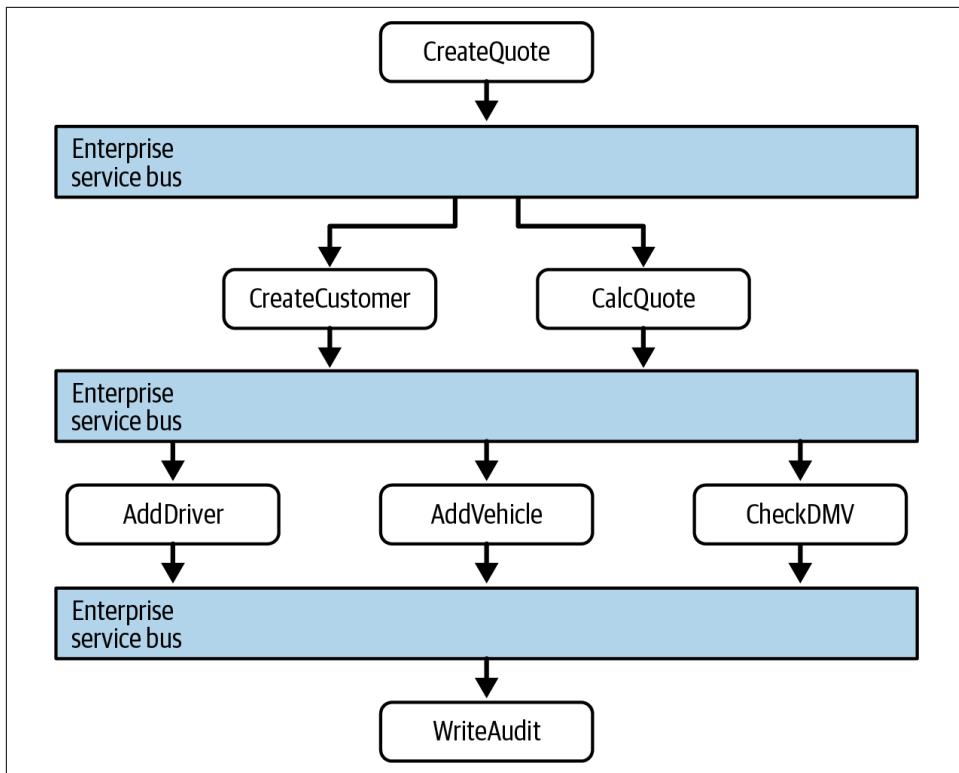


Figure 17-2. Message flow with service-oriented architecture

In [Figure 17-2](#), the `CreateQuote` business-level service calls the service bus, which defines the workflow. The workflow consists of calls to both `CreateCustomer` and `CalculateQuote`, each of which also makes calls to application services. The service bus acts as the intermediary for all calls within this architecture, serving as both an integration hub and orchestration engine.

Reuse...and Coupling

One of the primary goals of the architects who first utilized this architecture was reuse at the service level—the ability to gradually build business behavior that they could incrementally reuse over time. They were instructed to find reuse opportunities as aggressively as possible.

For example, consider the situation illustrated in [Figure 17-3](#). An architect realizes that each of six divisions within an insurance company contains a notion of Customer.

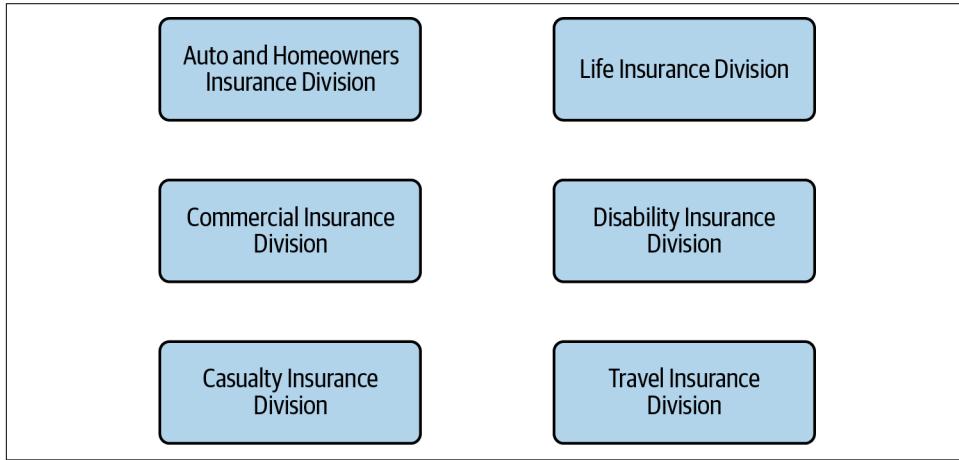


Figure 17-3. Seeking reuse opportunities in service-oriented architecture

Therefore, the proper SOA strategy entails extracting the Customer parts into a reusable service and then allowing the original services to reference the canonical Customer service. This is shown in [Figure 17-4](#): here the architect has isolated all customer behavior into a single Customer service, achieving the obvious reuse goals.

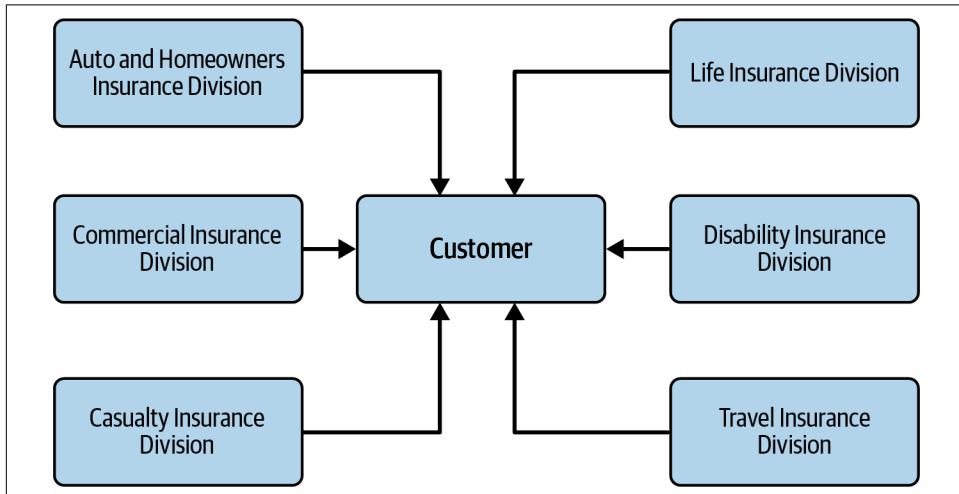


Figure 17-4. Building canonical representations in service-oriented architecture

Architects only slowly realized the negative trade-offs of this design. First, when a team builds a system primarily around reuse, it also incurs a huge amount of coupling between components. After all, reuse is implemented *via* coupling. For example, in [Figure 17-4](#), a change to the Customer service ripples out to all the other services. This makes even incremental change risky: each change has a potentially huge ripple effect. That in turn requires coordinated deployments, holistic testing, and other drags on engineering efficiency.

Another negative side effect of consolidating behavior into a single place: consider the case of auto and disability insurance in [Figure 17-4](#). To support a single Customer service, each division must include all the details the organization knows about its customers. Auto insurance requires a driver's license, which is a property of the person, not the vehicle. Therefore, the Customer service will have to include details about driver's licenses that the disability insurance division cares nothing about. Yet the disability insurance team must deal with the extra complexity of a single customer definition. In many ways, DDD's insistence on *avoiding* holistic reuse derives from experiences with these kinds of architectures.

Perhaps the most damaging revelation about orchestration-driven SOA is the impracticality of building an architecture so focused on technical partitioning. While this makes sense from a separation and reuse philosophy standpoint, it is a practical nightmare.

For example, developers commonly work on tasks like “add a new address line to `CatalogCheckout`.” Domain concepts like `CatalogCheckout` are spread so thinly throughout this architecture that they were virtually ground to dust. In an SOA, this task could involve dozens of services in several different tiers, plus changes to a single database schema. What’s more, if the current enterprise services aren’t defined at the correct transactional granularity, the developers will either have to change their design or build a nearly identical new service to change the transactional behavior. So much for reuse.

Data Topologies

Unlike many of the architecture styles we discuss in this book, the data topologies for orchestration-driven SOA aren’t very interesting, given this style’s historical origins. Even though it is a distributed architecture consisting of many parts, it generally uses a single (or a few) relational databases, which was the common practice in all distributed architectures in the late 1990s. Even transactionality was generally relegated to this architecture and away from databases: the message bus often included declarative transactional interactions for each of the entities within the topology, allowing developers, architects, or others to determine transactional behavior independently of the database or even the situational reuse of the entity.

Architects in this era considered data a foreign country. While it is an inevitable part of the plumbing in both SOA and event-driven architectures, back then, they treated it more as an integration point than as part of the problem domain.

Really? Declarative Transactions?!?

Yes, really. One of the “features” of many application servers in the heyday of orchestration-driven SOA was allowing configuration managers to change the transactional scope of individual entities, depending on the transactional context within which they wanted to operate. (This was declared, of course, in XML, given the era and its love for verbose but easy-to-parse configuration formats.) A portion of the declaration of an entity (called `EntityBeans`, a specialized type of JavaBean) determines transactional scope as it participates in workflows, which architects themselves declare to be transactional or not. In turn, the application server interacts with the database to create and manage database transactions that match the desired behavior of the entities and/or workflows.

This has largely failed, for two reasons. First, if a developer doesn’t know what the transactional behavior will be at runtime, it adds considerable complexity to entities and dependencies. This forces developers to create almost identical versions of the entities, differing only in their transactional scope. Second, no matter how much sophistication vendors build into their message buses, edge cases constantly appear where the myriad failure modes prevent the system from cleanly managing transactions, creating tangled messes of inconsistency for humans to untangle. Some complex, multifaceted features of systems (like transactions) cannot be cleanly abstracted away. Too many leaks in the abstraction prevent it from achieving reliability.

Cloud Considerations

Orchestration-driven SOA predates the cloud by several decades, so no consideration exists for building this architecture (in its original incarnation) in the cloud.

However, the current-day use of this style makes it a good integration architecture for cloud and on-premises services that must integrate and participate in workflows. As primarily an integration architecture, it works well with cloud-based services and facilities.

Common Risks

At the end of the last century and the beginning of this one, the big risks for this architecture were primarily about how much it cost, how long it took to implement, and (a shocking surprise) how difficult these systems are to maintain and update. Many of these projects were very expensive multiyear endeavors, with critical deci-

sions made high in the company hierarchy. Rather than call these projects “failures,” companies mostly just transformed them into integration architectures with better boundaries, more closely aligned with the ideas of DDD.

When architects use an ESB in a modern system as an integration facility, the biggest risk is the slippery slope of allowing the ESB to gradually encapsulate the entire architecture. This is called the *Accidental SOA* antipattern: where an architect gradually and unintentionally builds a fully orchestration-driven SOA without realizing it. To avoid Accidental SOA, an architect must ensure reasonable encapsulation boundaries for orchestration and pay close attention to issues like transactional boundaries.

Governance

When this architecture was popular, modern-day holistic testing was uncommon. Teams rarely tested SOA outside of formal quality-assurance-level testing, so tool and framework creators gave little consideration to facilitating testing the individual parts. Some testing frameworks emerged to create mocks and stubs for the massive machinery of the message bus and its related moving parts, but they were always cumbersome and inconsistent.

Governance suffered from the same limitations. The idea of automating architectural governance was even more foreign than automating testing. In this era, “governance” meant heavyweight frameworks, meetings, and code reviews—all manual.

However, architects still use ESBs strategically within organizations that need the particular mix of capabilities they offer. In particular, many companies have legacy systems that must interact with more modern systems, often combining results and aggregating behavior—all of which describes the central functionality of an ESB. In these scenarios, fitness functions can serve a critical role in preventing data or bounded contexts from “leaking” across parts of the ecosystem where they shouldn’t appear.

For example, consider a system that uses an ESB to coordinate between an enterprise resource planning (ERP) package, an online sales tool, and more modern microservices-based Accounting services. In this scenario, the system should only read from the ERP and sales systems and should write to the Accounting microservices. Architects can first build a fitness function that ensures that all communication is written consistently to logs, then write something like the following fitness function (given here in pseudocode):

```
READ logs for ERP into ERP-logs for past 24 hours
READ logs for Sales into Sales-logs for past 24 hours
FOREACH entry IN ERP-logs
    IF 'operation' is 'update' and 'target' != 'accounting' THEN
        raise fitness function violation
        "Invalid communication between integration points"
```

```

    END IF
FOREACH entry IN Sales-logs
    IF 'operation' is 'update' and 'target' != 'accounting' THEN
        raise fitness function violation
        "Invalid communication between integration points"
    END IF

```

This fitness function reads the log entries from both integration points to ensure that no update operations take place whose target isn't the Accounting system.

Using such fitness functions, architects can use tools like ESBs strategically while building guardrails around the places where teams typically misuse them.

Team Topology Considerations

Just as architects did not consider data topologies for orchestration-driven SOA, the same is true for team topologies, which was an unknown topic when this architecture style was popular.

In fact, the strict taxonomy of this style serves as a communication antipattern that *led* architects to develop the principles of team topologies. The *goal* in this architecture is extreme separation of responsibilities, with a corresponding separation of team members. Among the companies that adopted it, it was rare indeed for someone building *business services* to chat with someone building *enterprise services*. They were expected to communicate through technical artifacts, such as contracts and interfaces. The level of abstraction in this style creates many integration layers, each implemented by different teams, using enterprise-level ticketing tools to communicate. It should be easy to see why developers find it time-consuming to build features in this style.

Style Characteristics

Many of the criteria we now use to evaluate architecture styles were not priorities when orchestration-driven SOA was popular. The Agile software movement had just started and had not yet penetrated the large organizations likely to use this architecture.

A one-star rating in the characteristics ratings table in [Figure 17-5](#) means the specific architecture characteristic isn't well supported in the architecture, whereas a five-star rating means the architecture characteristic is one of the style's strongest features. Definitions for the characteristics identified in the scorecard can be found in [Chapter 4](#).

SOA is perhaps the most technically partitioned general-purpose architecture ever attempted! In fact, the backlash against the disadvantages of this structure led to more modern architectures, such as microservices. SOA has a single quantum, even

though it is a distributed architecture, for two reasons. First, it generally uses a single database or just a few, creating coupling points across many different concerns. Second, and more importantly, the orchestration engine acts as a giant coupling point —no part of the architecture can have different characteristics than the mediator that orchestrates all behavior. Thus, this architecture manages to find the disadvantages of both monolithic *and* distributed architectures.

| | Architectural characteristic | Star rating |
|-------------|------------------------------|-------------|
| Structural | Overall cost | \$\$\$\$ |
| | Partitioning type | Technical |
| | Number of quanta | 1 to many |
| | Simplicity | ★ |
| | Modularity | ★★★★★ |
| Engineering | Maintainability | ★ |
| | Testability | ★ |
| | Deployability | ★ |
| | Evolvability | ★ |
| Operational | Responsiveness | ★★ |
| | Scalability | ★★★★★ |
| | Elasticity | ★★★★ |
| | Fault tolerance | ★★★★ |

Figure 17-5. Service-oriented architecture characteristics ratings

Modern engineering goals such as deployability and testability score disastrously in this architecture, both because they are poorly supported and because those were not important (or even aspirational) goals when it was developed.

This architecture does support some goals, such as elasticity and scalability, despite the difficulties in implementing them. Tool vendors poured enormous effort into making these systems scalable by building session replication across application servers and other techniques. However, this being a distributed architecture, performance has never been a highlight, because each business request is split across so much of the architecture.

Because of all these factors, simplicity and cost have the inverse of the relationship most architects would prefer. Orchestration-driven SOA was an important milestone because it taught architects the practical limits of technical partitioning and how difficult distributed transactions can be in the real world.

Examples and Use Cases

The primary examples of this architecture existed in the late 1990s and early 2000s in many large enterprises. They have gradually been displaced by more agile and domain-based distributed architectures, like microservices. Even large enterprises have realized that change is inevitable and that software isn't static and needs to change with market forces and new capabilities.

Architects built orchestration-driven SOA architectures to try to achieve effective reuse across large organizations, but eventually realized how difficult their strict and elaborate taxonomy makes implementing common changes and updates. For example, a common domain change might be to update details about a single entity. On a lucky day, developers might only need to change components in the enterprise services layer to do the job. However, on a bad day (one where enterprise architects and/or business stakeholders didn't anticipate this type of change), developers might have to update four or five layers in the architecture, making highly coupled changes in each. Architects working in this style dread hearing the word *change* because it requires deep analysis, and the scope of the work is so variable.

As we mentioned in “[Governance](#)” on page 324, architects still use the building blocks of orchestration-driven SOA (such as the ESB), particularly for integration architectures. For example, an ESB includes both an integration hub (to facilitate communication, protocol, and contract transformation) and an orchestration engine (to allow architects to build workflows between various integration endpoints). Because the orchestration-driven SOA includes many layers of indirection, it allows architects to implement enterprise services as integration points, as package software, or as bespoke code, among other possibilities, as illustrated in [Figure 17-6](#).

Client requests use the message bus to determine which enterprise services to call, in what order to call them, and what information to aggregate. The enterprise services, in turn, communicate via APIs to custom code, old systems, or package software, and so on.

Orchestration-driven SOA represents an interesting innovation for how architects handle problems of integration at scale within the constraints of their ecosystem. For example, since most organizations weren't using open source operating systems when this architecture was popular, alternative architectures like microservices were impossibly expensive. Architects should learn from past approaches. We can continue

using the parts that still make sense, while internalizing the lessons of what failed and why.

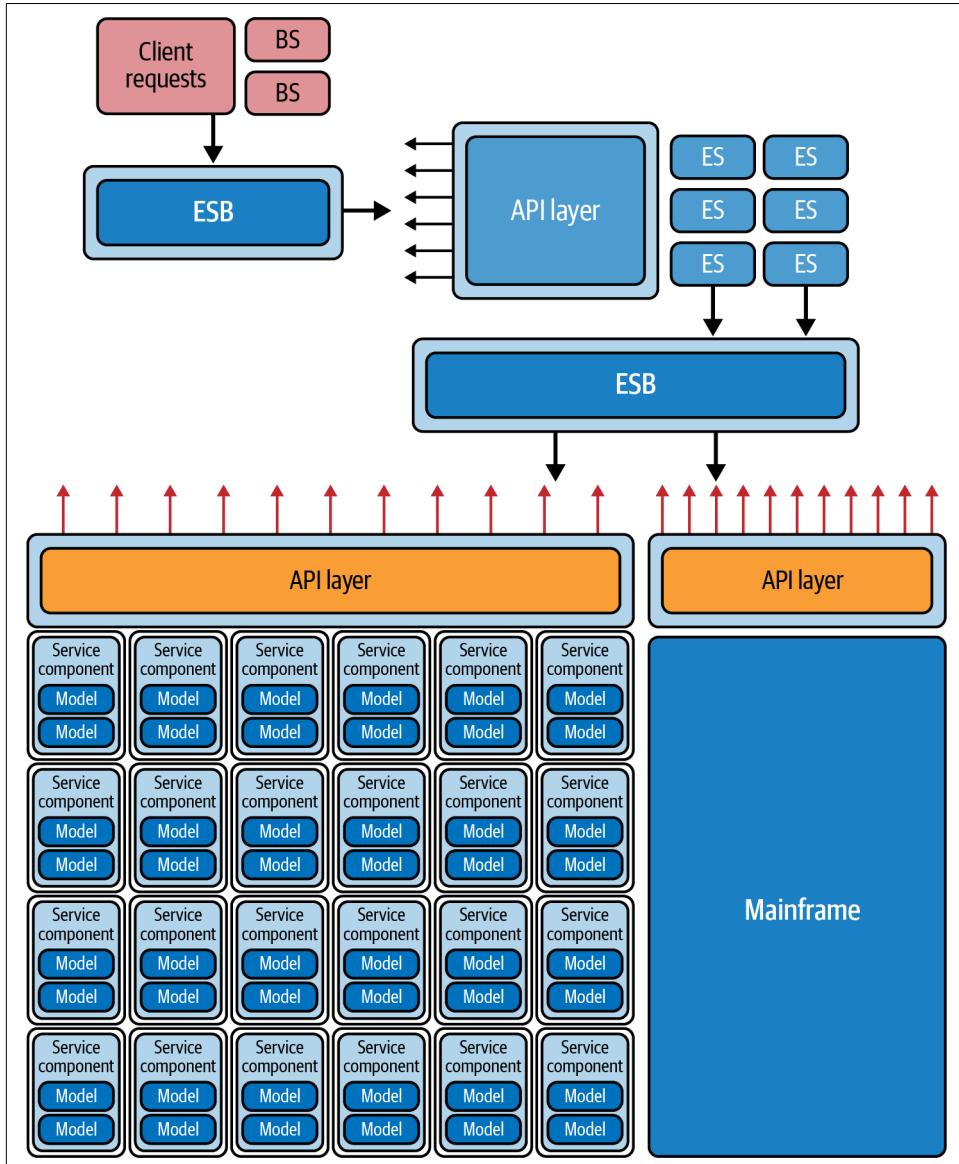


Figure 17-6. The layers of abstraction in this architecture style allow for implementation flexibility

Microservices Architecture

Microservices is an extremely popular architecture style that has gained significant momentum in recent years. In this chapter, we provide an overview of the important characteristics that set this architecture apart, both topologically and philosophically.

Most architecture styles are named after they are created by architects who notice that a particular pattern keeps reappearing. There is no secret group of architects who decide what the next big movement will be—architects make decisions as the software development ecosystem shifts and changes, and of the most common ways of dealing with and profiting from those shifts, those that emerge as the best become architecture styles that others emulate.

Microservices differs in this regard—it was named fairly early in its usage. Martin Fowler and James Lewis popularized it in a famous 2014 [blog post](#) in which they recognized and delineated the characteristics of this relatively new architectural style. Their blog post shaped the definition of the architecture and helped curious architects understand the underlying philosophy.

Microservices is heavily inspired by the ideas in domain-driven design (DDD), a logical design process for software projects. One concept in particular from DDD, the *bounded context*, decidedly inspired microservices. The concept of a bounded context represents a decoupling style (as discussed previously in “[Domain-Driven Design’s Bounded Context](#)” on page 97 in Chapter 7), which is why microservices is sometimes called a “share nothing” architecture.

When a developer defines a domain, that domain includes many entities and behaviors, identified in artifacts such as code and database schemas. For example, an application might have a domain called `CatalogCheckout` that includes notions such as catalog items, customers, and payment. In a traditional monolithic architecture, developers would share many of these concepts, building reusable classes and linked

databases. Within a bounded context, internal parts such as code and data schemas can be coupled together to produce work, but they are *never* coupled to anything outside the bounded context, such as a database or class definition from another bounded context. This allows each context to define only what it needs rather than accommodating other constituents, limiting reuse between bounded contexts.

While reuse is generally beneficial, remember the First Law of Software Architecture? Everything's a trade-off. The downside of reuse is that achieving it usually requires increasing the system's coupling, either by inheritance or composition.

If the architect's goal is a highly decoupled system—the primary goal of microservices—then they will favor duplication over reuse, physically modeling the logical notion of bounded context to include a service and its corresponding data.

Topology

The basic topology of microservices is shown in [Figure 18-1](#). Due to its single-purpose nature, services in this architecture style are much smaller than in other distributed architectures, such as orchestration-driven SOA ([Chapter 17](#)), event-driven architecture ([Chapter 15](#)), and service-based architecture ([Chapter 14](#)). Architects expect each service to include all parts necessary for it to operate independently, including databases and other dependent components.

Microservices is a *distributed* architecture style: each service runs in its own process, in either a virtual machine or container. Decoupling the services to this degree allows for a simple solution to a common problem in architectures that heavily feature multitenant infrastructure for hosting applications. For example, when the system is using an application server to manage multiple running applications, the application server allows operational reuse of network bandwidth, memory, and disk space, among other things. However, if all the supported applications continue to grow, eventually some resource on the shared infrastructure will become constrained.

Another problem concerns improper isolation between shared applications. Separating each service into its own process solves all the problems brought on by sharing. Before the evolution of freely available open source operating systems and automated machine provisioning, it was impractical for each domain to have its own infrastructure. Now, however, with cloud resources and container technology (see “[Cloud Considerations](#)” on page 351), teams can reap the benefits of extreme decoupling, both at the domain and operational level.

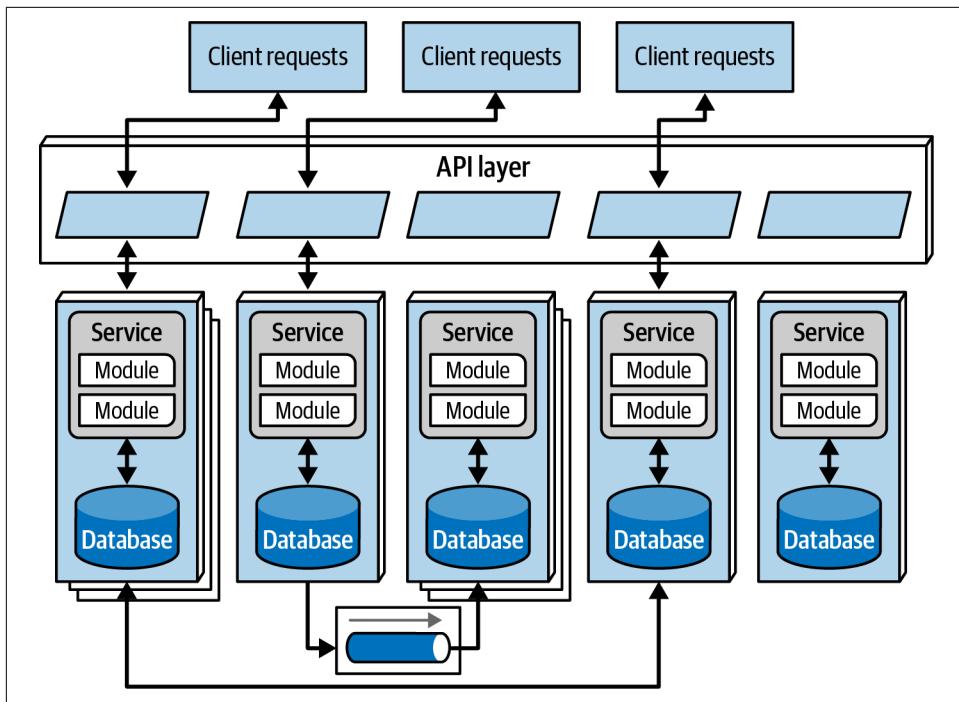


Figure 18-1. The topology of the microservices architecture style

Performance is often the downside of microservices' distributed nature. Network calls take much longer than method calls, and security verifications at every endpoint add additional processing time, requiring architects to think carefully about the implications of granularity.

Because microservices is a distributed architecture, experienced architects advise against using transactions across service boundaries. Determining the granularity of services is really the key to success in this architecture.

Style Specifics

The following sections, while not exhaustive, describe some of the important aspects of the microservices topology, including some unique elements that are unique to microservices.

Bounded Context

Let's dive more deeply into the notion of *bounded context*, which we mentioned as the driving philosophy of microservices. Each service models a particular function, subdomain, or workflow. Each bounded context, then, includes everything necessary

to operate within that function or subdomain—including services consisting of logical components and classes, database schemas, and the corresponding database the service requires to carry out its functions. Each service is meant to represent a particular subdomain or function.

This philosophy drives many of the decisions architects make within microservices. For example, in a monolith, it is common for developers to share common classes, such as `Address`, between disparate parts of the application. However, because microservices architectures try to avoid coupling, an architect building in this architecture style would use duplication rather than coupling to keep *all* code within the bounded context of that function or subdomain.

Microservices takes the concept of a domain-partitioned architecture to the extreme; in many ways, this architecture physically embodies the logical concepts in domain-driven design.

Granularity

Architects designing microservices often struggle to find the correct level of granularity, and end up making their services too small by taking the term *micro* literally. Then they have to build communication links back between the services to do useful work, which negates the point and results in a Big Ball of Distributed Mud.

The term *microservice* is a label, not a description.

—Martin Fowler

In other words, the originators of the term needed to call this new style *something*, and they chose *microservices* to contrast it with the dominant architecture style at the time (circa 2007), the service-oriented architecture, which could have been called “gigantic services.” However, many developers take the term *microservices* as a commandment, not a description, and create services that are too fine-grained.

The purpose of service boundaries in microservices is to capture a domain or workflow. In some applications, those natural boundaries might be large for parts of the system, simply because some business processes are more highly coupled than others. Here are some guidelines architects can use to help find the appropriate boundaries:

Purpose

The most obvious boundary relies on the inspiration for the architecture style: the problem domain. Ideally, each microservice should be functionally cohesive, contributing one significant behavior on behalf of the overall application.

Transactions

Bounded contexts are business workflows, and often the entities that need to cooperate in a transaction suggest a good service boundary. Because transactions

cause issues in distributed architectures, designing systems with the goal of avoiding them tends to result in better designs.

Choreography

A set of services offers excellent domain isolation but requires extensive communication to function. The architect may consider bundling these services back into a larger service to avoid the communication overhead.

Iteration is the only way to ensure good service design. Architects rarely discover the perfect granularity level, data dependencies, and communication styles on their first pass: they iterate over the options to refine their designs, particularly as they learn more about the system and its business functionality.

Data Isolation

Another requirement of microservices, also driven by the bounded-context concept, is *data isolation*. Many architecture styles use a single database for persistence. However, microservices tries to avoid *all* kinds of coupling, *including* using shared schemas and databases as integration points.

Data isolation is another factor to consider when looking at service granularity. Be wary of the Entity Trap (discussed in “[The Entity Trap](#) on page 115)—don’t simply model services to resemble single entities in a database. Architects are accustomed to using relational databases to unify values within a system, creating a single source of truth, but that’s no longer an option when you’re distributing data across the architecture. So every architect must decide how they want to handle this problem: either identifying one domain as the source of truth for some fact and coordinating with it to retrieve values, or distributing information through database replication or caching.

While this level of data isolation creates headaches, it also provides opportunities. Now that they aren’t forced to unify around a single database, each team can choose the most appropriate database technology for their service’s budget, storage structure type, operational characteristics, process characteristics, and so on. Another advantage of a highly decoupled system is that any team can change course and choose a more suitable database (or other dependency) without affecting other teams, which aren’t allowed to couple to implementation details. (We cover data isolation and database considerations in more detail in “[Data Topologies](#)” on page 348.)

API Layer

Most microservices architectures include an API layer (usually called an *API Gateway*) between the consumers of the system (either user interfaces or calls from other systems) and the microservices. The API layer can be implemented as a simple reverse-proxy or a more sophisticated gateway containing cross-cutting concerns such as security, naming services, and so on (these are covered in more detail in “[Operational Reuse](#)”).

While API layers have many uses, to stay true to the underlying philosophy of this architecture, they should not be used as mediators or orchestrators. All interesting business logic in this architecture should reside inside a bounded context, and putting orchestration or other business logic into a mediator violates that rule. Architects typically use mediators in technically partitioned architectures, whereas microservices is firmly domain partitioned.



When using an API layer in a microservices architecture, only include request routing and cross-cutting concerns, such as security, monitoring, logging, and so on. Be careful to avoid putting any business-related logic within the API layer.

Operational Reuse

Given that microservices prefers duplication to coupling, how do architects handle the parts of this architecture that really do benefit from coupling, such as operational concerns like monitoring, logging, and circuit breakers? Traditional service-oriented architecture philosophy was to reuse as much functionality as possible, domain and operational alike. In microservices, however, architects try to split these two concerns.

Once a team has built several microservices, its members start to realize that each microservice has common elements that benefit from similarity. For example, if an organization allows each service team to implement monitoring independently, how can it ensure that each team does so? And how does the organization handle concerns like upgrades—does each team become responsible for upgrading to the new version of the monitoring tool, and how long will that take? The *Sidecar* pattern offers a solution to this problem ([Figure 18-2](#)).

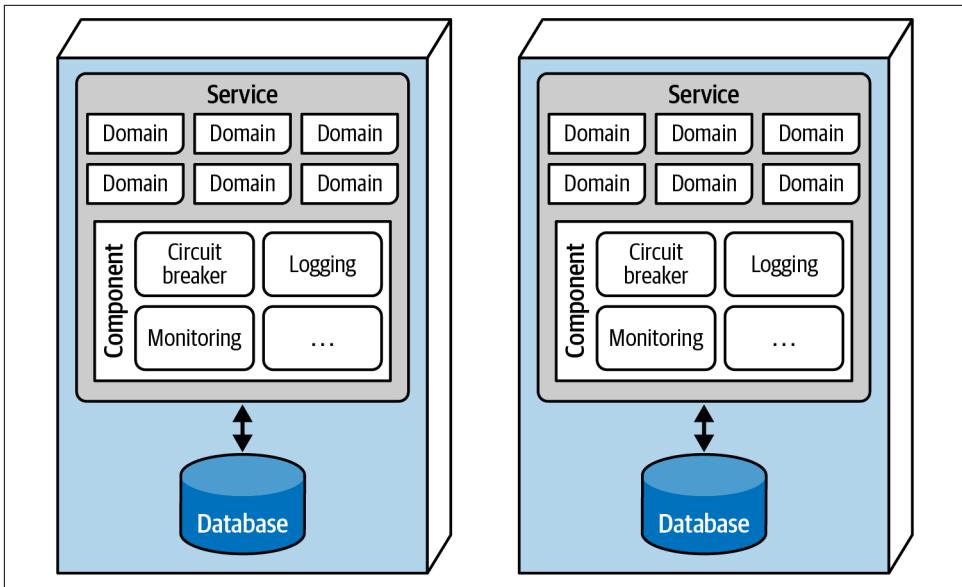


Figure 18-2. The Sidecar pattern in microservices

In [Figure 18-2](#), the common operational concerns (circuit breaker, logging, monitoring) appear within each service as separate components, each of which can be owned by individual teams or by a shared infrastructure team. The **Sidecar** component handles all the operational concerns that benefit from coupling, so when it comes time to upgrade the monitoring tool, the shared infrastructure team can update the sidecar, and each microservice will receive the new functionality (see “[Team Topology Considerations](#)” on page 353).

When each service includes a common Sidecar component, the architect can build a *service mesh* to allow teams unified control of these common concerns across the architecture. The sidecar components connect to form a consistent operational interface across all microservices, as shown in [Figure 18-3](#).

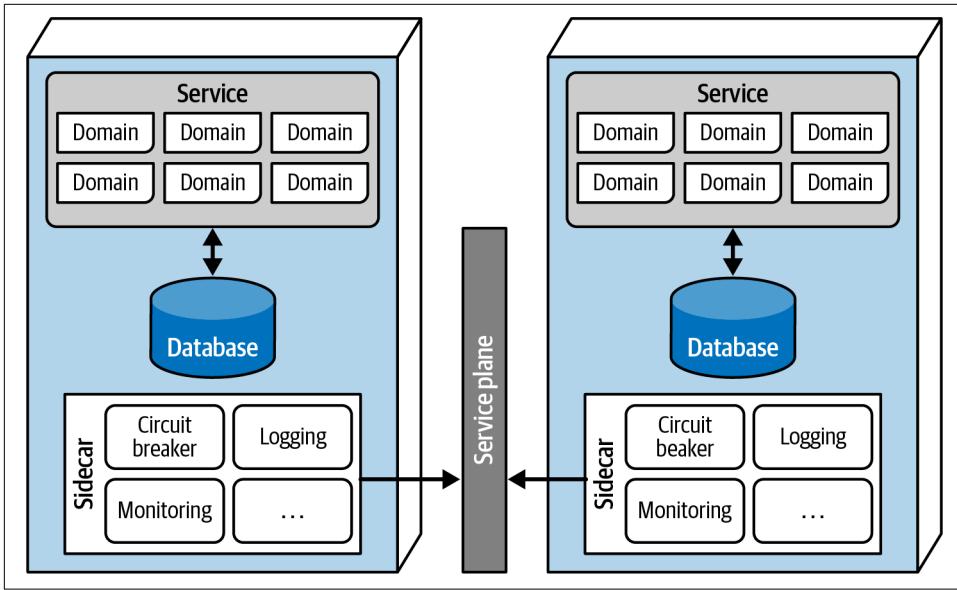


Figure 18-3. The service plane connects the sidecars in a service mesh

In [Figure 18-3](#), each sidecar wires into the *service plane*. A service plane is integration software (usually in the form of a product such as [Istio](#)) that connects each sidecar using a consistent interface, thus forming the service mesh.

Each service forms a node in the overall mesh, as shown in [Figure 18-4](#). The service mesh forms a console that allows teams to globally control operational coupling, such as monitoring levels, logging, and other cross-cutting operational concerns.

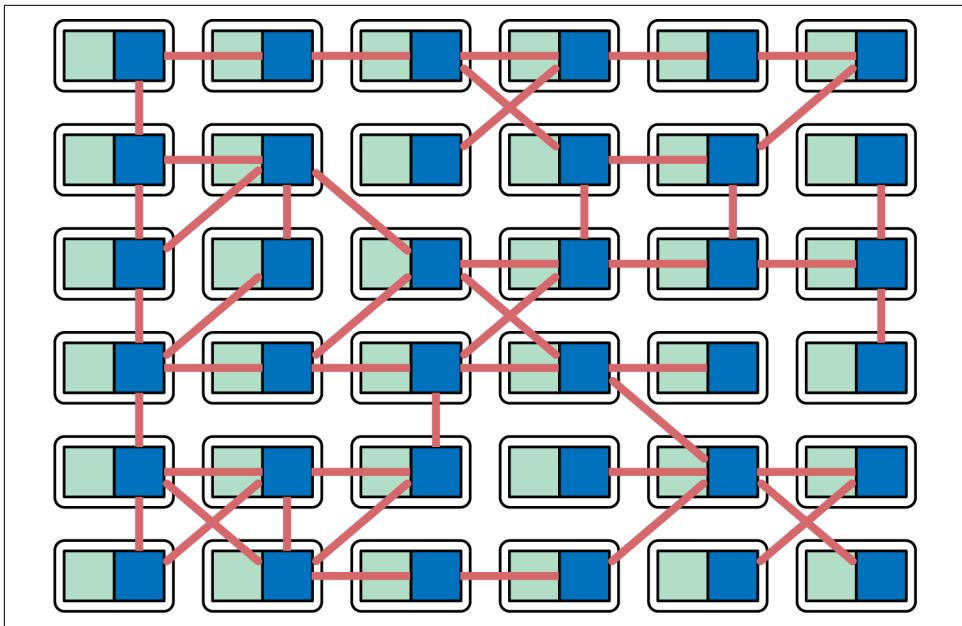


Figure 18-4. The service mesh forms a holistic view of the operational aspect of microservices

Architects use **service discovery** as a way to build elasticity into microservices architectures. *Service discovery* is a way of automatically detecting and locating services within a network. When a request comes in, rather than invoking a single service, it goes through a service discovery tool, which can monitor the number and frequency of requests and spin up new instances of services to handle scale or elasticity concerns. Architects often include service discovery in the service mesh, making it part of every microservice. The API layer is often used to host service discovery, allowing a single place for user interfaces or other calling systems to find and create services in an elastic, consistent way.

Frontends

Microservices favors decoupling, which would ideally encompass the user interfaces as well as backend concerns. In fact, the original vision for microservices included UI as part of the bounded context, faithful to the bounded-context principle in DDD. However, the practicalities of the partitioning required by web applications (and other external constraints) make that a difficult goal. That's why we commonly see two UI styles for microservices architectures.

The first style, shown in [Figure 18-5](#), is the *monolithic frontend*, which features a single UI that calls through the API layer to satisfy user requests. This frontend could be a rich desktop, mobile, or web application. For example, many web applications now use a JavaScript web framework to build a single UI.

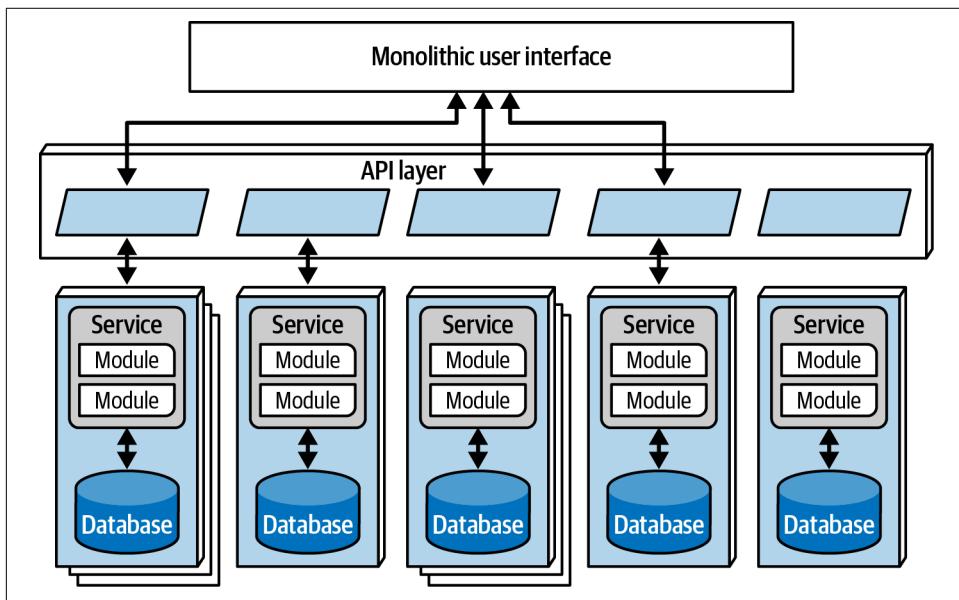


Figure 18-5. Microservices architecture with a monolithic user interface

The second UI option is *micro-frontends*, shown in [Figure 18-6](#).

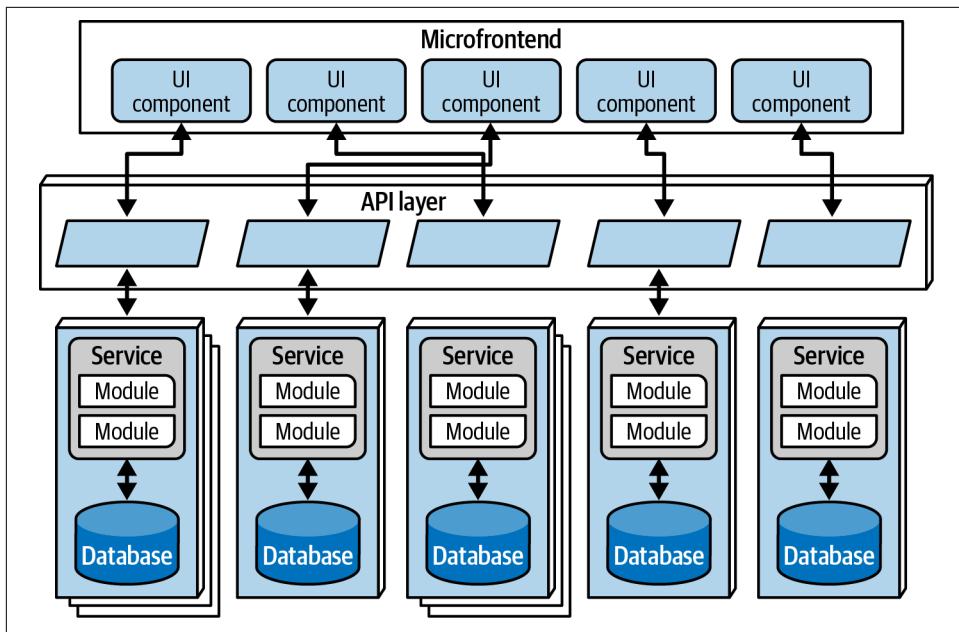


Figure 18-6. Micro-frontend pattern in microservices

The micro-frontend approach uses components at the UI level to create a synchronous level of granularity and isolation in the UI and the backend services, thus forming relationships between the UI components and corresponding backend services.

To learn more about micro-frontends, we highly recommend the book *Building Micro-Frontends*, 2nd Edition, by Luca Mezzalira (O'Reilly, 2025).

Communication

In microservices, architects and developers struggle to find the appropriate service granularity, which affects both data isolation and communication. Finding the correct communication style helps teams keep services decoupled, yet still coordinate them in useful ways.

Fundamentally, architects must decide on *synchronous* or *asynchronous* communication. Synchronous communication requires the sender to wait for a response from the receiver. Microservices architectures typically utilize *protocol-aware heterogeneous interoperability* when communicating to and between services. Let's break that complicated term down into its parts to better understand what it means and why it's important:

Protocol-aware

Because microservices don't include a centralized integration hub, each service needs to know how to call other services. Thus, architects commonly standardize on *how* particular services call each other: a certain level of REST, message queues, and so on. That means that services must know (or discover) which protocol to use to call other services.

Heterogeneous

Because microservices is a distributed architecture, each service could be written in a different technology stack. *Heterogeneous* indicates that microservices fully supports polyglot environments, in which different services use different platforms.

Interoperability

Describes services calling one another. While architects in microservices try to discourage transactional method calls, services commonly call other services via the network to collaborate and exchange information.

Enforced Heterogeneity

A well-known architect who was a pioneer in the microservices style was the chief architect at a startup, building personal-information management software for mobile devices. Because mobile is such a fast-moving problem domain, the architect wanted to ensure that none of the development teams would accidentally create coupling points that could hinder their ability to move independently. It turned out that the teams had a wide mix of technical skills, so the architect mandated a new rule: each development team had to use a *different* technology stack. If one team was using Java and the other was using .NET, it would be impossible for them to share classes accidentally!

This approach is the polar opposite of most enterprise governance policies, which insist on standardizing on a single technology stack. In the microservices world, the goal isn't to create the most complex ecosystem possible, but to choose the correct scale of technology for the narrow scope of the problem. Not every service needs an industrial-strength relational database, and forcing one on a small team is more likely to slow them down than to benefit them. This concept leverages the highly decoupled nature of microservices.

For asynchronous communication, architects often use events and messages, similar to what we described in event-driven architecture in [Chapter 15](#).

Choreography and Orchestration

Choreography utilizes the same communication style as EDA. Choreographed architectures have no central coordinator, respecting the bounded context philosophy and making it natural to implement decoupled events between services.

In choreography, each service calls other services as needed, without a central mediator. For example, consider the scenario shown in [Figure 18-7](#). The user requests details about another user's wish list. Because the `CustomerWishList` service doesn't contain all the information it needs, it makes a call to `CustomerDemographics` to retrieve the missing information, then returns the result to the user.

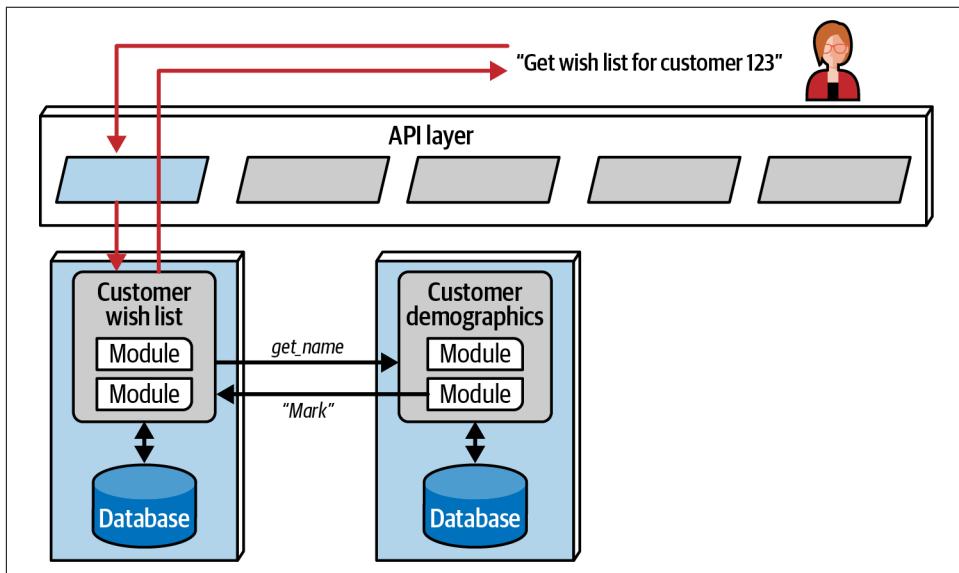


Figure 18-7. Using choreography in microservices to manage coordination

Because microservices architectures don't include a global mediator like other service-oriented architectures, if an architect needs to coordinate across several services, they can create their own localized mediator (usually called an *orchestration service*).

In Figure 18-8, the developers create a service whose sole responsibility is coordinating calls. For instance, the user calls the `ReportCustomerInformation` mediator, which calls all necessary other services to get the requested information.

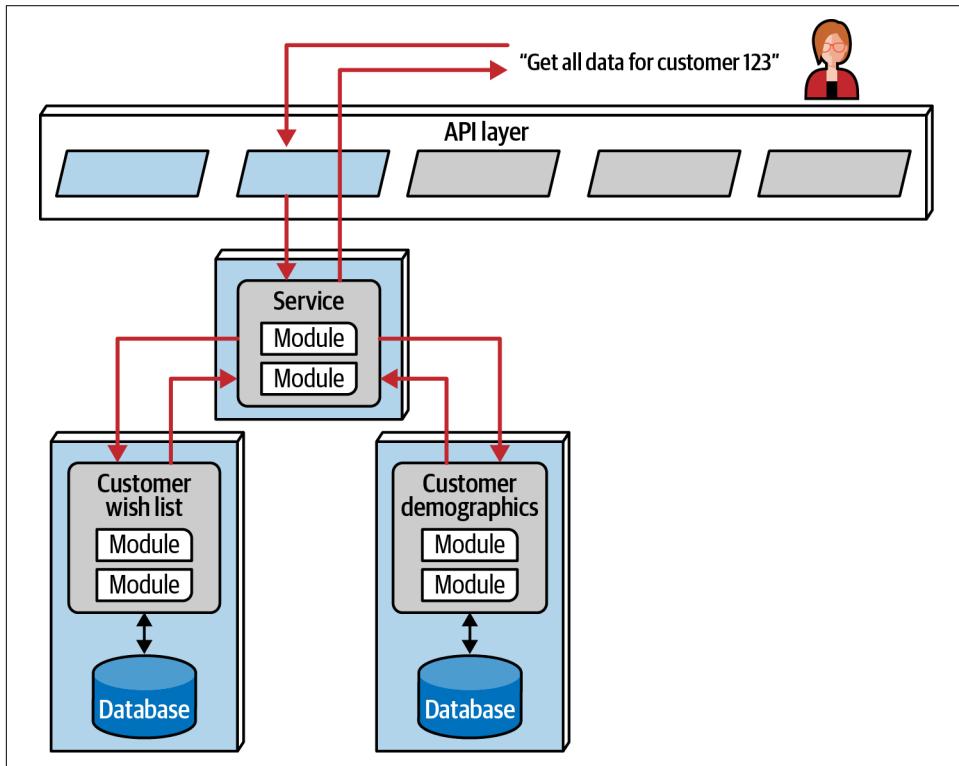


Figure 18-8. Using orchestration in microservices

The First Law of Software Architecture suggests that neither of these solutions is perfect—each has trade-offs. Choreography preserves the highly decoupled philosophy of microservices to reap its maximum benefits. However, it also makes common problems like error handling and coordination more complex.

Consider an example with a more complex workflow. In [Figure 18-9](#), the first service called must coordinate across a wide variety of other services, basically acting as a mediator in addition to its other domain responsibilities. This is called the *Front Controller* pattern, where a nominally choreographed service becomes a more complex mediator for some problem. The downside to this pattern is that the service taking on multiple roles adds complexity.

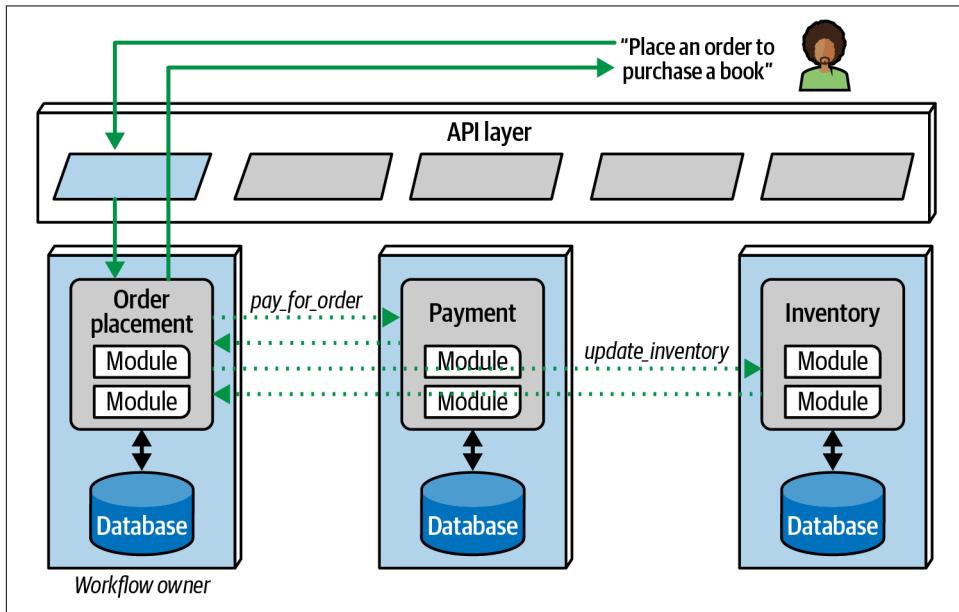


Figure 18-9. Using choreography for a complex business process

Alternatively, architects may choose to use orchestration for complex business processes, illustrated in [Figure 18-10](#). The architect builds a mediator service to coordinate the business workflow creating coupling between these services, but also allowing the architect to focus coordination into a single service, leaving the others less affected. Domain workflows are often inherently coupled, so the architect's job entails finding a way to represent that coupling that best supports the goals of both the domain and the architecture.

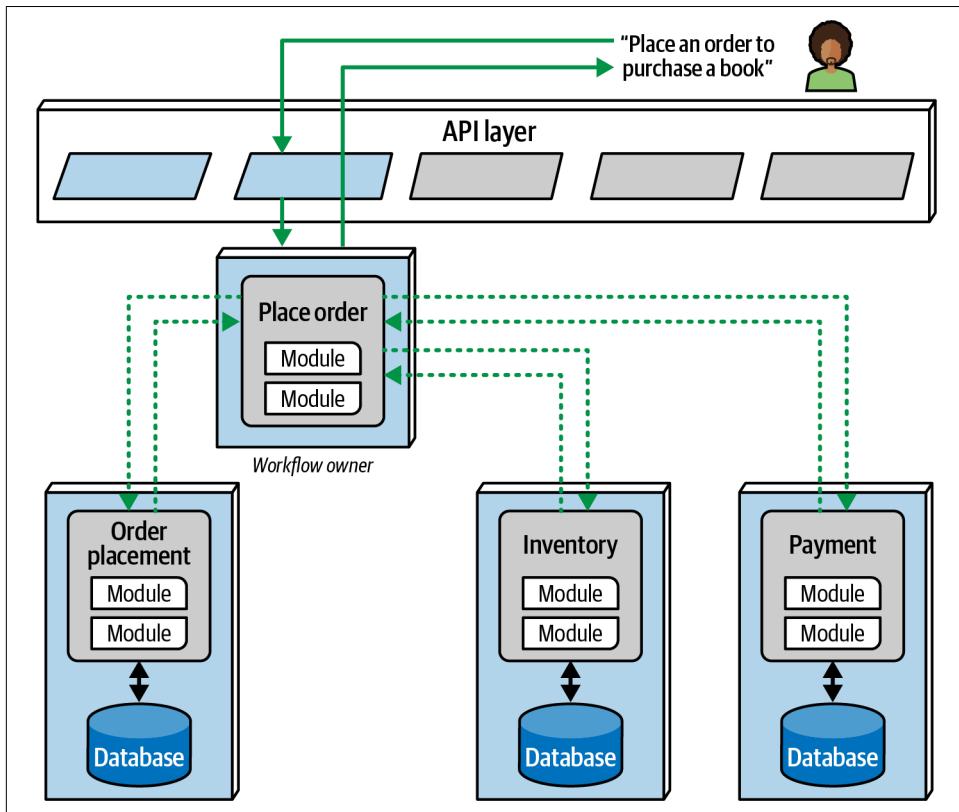


Figure 18-10. Using orchestration for a complex business process

Transactions and Sagas

Architects aspire to extreme decoupling in microservices, but often encounter the problem of how to coordinate transactions across services. Because microservices encourages the same level of decoupling in the databases as in the architecture, the atomicity that was trivial in monolithic applications becomes a problem in distributed ones.

Building transactions across service boundaries violates the core decoupling principle of microservices, and also creates the worst kind of dynamic connascence, *Connascence of Values* (see “[Connascence](#)” on page 49). The best advice we can offer architects who want to do transactions across services is: *don’t!* Fix the service granularity instead. If you’re finding that you need to wire your microservices architecture together with transactions, that’s a sign that your design is too granular.



Try to avoid transactions that span multiple microservices—fix the service granularity instead!

In keeping with the “it depends” rule, there are always exceptions. For example, a situation could arise where two different services need vastly different architecture characteristics that require distinct service boundaries, yet still need transactional coordination. The architect in that case could, after considering the trade-offs carefully, leverage certain transactional patterns to orchestrate transactions.

Distributed transactions in microservices are usually handled by what is known as a *Saga* pattern. In literature, a *saga* is an epic story describing a long sequence of events that leads to some sort of heroic conclusion, hence the name of this transactional pattern.

In Figure 18-11, a service acts as a mediator across multiple service calls to coordinate a transaction. The mediator calls each part of the transaction, records success or failure, and coordinates the results. If everything goes as planned, all the values in the services and their contained databases update synchronously. If there's an error condition and one part of the transaction fails, the mediator must ensure that no part of the transaction succeeds. Consider the situation shown in Figure 18-12.

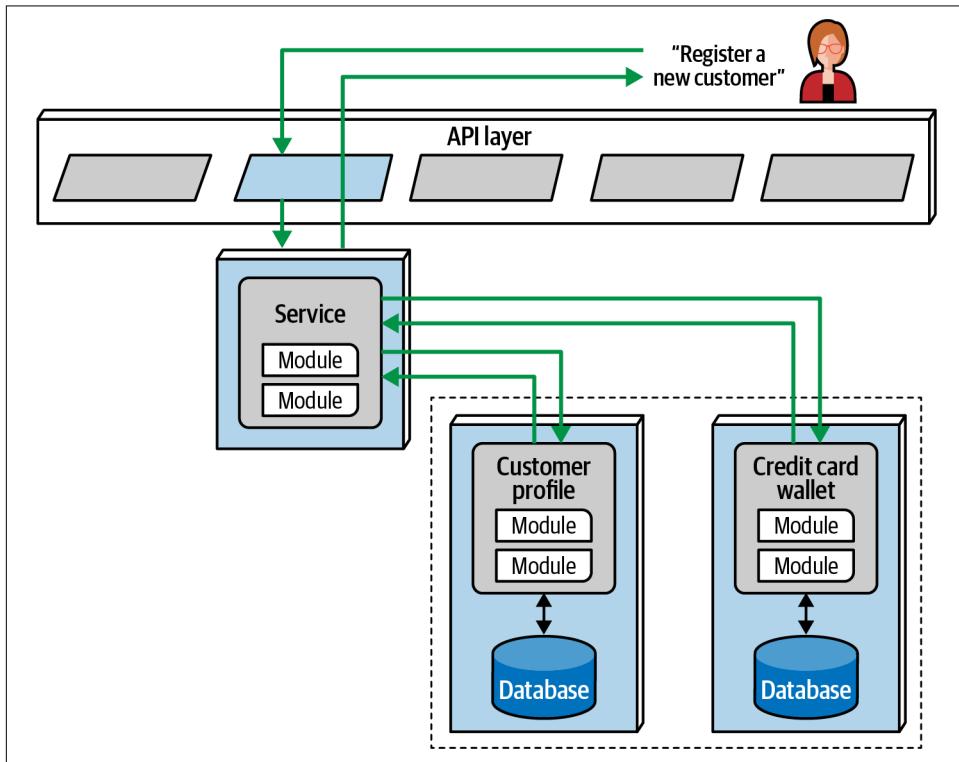


Figure 18-11. The Saga pattern in microservices architecture

If the first part of the transaction succeeds but the second part fails, the mediator must send a request to all other participating services of the transaction that were successful and tell them to undo the previous request. This style of transactional coordination is called a *compensating transaction framework*. Developers usually implement this pattern by having each request from the mediator enter a pending state until the mediator indicates overall success. However, juggling asynchronous requests can become complex, especially if new requests appear that are contingent on pending transactional state. Regardless of the protocol used, compensating transactions create a lot of coordination traffic at the network level.

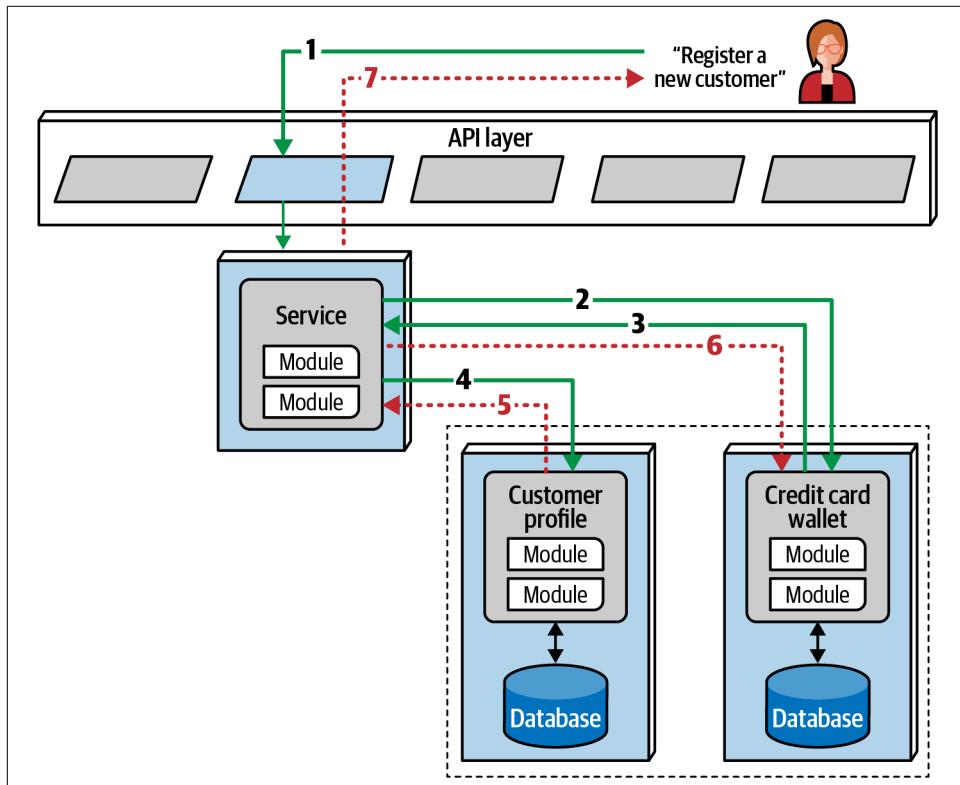


Figure 18-12. Saga pattern compensating transactions for error conditions



It's sometimes necessary for a few transactions to cross services; however, if it's the dominant feature of the architecture, then microservices is likely not the right choice!

Managing transactions in microservices is complicated, and you can dive much deeper. We've identified eight different transactional Saga patterns to solve a variety of scenarios, which you can find in Chapter 12 of our book *Software Architecture: The Hard Parts* (O'Reilly, 2021), coauthored with Pramod Sadalage and Zhamak Dehghani.

Data Topologies

As we've discussed throughout this chapter, data plays a critical role in microservices architectures. As a matter of fact, microservices is the only architectural style that *requires* architects to break apart data. While it's *possible* (if not always effective) to use a monolithic database in other distributed architectures, it's simply not an option in microservices. Neither are domain databases, as we discuss in “[Data Topologies](#)” on page 215 and also in “[Data Topologies](#)” on page 268. This is due to the fine-grained nature of microservices, bounded context, and the large number of services found in most microservices ecosystems.

To illustrate why a monolithic database isn't feasible in microservices, consider a scenario where 60 services share the same database. The first issue that arises will be about controlling change within the architecture. As [Figure 18-13](#) illustrates, changing the structure of that database (such as by changing a column name or dropping a table) would require corresponding changes to all 60 services using that data. Imagine trying to coordinate the maintenance, testing, and release of five dozen separately deployed services while at the same time releasing the database changes! This task would be daunting, to say the least, and would likely end in disaster.

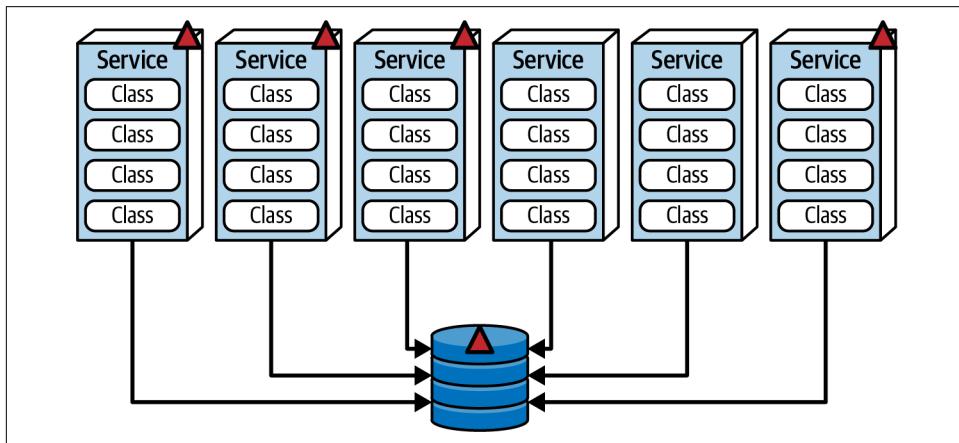


Figure 18-13. Controlling change with a monolithic database is a very challenging task

Perhaps the biggest issue with combining a monolithic data topology with microservices is that it breaks the entire notion of a physical bounded context. Recall that the

bounded context includes *all* functionality required to execute a particular business function or subdomain, *including the database and corresponding data structures*. If every service shares the same database and data structures, the bounded context goes away.

Another issue with monolithic data is about scalability and elasticity. While many operational tools and products automatically monitor concurrent load and adjust the number of service instances to address increases, there aren't many databases that scale accordingly. This imbalance can cause overall responsiveness issues and request timeouts. Managing the database connections, which are typically located in each service *instance*, is another concern. As the number of services and service instances increase, the services can quickly run out of available database connections, resulting in further connection waits and request timeouts.

Last, if the database becomes unavailable due to a crash, planned maintenance, or backups, the entire microservices ecosystem goes down—as would any architectural style using a monolithic database. While not as extreme, a domain-based database topology can suffer from the same issues of scalability, database connection pool management, availability, and fault tolerance.

For these reasons, the standard database topology for microservices is the *Database-per-Service* pattern. With this database topology, each microservices owns its own data, which is contained as tables within a separate database or schema, as illustrated in [Figure 18-14](#).

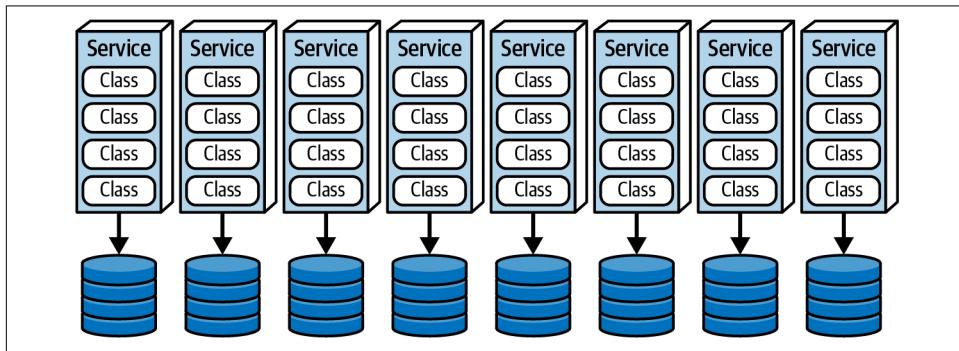


Figure 18-14. The Database-per-Service pattern is the typical database topology for microservices

This topology preserves the bounded context, making it easier for architects to control change. Because other services that need data must request it from the owning service through some sort of contract, they are decoupled from the data's internal structure. Changing the database structure only affects the owning service within the bounded context. This allows architects the freedom to change the database type

(such as from a relational database to a document database) without affecting other services.

The Database-per-Service topology also provides excellent scalability, elasticity, availability, and fault tolerance, architectural characteristics that all suffer under the monolithic or domain-based database topologies. Furthermore, managing connections to the database within a bounded context is much easier than with a monolithic or domain-based database.

While the Database-per-Service topology is widely used within microservices, it does have its drawbacks. For example, what if two or more services write to the same database table? What if a service outside of the bounded context *must* query the database directly, for performance reasons? In these cases (which tend to be fairly common), it is possible for a couple of services to share a database, as shown in [Figure 18-15](#). We recommend that no more than five or six services share a single database (or schema). Any more than that and you'll start to experience the same issues with change control, scalability, elasticity, availability, fault tolerance, and so on.

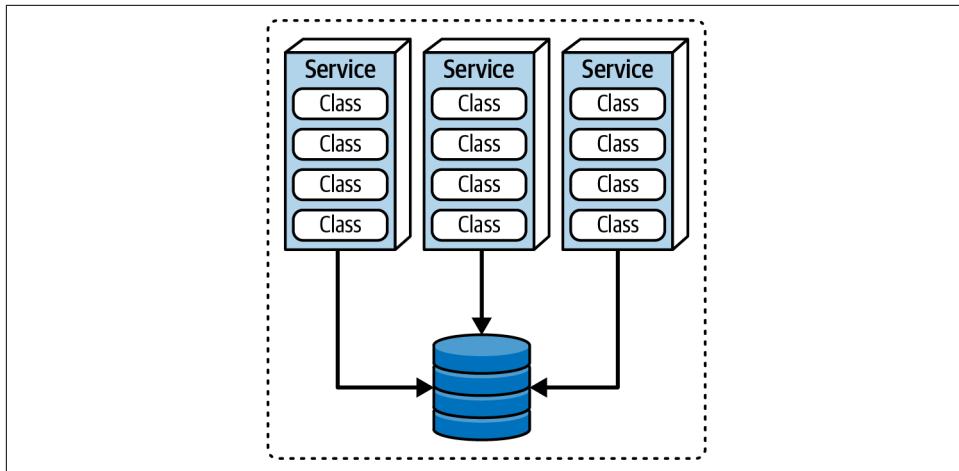


Figure 18-15. It's possible to share data between a few microservices

The box around the services and database in [Figure 18-15](#) denotes the bounded context. Just because services share a database doesn't mean there is no bounded context; it just means the architect has formed a *broader* bounded context. For example, they may have valid reasons for breaking payment processing apart into different payment types (such as credit card, gift card, PayPal, rewards points, and so on), but these individual services still need to update and access the same data. Similarly, an architect may break up a single shipping service into separately deployed services, one for each shipping method, but those services will all still require updates from and access to the same data.

The primary trade-off of sharing data between microservices within a broader bounded context is controlling database changes. When the database schema changes, the architect must now coordinate the change and deployment of multiple services, making database changes riskier and less agile. There might also be negative consequences for scalability, elasticity, and fault tolerance, depending on the business problem and the situation.

Cloud Considerations

While microservices can certainly be deployed in on-prem systems, particularly with the popularity of service orchestration platforms such as [Kubernetes](#) and [Cloud Foundry](#), this architectural style is very well suited for cloud-based deployments—so much so that microservices is sometimes referred to as a “cloud-native” architecture. On-demand provisioning of virtual machines, containers, and databases, combined with the services-based approach found in cloud environments (such as AWS), fits well with the microservices architectural style.

The attentive reader might wonder why we didn’t include [serverless](#) as an architecture style. [Serverless](#) refers to a cloud-computing model where functions are triggered upon request, allocating necessary machine resources on an on-demand basis. Serverless functions include such artifacts as [AWS Lambdas](#), [Google cloud functions](#), and [Azure cloud functions](#). We believe that serverless is not an architectural style, but a *deployment model* of the microservices architectural style.

Recall from earlier in the chapter that a *microservice* is defined as a single-purpose, separately deployed unit of software that does *one thing* really well (hence the prefix *micro*). As such, microservices tends to be relatively finer-grained than services in other architectural styles. This also essentially describes a *serverless function*, which is why we consider serverless part of microservices.

That said, microservices in cloud environments does not *have* to be deployed as serverless functions; it can be deployed as containerized services just as easily. Most cloud vendors have adopted Kubernetes (or some form of the Kubernetes platform), allowing developers to deploy containerized microservices as easily as serverless ones.

Common Risks

One of the biggest risks in microservices is making services *too small*. In 2016 one of your authors (Mark) coined the name *Grains of Sand* for the antipattern of making services too fine-grained, just like grains of sand on a beach. As we previously mentioned, the *micro* in microservices refers to *what* the service does, not how *big* it is. Service granularity is such an important aspect of microservices that we devote an entire chapter to it (Chapter 7) in our book [Software Architecture: The Hard Parts](#).

Another common risk in microservices is creating too much communication between services. The fine-grained nature of microservices, combined with tight bounded contexts, necessitates that services within a microservices ecosystem will invariably need to communicate with one another. This communication may be due to workflow processing (such as choreography or AWS step functions for serverless microservices) or needing another service's data (due to the bounded context within each service and its data). Regardless of the reason, take care to avoid too much dynamic coupling and interservice communication. Again, this is often the result of making services too fine-grained, and can be fixed by combining services together into coarser-grained microservices.

Another risk in microservices is going overboard with sharing data. While we've mentioned that it is possible (and sometimes necessary) to share data in microservices, too much data sharing creates risks to the system's in change control, scalability, fault tolerance, and overall agility—the things microservices does really well (see "[Style Characteristics](#)" on page 354). Know when it's necessary to share data and when to fix data sharing through service consolidation.

A last risk often overlooked in the microservices architectural style is that of reusing code and sharing functionality. Code reuse is a necessary part of software development. However, reusing code and functionality goes directly against the principles of microservices, hence the term "share nothing" architecture described earlier in this chapter. When an architect shares common functionality between services through custom libraries (such as JAR files or DLLs), a part of the bounded context falls apart. In other words, reused code is spread across multiple bounded contexts, meaning that not *all* of the functionality for that particular function or subdomain is contained within its bounded context, and thus a change to shared code could break services in other bounded contexts. While versioning does help address this issue, sharing code nevertheless adds significant complication to a microservices ecosystem.

Governance

Many of the governance rules and techniques used in microservices address the common risks we described in the previous section. Governance within a microservices architecture is largely about avoiding structural decay.

First and foremost, architects should apply governance to monitor and control the amount of static and dynamic coupling between services. Recall from [Chapter 7](#) that static coupling occurs when microservices share common custom or third-party libraries, as well as in the form of contracts when services need to communicate. Contracts are particularly important: while architects can use asynchronous communication protocols to *dynamically* decouple services, they might nevertheless still be *statically* coupled by the contract used between them (regardless of communication type).

A software bill of materials, deployment scripts, and dependency-management tools can help architects to better understand and govern the number of artifacts shared between services. While we cannot prescribe exactly how much static coupling is *too much*, we recommend striving to minimize coupling between services.

Governing dynamic coupling is much more difficult than governing static coupling. Gathering proper metrics requires some creativity and consistency. One common governance technique is to use logs to identify calls to other services. As services make calls to internal or third-party services, those services log the interactions along with information about what service is being invoked, the protocol used, and so on. Architects can analyze this information through fitness functions to better understand dynamic coupling levels throughout the microservices ecosystem. This approach, however, requires keen governance to ensure each service is exposing this information through logging in a consistent manner. A custom library (such as a JAR file or DLL) is one way to provide a consistent API that all services can bind to at compile time and use to ensure consistent logging.

Another way to gather dynamic coupling metrics in microservices is through registry entries. Whenever the first instance of a service starts, that service registers its inter-service calls through some sort of contract (such as JSON) to a custom configuration service or configuration server, such as [Apache ZooKeeper](#). The architect can then query the configuration server to get a map of all interservice calls throughout the microservices ecosystem, which they can use to govern and control the amount of communication between services.

Team Topology Considerations

Since microservices architectures are domain partitioned, they work best when teams are also aligned by domain area (such as cross-functional teams with specialization). When a domain-based requirement comes along, a domain-focused cross-functional team can work together on that feature within a specific domain service, without interfering with other teams or services. Conversely, technically partitioned teams (such as UI teams, backend teams, database teams, and so on) do not work well with this architectural style because of its domain partitioning. Assigning domain-based requirements to a technically organized team requires a level of interteam communication and collaboration that proves difficult in most organizations.

Here are some considerations for architects who want to align a microservices architecture with the specific team topologies outlined in “[Team Topologies and Architecture](#)” on page 151:

Stream-aligned teams

If the domain boundaries are properly aligned, stream-aligned teams work well with this architectural style, particularly if their streams are focused on a specific

domain. However, microservices architecture becomes more challenging for teams whose streams cross multiple bounded contexts and services, outside of a particular subdomain or domain. In this case, we recommend analyzing the microservices' bounded contexts and granularity and either realigning them to the streams or choosing a different architectural style.

Enabling teams

Enabling teams are most effective within a microservices architecture when they can use shared services for specialized or cross-cutting concerns. Due to the high degree of modularity found in microservices, enabling teams can work independently from stream-aligned teams to provide additional specialized and shared functionality without getting in the way. Working with platform teams, they can also assist with creating the sidecar components that make up a service mesh (see “[Operational Reuse](#)” on page 334).

Complicated-subsystem teams

Complicated-subsystem teams can leverage this architecture style’s service-level modularity to focus on complicated domain or subdomain processing, staying independent of other team members (and services).

Platform teams

The high degree of modularity found in microservices helps stream-aligned teams leverage the benefits of the platform-teams topology by utilizing common tools, services, APIs, and tasks. In many cases, platform teams (sometimes working with enabling teams) focus on creating and maintaining the cross-cutting operational functionality found in sidecars (see “[Operational Reuse](#)” on page 334) and the service mesh, freeing stream-aligned teams from these operational concerns.

Style Characteristics

The microservices architecture style offers several extremes on our standard ratings scale, shown in [Figure 18-16](#). A one-star rating means the specific architecture characteristic isn’t well supported in the architecture, whereas a five-star rating means the architecture characteristic is one of the strongest features in the architecture style. Definitions for each characteristic identified in the scorecard can be found in [Chapter 4](#).

Microservices offers notably high support for modern engineering practices such as automated deployment and testability. Microservices couldn’t exist without the DevOps revolution, with its relentless march toward automating operational concerns.

| | Architectural characteristic | Star rating |
|-------------|------------------------------|-------------|
| Structural | Overall cost | \$\$\$\$ |
| | Partitioning type | Domain |
| | Number of quanta | 1 to many |
| | Simplicity | ★ |
| | Modularity | ★★★★★ |
| | Maintainability | ★★★★★ |
| | Testability | ★★★★★ |
| | Deployability | ★★★★★ |
| | Evolvability | ★★★★★ |
| | Responsiveness | ★★ |
| Engineering | Scalability | ★★★★★ |
| | Elasticity | ★★★★ |
| | Fault tolerance | ★★★★★ |
| | | |
| Operational | | |
| | | |

Figure 18-16. Microservices characteristics ratings

The independent, single-purpose, and hence fine-grained nature of services in this architectural style generally leads to high fault tolerance, hence the high rating for this characteristic.

The other high points of this architecture are scalability, elasticity, and evolvability. Some of the most scalable systems ever written have used microservices to great success. Similarly, because this style relies heavily on automation and intelligent integration with operations, architects can build in support for elasticity. Because this architecture favors high decoupling at an incremental level, it also supports the modern business practice of evolutionary change, even at the architecture level. Modern businesses move fast, and software development has struggled to keep pace. An architecture structured with extremely small, highly decoupled deployment units can support a faster rate of change.

Performance is often an issue in microservices. Distributed architectures must make many network calls to complete work, and that has a high performance overhead. They must also invoke security checks to verify identity and access for each endpoint, incurring more latency. Microservices also suffers from *data latency*: when a request requires multiple services to coordinate, that means multiple database calls.

For this reason, the microservices world has many architecture patterns to increase performance, including intelligent data caching and replication to prevent an excess of network calls. Performance is another reason that microservices often use choreography rather than orchestration: less coupling allows for faster communication and fewer bottlenecks.

Microservices is decidedly a domain-partitioned architecture, where each service boundary should correspond to domains. Thanks to the bounded context, it also has the most distinct quanta of any modern architecture—in many ways, it exemplifies what the quantum measure evaluates. The driving philosophy of extreme decoupling creates headaches, but yields tremendous benefits when done well. As in any architecture, architects must understand the rules to break them intelligently.

Examples and Use Cases

Systems that have a high degree of functional and data modularity are good candidates for the microservices architecture. A good use case that exemplifies this style's power is a medical monitoring system that monitors a patient's vital signs: their heart rate, blood pressure, oxygen levels, and so on. Each vital sign the system monitors is a separate, independent function that manages its own data, which fits well with this architectural style and the bounded-context concept.

This patient-monitoring system reads inputs from patient-monitoring devices, records vital signs, analyzes those vital signs for any issues or discrepancies, and alerts a medical professional if it finds problems. Each vital sign can be represented by a separate microservice that is largely independent of other services and data. One exception to this independence, however, would be if a particular vital sign (such as heart rate) needed additional information from another vital sign (such as a sleep-monitor service) to analyze its vital sign for warnings or issues.

Figure 18-17 illustrates how this system might be designed using a microservices architecture. Notice how each vital sign is implemented as a separate microservice, with each maintaining its own data store for vital-sign readings and historical data.

The alert functionality, which is common to all vital-signs services, is represented by a *shared service* called `Alert Staff`. It alerts a nurse or doctor if a particular service notices something wrong with a reading. Each service asynchronously sends its latest vital-signs reading to a monitor located in the patient's room, represented by the `Display Vital Signs` shared service.

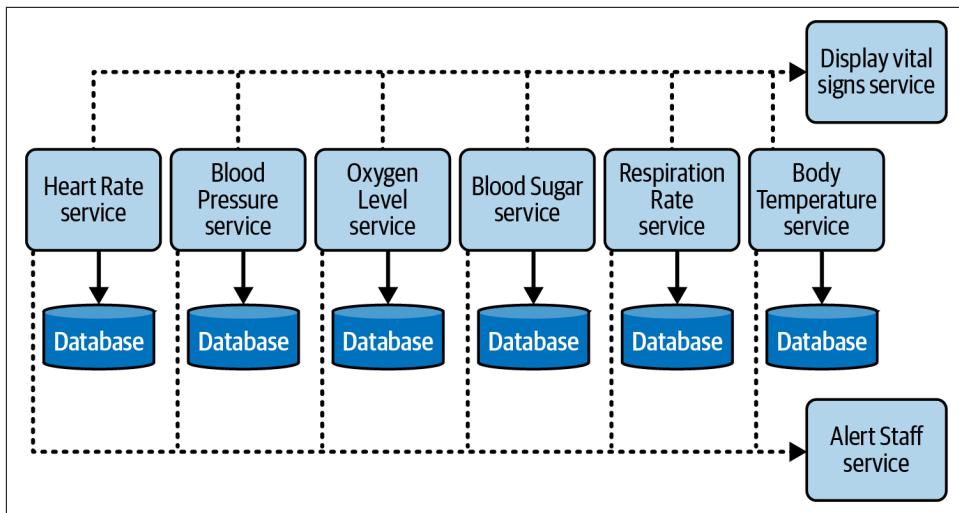


Figure 18-17. A patient medical-monitoring system implemented using a microservices architecture

This example clearly illustrates some of the strengths and advantages of the microservices architecture. Take *fault tolerance*: if one of the vital-sign services happens to crash or become unresponsive, all other vital-sign monitoring services remain fully operational. This is especially important in medical monitoring. Another superpower is *testability*. If a developer performs some maintenance on the blood-pressure monitoring service, the scope of testing is small enough that they can be assured that this particular vital sign is fully tested, and that the maintenance didn't impact other vital-sign services. Finally, a sign of *evolvability* (another microservices superpower) is that the architect could easily add another vital-sign monitor without affecting the other services.

We've touched on many of the significant aspects of microservices architecture in this chapter, and there are many excellent resources you can use to learn more. For a microservices deep dive, we recommend the following:

- *Building Microservices*, 2nd Edition, by Sam Newman (O'Reilly, 2021)
- *Building Micro-Frontends*, 2nd Edition, by Luca Mezzalira (O'Reilly, 2025)
- *Microservices vs. Service-Oriented Architecture* by Mark Richards (O'Reilly, 2016)
- *Microservices AntiPatterns and Pitfalls* by Mark Richards (O'Reilly, 2016)

Choosing the Appropriate Architecture Style

It depends! With all the choices available (and new ones arriving almost daily), we would like to tell you which architecture style you should use—but we can't. Nothing is more contextual to a number of factors within an organization and what software it builds. Choosing an architecture style represents the culmination of a whole process of analysis and thought about trade-offs for architecture characteristics, domain considerations, strategic goals, and a host of other things. That's why, as we said back in [Chapter 2](#), it depends.

However contextual the decision is, this chapter offers some general advice around choosing an appropriate architecture style.

Shifting “Fashion” in Architecture

The software industry’s preferences in architecture styles shift over time, driven by a number of factors. These include:

Observations from the past

New architecture styles generally arise from observations about past experiences, especially observations about pain points. Architects’ experiences working with systems—which are often what lead us to become architects in the first place— influence our thoughts about future systems. New architecture designs often reflect fixes for the specific deficiencies encountered in past architecture styles. For example, after building architectures that centered on code reuse, architects realized the negative trade-offs and seriously rethought the implications of reusing code.

Changes in the ecosystem

In the software development ecosystem, everything changes all the time. This constant change is so chaotic that it's impossible to even predict what type of change will come next. Not too many years ago, no one knew what Kubernetes was, and now it is a daily feature of many developers' lives. In a few more years, Kubernetes may be replaced with some other tool that hasn't been written yet.

New capabilities

Architects must keep a keen eye open not only for new tools but for new paradigms. When new capabilities arise, architecture can shift to an entirely new paradigm, rather than merely replacing one tool with another. For example, few anticipated the tectonic shift in the software development world caused by the advent of containers such as Docker. While that was an evolutionary step, it had an astounding impact on architects, tools, engineering practices, and so much more. Constant change in the ecosystem also regularly delivers new tools and capabilities. Even something that looks like a new one-of-something-we-already-have could include nuances that make it a game changer. New features don't even have to rock the entire development world: a minor change that aligns exactly with an architect's goals can change everything.

Acceleration

Not only does the ecosystem constantly change, but change keeps getting faster and more pervasive. New tools create new engineering practices, which lead to new designs and capabilities, keeping software architects in a constant state of flux. The rise and influence of generative AI is an outstanding example of this constant evolution and corresponding unpredictability.

Domain changes

The domains for which we write software constantly shift and change, as businesses evolve or merge with other companies.

Technology changes

Organizations try to keep up with at least some technological changes, especially those with obvious bottom-line benefits.

External factors

Many external factors only peripherally associated with software development can drive change within an organization. For example, architects and developers might be perfectly happy with a particular tool, but if its licensing cost becomes prohibitive, the business might be forced to migrate to another option.

Architects should understand current industry trends so they can make intelligent decisions about which trends to follow and when to make exceptions, regardless of how closely their organization follows current architecture fashion.

Decision Criteria

When choosing an architectural style, architects must account for all the various factors that contribute to the domain design structure. Fundamentally, an architect designs two things: whatever domain has been specified, and the structural elements required to make the system a success (provided by architectural characteristics).

Only approach choosing an architectural style when you have sufficient knowledge about the following factors:

The domain

Understand as many important aspects of the business domain as you can, especially those that affect operational architecture characteristics. Architects don't have to be subject matter experts, but should at least have a good general understanding of the major aspects of the domain under design. Other specialists, such as business analysts, can help you fill any gaps in your domain knowledge.

Architecture characteristics that impact structural decisions

Identify and elucidate which architecture characteristics are needed to support the domain and other external factors by conducting an architectural characteristics analysis, one of the core activities in choosing a style.

It's possible to implement any of the generic architecture styles in pretty much any problem domain—*generic* implies that they are general-purpose, after all. The exceptions are domains that require special operational architectural characteristics, like a highly scalable auction site. However, in most cases, the real differences between architectural styles concern not the domain, but how well each style supports various architectural characteristics.

You may have noticed that the star charts we've used in Part II of this book to compare each architecture style focus on architectural *characteristics*, not domains. This reflects the importance of understanding architectural characteristics when choosing a style.

Data architecture

Architects and data developers must collaborate on databases, schemas, and other data-related concerns. Data architecture is a specialization in its own right, and we don't cover it much in this book, outside of style-specific considerations. However, you need to understand the impact a given data design might have on your architectural design, particularly if the new system must interact with an older data architecture or one that's already in use.

Cloud deployments

Using the cloud as an architectural destination is the latest in a long line of fundamental shifts in where computation and data reside. The trade-offs involved in designing an application to run on-premises are quite different from those

involved in designing one for the cloud. It's important to know how much data the application will need to store and how much data can move around (which can incur significant costs), among a host of other concerns.

The cloud is a great example of how sophisticated capabilities become commodities over time. A decade ago, building a highly elastic and scalable on-prem system required esoteric skills and was seen as almost magical. Now, architects can achieve the same results just by changing their cloud provider's configuration parameters.

Organizational factors

Many external factors influence design. For example, the cost of a particular cloud vendor may prevent a business from adopting what would otherwise be the ideal design. Likewise, knowing that the company plans to engage in mergers and acquisitions might encourage an architect to gravitate toward open solutions and integration architectures.

Knowledge of process, teams, and operational concerns

Many specific project factors influence architects' designs: the software development process, an architect's interaction (or lack of) with operations, and the QA process. For example, if an organization lacks maturity in Agile engineering practices, architecture styles that rely on those practices for success (such as microservices) will present difficulties.

Domain/architecture isomorphism

Architecture isomorphism is a fancy term for the generic “shape” of an architecture—in other words, the way its components depend on each other within the overall topology. The word *isomorphism* means “a map that preserves sets and relations among elements”; it derives from the Greek *isos*, meaning “equal,” and *morph*, meaning “form” or “shape.”

Architects think about the generic shape of the architecture when considering how suitable it is. For example, consider the differences between the two architecture isomorphism diagrams for the layered and modular monolith architectural styles shown in [Figure 19-1](#). The internal shape of each architecture is apparent: separation by layers versus by domains.

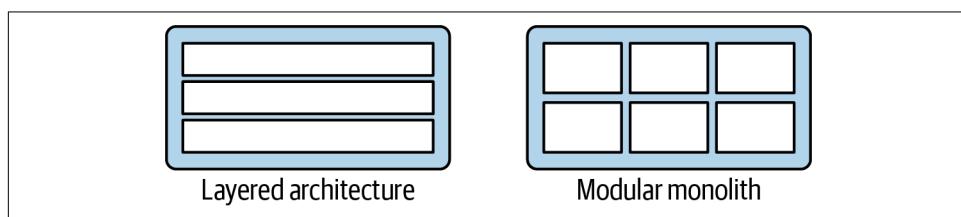


Figure 19-1. Comparison between the isomorphic representations of layered and modular monoliths

The difference between monolithic and distributed is similarly clear from isomorphic drawings, as illustrated in [Figure 19-2](#). Here, the distribution of core components makes the macro-level structure of the architecture clear.

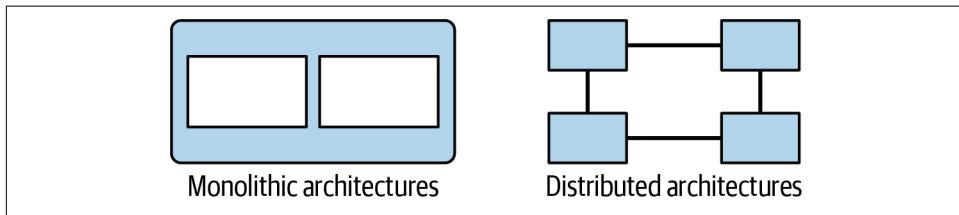


Figure 19-2. Comparing isomorphic representations of monolithic and distributed architecture styles

Some problem domains match the topology of the architecture. For instance, the microkernel architecture style is perfectly suited to systems that require customizability—the architect can design customizations as plug-ins. For another example, a system designed for genome analysis, which requires a large number of discrete operations, might be a good fit for space-based architecture, which offers a large number of discrete processors.

Similarly, some problem domains may be particularly ill-suited for certain architecture styles. Highly scalable systems struggle with large monolithic designs, because it's difficult for a highly coupled code base to support a large number of concurrent users. A problem domain that includes a huge amount of semantic coupling would match poorly with a highly decoupled, distributed architecture: for instance, an insurance application consisting of multipage forms, each based on the context of previous pages, is a highly coupled problem. It would be difficult to model in a decoupled architecture like microservices. An intentionally coupled architecture like service-based architecture would suit this problem better.

Taking all these things into account, you must make several determinations in choosing an architecture style:

Monolith versus distributed?

Will a single set of architecture characteristics suffice for the design, or do different parts of the system need differing architecture characteristics? A single set implies that a monolith would be suitable (although other factors may suggest a distributed architecture); different sets of architecture characteristics imply a distributed architecture. The concept of architecture quantum ([Chapter 7](#)) is useful in making this determination.

Where should data live?

If the architecture is monolithic, architects commonly assume it will use a single relational database, or perhaps a few of them. In a distributed architecture, you

must decide which services should persist data, which also implies thinking about how data will flow through the architecture to build workflows. Consider both structure and behavior when designing architecture, and don't be afraid to iterate on your design to find better combinations.

Should services communicate synchronously or asynchronously?

Once you've determined where data should live, the next design consideration is communication between services—should it be synchronous or asynchronous? Synchronous communication is often more convenient, but it can mean trading off on scalability, reliability, and other desirable characteristics. Asynchronous communication can provide unique benefits in terms of performance and scale, but also plenty of headaches around data synchronization, deadlocks, race conditions, debugging, and so on. (We cover many of these issues in [Chapter 15](#).)

Because synchronous communication presents fewer design, implementation, and debugging challenges, we recommend defaulting to synchronous when possible, and using asynchronous communication only when necessary.



Use synchronous communication by default, asynchronous when necessary.

The output of this design process is an *architecture topology* that encompasses the chosen architecture style (and any hybridizations), Architectural Decision Records (ADRs) about the parts of the design that require the most effort, and architecture fitness functions to protect important principles and operational architecture characteristics.

Monolith Case Study: Silicon Sandwiches

In the Silicon Sandwiches architecture kata in [Chapter 5](#), after our architecture characteristics analysis, we determined that a single quantum was sufficient to implement the system. Since we were looking at a simple application without a huge budget, the simplicity of a monolith was appealing.

However, we created two different component designs for Silicon Sandwiches: one domain partitioned, and another technically partitioned—you might want to flip back to [Chapter 5](#) if you want a refresher. In this chapter, we return to those simple solutions to create designs for each option and discuss their trade-offs. We'll begin with a monolithic architecture.

Modular Monolith

A modular monolith builds domain-centric components with a single database, deployed as a single quantum; the modular monolith design for Silicon Sandwiches appears in [Figure 19-3](#).

This is a monolith with a single relational database, implemented with a single web-based UI (with careful design considerations for mobile devices) to keep overall cost down. Each of the domains we identified appears as a component. If time and resources are sufficient, consider separating the tables and other database assets in the same way as the domain components, which would make it much easier to migrate this architecture to a distributed architecture if future requirements warrant it.

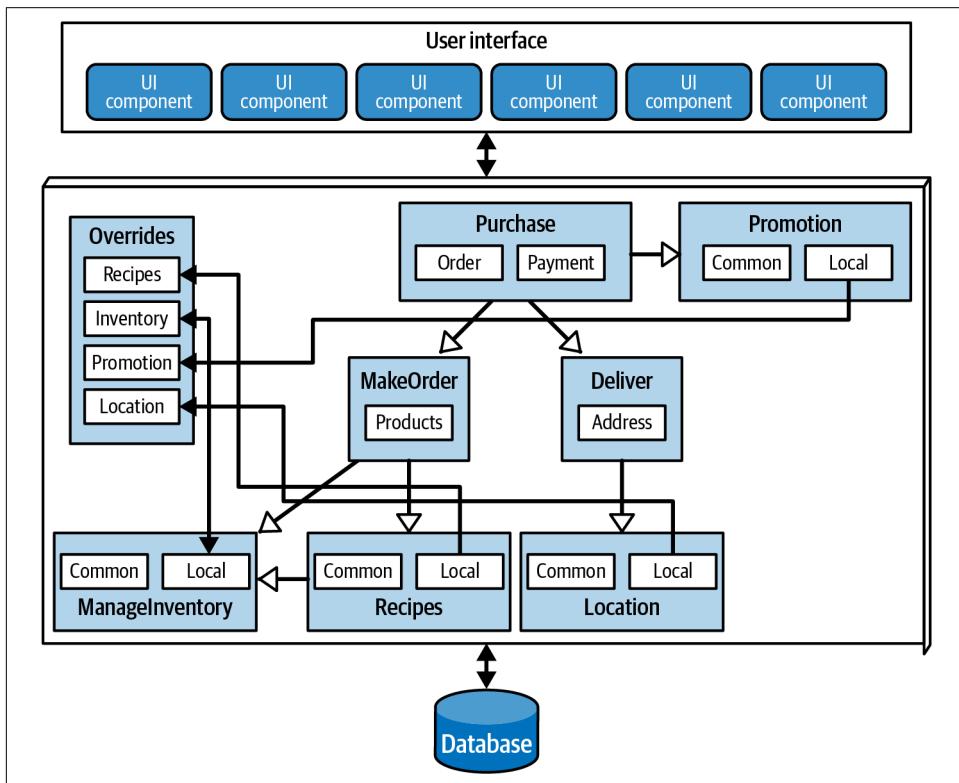


Figure 19-3. A modular monolith implementation of Silicon Sandwiches

Because the architecture style itself doesn't inherently handle customization, that feature should become part of the domain design. In this case, the architect designs an `Override` endpoint, where developers can upload individual customizations. Correspondingly, they must ensure that every domain component references the `Override`

component for each customizable characteristic. (Checking this would be a perfect job for an architectural fitness function.)

Microkernel

One of the architecture characteristics we identified in Silicon Sandwiches was customizability. [Figure 19-4](#) uses domain/architecture isomorphism to show how this could be implemented using a microkernel architecture.

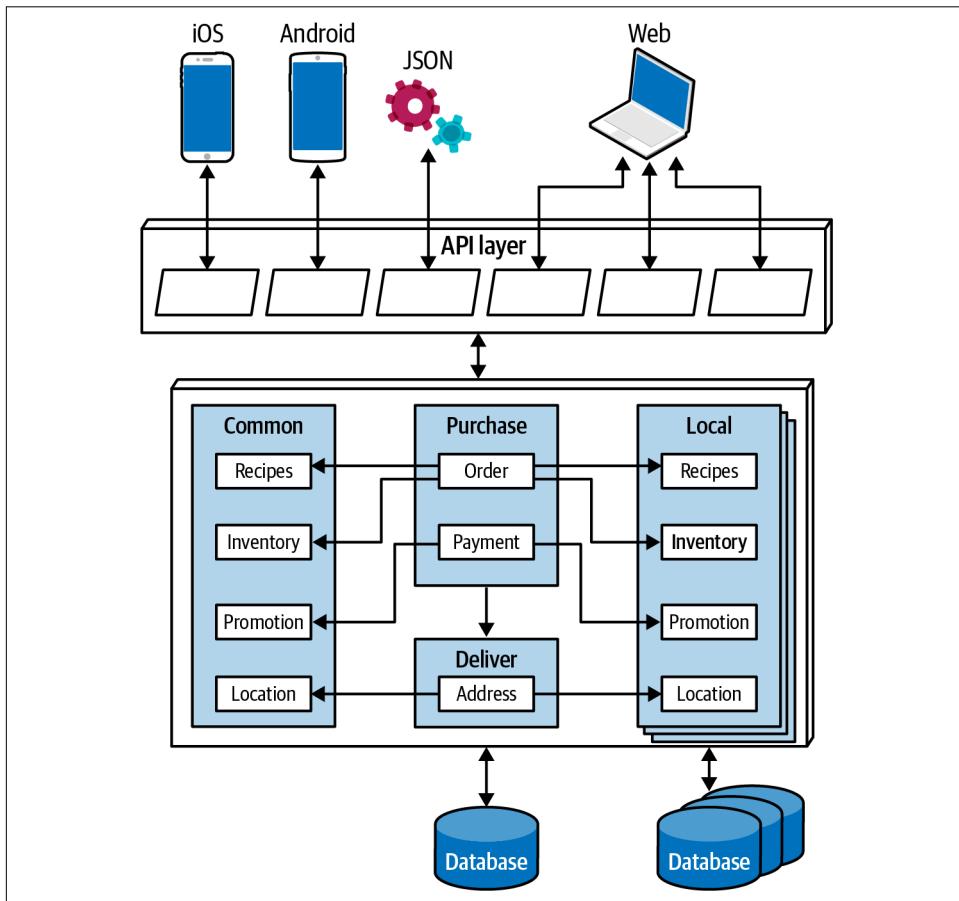


Figure 19-4. A microkernel implementation of Silicon Sandwiches

In [Figure 19-4](#), the core system consists of the domain components and a single relational database. As in the modular monolith design, careful synchronization between the domains and the data design will facilitate migrating the core to a distributed architecture in the future. Each customization appears in a plug-in; the common ones appear in a single set of plug-ins (with a corresponding database) and a series of local

ones, each with its own data. Because none of the plug-ins need to be coupled to the other plug-ins, they can each maintain their data and remain decoupled.

The other unique element of this design is that it utilizes the **Backends for Frontends (BFF) pattern**, making the API layer a thin microkernel adapter in addition to the core architecture. The API layer supplies general information from the backend, which the BFF adapters translate into a suitable format for the frontend device. For example, the BFF for iOS takes the generic backend output and customizes the data format, pagination, latency, and other factors to fit what the iOS native application expects. Building each BFF adapter allows for the richest possible user interfaces and makes it possible to expand the architecture to support other devices in the future—one of the benefits of the microkernel style.

Communication within either of these two Silicon Sandwich architectures can be synchronous, since the architecture doesn't require extreme performance or elasticity requirements, and none of the operations will be lengthy.

Distributed Case Study: Going, Going, Gone

The Going, Going, Gone (GGG) kata from [Chapter 8](#) presents some more interesting architectural challenges. Based on the component analysis in [“Case Study: Going, Going, Gone—Discovering Components” on page 125](#), we know that different parts of this architecture need different characteristics. For example, the need for availability and scalability will differ between roles like auctioneer and bidder.

The system requirements for GGG also explicitly state some ambitious expectations for scale, elasticity, performance, and other tricky operational architecture characteristics. The architecture pattern should allow for a high degree of customization at a fine-grained level. Of the candidate distributed architectures, low-level event-driven architecture and microservices are the two that best match most of the required architecture characteristics. Of the two, microservices is better at supporting variation among operational architecture characteristics. (Purely event-driven architectures typically separate pieces not by their architecture characteristics, but by whether they use orchestrated or choreographed communication.)

Achieving the stated performance goal would be a challenge in microservices, but the best way to address any weak point of an architecture is by designing to accommodate it. For example, while microservices by its nature offers a high degree of scalability, it often develops specific performance issues caused by too much orchestration or too aggressive data separation, among other issues.

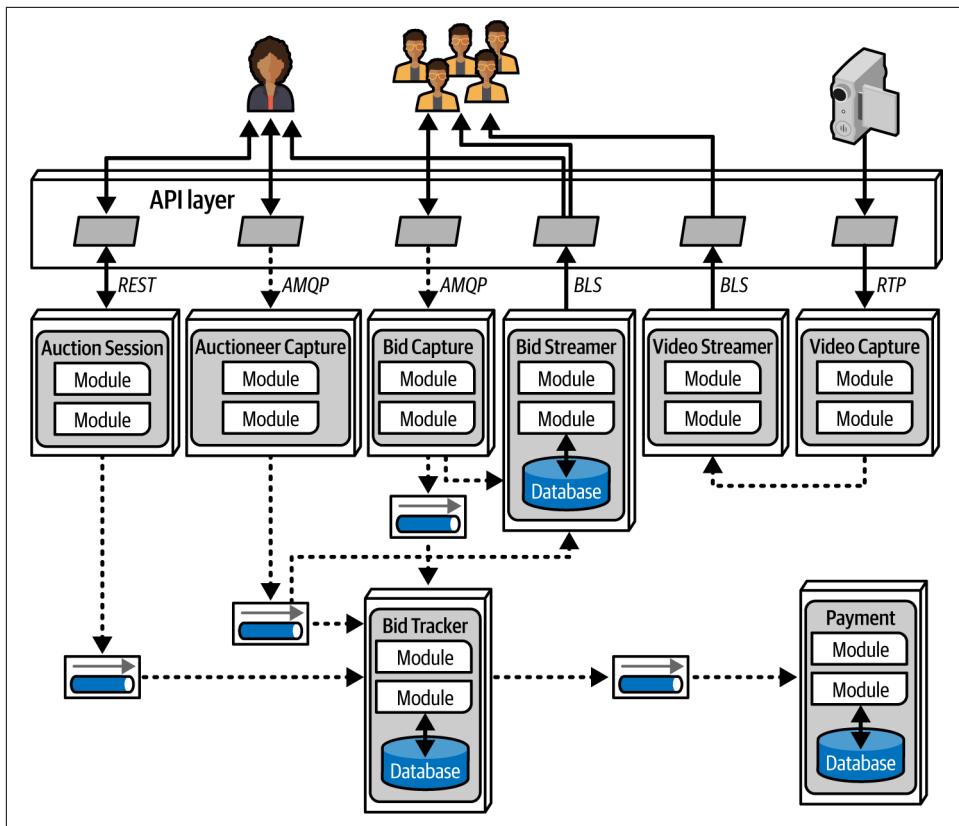


Figure 19-5. A microservices implementation of Going, Going, Gone

In [Figure 19-5](#), each identified component becomes a service in the architecture, matching components with service granularity. GGG has three distinct user interfaces:

Bidder

The numerous bidders for the online auction.

Auctioneer

One per auction.

Streamer

A service responsible for streaming video and bids to the bidders. This is a read-only stream, which allows for optimizations that wouldn't be available if updates were necessary.

The following services appear in this design for the GGG architecture:

Bid Capture

Captures online bidder entries and asynchronously sends them to **Bid Tracker**. This service needs no persistence because it acts as a conduit for the online bids.

Bid Streamer

Streams the bids back to online participants in a high-performance, read-only stream.

Bid Tracker

Tracks bids from both **Auctioneer Capture** and **Bid Capture**, unifying these two information streams and ordering the bids in as close to real time as possible. Both inbound connections to this service are asynchronous, allowing the developers to use message queues as buffers to handle very different message flow rates.

Auctioneer Capture

Captures bids for the auctioneer. The analysis in [“Case Study: Going, Going, Gone—Discovering Components” on page 125](#) showed that the architecture characteristics of **Bid Capture** and **Auctioneer Capture** are quite different, leading us to separate them.

Auction Session

Manages the workflow of individual auctions.

Payment

Third-party payment provider that handles payment information after **Auction Session** completes the auction.

Video Capture

Captures the video stream of the live auction.

Video Streamer

Streams the auction video to online bidders.

We were careful to identify both synchronous and asynchronous communication styles in this architecture. The choice of asynchronous communication was primarily to accommodate operational architecture characteristics varying between services. For example, if the **Payment** service can only process a new payment every 500 ms, and a large number of auctions end at the same time, using synchronous communication between the services would cause timeouts and other reliability headaches. Message queues add reliability to a critical but fragile part of the architecture.

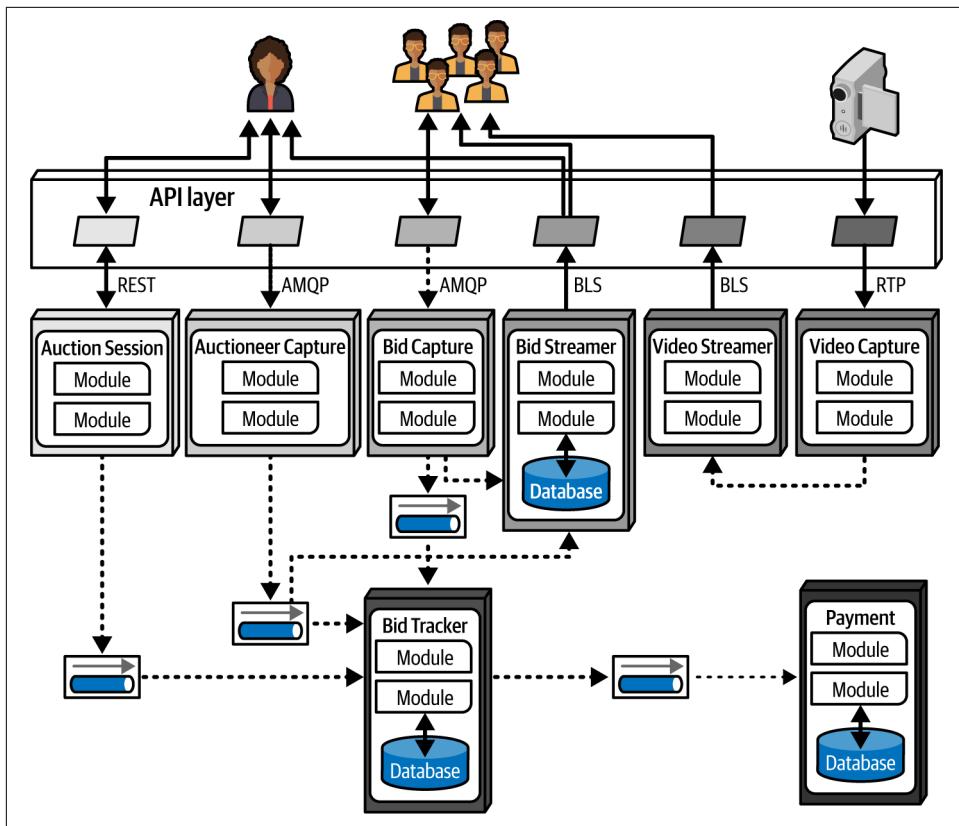


Figure 19-6. The quantum boundaries for GGG

In the final analysis, this design resolved to five quanta, identified in [Figure 19-6](#), roughly corresponding to the services: Payment, Auctioneer, Bidder, Bidder Streams, and Bid Tracker. Multiple instances are indicated in the diagram by stacks of containers. Using quantum analysis at the component-design stage made it easier to identify service, data, and communication boundaries.

Note that this isn't the “correct” design for GGG, and it's certainly not the only one. We don't even suggest that it's the best possible design—but it seems to have the *least worst* set of trade-offs. Choosing microservices, then using events and messages intelligently, allows this architecture to get the most out of a generic architecture pattern while still building a foundation for future development and expansion.

Architectural Patterns

We distinguish between architectural styles and architectural patterns in [Chapter 9](#): to recap, *styles* are named topologies that architects distinguish by differences in topologies, physical architectures, deployments, communication styles, and data topologies. Architecture *patterns*, inspired by the often-mentioned book [Design Patterns](#), are contextualized solutions to problems.

It's important to distinguish between architecture patterns and “best practices” (discussed in more depth in [Chapter 21](#)). Calling something a “best practice” implies that the architect has a clear duty, anytime a particular situation arises, to utilize that practice. Calling it a *better* practice would at least brook some argument, but no—we call it the *best* practice, allowing architects to shut off their brains and always follow the same solution.

It's also important to distinguish between patterns and *solutions*. Many tools, frameworks, libraries, and other software-development artifacts encapsulate one or more patterns—depending on how they are implemented, with differing degrees of fidelity and intermingling with other patterns. Focus on identifying the most appropriate pattern first, then choose the most appropriate implementation for it.

We provide a smattering of representative patterns in this chapter to contrast and provide context for the styles we've introduced in Part II of this book. Our first example, an architecture reuse pattern, clarifies the difference between patterns and implementations.

Reuse

The split between domain coupling and operational coupling is a common architectural concern in distributed architectures such as microservices.

Separating Domain and Operational Coupling

One of the design goals of microservices architectures is a high degree of decoupling, often manifested in the advice: “duplication is preferable to coupling.”

Let’s say that two services need to pass information about customer profiles back and forth, but the architecture’s domain-driven bounded context insists that implementation details should remain private to each service. A common solution is to give each service its own internal representations of entities such as `Profile`, then passing that information in loosely coupled ways, such as name-value pairs in JSON. This lets each service change its internal representation—including the technology stack—at will, without breaking the integration. Developers generally frown on duplicating code, which can cause issues with synchronization, semantic drift, and more, but some things are worse than duplication...and in microservices, that includes coupling.

Architects who design microservices generally resign themselves to the reality that sometimes they have to duplicate implementations to preserve decoupling. But what about capabilities that *benefit* from high coupling? Each service should have certain common operational capabilities, such as monitoring, logging, authentication and authorization, and circuit breakers. However, allowing each team to manage these dependencies often descends into chaos.

For example, consider a company trying to choose one standard monitoring solution for all of its services to make it easier to operationalize them. The architect decides to make each team responsible for implementing monitoring for its service: the `Payment` service team, the `Inventory` service team, and so on. But how can the operations team be sure that each team actually did this? Also, what about issues such as unified upgrades? If the standardized monitoring tool needs an upgrade across the organization, how should the teams all coordinate that?

Hexagonal architecture

In the *Hexagonal* architecture pattern, shown in [Figure 20-1](#), the domain logic resides in the center of the hexagon, which is surrounded by ports and adapters connected to other parts of the ecosystem (in fact, this pattern is alternately known as the *Ports and Adapters* pattern). A visual representation of this pattern appears in [Figure 20-1](#).

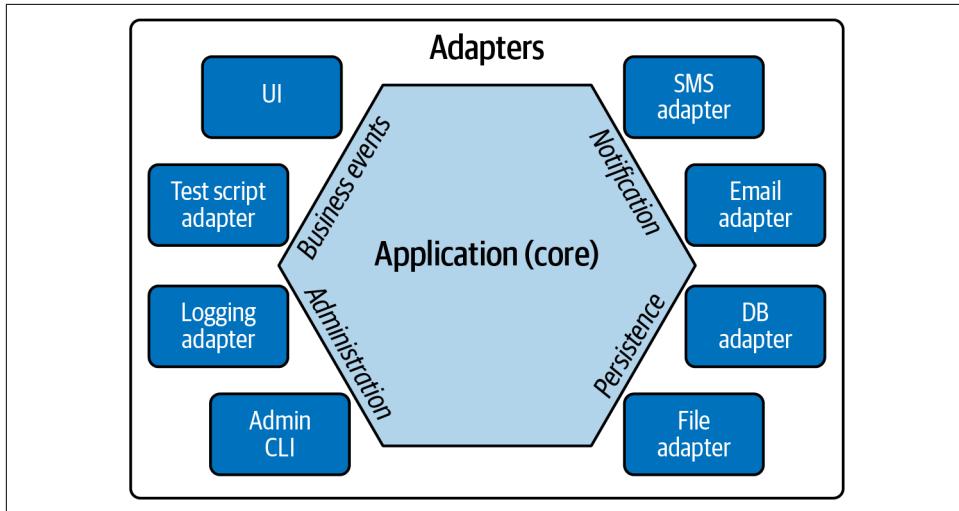


Figure 20-1. The Hexagonal architecture pattern

Sharp-eyed readers may notice that only four of the hexagon's six sides are used. This pattern's creator, Alistair Cockburn, initially drew it as a hexagon and called it the Hexagonal architecture pattern. He almost instantly regretted it, since the name Ports and Adapters is more descriptive, but it was too late. Too many architects thought “Hexagonal” sounded cool, so the name stuck.

Mistaking the pattern for the implementation is a common hazard, and this is a good example. The Hexagonal architecture has a potentially serious flaw when used to describe microservices, but only for those who understand the pattern's original intent. Hexagonal architecture predates modern microservices, but there are many similarities between them. There's also one significant difference: data fidelity. Hexagonal architecture treats the database as just another adapter that can be plugged in. It didn't include the data schema as business logic, because of the misperception (which was common at the time that the Hexagonal pattern name was coined) that the database was an entirely separate piece of machinery. Eric Evans had the insight to correct this mistake in his book *Domain-Driven Design* by recognizing that, regardless of where they reside, database schemas must change to reflect the system's business logic.

This makes the Hexagonal pattern a constant source of confusion among architects. Whenever someone uses it, are they describing the separation of operational and domain concerns, or are they referring to the literal pattern, which would isolate the data and therefore violate a core microservices design principle? Using the pattern name as shorthand for “separation of domain and operational concerns” is fine, as long as it isn’t misleading in context. However, architects today have no need for this implementation. We now have a more suitable mechanism to implement the Hexagonal pattern: the *Service Mesh* pattern.

Service Mesh

Back in “[Operational Reuse](#)” on page 334, we described the common architectural approach to separation of technical concerns from domain concerns: using the Sidecar and Service Mesh patterns.

The Sidecar pattern isn’t just a way to decouple operational capabilities from domains—it’s an orthogonal reuse pattern to address a specific kind of coupling (see “[Orthogonal Coupling](#)” on page 374). Often, architectural solutions require several types of coupling, such as our current example of domain versus operational coupling. An *Orthogonal Reuse* pattern presents a way to reuse some aspect represented by one or more concerns in the architecture that doesn’t fit within the preferred hierarchical organization. For example, microservices architectures are organized around domains, but operational coupling requires cutting across those domains. A sidecar allows an architect to isolate those concerns in a cross-cutting, but consistent, layer through the architecture.

Orthogonal Coupling

In mathematics, two lines are *orthogonal* if they intersect at right angles, which also implies independence. In software architecture, two parts of an architecture may be *orthogonally coupled*: they may have two distinct purposes that must still intersect to form a complete solution. The obvious example from this chapter is an operational concern such as monitoring, which is necessary but independent from domain behavior, like catalog checkout. Recognizing orthogonal coupling allows architects to find the intersection points that cause the least entanglement between concerns.

While the Sidecar pattern offers a nice abstraction, it has trade-offs like all other architectural approaches, as shown in [Table 20-1](#).

Table 20-1. Trade-offs for the Sidecar and Service Mesh patterns

| Advantages | Disadvantages |
|--|--|
| Offers a consistent way to create isolated coupling | Must implement a sidecar per platform |
| Allows consistent infrastructure coordination | Sidecar component may grow large/complex |
| Ownership per team, centralized, or some combination | Implementation “drift” between independent teams |

Both the Hexagonal and Service Mesh patterns illustrate ways to implement the reuse pattern to separate domain from operational concerns. The Hexagonal implementation is general purpose, while the Service Mesh is well suited to microservices and other distributed architectures. The key for architects is to identify the pattern first—separation—and then decide how best to implement it in their architecture.

Communication

Many architecture patterns, including communication patterns, come from event-driven architecture and apply to any distributed architecture that communicates via messages and/or events, such as discussed in [Chapters 15](#) and [18](#). In fact, you’ve seen examples of communication in each of those chapters—we just haven’t identified them as particular patterns because architects commonly implement patterns without realizing it. Patterns are, after all, solutions to common problems.

Orchestration Versus Choreography

Consider two communication patterns you’ve already seen, in both “[Mediated Event-Driven Architecture](#)” on page [258](#) and “[Choreography and Orchestration](#)” on page [341](#): choreography and orchestration, as summarized in [Figure 20-2](#).

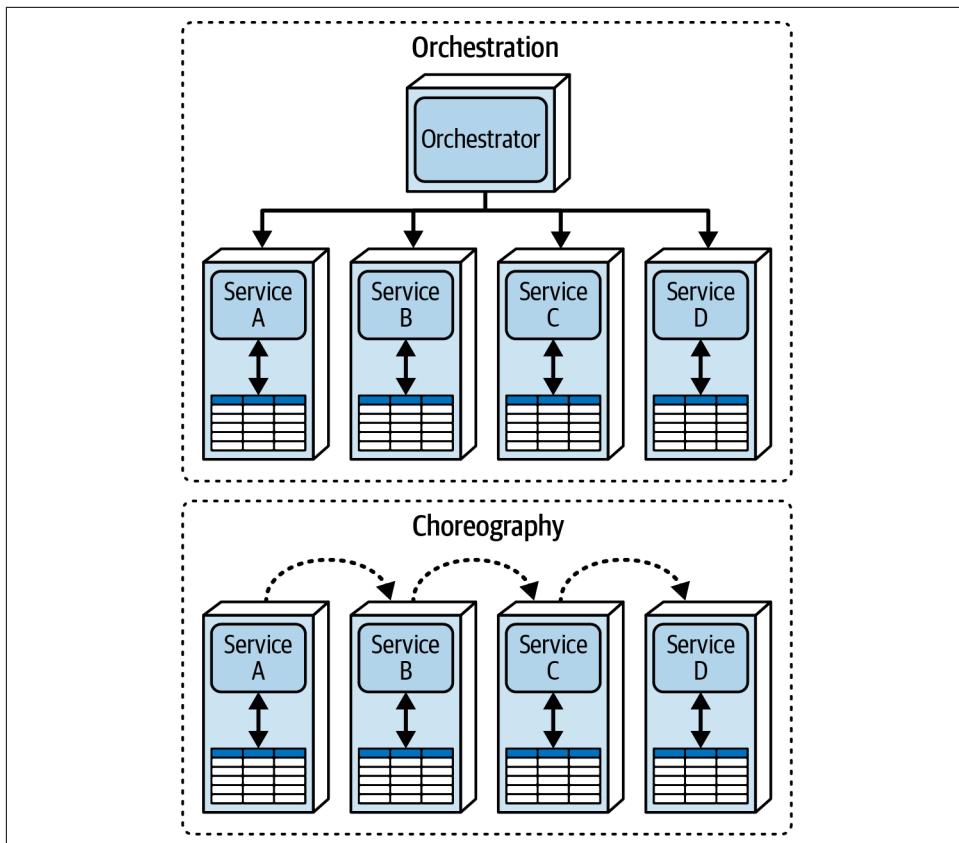


Figure 20-2. Orchestration and choreography in a microservices architecture

In each of the isomorphic workflows shown in Figure 20-2, four domain services (Services A through D) need to collaborate to form a workflow. In the orchestration case, there's also a separate service that acts as a coordinator for the workflow: the *orchestrator*.

While we've described this communication as both *orchestration* and *mediation*, the pattern remains the same. Architects benefit from being able to recognize patterns lurking inside implementations because their trade-offs become more apparent.

Notice that when we described the trade-offs for mediation and orchestration, we touched on many of the same issues. We'll summarize the advantages here:

Centralized workflow

As complexity goes up, architects benefit from utilizing a unified component for state, behavior, and boundary conditions.

Error handling

Error handling, a major part of many domain workflows, is assisted by having a state owner for the workflow.

Recoverability

Because an orchestrator monitors the state of the workflow, if one or more domain services suffers from a short-term outage, the architect can add logic to retry.

State management

Having an orchestrator makes the state of the workflow queryable, providing a place for other workflows and other transient states.

General disadvantages of orchestration include:

Responsiveness

All communication must go through the orchestrator, which has the potential to create a throughput bottleneck that can harm responsiveness.

Fault tolerance

While orchestration enhances recoverability for domain services, it creates a potential single point of failure for the workflow. This can be addressed with redundancy, but that also adds more complexity.

Scalability

This communication style doesn't scale as well as choreography because the orchestrator adds more coordination points, which cuts down on potential parallelism.

Service coupling

Having a central orchestrator creates tighter coupling between it and the domain components, which is sometimes necessary but is frowned upon in microservices architectures.

Similarly, we discussed choreography in both microservices and event-driven architectures. The trade-offs for choreographed workflows include:

Responsiveness

This communication style has fewer single chokepoints, thus offering more opportunities for parallelism.

Scalability

The lack of coordination points like orchestrators allows more independent scaling.

Fault tolerance

The lack of a single orchestrator allows the architect to use multiple instances to enhance fault tolerance. They could, of course, create multiple orchestrators, but because all communication must go through them, multiple orchestrators are more sensitive to the workflow's overall level of fault tolerance.

Service decoupling

No orchestrator means less coupling.

Disadvantages of the choreography communication style include:

Distributed workflow

Having no workflow owner makes managing errors and other boundary conditions more difficult.

State management

Having no centralized state holder hinders ongoing state management.

Error handling

Error handling is more difficult without an orchestrator because the domain services must have more workflow knowledge.

Recoverability

Recoverability becomes more difficult without an orchestrator to attempt retries and other remediation efforts.

These two patterns nicely illustrate our distinction between styles and patterns. Any distributed architecture can use any of these communication patterns, and architects should understand how to evaluate their trade-offs. Remember our Second Law of Software Architecture: *you can't just do trade-off analysis once and be done with it*. They also illustrate that common patterns exist everywhere, which is why they are common.

CQRS

Another common communication pattern that appears in many distributed architectures (and a few monolithic ones) is *Command-Query-Responsibility-Segregation* (CQRS). This simple communication and data pattern splits a commonly monolithic communication with a database into two, illustrated in [Figure 20-3](#).

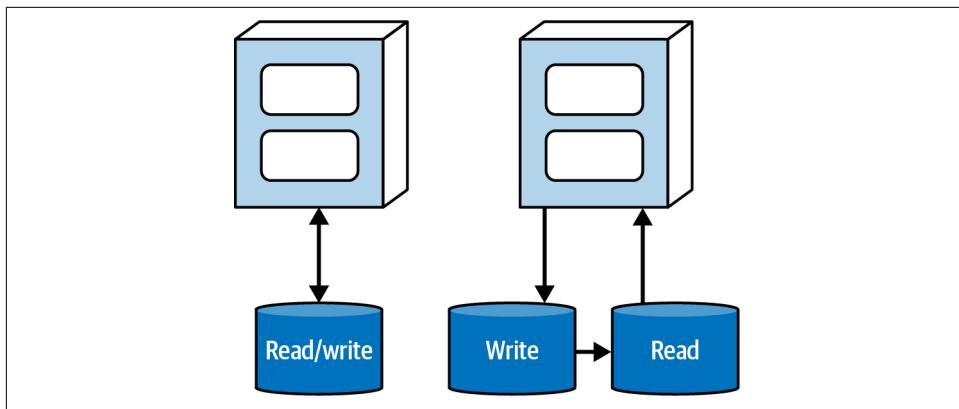


Figure 20-3. Client/server versus CQRS

In [Figure 20-3](#), the structure on the left illustrates a common *client/server* data interaction, where the application queries the database and performs transactional writes by utilizing the database as part of the application infrastructure. While this is a common pattern, some systems have stark differences between read and write volumes, or want to isolate reads from writes for the sake of security and other concerns. For those systems, CQRS, illustrated on the right in [Figure 20-3](#), solves this problem.

CQRS isolates writes into one datastore (usually a database; sometimes another infrastructure, such as a durable message queue), which synchronizes the data to another database (usually asynchronously), which services read requests.

By separating reads and writes, architects can isolate different architectural characteristics, depending on the data. This also allows them to use different data models for each database if necessary.

CQRS is a good example of a data communication pattern that facilitates differing architectural characteristics for different types of data capabilities, security concerns, or other factors that benefit from physical separation.

Infrastructure

Software architecture has patterns in every place where teams have found useful contextualized solutions to common problems, and these patterns often intersect with other parts of the ecosystem (see [Chapter 26](#) for a complete discussion).

Architects care about coupling: between components, data elements, APIs...and infrastructure, as exemplified by the *Broker-Domain pattern*.

Broker-Domain Pattern

In this section, we'll look at the same order-placement workflow, this time implemented with an event-driven architecture, as illustrated in [Figure 20-4](#).

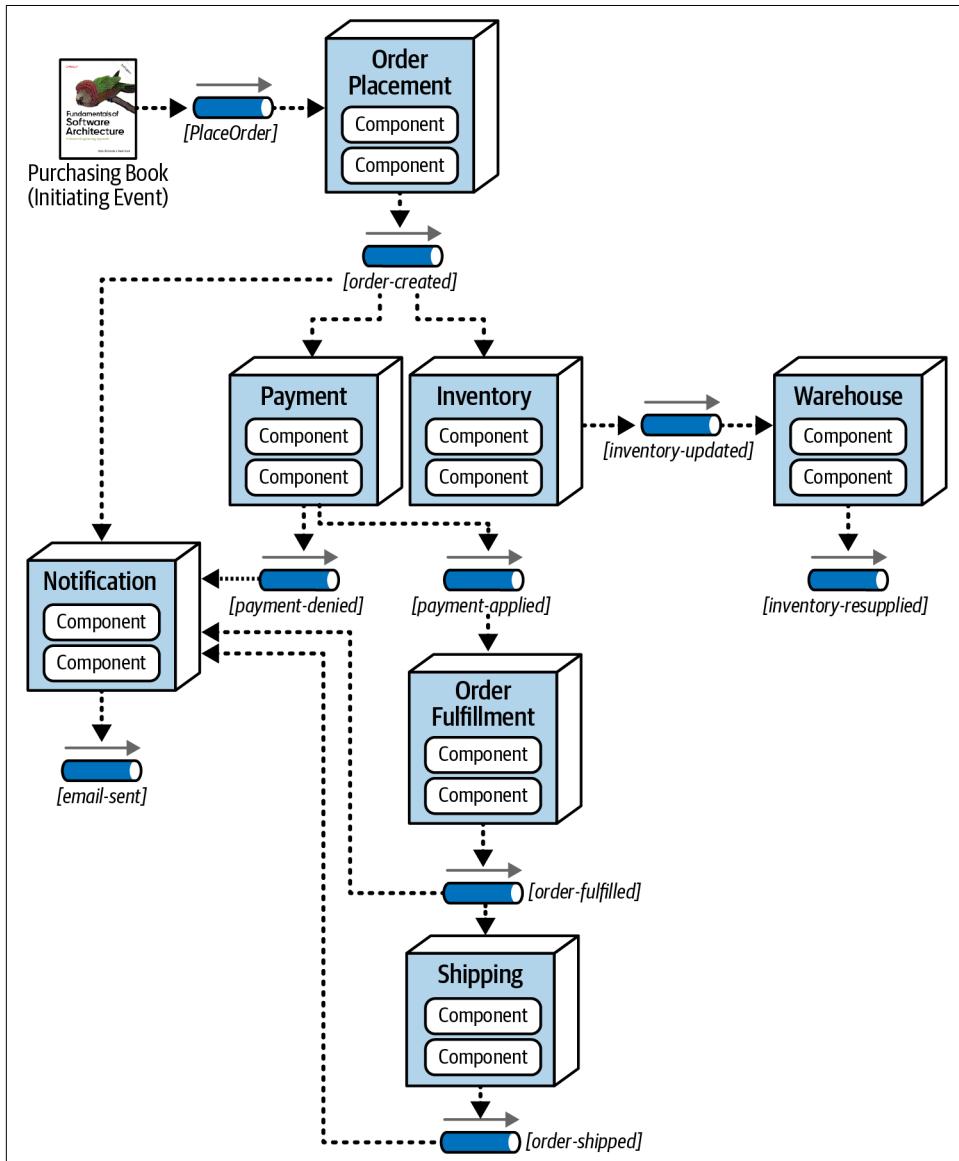


Figure 20-4. An order placement workflow implemented in EDA

As you've seen, EDA uses events to communicate between services, so event handlers need to subscribe to the proper services to build the workflow. Event handlers are implemented by *brokers*, which are part of the infrastructure of the architecture. In EDA, the topic or queue is typically owned by the sender: for example, Payment needs to know a topic's address to subscribe to it.

In [Figure 20-5](#), OrderPlacement "owns" the broker to which other processors will subscribe; in other words, the infrastructure required to support this service includes the broker. If a system uses only one broker for all communication, all of its services depend on a single part of the infrastructure.

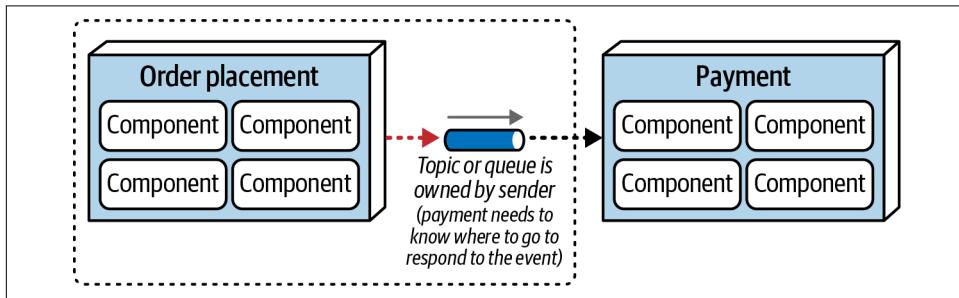


Figure 20-5. In event-driven choreography, the topic or queue is typically owned by the sender

The infrastructure for the workflow depicted in [Figure 20-6](#) includes a single broker, meaning that each event processor “knows” where to go to subscribe to workflow collaborators. A single broker also allows a single place for logging, monitoring, and other governance.

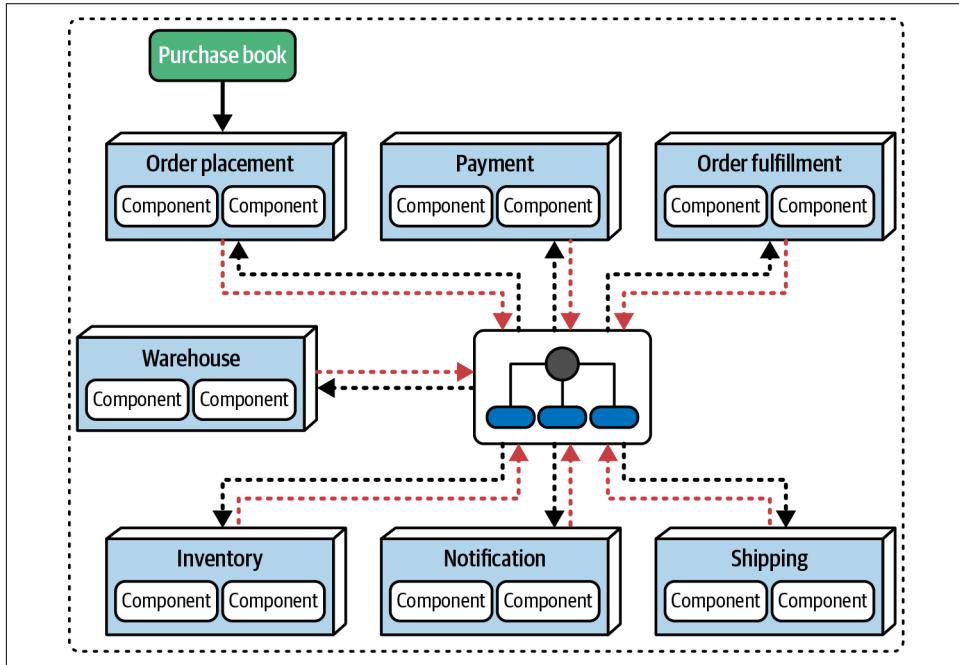


Figure 20-6. Using one broker for the entire workflow

However, one of the goals of a distributed architecture is to improve fault tolerance. What happens if the single broker in [Figure 20-6](#) goes down? The entire workflow stops working. Another potential issue is scale. If all messages must go through a single broker, there's a risk that the broker will become swamped as the volume of messages increases.

An alternative approach is the *Domain-Broker* pattern, which treats infrastructure in a similar manner to the granularity of domains. Consider Figure 20-7.

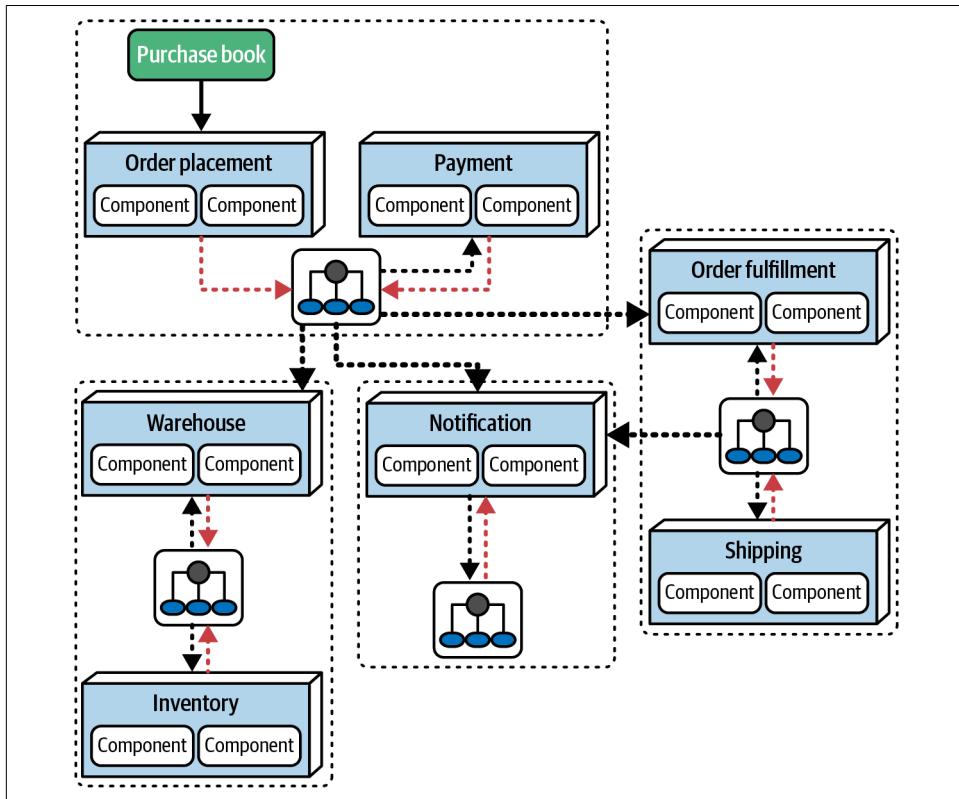


Figure 20-7. Utilizing the Domain-Broker pattern to assign ownership to infrastructure

In the alternative architecture shown in Figure 20-7, each group of related services shares a broker, reflecting the architecture's overall domain partitioning. This solution still allows good discovery while increasing fault tolerance, scalability, elasticity, and a host of other operational architectural characteristics. Note, however, that neither of these approaches is a “best practice.” The Single-Broker pattern’s trade-offs appear in Table 20-2.

Table 20-2. Single-Broker pattern trade-offs

| Advantages | Disadvantages |
|-------------------------------|-------------------|
| Centralized discovery | Fault tolerance |
| Least possible infrastructure | Throughput limits |

The Domain-Broker pattern also features trade-offs, shown in [Table 20-3](#).

Table 20-3. Domain-Broker pattern trade-offs

| Advantages | Disadvantages |
|---------------------------|---|
| Better isolation | More difficult discovery of queues/topics |
| Matches domain boundaries | More infrastructure = more expensive |
| More scalable | More moving parts to maintain |

Architects must balance discovery with the need to isolate domains as they decide which of these infrastructure patterns make the most sense for their particular systems.

PART III

Techniques and Soft Skills

An effective software architect must not only understand the technical aspects of software architecture, but also the primary techniques and soft skills necessary to think like an architect, guide development teams, and effectively communicate the architecture to various stakeholders. This section of the book addresses the key techniques and soft skills necessary to become an effective software architect.

Architectural Decisions

One of the core expectations of an architect is to make architectural decisions. Architectural decisions usually involve the structure of the application or system, but they may involve technology decisions as well, particularly when those technology decisions impact architectural characteristics. Whatever the context, a good architectural decision is one that helps guide development teams in making the right technical choices. Making architectural decisions involves gathering enough relevant information, justifying the decision, documenting the decision, and effectively communicating that decision to the right stakeholders.

Architectural Decision Antipatterns

The programmer [Andrew Koenig](#) defines an *antipattern* as something that seems like a good idea when you begin, but leads you into trouble. Another definition of an antipattern is a repeatable process that produces negative results. The three most common architectural decision antipatterns that can (and usually do) emerge when an architect makes decisions are the Covering Your Assets antipattern, the Groundhog Day antipattern, and the Email-Driven Architecture antipattern. These three antipatterns usually follow a progressive flow: overcoming the Covering Your Assets antipattern leads to the Groundhog Day antipattern, and overcoming that leads to the Email-Driven Architecture antipattern. Making effective and accurate architectural decisions requires overcoming all three.

The Covering Your Assets Antipattern

The Covering Your Assets antipattern occurs when an architect avoids or defers making an architectural decision out of fear of making the wrong choice. There are two ways to overcome this. The first is to wait until the *last responsible moment* to make an important architectural decision: that is, when there's enough information

to justify and validate the decision, but not so long that it holds up development teams or sends the architect into the *Analysis Paralysis* antipattern, where they are stuck forever in analyzing the decision. A good way to determine the last responsible moment is to ask when the cost of deferring the decision exceeds the risk associated with deciding. As illustrated in Figure 21-1, notice that in the early stages of the decision-making time scale, the cost (denoted by the solid line) is low because there's less time spent on making the decision, but the risk (denoted by the dotted line) is high because less is known about the problem or solution. Spending more time deferring the decision increases the cost, but also reduces risk because the architect can make a more complete analysis of the problem and possible alternatives. The time to make a decision is where these two factors intersect and the cost increase exceeds the reduced risk.

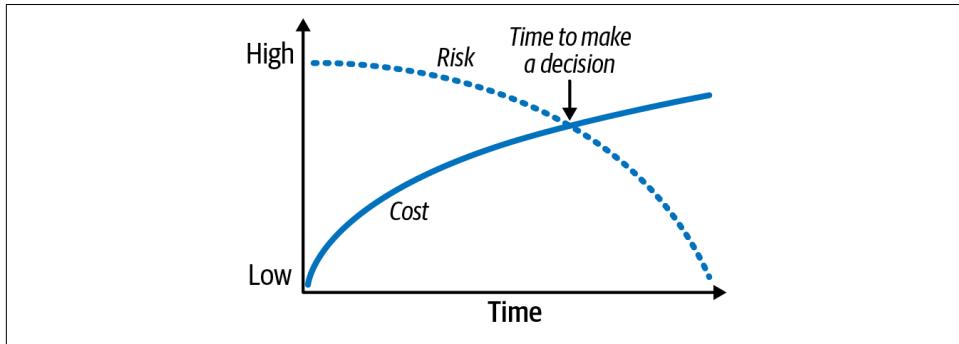


Figure 21-1. Last responsible moment

Another way to avoid this antipattern is to collaborate with development teams to ensure that the decision can be implemented as expected. This is vitally important because no architect can possibly know every single detail about any issue associated with a particular technology. By closely collaborating with development teams, the architect can respond quickly, gain more insight, and reduce the risk of the decision being the incorrect one.

To illustrate this point: suppose you, as the architect, decide that all product-related reference data (such as product description, weight, and dimensions) should be cached in all service instances needing that information. You decide that this will be done using a read-only replicated cache, with the primary cache owned by the Catalog service. (A *replicated*, or *in-memory*, cache means that if there are any changes to product information or new products added, the Catalog service updates its cache, which is then replicated to all other services requiring that data through a replicated cache product.) Your justification for this decision is to reduce coupling between the services and to share data effectively without having to make an interservice call. However, the development teams implementing this architectural decision find that, due to some services' scalability requirements, this decision would require more

in-process memory than is available. Because you are closely collaborating with these teams, you quickly become aware of the issue and adjust your architectural decision accordingly.

Groundhog Day Antipattern

The Groundhog Day antipattern occurs when people don't know why an architect made a particular decision, so they keep discussing it over and over and over, never coming to a final resolution or agreement. It gets its name from the 1993 movie *Groundhog Day*, in which Bill Murray's character must relive February 2 over and over, every day.

This antipattern occurs because architects fail to justify their decision (or fail to justify it completely). It's important to provide both technical and business justifications for an architectural decision.

For example, say you decide to break apart a monolithic application into separate services. Your justification is to decouple the functional aspects of the application so that each part of the application uses fewer virtual machine resources and can be maintained and deployed separately. While this is a good technical justification, what's missing is the *business* justification—in other words, why should the business pay for this architectural refactoring? A good business justification for this decision might be to deliver new business functionality faster, thereby improving time to market. Another might be to reduce the costs associated with developing and releasing new features.

Providing the business value is vitally important when justifying an architectural decision. It is also a good litmus test for determining whether the architectural decision should be made in the first place. If it doesn't provide any business value, perhaps the architect should reconsider the decision.

Four of the most common business justifications are cost, time to market, user satisfaction, and strategic positioning. Consider what's important to the business stakeholders. Justifying a particular decision based on cost savings alone might not be the right call if the business stakeholders are more concerned about time to market.

Email-Driven Architecture Antipattern

Once an architect makes and fully justifies their decisions, another architecture antipattern often emerges: Email-Driven Architecture. This antipattern occurs when people lose or forget an architectural decision, or don't even know that it's been made and therefore cannot possibly implement it. Overcoming this antipattern is all about communicating architectural decisions effectively. Email is a great tool for communication, but it makes a poor document repository system.

Fortunately, an architect can easily avoid the Email-Driven Architecture antipattern by learning how to communicate architectural decisions effectively. First, be sure not to include the decision in the body of an email. Doing so creates multiple systems of record for that decision, because each email contains a copy of the decision rather than having it in only one place. Many such emails leave out important details about the decision (including the justification), creating the Groundhog Day antipattern all over again. Also, if that architectural decision is ever changed or superseded, it's difficult to know whether all the relevant people receive the revised decision.

A better approach is to mention only the nature and context of the decision in the body of the email and provide a link to the single system of record, where the architectural decision and corresponding details are stored (whether that's a link to a wiki page or a reference to a document in a filesystem).

Consider the following email about an architectural decision:

“Hi, Sandra, I've made an important decision regarding communication between services that directly impacts you. Please see the decision using the following link....”

Notice that, in the phrasing of the beginning of the first sentence, the context is mentioned (communication between services), but not the actual decision itself. The second part of the first sentence is also important: if an architectural decision doesn't directly impact the person, then why bother that person with it? This is a great litmus test for determining which stakeholders (including developers) should be notified directly of an architectural decision. The second sentence in this example provides a link to the single location of the architectural decision, providing a single system of record for decisions.

Architectural Significance

Many architects believe that if a decision involves a specific technology, then it's not an architectural decision, but rather a technical decision. This is not always true. If an architect decides to use a particular technology because it directly supports a particular architecture characteristic (such as performance or scalability), then it's still an architectural decision.

Michael Nygard, a well-known software architect and author of the second edition of *Release It!* (Pragmatic Bookshelf, 2018), addresses the problem of what decisions an architect should be responsible for (and hence what constitutes an architectural decision) by coining the term *architecturally significant*. According to Nygard, architecturally significant decisions are those that affect a system's structure, non-functional characteristics, dependencies, interfaces, or construction techniques.

Structure, in this context, refers to decisions that affect the architecture patterns or styles being used. For example, a decision by an architect to share code between a set

of microservices impacts the bounded context of the microservice, and thus affects the structure of the system.

The system's *non-functional characteristics* are the architectural characteristics that are important for the system being developed or maintained. For example, if a choice of technology affects performance, and performance is an important aspect of the application, that choice becomes an architectural decision, even though it may specify a particular product, framework, or technology.

Dependencies refer to the coupling points between components and/or services within the system. Dependencies can affect architecture characteristics such as scalability, modularity, agility, testability, reliability, and so on, so decisions about dependencies become architectural decisions.

Interfaces refer to how services and components are accessed and orchestrated: usually through a gateway, integration hub, service bus, adapter, or API proxy. Making decisions about interfaces usually involves defining contracts, including versioning and deprecation strategies. Interfaces impact others using the system and hence are architecturally significant.

Finally, *construction techniques* refer to decisions about platforms, frameworks, tools, and even processes that, although technical in nature, might impact some aspect of the architecture.

Architectural Decision Records

One of the most effective ways of documenting architectural decisions is through *Architectural Decision Records* ([ADRs](#)). Michael Nygard first evangelized ADRs in a 2011 [blog post](#), and in 2017, [Thoughtworks Technology Radar](#) recommended the technique for widespread adoption.

An ADR consists of a short text file (usually one to two pages long) describing a specific architectural decision. While ADRs can be written using plain text or in a wiki page template, they are usually written in some sort of text document format, like [AsciiDoc](#) or [Markdown](#).

Tooling is also available for managing ADRs. Nat Pryce, coauthor of *Growing Object-Oriented Software, Guided by Tests* (Addison-Wesley, 2009), has written an open source tool called [ADR Tools](#) that provides a command-line interface to manage ADRs, including numbering schemes, locations, and superseded logic. Micha Kops, a software engineer from Germany, provides some [great examples](#) of using ADR tools to manage architectural decision records.

Basic Structure

The basic structure of an ADR consists of five main sections: *Title*, *Status*, *Context*, *Decision*, and *Consequences*. We usually add two additional sections as part of the basic structure: *Compliance* and *Notes*. The Compliance section is a space to think about and document how the architectural decision will be governed and enforced (manually or through automated fitness functions). The Notes section is a space to include metadata about the decision, such as the author, who approved it, when it was created, and so on.

It's fine to extend this basic structure (as illustrated in [Figure 21-2](#)) to include any other needed section. Just keep the template consistent and concise. A good example of this might be to add an *Alternatives* section analyzing all the other possible solutions.

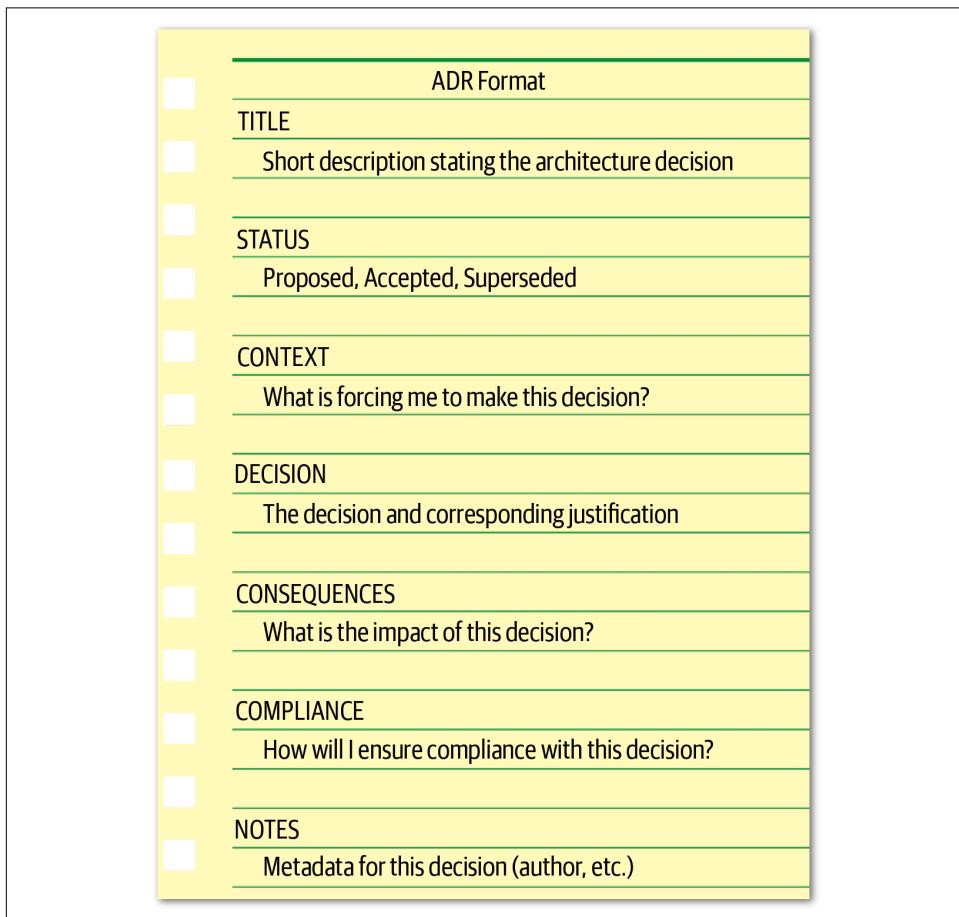


Figure 21-2. Basic ADR structure

Title

ADR titles are usually numbered sequentially and contain a short phrase describing the architectural decision. For example, an ADR title describing a decision to use asynchronous messaging between the Order service and the Payment service might read: “42. Use of Asynchronous Messaging Between Order and Payment Services.” The title should be short and concise, but descriptive enough to remove any ambiguity about the nature and context of the decision.

Status

Every ADR has one of three statuses: *Proposed*, *Accepted*, or *Superseded*. The *Proposed* status means the decision must be approved by a higher-level decision maker or some sort of architectural governance body (such as an architecture review board). *Accepted* means the decision has been approved and is ready for implementation. *Superseded* means the decision has been changed and superseded by another ADR. The *Superseded* status always assumes that the prior ADR’s status was *Accepted*; in other words, a *Proposed* ADR would never be superseded by another ADR; it would be modified until *Accepted*.

The *Superseded* status is a powerful way of keeping historical records of what decisions have been made, why they were made at the time, what the new decision is, and why it was changed. Usually, when an ADR has been superseded, it is marked with the number of the decision that superseded it. Similarly, the decision that supersedes another ADR is marked with the number of the ADR it superseded.

For example, let’s say ADR 42 (“Use of Asynchronous Messaging Between Order and Payment Services”) is in *Approved* status. Due to later changes to the implementation and location of the Payment service, you decide that REST should now be used between the two services. You therefore create a new ADR (number 68) to document this changed decision. The corresponding statuses look as follows:

ADR 42. Use of Asynchronous Messaging Between Order and Payment Services

Status: Superseded by 68

ADR 68. Use of REST Between Order and Payment Services

Status: Accepted, supersedes 42

The link and history trail between ADRs 42 and 68 lets you avoid the inevitable “what about using messaging?” question regarding ADR 68.

ADRs and Request for Comments (RFC)

Sending a draft ADR out or comments can help an architect validate their assumptions and assertions with a larger audience of stakeholders. An effective way of engaging developers and initiating collaboration is creating a new kind of status called *Request for Comments* (RFC) and specifying a deadline for reviewers to complete their feedback. Once that date is reached, the architect can analyze the comments, make any necessary adjustments to the decision, make the final decision, and set the status to Proposed (or Accepted, if the architect has authority to approve the decision).

An ADR with RFC status would look as follows:

STATUS

Request For Comments, Deadline 09 JAN 2026

Another significant aspect of the Status section of an ADR is that it forces the architect and their boss or lead architect to discuss the criteria for approving an architectural decision and determine whether the architect can do so on their own, or whether it must be approved through a higher-level architect, architecture review board, or some other governing body.

Three good starting places for these conversations are cost, cross-team impact, and security. Cost should include software purchase or licensing fees, additional hardware costs, and the overall level of effort to implement the architectural decision. To estimate this, multiply the estimated number of hours to implement the architectural decision by the company's standard *full-time equivalency* (FTE) rate. The project owner or project manager usually has the FTE amount. This conversation might lead everyone to agree that, for example, if the cost of the architectural decision exceeds a certain amount, then it must be set to a Proposed status and approved by someone else. If the architectural decision impacts other teams or systems or has any sort of security implications, then it must be approved by a higher-level governing body or lead architect.

Once the team establishes and agrees on the criteria and corresponding limits (such as “costs exceeding \$5,000 must be approved by the architecture review board”), document it well so that all architects creating ADRs will know when they can and cannot approve their own architectural decisions.

Context

The Context section of an ADR specifies the forces at play. In other words, “What situation is forcing me to make this decision?” This section of the ADR allows the architect to describe the specific circumstances and concisely elaborate on the possible alternatives. If the architect is required to document the analysis of each

alternative in detail, add an Alternatives section rather than including that analysis in the Context section.

The Context section also provides a place to document a specific area of the architecture itself. By describing the context, the architect is also describing the architecture. Using the example from the prior section, the Context section might read as follows: “The Order service must pass information to the Payment service to pay for an order currently being placed. This could be done using REST or asynchronous messaging.” Notice that this concise statement specifies not only the scenario, but also the alternatives considered.

Decision

The Decision section of the ADR contains a description of the architectural decision, along with a full justification. Nygard recommends stating architectural decisions in an affirmative, commanding voice rather than a passive one. For example, the decision to use asynchronous messaging between services would read, “*We will use asynchronous messaging between services.*” This is much better than “*I think asynchronous messaging between services would be the best choice,*” which does not make clear what the decision is or if a decision has even been made—only the opinion of the architect.

One of the most powerful aspects of the Decision section of ADRs is that it lets the architect emphasize the justification for the decision. Understanding *why* a decision was made is far more important than understanding *how* something works. This helps developers and other stakeholders better understand the rationale behind a decision, and therefore makes them more likely to agree with it.

To illustrate this point, let’s say you decide to use Google’s Remote Procedure Call ([gRPC](#)) to communicate between two particular services to reduce network latency due to very high responsiveness needs. Several years later, a new architect on the team decides to use REST instead of gRPC to make communications between services more consistent. Because the new architect doesn’t understand *why* gRPC was chosen in the first place, their decision ends up having a significant impact on latency, causing timeouts in upstream systems. If the new architect had access to an ADR, they would have understood that the original decision to use gRPC was meant to reduce latency (at the cost of tightly coupled services) and could have prevented this problem.

Consequences

Every decision an architect makes has some sort of impact, good or bad. The Consequences section of an ADR forces an architect to describe the overall impact of an architectural decision, allowing them to think about whether the negative impacts outweigh the benefits.

This section is also a good place to document the trade-off analysis performed during the decision-making process. For example, let's say you decide to use asynchronous (fire-and-forget) messaging for posting reviews on a website. Your justification for this decision is to improve responsiveness (from 3,100 milliseconds down to 25 milliseconds) because users wouldn't have to wait for the actual review to be posted—only for the message to be sent to a queue. One member of your development team argues that this is a bad idea due to the complexity of the error handling associated with an asynchronous request: "What happens if someone posts a review with some bad words?" What this team member doesn't know is that you discussed that very problem with the business stakeholders and other architects when analyzing the trade-offs of this decision, and decided together that it was better to improve responsiveness and deal with complex error handling rather than increase the wait time and provide feedback as to whether the review post was successful or not. If this decision was documented using an ADR, you could have provided this trade-off analysis in the Consequences section, preventing this sort of disagreement.

Compliance

The Compliance section is not one of the standard sections of an ADR, but it's one we highly recommend adding. The Compliance section states how the architectural decision will be measured and governed. Will the compliance check for this decision be manual, or can it be automated using a fitness function? If it can be automated, the architect can specify how the fitness function will be written, along with any other changes to the code base needed to measure this architectural decision for compliance.

For example, suppose you make the decision within a traditional n-tiered layered architecture (as illustrated in [Figure 21-3](#)) that all shared objects used by business objects in the Business layer must reside in the Shared Services layer to isolate and contain shared functionality.

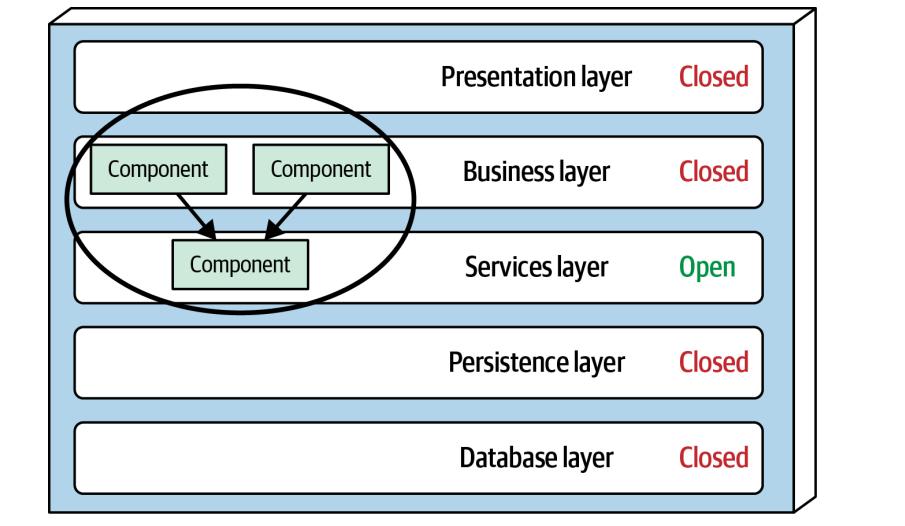


Figure 21-3. An example of an architectural decision

This architectural decision can be measured and governed using a host of automation tools, including [ArchUnit](#) in Java and [NetArchTest](#) in C#. With ArchUnit in Java, the automated fitness-function test for this architectural decision might look like this:

```
@Test
public void shared_services_should_reside_in_services_layer() {
    classes().that().areAnnotatedWith(SharedService.class)
        .should().resideInAPackage("..services..")
        .check(myClasses);
}
```

This automated fitness function would require writing new stories to create a Java annotation (`@SharedService`) and add it to all shared classes to support this method of governance.

Notes

Another section that is not part of a standard ADR but that we highly recommend adding is a Notes section. This section includes various metadata about the ADR:

- Original author
- Approval date
- Approved by
- Superseded date
- Last modified date
- Modified by
- Last modification

Even when storing ADRs in a version-control system (such as Git), it's useful to have additional metadata beyond what the repository can support. We recommend adding this section regardless of how and where the architect stores ADRs.

Example

Our example of the Going, Going, Gone (GGG) auction system includes dozens of architectural decisions. Splitting up the bidder and auctioneer user interfaces, using a hybrid architecture consisting of event-driven and microservices, leveraging the Real-time Transport Protocol (RTP) for video capture, using a single API Gateway, and using separate queues for messaging are just a few of the architectural decisions an architect would make. Every architectural decision an architect makes, no matter how obvious, should be documented and justified.

Figure 21-4 illustrates one of the architectural decisions within the GGG auction system: using separate point-to-point queues between the bid capture, bid streamer, and bid tracker services rather than a single publish-and-subscribe topic (or even REST, for that matter).

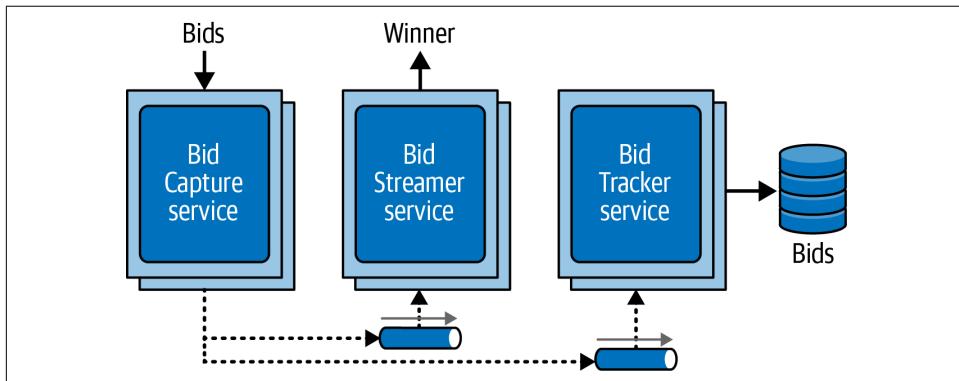


Figure 21-4. Use of pub/sub between services

Without an ADR to justify this decision, others involved with designing and developing this system might disagree and decide to implement it in a different way.

The following is an example of an ADR for this architectural decision:

ADR 76. Separate Queues for Bid Streamer and Bidder Tracker Services

STATUS

Accepted

CONTEXT

The `Bid Capture` service, upon receiving a bid, must forward that bid to the `Bid Streamer` service and the `Bidder Tracker` service. This could be done using a single topic (pub/sub), separate queues (point-to-point) for each service, or REST via the Online Auction API layer.

DECISION

We will use separate queues for the `Bid Streamer` and `Bidder Tracker` services.

The `Bid Capture` service does not need any information from the `Bid Streamer` service or `Bidder Tracker` service (communication is only one-way).

The `Bid Streamer` service must receive bids in the exact order they were accepted by the `Bid Capture` service. Using messaging and queues automatically guarantees the bid order for the stream by leveraging first-in, first out (FIFO) queues.

Multiple bids come in for the same amount (for example, “Do I hear a hundred?”). The `Bid Streamer` service only needs the first bid received for that amount, whereas the `Bidder Tracker` needs all bids received. Using a topic (pub/sub) would require the `Bid Streamer` to ignore bids that are the same as the prior amount, forcing the `Bid Streamer` to store shared state between instances.

The `Bid Streamer` service stores the bids for an item in an in-memory cache, whereas the `Bidder Tracker` stores bids in a database. The `Bidder Tracker` will therefore be slower and might require backpressure. Using a dedicated `Bidder Tracker` queue provides this dedicated backpressure point.

CONSEQUENCES

We will require clustering and high availability of the message queues.

This decision will require the `Bid Capture` service to send the same information to multiple queues.

Internal bid events will bypass security checks done in the API layer.

UPDATE: Upon review at the January 14, 2025, ARB meeting, the ARB decided that this was an acceptable trade-off and that no additional security checks are needed for bid events between these services.

COMPLIANCE

We will use periodic manual code reviews to ensure that asynchronous pub/sub messaging is being used between the `Bid Capture` service, `Bid Streamer` service, and `Bidder Tracker` service.

NOTES

Author: Subashini Nadella

Approved: ARB Meeting Members, 14 JAN 2025

Last Updated: 14 JAN 2025

Storing ADRs

Once an architect creates an ADR, they need to store it somewhere. Regardless of where that is, each architectural decision should have its own file or wiki page. Some architects like to keep ADRs in the same Git repository as the source code, allowing the team to version and track ADRs as they would source code.

However, for larger organizations, we caution against this practice for several reasons. First, everyone who needs to see the architectural decision might not have access to the Git repository containing the ADRs. Second, the application's Git repository is not a good place to store ADRs that have a context outside of them (such as integration architectural decisions, enterprise architectural decisions, or decisions that are common to every application). For these reasons, we recommend storing ADRs in a dedicated ADR Git repository to which everyone has access, a wiki (using a wiki template), or a shared directory on a shared file server that can be accessed easily by a wiki or other document-rendering software.

Figure 21-5 shows what this directory structure (or wiki page navigation structure) might look like.

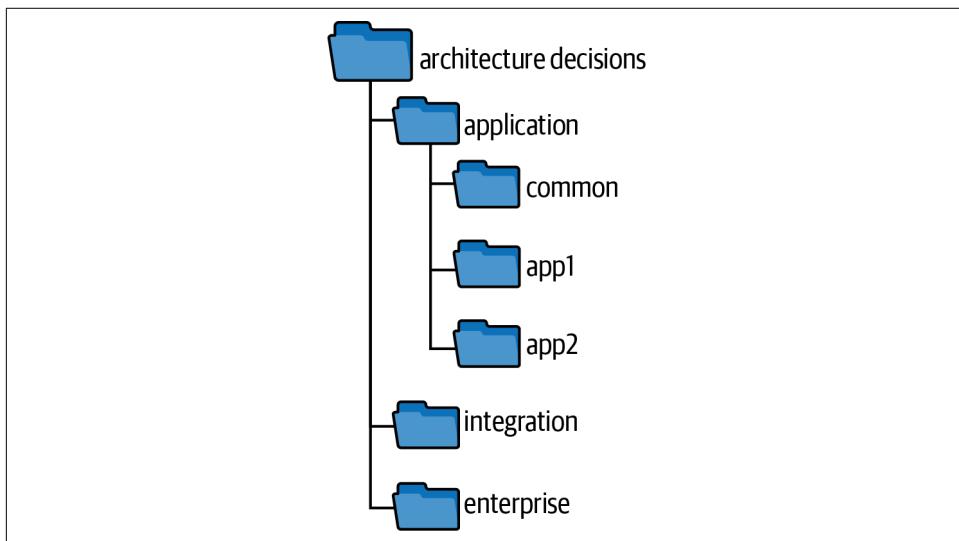


Figure 21-5. Example directory structure for storing ADRs

The *application* directory contains architectural decisions that are specific to some sort of application (or product) context. This directory is subdivided into further directories:

common

The *common* subdirectory is for architectural decisions that apply to all applications, such as, “All framework-related classes will contain an annotation (@Framework in Java) or attribute ([Framework] in C#) identifying the class as belonging to the underlying framework code.”

application

Subdirectories under the *application* directory correspond to the specific application or system context and contain architectural decisions specific to that application or system (in this example, the *app1* and *app2* applications).

integration

The *integration* directory contains ADRs that involve communication between application, systems, or services.

enterprise

Enterprise architecture ADRs are contained within the *enterprise* directory, indicating that these are global architectural decisions impacting all systems and applications. An example of an enterprise architecture ADR would be “All access to a system database will only be from the owning system,” preventing databases from being shared across multiple systems.

When storing ADRs in a wiki, the same structure applies, with each directory structure representing a navigational landing page. Each ADR is represented as a single wiki page within each navigational landing page (application, integration, or enterprise).

The directory and landing-page names indicated in this section are only recommendations and examples. Choose whatever names fit the company’s situation, as long as those names are consistent across teams.

ADRs as Documentation

Documenting software architecture has always been difficult. While some standards are emerging for diagramming architecture (such as software architect Simon Brown’s [C4 Model](#) or The Open Group [ArchiMate](#) standard), no agreed-upon standard exists for documenting software architecture. That’s where ADRs come in.

ADRs can be an effective means to document a software architecture. The Context section provides an excellent opportunity to describe the specific area of the system that requires an architectural decision to be made, as well as to describe the alternatives. More importantly, the Decision section describes the reasons why a particular decision is being made, which is by far the best form of architectural documentation. The Consequences section adds the final piece of the puzzle by describing the trade-off analysis for the decision—for example, the reasons (and trade-offs) for choosing performance over scalability.

Using ADRs for Standards

Very few developers like standards. Unfortunately, standards are sometimes more about control rather than providing a useful purpose. Using ADRs for standards can change this bad practice. For example, the Context section of an ADR describes the situation that is forcing the organization to adopt the particular standard. The Decision section of an ADR can thus indicate not only what the standard is, but more importantly, why it needs to exist.

This is a great way to qualify whether a particular standard should even exist in the first place. If an architect can't justify it, then perhaps it isn't a good standard to set and enforce. Furthermore, the more developers understand *why* a particular standard exists, the more likely they are to follow it (and, correspondingly, not challenge it). The Consequences section of an ADR is another great place to qualify whether a standard is valid: it requires the architect to think about and document the standard's implications and consequences, and whether the particular standard should be implemented or not.

Using ADRs with Existing Systems

Many architects question the usefulness of ADRs for existing systems. After all, architectural decisions have already been made and the system is in production. Do ADRs serve any real purpose at this point? In fact, they do. Remember, ADRs are more than just documentation—they help architects and developers understand *why* a decision was made and if it was the most appropriate one.

Start by writing some ADRs for the more significant architectural decisions made and questioning whether those decisions are the right ones or not. For example, maybe a group of services are sharing a single database. Why? Is there a good reason? Should the data be broken apart?

Part of the journey of incorporating ADRs into an existing system is doing a little investigative work to uncover these *why* questions. Unfortunately, the person who made the original decision might have left the company a long time ago, so no one knows the answer. In these cases it's up to the architect to identify and analyze the alternatives and the trade-offs of each option and attempt to validate (or invalidate) the existing decision. In either case, writing ADRs for these sorts of significant decisions starts to build up the justifications and rationales (and brain trust) for the system and can help identify architectural inefficiencies and incorrect system design.

Leveraging Generative AI and LLMs in Architectural Decisions

One of the many intriguing aspects of generative AI is whether architects can use it to help make and validate decisions. Should services use messaging, streaming, or event sourcing when sending downstream data? Should the database remain as a single

monolith, or should it be broken down into domain databases? Should **Payment Processing** be deployed as a single service or broken apart into multiple services, one for each payment type?

Most architects already know the answer to these questions—it depends! Coming back to our First Law of Software Architecture, *everything in software architecture is a trade-off*. Decisions like these depend on a lot of factors, including the specific context in which the decision is applied. Every situation and every environment is different, which is why there are no “best practices” for these sorts of structural questions.

Most LLMs base their results largely on probability. In other words, what is the most probable answer given the context of the prompt, and what is the “best practice” for this problem? However, probability and “best practices” have no place in making architectural decisions. Answering architectural questions requires a careful analysis of the trade-offs involved, and applying a specific business and technical context to identify the most appropriate choice. For example, if the business is most concerned about time to market (getting changes and new features out to customers as fast as possible), then *Maintainability* is going to be far more important than *Performance*, and will drive most of the decision-making process toward optimizing maintainability.

Architectural decisions require translating business concerns (such as time to market or sustained growth) into architectural characteristics (such as maintainability, testability, deployability, and so on). This translation is not always obvious, and getting it right requires years of experience. Once completed, it serves as a basis for trade-off analysis. For example, deciding whether to have a single service for payment processing or a service per payment type boils down to a trade-off between maintainability and performance: a single service provides better performance, but multiple services provide better maintainability. If the business is primarily concerned about time to market, maintainability is far more important than performance, so separate services would be the appropriate choice for this specific context.

Because of the very specific and individualistic nature of analyzing trade-offs and business context, it is difficult for generative AI as it currently exists to arrive at the most appropriate architectural decision. The best-case scenario, based on recent experiments your authors have done, is to have a generative AI tool outline the possible trade-offs of a decision, to assist in identifying any missed trade-offs. While generative AI tools have plenty of *knowledge*, they lack the *wisdom* required to make the most appropriate architectural decision.

Analyzing Architecture Risk

Every architecture comes with risks: some operational (such as availability, scalability, and data integrity), some structural (such as static coupling between logical components). Analyzing architecture risk is one of architects' most important activities, allowing them to address deficiencies and structural decay within the architecture and take corrective action. In this chapter, we show you some key techniques and practices for quantifying, assessing, and identifying risk, and introduce an activity called *risk storming*.

Risk Matrix

The first thing to determine in assessing architecture risk is its level: whether risk to a particular part of the architecture is low, medium, or high. The challenge here is that assessing risk can be *subjective*. One architect might hold the *opinion* that some aspect of the architecture is high risk, while another architect's *opinion* might be that the same aspect is medium risk. We italicize *opinion* here to emphasize the subjectivity of assessing risk. Fortunately, architects have a useful risk-assessment matrix that helps us make risk more measurable.

The architecture risk-assessment matrix ([Figure 22-1](#)) uses two dimensions to qualify risk: the overall impact of the risk involved and the likelihood of that risk occurring. The architect rates each dimension as low (1), medium (2), or high (3), then multiplies the numbers within each intersection of the matrix. This provides a numerical representation of risk, making the process of qualifying risk more *objective*. Ratings of 1 and 2 are considered low risk (usually depicted in green), numbers 3 and 4 are considered medium risk (usually yellow), and numbers 6 through 9 are considered high risk (usually red). Using shading can be helpful for grayscale rendering and for people unable to distinguish colors.

| | | Likelihood of risk occurring | | |
|------------------------|------------|------------------------------|------------|----------|
| | | Low (1) | Medium (2) | High (3) |
| Overall impact of risk | Low (1) | 1 | 2 | 3 |
| | Medium (2) | 2 | 4 | 6 |
| | High (3) | 3 | 6 | 9 |

Figure 22-1. Matrix for determining architecture risk

To show you its utility, we'll use the risk matrix in an example. Suppose you're concerned about the availability of your application's primary central database.



When using this matrix to qualify risk, consider impact first and likelihood second. If you are unsure of the likelihood, use a high (3) rating until you can confirm.

First, you'll consider overall impact: what happens if the database goes down or becomes unavailable? Let's say you might deem the impact high risk and map that risk to the last row of the matrix in [Figure 22-1](#) as a 3 (medium), 6 (high), or 9, (high). However, when you consider the second dimension—the likelihood of that risk occurring—you realize that the database is on highly available servers in a clustered configuration, and the *likelihood* that the database would become unavailable is low (first column in the matrix). The intersection between high impact and low likelihood gives you an overall risk rating of 3 (medium risk) for availability in the primary central database.

Risk Assessments

You can use the risk matrix to build what is called a *risk assessment*: a summarized report of an architecture's overall risk, with meaningful assessment criteria based on a context (services, subdomain areas, or domain areas of a system). We've performed many risk assessments, and have found architectural characteristics to be excellent

risk-assessment criteria. Why spend time analyzing performance risk when the system's critical architectural characteristics are scalability, elasticity, and data integrity? Knowing characteristics such as those described in [Chapter 4](#) is the first step in analyzing architectural risk.



The architectural characteristics that are most critical for the architecture to support make great risk-assessment criteria.

The basic format of a risk-assessment report is shown in [Figure 22-2](#). Here, 1 and 2 represent low risk, 3 and 4 represent medium risk, and 6 and 9 represent high risk. The risk criteria run along the left side of the spreadsheet, and the context runs across the top.

| RISK CRITERIA | Customer registration | Catalog checkout | Order fulfillment | Order shipment | TOTAL RISK |
|----------------|-----------------------|------------------|-------------------|----------------|------------|
| Scalability | (2) | (6) | (1) | (2) | 11 |
| Availability | (3) | (4) | (2) | (1) | 10 |
| Performance | (4) | (2) | (3) | (6) | 15 |
| Security | (6) | (3) | (1) | (1) | 11 |
| Data integrity | (9) | (6) | (1) | (1) | 17 |
| TOTAL RISK | 24 | 21 | 8 | 11 | |

Figure 22-2. Example of a standard risk assessment

We use an ecommerce ordering system as the example for this risk assessment. It assesses five criteria, representing the system's critical architectural characteristics. Across the top are four different contexts, each representing a separate domain (customer registration, catalog checkout, order fulfillment, and order shipping). A domain or subdomain context works well; analyzing risk at the level of services is usually too fine-grained and doesn't account for risk involving communication or coordination between multiple services.

The nice thing about using quantified risk is that it considers both the risk criteria and the context. For example, in [Figure 22-2](#), the total accumulated risk for data

integrity is 17, making it the highest risk area from a criteria standpoint. The accumulated risk for availability is only 10 (the lowest risk criteria-wise). In terms of the relative risk of each context area, however, customer registration is the domain that carries the highest risk context-wise, whereas order fulfillment is the lowest-risk context. This is useful information when determining priorities and where to put additional effort into reducing risk.

This risk assessment example contains all of the risk-analysis results, but it's sometimes useful to filter out details to highlight certain problems. For example, suppose that you, as the architect of this system, are in a meeting, presenting to stakeholders about the high-risk areas of the system. Rather than presenting the full risk assessment, as [Figure 22-2](#) does, you could filter out the low- and medium-risk areas (the noise) to highlight the high-risk areas (the signal). Improving the overall signal-to-noise ratio lets you deliver a more effective, less distracting message. [Figure 22-3](#) shows a filtered version of the same risk assessment. Compare the two images to see how much clearer the message is with the second filtered assessment.

| RISK CRITERIA | Customer registration | Catalog checkout | Order fulfillment | Order shipment | TOTAL RISK |
|-------------------|-----------------------|------------------|-------------------|----------------|------------|
| Scalability | | 6 | | | 6 |
| Availability | | | | | 0 |
| Performance | | | | 6 | 6 |
| Security | 6 | | | | 6 |
| Data integrity | 9 | 6 | | | 15 |
| TOTAL RISK | 15 | 12 | 0 | 6 | |

Figure 22-3. Filtering the risk assessment to show only high risks

One problem with the full version of the risk assessment is that it only shows a snapshot in time, not whether things are improving or getting worse. In other words, [Figure 22-2](#) does not show the *direction of risk*. You can determine the direction of risk by using continuous measurements through fitness functions, as described in [Chapter 6](#). Objectively analyzing each risk criterion lets you observe trends to spot the direction of each risk criterion.

In [Figure 22-4](#), we add a third dimension to the risk assessment: *direction*. We use a right-side-up triangle to indicate that the risk for that particular criteria and context is getting worse—the tip of the triangle points up, toward a *higher* number. Conversely, we use an upside-down triangle to indicate that the risk is lessening (in other words, the tip is pointing down to a *lower* number). Finally, we use a circle to represent that the risk is not moving—getting neither better nor worse. This can get confusing, so we always recommend including a key when using any sort of symbol to represent direction.

| RISK CRITERIA | Customer registration | Catalog checkout | Order fulfillment | Order shipment | TOTAL RISK |
|----------------|-----------------------|------------------|-------------------|----------------|------------|
| Scalability | (2) | ▲6 | (1) | (2) | 11 |
| Availability | ▼3 | ▼4 | (2) | △1 | 10 |
| Performance | (4) | (2) | ▲3 | (6) | 15 |
| Security | ▼6 | ▼3 | (1) | (1) | 11 |
| Data integrity | ●9 | ▲6 | △1 | △1 | 17 |
| TOTAL RISK | 24 | 21 | 8 | 11 | |

Figure 22-4. Showing direction of risk using triangles

This revised architectural risk assessment, now showing direction, tells a different story than the original. First, we can see that data integrity is getting worse based on continuous measurements (triangle pointing up) for catalog checkout, order fulfillment, and order shipping, which could indicate a database issue. However, security and availability are getting better overall (triangle pointing down) for customer registration and catalog checkout, indicating improvements in those areas.

Next, we describe a process called *risk storming* that teams can use to determine *how* to identify the risk level for particular contexts and criteria.

Risk Storming

No architect can singlehandedly determine the overall risk of a system, for two reasons. First, an architect working alone might miss or overlook a risk area; second, very few architects have full knowledge of *every* part of the system. This is where risk storming can help.

Risk storming is a collaborative exercise to determine architectural risk within a specific dimension (either context or criteria). While most risk-storming efforts involve multiple architects, we strongly recommend including senior developers and tech leads as well. Not only will they provide an implementation perspective on architectural risk, but involving them helps them better understand the architecture.

Risk storming involves three phases: identification, consensus, and mitigation. In the individual phase (phase 1), all participants, working alone, use the risk matrix to assign risk to various areas of the architecture. This individual phase of risk storming is essential so that participants don't influence other participants or direct people's attention away from particular areas of the architecture. In the two collaborative phases, all participants work together to gain consensus on what the risk areas are and discuss them (phase 2), and come up with solutions for mitigating the risk (phase 3).

All three phases utilize a comprehensive or contextual architecture diagram (see [Chapter 23](#)). The architect conducting the risk-storming effort—we'll call this person the *facilitator*—is responsible for sending all participants updated diagrams for the risk-storming session.

[Figure 22-5](#) shows an example architecture we'll use to illustrate the risk-storming process. In this architecture, an Elastic Load Balancer forwards a request to each EC2 instance containing the web servers (Nginx) and application services. The application services make calls to a MySQL database, a Redis cache, and a MongoDB database (for logging). They also make calls to the Push Expansion Servers, which in turn interface with the MySQL database, Redis cache, and MongoDB logging facility. (Don't worry if you don't understand all of these products and buzzwords—this overly vague, generalized architecture is only meant to illustrate how risk storming works.)

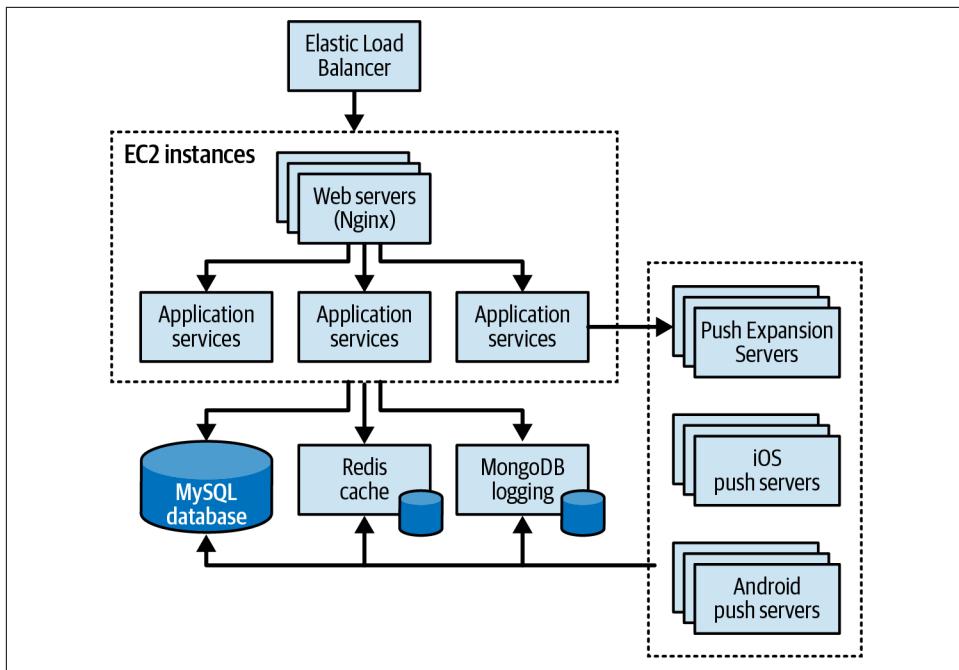


Figure 22-5. Example architecture diagram for risk storming session

We'll begin with phase 1.

Phase 1: Identification

The *identification* phase of risk storming involves each participant individually identifying areas of risk within the architecture. It's critical that, in this phase, each participant should record their unbiased view of the risk involved, without being redirected or swayed by other participants. The identification phase has three steps:

1. The facilitator sends all participants an invitation to the collaborative phases. The invitation contains the architecture diagram (or where to find it), the risk criteria and context to be analyzed, and the date, time, and location (physical or virtual) of the collaborative session, along with any other logistical details.
2. Participants use the risk matrix to analyze the architecture risks individually.
3. Participants classify each risk as low (1–2), medium (3–4), or high (6–9), and write the numbers on small green, yellow, or red sticky notes.

Most risk-storming efforts analyze only one particular criterion or context (such as “Where are our security risks?” or “What areas are at risk within customer registration?”). However, if there are issues with staff availability or timing, the

risk-assessment team may have to analyze multiple dimensions within a specific context (such as performance and scalability). In these cases, participants typically write the specific criterion next to the risk number on the sticky notes. For example, suppose three participants identify risk with a central database. All three participants identify the risk as high (6), but one participant sees this as a risk to availability, while the other two see it as a risk to performance. The participants should discuss these two criteria separately.



Whenever possible, restrict risk-storming efforts to a single criterion or context. This allows participants to focus their attention on that specific dimension and avoids confusion about what the actual risk is.

Phase 2: Consensus

The *consensus* phase of risk storming is highly collaborative, with the goal of gaining consensus among all participants regarding the risk or risks within the architecture. This activity is most effective when the facilitator posts a large printed architecture diagram on the wall (or an electronic version on a large screen). When the participants arrive at the risk-storming session, the facilitator instructs them to begin placing their risk-level sticky notes on the relevant area of the architecture diagram (see [Figure 22-6](#)).

Once all of the sticky notes are in place, the collaborative phase can begin. The goal here is to analyze the risk areas as a team and reach a consensus about their risk level. In the example pictured in [Figure 22-6](#), the team has identified several areas of risk. (The actual criteria aren't important for this example.) We can see that:

- Two participants identified the Elastic Load Balancer as medium risk (3), whereas one participant identified it as high risk (6).
- One participant identified the Push Expansion Servers as high risk (9).
- Three participants identified the MySQL database as medium risk (3).
- One participant identified the Redis cache as high risk (9).
- Three participants identified MongoDB logging as low risk (2).
- No one identified any other areas of the architecture as carrying any risk, so there are no sticky notes on any other areas.

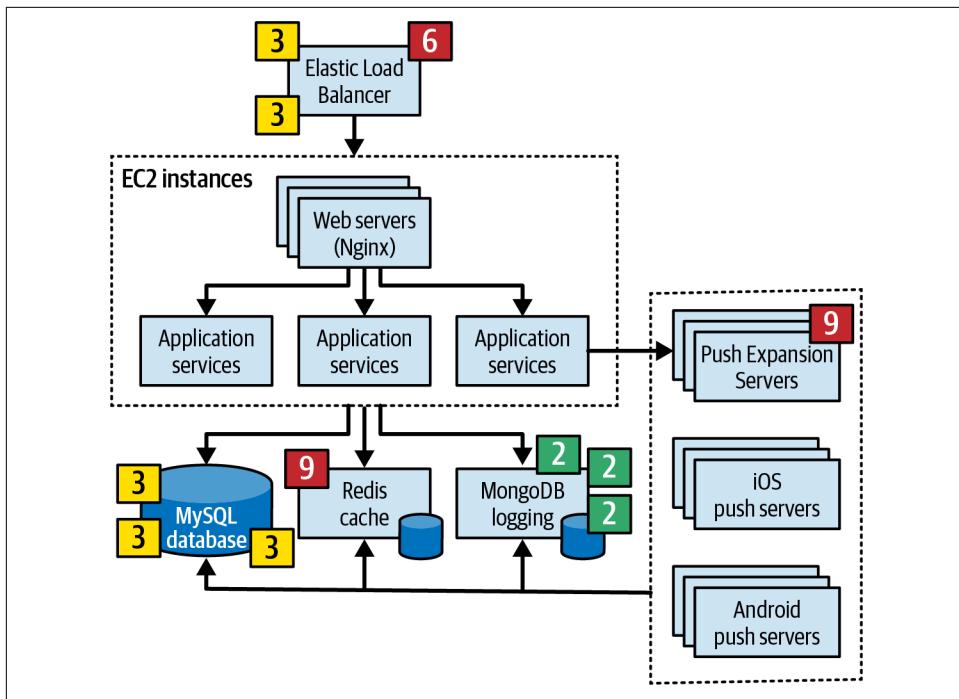


Figure 22-6. Initial identification of risk areas

The MySQL database and MongoDB logging need no further discussion in this session, since all participants agree on their risk levels. However, there's a difference of opinion about the Elastic Load Balancer, and the Push Expansion Servers and Redis cache were only identified as risks by one participant each. Working through these discrepancies is what the collaborative phase is all about.

Two participants (Austen and Logan) identified the Elastic Load Balancer as medium risk (3); one (Addison) identified it as high risk (6). Austen and Logan ask Addison why they identified the risk as high. Addison responds that if the Elastic Load Balancer goes down, the entire system will become inaccessible. While this is true—and brings the *impact* rating up to high—the other two participants convince Addison that there is low risk of this happening due to clustering. Addison agrees, and the group brings the *likelihood* risk level down to a medium (3).

This could have gone a different way, however. If Austen and Logan missed a particular aspect of risk in the Elastic Load Balancer that Addison saw, Addison might have convinced the other two participants to classify this risk's level as high instead of medium. That's why the collaboration phase of risk storming is so important.

One participant identified the Push Expansion Servers as high risk (9), but no other participant identified any risk in this area of the architecture at all. The person who identified the risk explains that they rated the risk as high because they've had bad experiences with Push Expansion Servers continually crashing under high loads similar to this architecture's load. This example shows the value of risk storming—without that participant's involvement, no one would have seen the high risk until well into production.

The Redis cache is an interesting case. One participant, a developer named Devon, identified it as high risk (9), but no one else saw that cache as having any risk. When the other participants ask Devon about their rationale for rating this risk as high, Devon responds, "What's a Redis cache?" Whenever a risk-storming participant identifies a technology as unknown to them, that area automatically gets assigned a high risk level (9).



Always assign unproven or unknown technologies the highest risk rating (9), since the risk matrix cannot be used for this criterion or context.

The example of the Redis cache illustrates why it's important to bring developers into risk-storming sessions. The fact that this participant didn't know a given technology is valuable information for the architect about overall risk. The architect might decide to change the technology or incur training costs to bring the development team up to speed.

This phase continues until all participants agree on the risk areas identified. Once all the sticky notes are consolidated, this phase ends. Its final outcome is shown in [Figure 22-7](#).

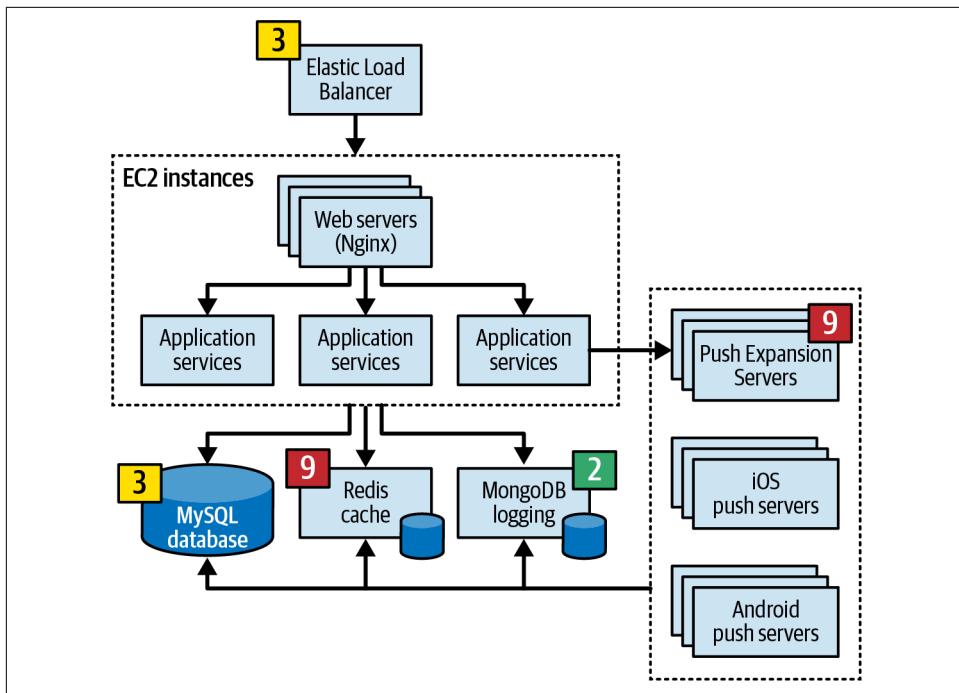


Figure 22-7. Consensus of risk areas

Phase 3: Risk Mitigation

Once all participants agree on the risk levels of the architecture, the *risk mitigation* phase begins. Mitigating risk usually involves changing certain areas of the architecture that otherwise might have been deemed perfect the way they were.

This phase, which is also collaborative, seeks ways to reduce or eliminate the risks identified in the second phase. Depending on the risks identified, the original architecture may need to be completely changed, or the changes could be limited to straightforward architectural refactoring in specific areas, such as adding a queue for backpressure to reduce a throughput bottleneck.

Whatever the changes required, the risk mitigation phase usually incurs additional costs. For that reason, it's important for this phase to involve key business stakeholders with the authority to decide whether the cost of a given mitigation solution outweighs the risk.

For example, suppose that, in our example risk-storming session the team identifies the central database as having medium risk (4) with regard to overall system availability. The participants agree that clustering the database and breaking it into separate physical databases would mitigate that risk. However, that solution would cost

\$50,000. The facilitating architect meets with key business stakeholders, including the owner, to discuss the trade-offs of availability risk and cost. The business owner decides that the price tag is too high and that the cost does not outweigh the risk of availability. The architect then suggests a different approach: instead of expensive clustering, what about splitting the database into two separate domain-based databases? This solution would only cost \$16,000, while still reducing the availability risk. The stakeholders agree to this compromise.

This scenario shows how risk storming shapes not only the overall architecture, but also the negotiations between architects and business stakeholders. In combination with the risk assessments we described at the start of this chapter, risk storming is an excellent vehicle for identifying and tracking risk, improving the architecture, and structuring negotiations between key stakeholders.

User-Story Risk Analysis

Risk storming is useful in many aspects of software development aside from identifying architectural risk. For example, a development team could use risk storming to determine the overall risk associated with user-story completion within a given iteration (and consequently the overall risk assessment of that iteration) during story grooming. Using the same risk matrix, the team can identify user-story risk by identifying the overall impact if the story is not completed within the iteration, and the likelihood that the story will not be completed in the current iteration. Then they can identify high-risk stories, track them carefully, and better prioritize them.

Risk-Storming Use Case

To illustrate the power of risk storming and how it can improve overall architecture, let's consider the example of a support system for a call center where nurses advise patients on various health conditions. The system requirements are:

- A third-party diagnostics engine will serve up questions and guide nurses and patients through their medical issues. This engine can handle about 500 requests a second.
- Patients can either call in using the call center to speak to a nurse or use a self-service website that accesses the same diagnostic engine directly.
- The system must support 250 concurrent nurses nationwide and up to hundreds of thousands of concurrent self-service patients nationwide.
- Nurses can access patients' medical records through a medical records exchange, but patients cannot access their own medical records.

- The system must be **HIPAA** (Health Insurance Portability and Accountability Act) compliant, meaning it's essential that no one but nurses can access patients' medical records. The self-service option cannot guarantee HIPAA compliance.
- The system needs to be able to handle high volume during cold, flu, and COVID outbreaks.
- Calls are routed to nurses based on each nurse's skill profile (such as languages spoken or medical specializations).

After analyzing these requirements, Logan, the architect responsible for this system, creates the high-level architecture diagrammed in [Figure 22-8](#). This architecture has three separate web-based UIs: one for self-service, one for nurses receiving calls, and one for administrative staff to add and maintain nurse profiles and configuration settings. The call-center portion of the system consists of a **Call Acceptor** service, which receives calls, and the **Call Router** service, which routes the caller to the next available nurse based on the nurse's skill profile. The **Call Router** service accesses the central database to get nurse profile information. Central to this architecture is a diagnostics-system API Gateway, which performs security checks and directs requests to the appropriate backend service.

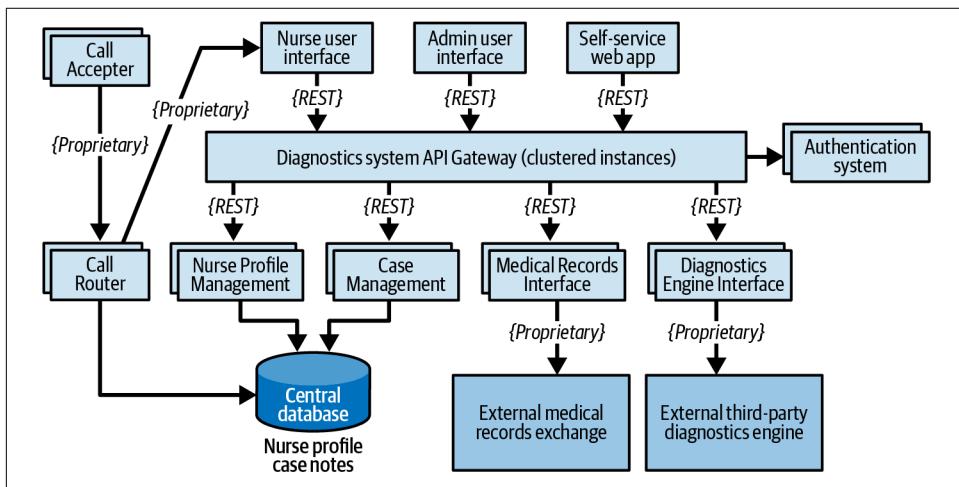


Figure 22-8. High-level architecture for a nursing-hotline diagnostics system

The four main services in this system are the **Case Management** service, the **Nurse Profile Management** service, the **Medical Records Interface** service to the medical records exchange, and the external third-party **Diagnostics Engine Interface** service. All communications use REST, except for proprietary protocols to the external systems and call-center services. The areas the architecture must support are, in summary, availability, elasticity, and security.

After many reviews, Logan believes the architecture is ready for implementation. However, being a responsible and effective architect, Logan decides to hold a risk-storming exercise.

Availability

As facilitator, Logan decides to focus the first risk-storming exercise on availability, which is critical to the system's success. After the identification and collaboration phases, the participants come up with the following risk areas (illustrated in Figure 22-9):

- Central database availability: high risk (6) due to high impact (3) and medium likelihood (2) of the database being unavailable when needed.
- Diagnostics Engine availability: high risk (9) due to high impact (3) and unknown likelihood (3) of unavailability.
- Medical Records Interface availability: low risk (2); this component is not required to determine a particular medical outcome.
- The team has deemed no other parts of the system to be availability risks, since the architecture includes multiple instances of each service and clusters the API Gateway.

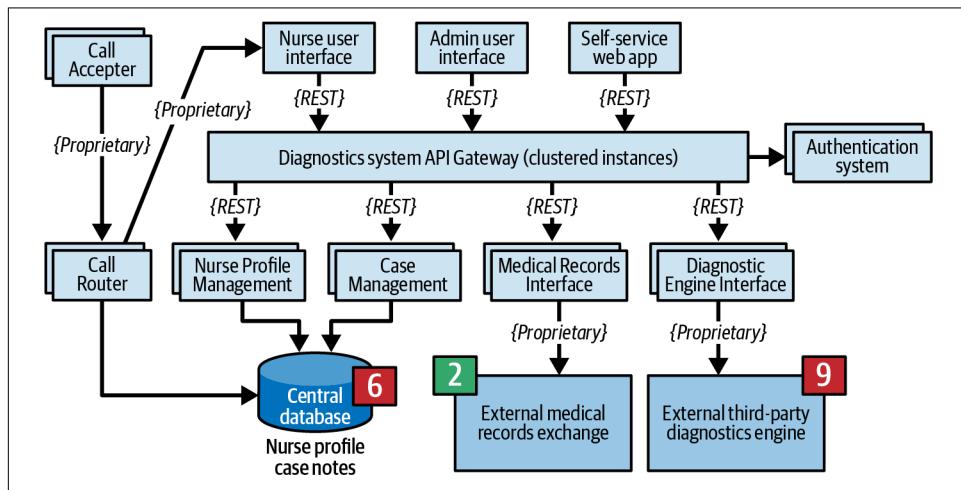


Figure 22-9. Availability risk areas, as identified by the risk-storming team

All participants agree that if the database went down, nurses could write down case notes manually, but the call router could not function. To mitigate this risk, they collectively decide to break apart the single physical database into two separate databases: one clustered database containing nurse-profile information, and

one single-instance database for case notes. Not only does this architecture change address the database availability concerns, it also helps secure the case notes.

It's much harder to mitigate availability risk in external systems (in this case, the **Diagnostics Engine** and **Medical Records Interface**) because they are controlled by a third party. The team decides to research whether these systems have published service-level agreements (SLAs) or service-level objectives (SLOs). An SLA is usually a legally binding contractual agreement; an SLO is usually not legally binding. They find SLAs for both systems. The **Diagnostics Engine** SLA guarantees 99.99% availability (that's 52.60 minutes of downtime per year), and the **Medical Records Interface** guarantees at 99.90% availability (that's 8.77 hours of downtime per year). Based on this analysis, this information was enough for the risk assessment team to remove the identified risk.

After this risk-storming session, the team changes the architecture, as illustrated in [Figure 22-10](#), creating two databases and adding the SLAs to the architecture diagram.

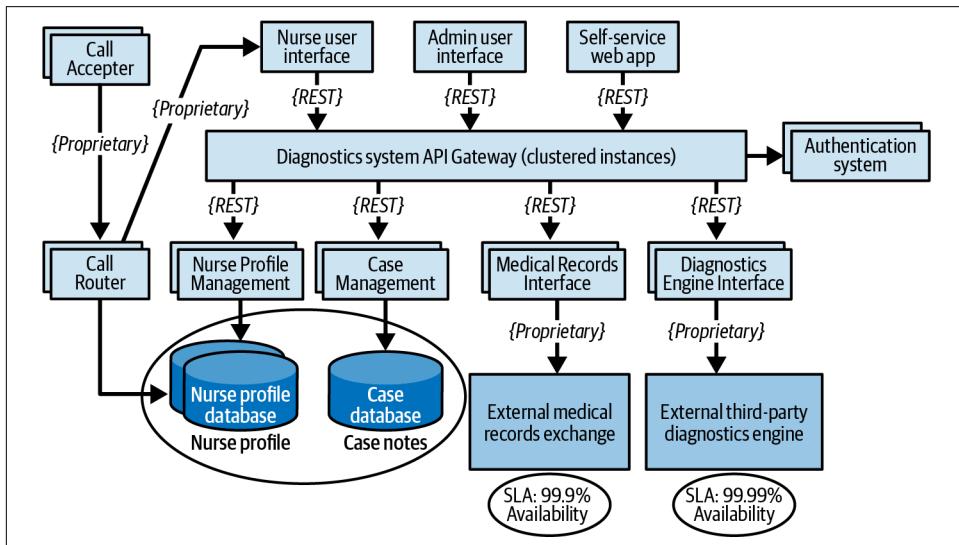


Figure 22-10. Availability risk areas can be mitigated by using separate databases

Elasticity

The second risk-storming exercise focuses on elasticity—spikes in user load (otherwise known as variable scalability). Although there are only 250 nurses (which means no more than 250 nurses will be accessing the **Diagnostics Engine**), the self-service portion of the system can access the **Diagnostics Engine** as well, which significantly increases the number of requests to the **Diagnostics Engine** interface. The risk-storming participants are concerned about flu season and COVID outbreaks, which will significantly increase the anticipated load on the system.

The participants unanimously identify the **Diagnostics Engine** interface as high risk (9). Since it can handle only 500 requests per second, they correctly calculate a high risk that it won't be able to keep up with the anticipated throughput, particularly with REST as its interface protocol.

The team decided that one way to mitigate this risk is to use asynchronous queues (messaging) for communication between the API Gateway and the **Diagnostics Engine** interface. This will provide a backpressure point if calls to the **Diagnostics Engine** get backed up. While this is a good practice, it still doesn't quite mitigate all the risk: nurses and self-service patients will still be waiting too long for responses from the **Diagnostics Engine**, and their requests will likely time out.

The participants decide to use a pattern known as the **Ambulance** pattern to separate these requests by using two message channels instead of just one. This would let the system prioritize nurses' requests over self-service requests. This would help mitigate the risk, but still doesn't address wait times. After more discussion, the group decides to reduce outbreak-related calls to the **Diagnostics Engine** by caching those particular diagnostics questions so that they never reach the **Diagnostics Engine** interface.

In addition to creating two messaging channels (one for the nurses and one for self-service patients), the team creates a new service called the **Diagnostics Outbreak Cache Server** that handles all requests related to a particular outbreak or flu-related question ([Figure 22-11](#)). This new architecture reduces the number of calls to the diagnostics engine, allowing for more concurrent requests related to other symptoms. Without the risk-storming effort, this risk might not have been identified until flu season.

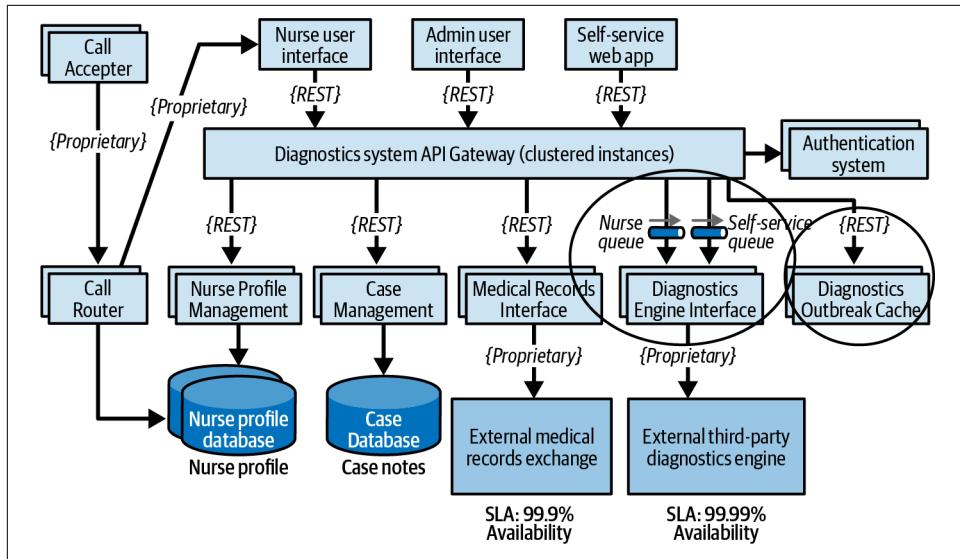


Figure 22-11. Architecture modifications to address elasticity risk

Security

Encouraged by these successes, the architect decides to facilitate a final risk-storming session focused on another crucial characteristic for this system—security. Due to HIPAA regulatory requirements, the Medical Records Interface must allow only nurses to access patients' medical records. The architect believes that the authentication and authorization checks in the API Gateway neutralize this risk, but is curious whether the participants will find any other security risks.

The participants all identify the diagnostics system's API Gateway as a high security risk (6). They cite the high impact if administrative staff or self-service patients access medical records (3), but rate the likelihood as medium (2). While the security checks for each API call help, all calls (self-service, admin, and nurses) are still going through the same API Gateway. They eventually convince the facilitator, who had initially rated this risk as low (2), that the risk is in fact high and needs to be mitigated.

Everyone agrees that having separate API Gateways for each type of user (admin staff, self-service users, and nurses) would prevent non-nurse calls from ever reaching the Medical Records Interface. The architect's final version of the architecture is shown in [Figure 22-12](#).

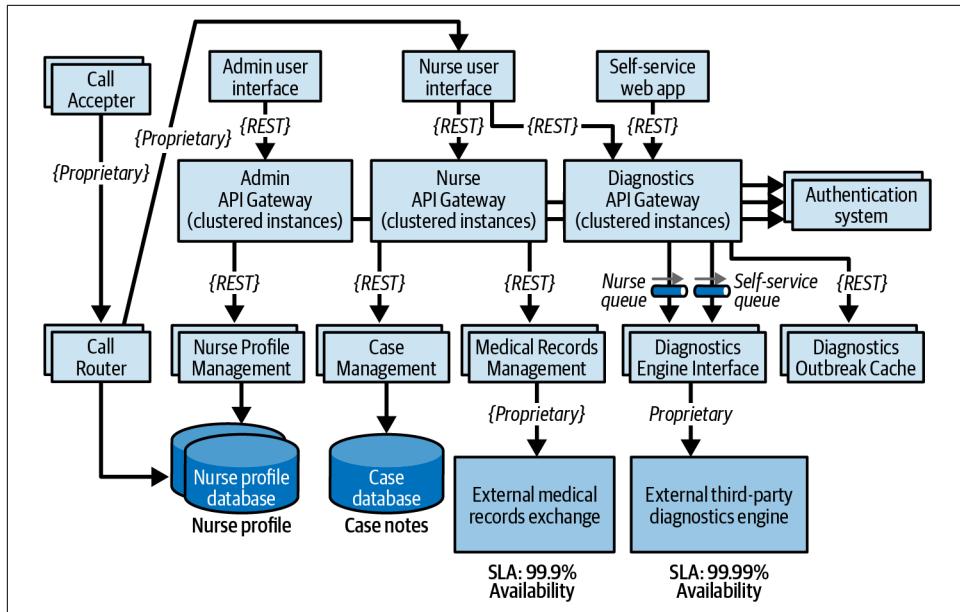


Figure 22-12. Architecture modifications to address security risk

This example illustrates the power of risk storming. Architects, developers, and key stakeholders collaborate, consider the architectural characteristics most vital to the system's success, and identify risk areas that would otherwise have gone unnoticed.

The original architecture (Figure 22-8) is significantly changed after risk storming (Figure 22-12) in ways that address concerns about availability, elasticity, and security and make this architecture more effective and more likely to succeed.

Summary

Risk storming is not a one-time process. It continues throughout a system's lifecycle, as teams work to identify and mitigate risk areas before they appear in production. How often an organization should hold risk-storming sessions depends on many factors, including the frequency of change, any architecture-refactoring efforts, and the architecture's incremental development. It's typical to do some risk storming on particular dimensions after adding a major feature or at the end of every iteration, to ensure that the architecture is correct and will address the business's needs.

CHAPTER 23

Diagramming Architecture

Newly minted software architects are often surprised at how varied the job is, outside the technical knowledge and experience that brought them into the role to begin with. In particular, effective communication is critical to an architect's success. No matter how brilliant your technical ideas, if you can't convince managers to fund those ideas and developers to build them, your brilliance will never manifest.

Diagramming is a critical communication skills for architects. While entire books exist about each topic, we'll hit some particular highlights for each.

To visually describe an architecture, the architect often must show different views: for example, they might start with an overview of the entire topology, then drill down into the design details of specific parts of the architecture. However, showing a portion without indicating its place within the overall architecture will confuse viewers.

Representational consistency is the practice of always showing the relationships between parts of an architecture before changing views, and it's equally important in diagrams and in presentations. For example, if you wanted to describe the details of how the plug-ins relate to one another in the Silicon Sandwiches solution, you'd start with an architecture diagram showing the system's entire topology, then the relationship between it and the plug-in structure, before going into the plug-in structure itself. [Figure 23-1](#) provides an example.

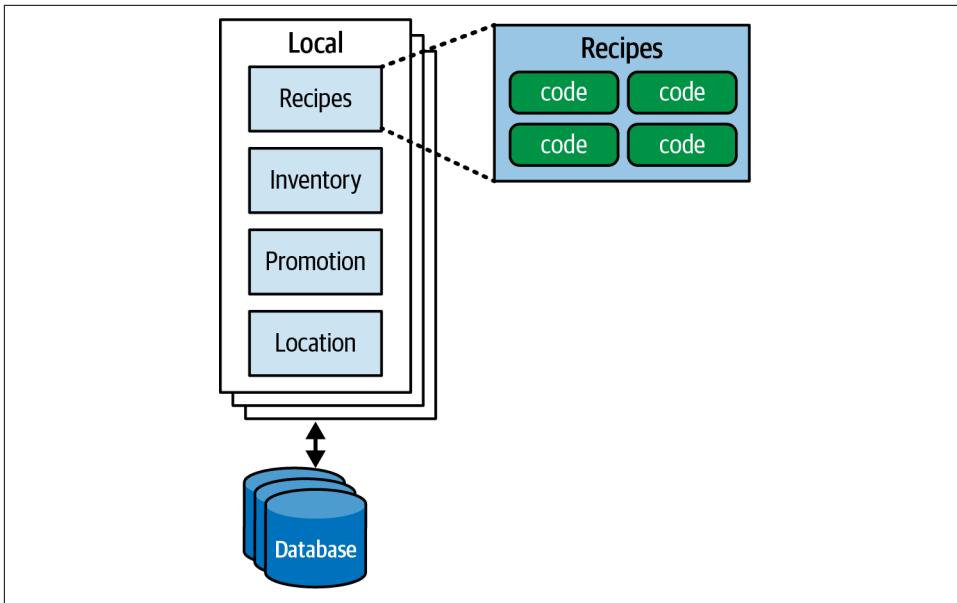


Figure 23-1. Using representational consistency to indicate context in a larger diagram

Use representational consistency carefully to ensure that viewers understand the scope of the items being presented and eliminate a common source of confusion.

Diagramming

The topology of an architecture is always of interest to architects and developers because it captures how the structure fits together, and allows teams to form a valuable shared understanding. We encourage all architects to hone their diagramming skills to razor sharpness.

Tools

The current generation of diagramming tools for architects is extremely powerful, and every architect should learn their tool of choice deeply. But don't neglect low-fidelity artifacts, especially early in the design process. Building very ephemeral design artifacts early prevents architects from becoming overly attached to what they have created, an antipattern we call *Irrational Artifact Attachment*.

Irrational Artifact Attachment

The Irrational Artifact Attachment antipattern describes the proportional relationship between a person's irrational attachment to some artifact and how long it took the person to produce that artifact. If an architect spends four hours creating a beautiful diagram in some tool like Visio, they'll be even more irrationally attached to that artifact than if they'd invested only two hours.

One benefit of the Agile approach to software development is that it involves creating "just-in-time" artifacts with as little ceremony and ritual as possible. This is one reason so many Agilists love index cards and sticky notes: using low-tech tools lets people throw away what's not right, freeing them to experiment and allow the true nature of the artifact to emerge through revision, collaboration, and discussion.

The classic ephemeral artifact is a cell-phone photo of a whiteboard diagram (along with the inevitable "Do Not Erase!" imperative). These days, many architects have abandoned the whiteboard in favor of a tablet attached to an overhead projector. This offers several advantages. First, the tablet has an unlimited canvas and can fit as many drawings as the team might need. Second, it allows the team to copy/paste "what if" scenarios, which would obscure the original if done on a whiteboard. Third, images created on a tablet are already digitized and don't have the inevitable glare of whiteboard photos. Fourth, using electronic images makes remote work much easier and more collaborative.

Eventually, you'll need to create nice diagrams in a fancy tool, but before you invest that time, make sure the team has iterated on the design sufficiently. Whatever platform you're using, you'll find powerful diagramming tools. We happily used **OmniGraffle** for the original versions of all of the diagrams in this book, which were then refined by O'Reilly illustrators. While we don't necessarily advocate one tool over another, we recommend looking for the following features, as a baseline:

Layers

Drawing tools often support layers, which architects should learn to use well. Layers allow the user to link groups of items together logically and show or hide them as needed. For example, you might build a comprehensive diagram for a presentation, but hide the overwhelming details when not directly discussing them. Layers also provide a good way to present pictures that build incrementally.

Stencils/templates

Stencil tools allow users to amass a library of common visual components, including composites of other basic shapes. For example, throughout this book, you have seen standard icons for things like individual microservices; these exist as single items in the authors' stencil tool. Building a set of stencils for patterns and artifacts that are common within an organization creates consistency across architecture diagrams and makes it faster to build new diagrams.

Magnets

Many drawing tools offer assistance for drawing lines between shapes. *Magnets* represent the places on those shapes where lines snap to connect automatically, providing alignment and other visual niceties. Some tools allow users to add more magnets or create their own.

Use Layers Semantically, Not Decoratively

We prefer drawing tools that support layers, but we suggest using them *semantically*—that is, in ways that contribute meaning to the overall image. For example, the base layer of each diagram should represent the topology of the architecture: its containers, databases, dependencies, brokers, and other core elements. This layer should focus on architecture rather than implementation; for instance, specify “synchronous communication” rather than naming a specific protocol. The next layer should contain implementation details: what type of database, which communication protocols, and so on.

Using layers in this manner makes diagrams extensible: it’s possible to add other contextualized layers to present domain-driven design boundaries, transactional scope, or any other meta-information the architect wants to contrast against the topology of the architecture.

In addition to these specific helpful features, the tool should, of course, support basic functions such as lines, colors, shapes, and other visual artifacts, and should be able to export to a wide variety of formats.

Diagramming Standards: UML, C4, and ArchiMate

The software industry uses several formal standards for technical diagrams. We'll look at three of the most popular here.

UML

Grady Booch, Ivar Jacobson, and Jim Rumbaugh created the Unified Modeling Language (UML) standard in the 1980s to unify their competing design philosophies. It was supposed to be the best of all worlds, but, like many things designed by committee, failed to create much impact outside organizations that mandated its use. Architects and developers still use UML class and sequence diagrams to communicate structure and workflow, but most other UML diagram types have fallen into disuse.

C4

C4 is a diagramming technique developed by Simon Brown between 2006 and 2011 to address UML's deficiencies and modernize its approach. The “four Cs” in C4 are:

Context

The entire context of the system, including the roles of users and external dependencies.

Container

The physical (and often logical) deployment boundaries and containers within the architecture. This view forms a good meeting point for operations teams and architects.

Component

The component view of the system; this most neatly aligns with an architect's view of the system.

Class

C4 uses the same style of class diagrams as UML, which are effective, so there is no need to replace them.

C4 offers a good alternative for any company seeking to standardize on a diagramming technique. Its creators have been active for years and have gathered a huge following. Critically, it has kept up with changes in the software development ecosystem, expanding to take advantage of new capabilities. Many diagramming tools contain templates for C4 diagrams, and the [C4 ecosystem](#) contains tools and frameworks to assist architects. C4 defines standards for components, lines, containers, databases, and other common artifacts.

A C4 diagram of the Silicon Sandwich modular monolith data design appears in [Figure 23-2](#).

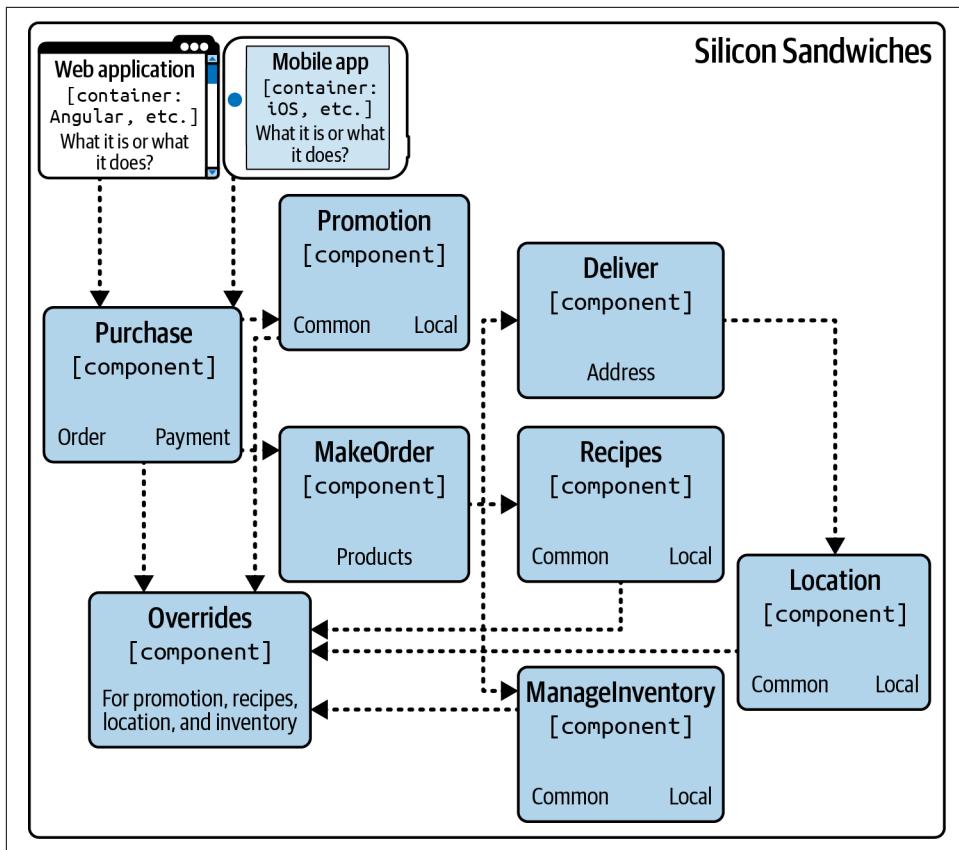


Figure 23-2. Silicon Sandwiches component diagram in C4

ArchiMate

ArchiMate (a portmanteau of *architecture* and *animate*) is an open source enterprise-architecture modeling language from The Open Group for describing, analyzing, and visualizing architectures within and across business domains. ArchiMate offers a lighter-weight modeling language for enterprise ecosystems. The goal of this technical standard is to be “as small as possible,” not to cover every edge case. It is a popular choice among architects.

Diagram Guidelines

Every architect should build their own diagramming style, whether they use their own modeling language or one of the formal ones. It’s OK to borrow from representations you think are particularly effective. This section offers some general guidelines for creating technical diagrams.

Titles

Make sure all the elements of the diagram have titles, unless they're very well-known to the audience. Use rotation and other effects to make titles "stick" to the right thing and to make efficient use of space.

Lines

Lines should be thick enough to be clearly visible. If the lines indicate information flow, use arrows to indicate directional or two-way traffic. Different types of arrowheads might suggest different semantics—just be consistent.

One of the few general standards in architecture diagrams is that solid lines almost always indicate synchronous communication, and dotted lines indicate asynchronous communication.

Shapes

While the formal modeling languages we've described all have standard shapes, there's no pervasive set of standard shapes used across the software development world. Most architects make their own set of standard shapes; sometimes these are adopted across an organization to create a standard language.

We tend to use three-dimensional boxes to indicate deployable artifacts, rectangles to indicate containers, and cylinders to represent databases, but we don't have any particular key beyond that.

Labels

It's important to label every item in a diagram, especially if there is any chance of ambiguity.

Color

Architects often don't use color enough. For many years, books were out of necessity printed in black and white, so architects and developers became accustomed to monochromatic drawings. While we still favor monochrome, we use color when it helps distinguish one artifact from another. For example, when discussing microservices architecture quantum in [Chapter 19](#), we use shades of gray in [Figure 19-6](#) (reproduced here as [Figure 23-3](#)) to indicate grouping.

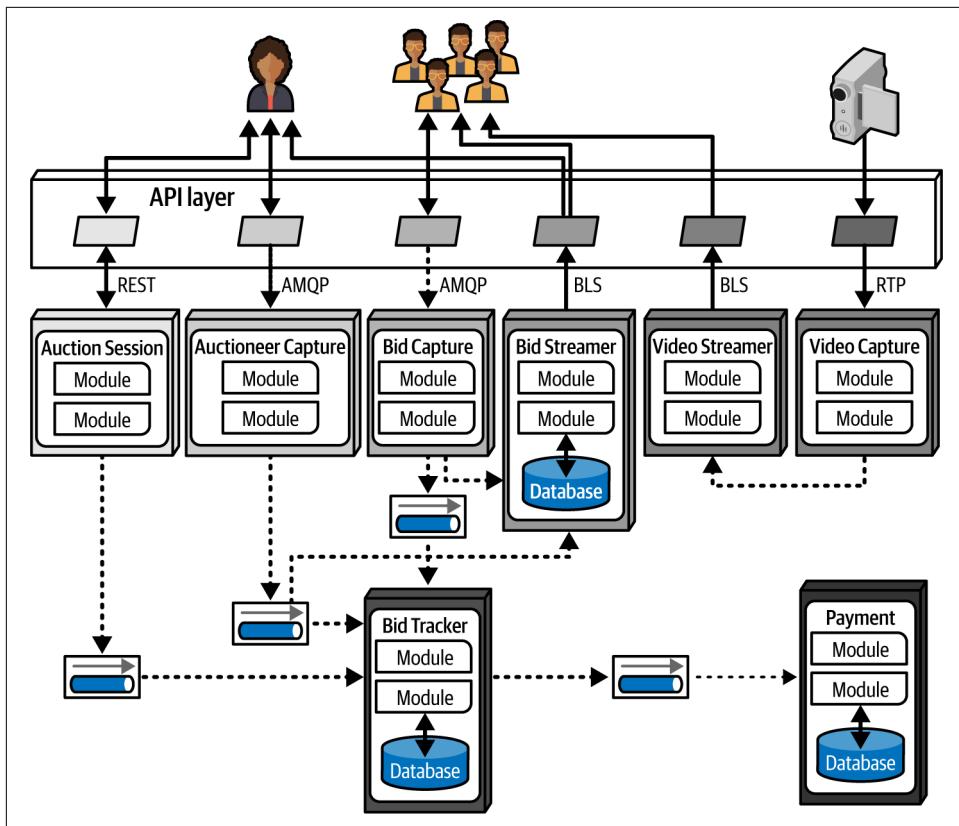


Figure 23-3. Reproduction of a microservices communication example from Chapter 19, showing different services in shades of gray

Be careful about using color to indicate critical differences, however; people who are colorblind or have other visual disabilities will not be able to see the distinction. We suggest using unique iconography in addition to color, so if someone can't distinguish the color, they can still understand the meaning—just as any street-crossing lights use green for go and red for stop, but also show unique figures for each color.

Keys

If a diagram's shapes are ambiguous for any reason, include a key that clearly indicates what each shape represents. An easily misinterpreted diagram is worse than no diagram at all.

Summary

Diagramming standards are a useful way for organizations to provide consistent communication. Architects, however, frequently break the rules, especially when the standard doesn't offer a good way to represent the design. We encourage organizations to establish standards but allow reasonable exceptions.

In the past, when heavyweight computer-assisted software engineering (CASE) tools were the norm, architects had to build elaborate models to represent simple things. They were often forced to include many useless details that were merely noise in that specific context. We favor lightweight diagramming tools and quick-and-dirty artifacts, especially early in the design process. Just don't become so enamored with your creations that you lose your objectivity.

Making Teams Effective

In addition to creating technical architectures and making architecture decisions, software architects are also responsible for leading the development team and guiding it through implementing the architecture. Architects who do this well create effective development teams that work together closely to solve problems and create winning solutions. While this may sound obvious, we've seen too many architects ignore their development teams and create an architecture alone, in a siloed environment. When this architecture is handed off to the development team, the developers often struggle to implement it correctly.

Making teams productive is one of the ways successful software architects differentiate themselves. In this chapter, we introduce some basic techniques for improving development teams' effectiveness.

Collaboration

All too often, the software industry treats architecture and development as entirely separate activities. Consider [Figure 24-1](#), which compares the traditional responsibilities of architects to those of developers. Architects are responsible for activities like analyzing business requirements to extract and define architectural characteristics, selecting architecture patterns and styles to solve the problem domain, and creating logical components. The development team uses the artifacts the architect creates during these activities to create class diagrams for components, build UI screens, and write and test source code.

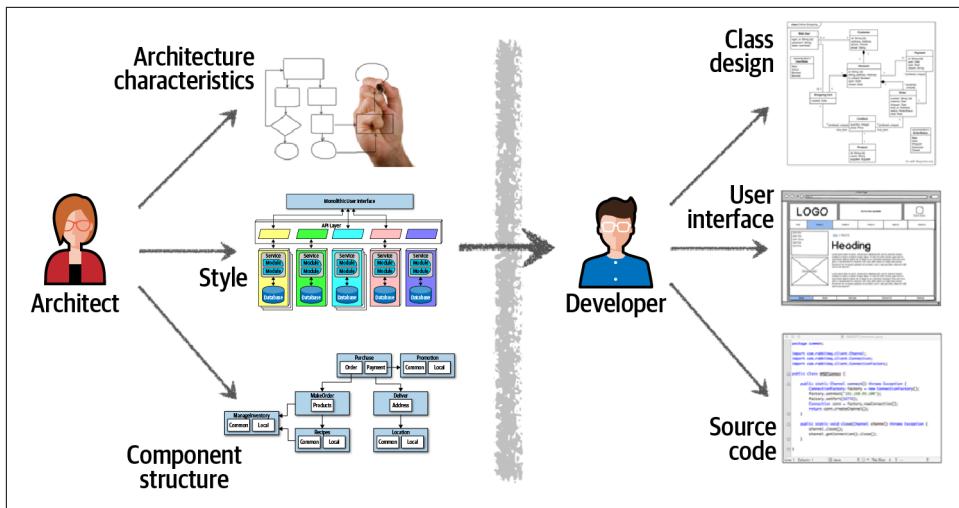


Figure 24-1. Traditional roles of architects versus developers

The image in [Figure 24-1](#) illustrates why this traditional approach to architecture rarely works. Look at the unidirectional arrow that passes through the virtual and physical barriers separating the architect from the developer: that's what causes all of the problems. Architects' decisions don't always make it to the development team, and when development teams change the architecture, it rarely gets back to the architect. In this model, because the architect is so disconnected from the development team, the architecture rarely accomplishes its goals.

The key to making architecture work is breaking down the barriers, both physical and virtual, between architects and developers, instead forming a strong bidirectional collaborative relationship between them. The architect and development team must be on the same virtual team, as in the collaborative model depicted in [Figure 24-2](#). Not only does this model facilitate strong bidirectional communication and collaboration, it also allows the architect to mentor and coach developers.

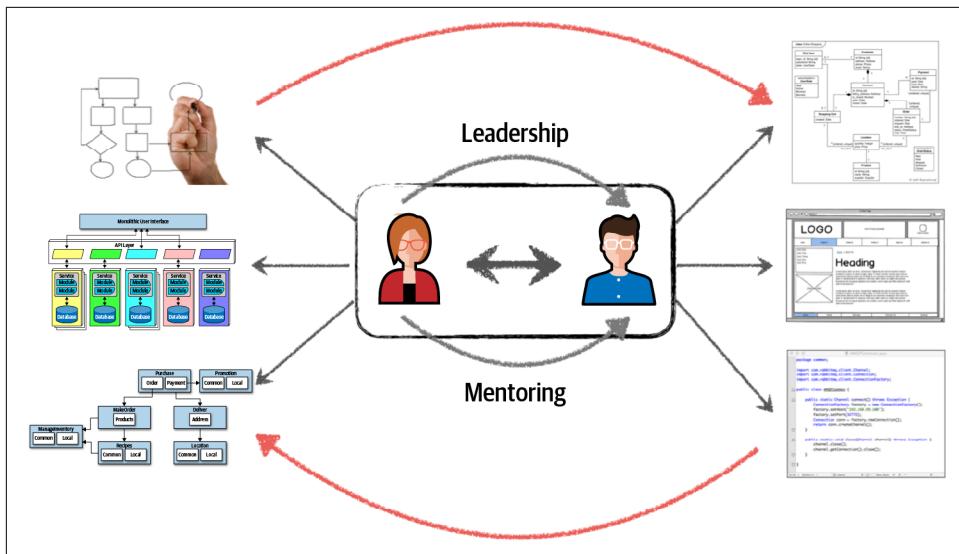


Figure 24-2. Making architecture work through collaboration

Unlike with the static, rigid old-school waterfall approaches, today's software architectures change and evolve with nearly every iteration or phase of a product effort. Tight collaboration between the architect and the development team is essential for success.

In the rest of this chapter and in [Chapter 25](#), we'll show you techniques for forming the healthy, bidirectional, collaborative relationships that not only make development teams more effective, but create more robust and successful architectures.

Constraints and Boundaries

It's been our experience that a software architect can significantly influence the success or failure of a development team. Teams that feel left out of the loop or estranged from their architects often lack knowledge about various constraints on the system; without the right level of guidance, they struggle to implement the architecture correctly.

One of the roles of a software architect is to create and communicate the constraints within which developers must implement the architecture. Think of these constraints as forming a “room” in which the development team works to implement the architecture. As [Figure 24-3](#) demonstrates, having boundaries that are too tight or too loose directly hampers teams' ability to successfully implement the architecture.

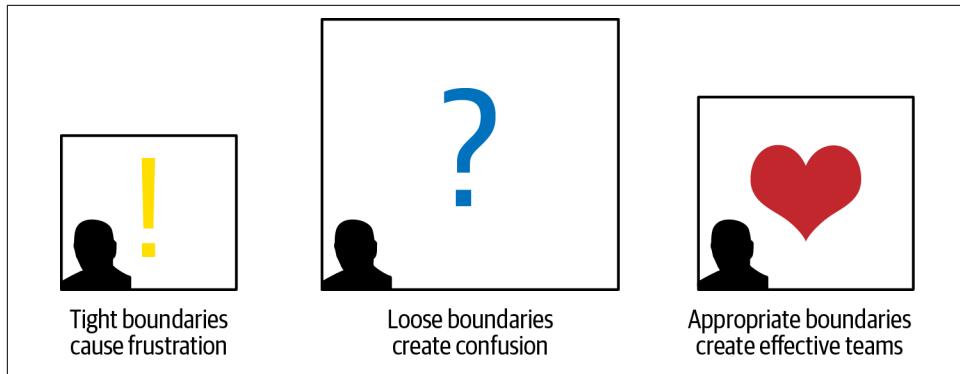


Figure 24-3. The boundaries a software architect creates affect the team's ability to implement the architecture

Too many constraints make the room too small for the development team, preventing them from accessing many of the tools, libraries, and practices they need to implement the system. This causes frustration, and usually results in developers leaving the project for happier and healthier environments.

The converse can also happen. Constraints that are too loose (or having no constraints at all) make the room too big. In this case there are too many choices available, forcing the development team to essentially take on the role of the architect to make all of the important architectural decisions. Lacking proper guidance, the developers find themselves performing too many proofs of concept, struggling over design decisions, and becoming unproductive, confused, and frustrated.

Effective software architects strive to provide the right level of guidance and appropriate constraints so that the team has everything it needs. The rest of this chapter is devoted to showing how to create these appropriate boundaries.

Architect Personalities

We're going to generalize wildly for the sake of conceptual clarity here, and tell you that architects' personalities come in three basic types: the *control-freak architect*, the *armchair architect*, and the *effective architect*. Control-freak architects tend to produce tight boundaries, armchair architects tend to produce loose boundaries, and effective architects produce appropriate boundaries. The following sections describe the details of each architecture personality.

The Control-Freak Architect

The control-freak architect tries to control every detail of the software development process. Every decision they make is usually too fine-grained and too low-level, resulting in tight boundaries and too many constraints on the development team.

For example, a control-freak architect might restrict the development team from downloading useful or even necessary open source or third-party libraries, or place tight restrictions on naming conventions, class designs, method lengths, and so on. They might go so far as to write pseudocode for the development teams to implement, essentially stealing the art of programming away from the developers. Developers find this frustrating and often lose respect for the architect.

Unfortunately, it's very easy to become a control-freak architect, particularly when you're transitioning into architecture from a software developer role. Architects' role is to create the building blocks of the application (the logical components) and determine how they all interact. The developers' corresponding role is to determine how best to implement those logical components, using class diagrams and design patterns. New architects, being used to creating the class diagrams and selecting the design patterns themselves as developers, often find this temptation difficult to resist.

For example, suppose an architect creates a logical component that manages the reference data within the system: things like the static name-value pair data used on the website, product codes, and warehouse codes. The architect's role is to identify the logical component (in this case, a `Reference Manager`), determine its core set of operations (for example, `GetData`, `SetData`, `ReloadCache`, and `NotifyOnUpdate`), and identify which other components need to interact with the `ReferenceManager`. A control-freak architect might think that the best way to implement this component is through a parallel loader pattern, leveraging an internal cache with a particular data structure. This might be an effective design, but it's not the only design, and, more importantly, it's not the architect's job to come up with an internal design for the `Reference Manager`—that's the developer's job.

As we'll talk about in this chapter, sometimes architects need to play the role of control freak, depending on the complexity of the project and the team's skill level. However, most of the time, a control-freak architect disrupts the development team, doesn't provide the right level of guidance, gets in the way, and is generally ineffective as a leader.

The Armchair Architect

The armchair architect is an architect who hasn't coded in a very long time (if ever) and doesn't account for implementation details when creating an architecture. They are typically disconnected from the development team and are rarely around, simply moving on to the next project after completing the initial architecture diagrams.

Some armchair architects are simply in way over their heads: they just don't know the technology or business domain well enough to provide leadership or guidance. Think about it: what do developers do? They write source code, of course. Writing source code is really hard to fake; either you can write source code or you can't. What does an architect do? No one knows! Draw lots of lines and boxes? It's all too easy to fake it as an architect.

For example, suppose an armchair architect designing a stock-trading system is in way over their head. Their architecture diagram might only contain two boxes: one representing the trading system, and one representing the trade compliance engine it communicates with. There's nothing *wrong* with this architecture—it's just too high-level to be of any use to anyone.

Armchair architects create loose boundaries around their development teams, so those teams end up doing the work the architect is supposed to be doing. Their velocity and productivity suffer as a result, and everyone gets confused about how the system should work.

It's as easy to become an armchair architect as it is to become a control freak. When an architect finds that they don't have time for the development teams that are implementing the architecture (or simply choosing not to spend time with them), that's an indicator that they might be falling into the armchair-architect personality. Development teams need an architect's support and guidance, and they need the architect to be available to answer questions. Other indicators of an armchair architect are the following:

- Not fully understanding the business domain, business problem, or technology being used
- Not enough hands-on experience developing software
- Not considering the implications associated with a certain implementation of the architecture solution (such as complexity, maintenance, and testing)

Few architects *intend* to become armchair architects; it just "happens" when they get spread too thin between projects or teams and lose touch with the technology or the business domain. To avoid this, we recommend getting more involved in the project's technologies and building a stronger understanding of the business problem and domain.

The Effective Architect

An *effective* software architect creates appropriate constraints and boundaries, ensures that team members are working well together, and provides them with the right level of guidance. The effective architect also makes sure that the team has the

correct tools and technologies in place and removes any other roadblocks between the development team and its goals.

While this sounds obvious and easy, it's not. There is an art to becoming an effective *leader* as well as an effective software architect. It requires collaborating closely with the development team and gaining their respect. In the following sections, we'll show you some techniques for determining how involved an architect should be with a development team.

How Much Involvement?

Becoming an effective architect is about knowing how much to be involved with a given development team—and when to stay out of their way. This concept, known as *Elastic Leadership*, is widely evangelized by author and consultant Roy Osherove. We're going to deviate a bit from Osherove's work in this area to focus on factors that are specific to software-architecture leadership.

Knowing how much to be involved with a development team can be challenging, and so is determining how many teams or projects you can manage at once. Here are five key factors to consider:

Team familiarity

How well do the team members know each other? Have they worked together before? Generally, the better team members know each other, the more they can self-organize and the less they need the architect involved. Conversely, the newer the team members are, the more they'll need the architect to facilitate collaboration and reduce cliques.

Team size

We consider more than a team with 12 developers to be a big team, and one with 5 or fewer to be a small team. The larger the team, the more the architect is needed. We discuss this topic in more detail in “[Team Warning Signs](#)” on page [443](#).

Overall experience

What is the team's mix of senior and junior (inexperienced) developers? How well do its members know the technology and the business domain? (If the business domain is particularly complex, consider assessing team members' overall level of technological experience separately from their business-domain experience.) Teams with lots of junior developers require more involvement and mentoring from the architect and more mentoring. In teams with more senior developers, the architect can become more of a facilitator than a mentor.

Project complexity

Highly complex projects require the architect to be more available to assist with issues, while relatively simple, straightforward projects require less involvement.

Project duration

Is the project short (say, two months), long (two years), or average duration (around six months)? The architect involvement needed grows with the length of the project.

While most of these factors might seem obvious, project duration is sometimes confusing. As we indicated, the shorter the project's duration, the less involvement is needed; the longer the project, the more involvement is needed. Does that seem counterintuitive? Consider a quick two-month project. Two months is not a lot of time to qualify requirements, experiment, develop code, test every scenario, and release into production. In this case, the architect should act more like an armchair architect; the development team already has a keen sense of urgency, and a control-freak architect would just get in the way and delay the project. Now think about a two-year project: the developers are more relaxed, not feeling a sense of urgency. They're likely to be planning vacations and taking long lunches. Thus, for longer duration projects the architect is needed to ensure that the project moves along on schedule and that the team accomplishes the most complex tasks first.

To illustrate how to use these factors to determine an appropriate level of involvement, assume a fixed scale of 20 points for each factor, as shown in [Figure 24-4](#). The side of the scale with negative values indicates less involvement, ending in the extreme of the armchair architect. The positive values indicate more involvement, ending in the extreme of the control-freak architect.

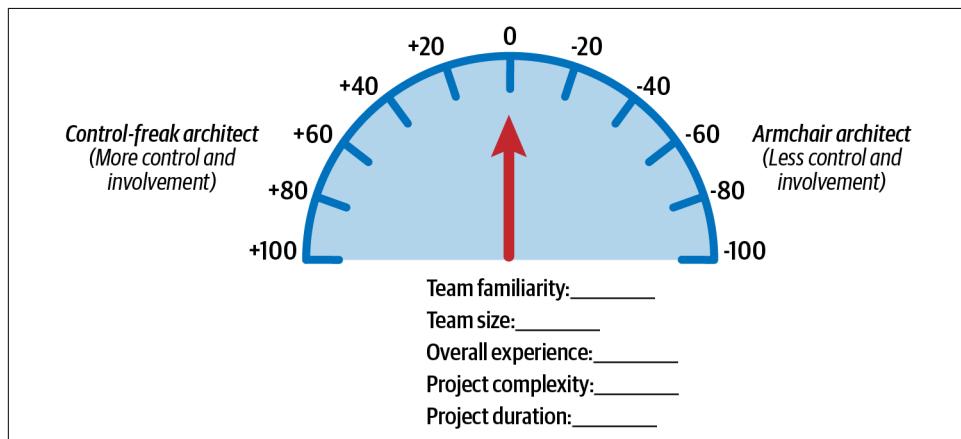


Figure 24-4. Scale for measuring architects' amount of involvement with development teams

This scale is not exact, of course, but it does help in determining the amount of involvement the architect should expect to have with a development team. For example, consider the project scenario, Scenario 1, shown in [Table 24-1](#) and [Figure 24-5](#). The values for each factor in the table move the “needle” toward either extreme: +20 for a factor that indicates more involvement, -20 for one that indicates less involvement. The factor scores rated here for Scenario 1 total -60, indicating that the architect should limit their involvement in daily interactions, facilitating but staying out of the team’s way. They will be needed to answer questions and make sure the team is on track, but for the most part the architect should be largely hands-off and let the experienced team do what they do best—develop software quickly.

Table 24-1. Scenario 1 example for amount of involvement

| Factor | Value | Rating | Personality |
|--------------------|-------------------|--------|--------------------|
| Team familiarity | New team members | +20 | Control freak |
| Team size | Small (4 members) | -20 | Armchair architect |
| Overall experience | All experienced | -20 | Armchair architect |
| Project complexity | Relatively simple | -20 | Armchair architect |
| Project duration | 2 months | -20 | Armchair architect |
| Accumulated score | | -60 | Armchair architect |

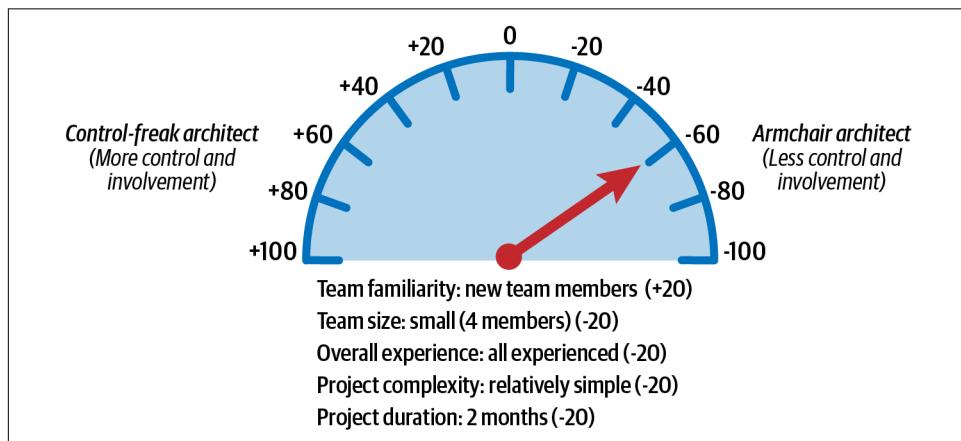


Figure 24-5. Amount of involvement for Scenario 1

Now consider Scenario 2, described in [Table 24-2](#) and illustrated in [Figure 24-6](#), where the team members know each other well, but the team is large (12 team members) and consists mostly of junior developers. The project is relatively complex, with a duration of six months. In this case, the accumulated score comes out to +20, indicating that the effective architect should take on a mentoring and coaching role and be fairly involved in day-to-day activities, but not so much as to disrupt the team.

Table 24-2. Scenario 2 example for amount of involvement

| Factor | Value | Rating | Personality |
|--------------------|----------------------|--------|--------------------|
| Team familiarity | Know each other well | -20 | Armchair architect |
| Team size | Large (12 members) | +20 | Control freak |
| Overall experience | Mostly junior | +20 | Control freak |
| Project complexity | High complexity | +20 | Control freak |
| Project duration | 6 months | -20 | Armchair architect |
| Accumulated score | | +20 | Control freak |

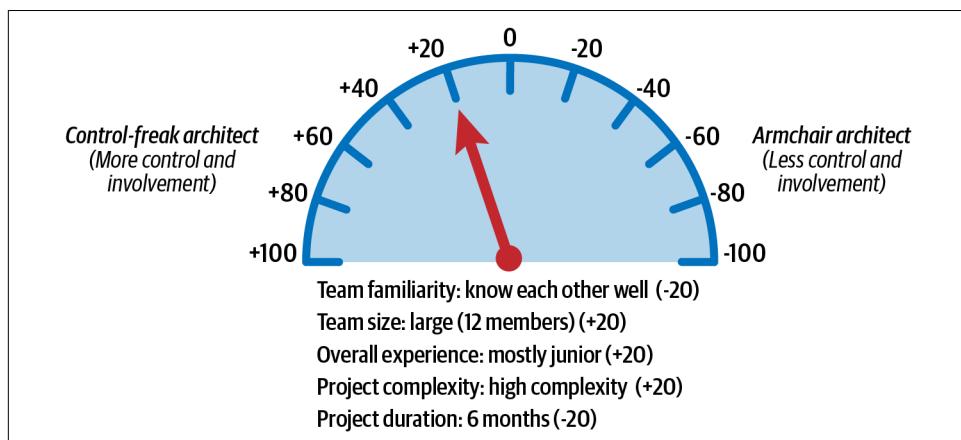


Figure 24-6. Amount of involvement for Scenario 2

Architects use these factors at the start of a project to determine how involved they plan to be, but as the project progresses, their level of involvement usually changes. We thus recommend analyzing these factors continually throughout the project's lifecycle.

It is difficult to make these factors objective, since some (such as the team's overall experience level) might carry more significance than others. In these cases the metrics can easily be weighted or modified to suit a particular situation.

The primary message here is that the appropriate amount of architect involvement on a development team varies according to these five factors. By using these factors to gauge their level of involvement, architects can draw the appropriate boundaries and create the right size "room" for the team.

Team Warning Signs

We mentioned team size as one of the five factors that help an architect determine the level of involvement on a development team: the larger a team, the more architect involvement is needed; the smaller the team, the less involvement is needed. However, what constitutes a “large team”? In this section, we’ll look at three factors that can help an architect determine if the team is too large, and hence not as effective as it can be.

Process Loss

The term *process loss* was coined by Fred Brooks in his book *The Mythical Man-Month* (Addison-Wesley, 1995). The basic idea of process loss, otherwise known as **Brooks's Law**, is that *the more people you add to a project, the more time the project will take*. As Figure 24-7 shows, *group potential* is defined by the collective efforts of everyone on the team. However, Brooks states that any team’s *actual productivity* will always be less than its *potential productivity*; the difference is called the team’s *process loss*.

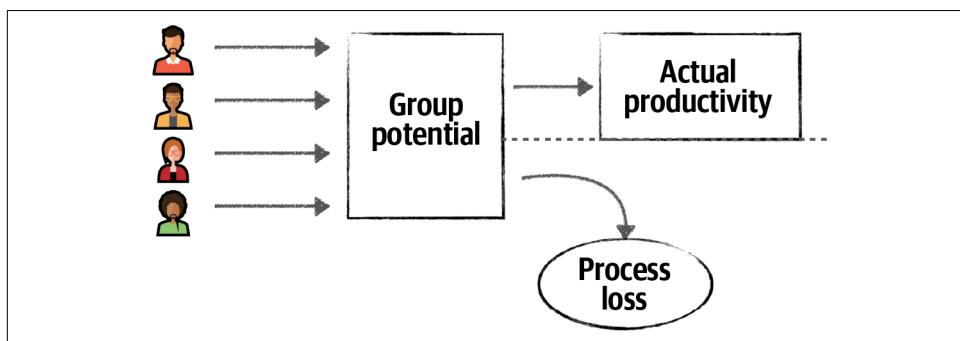


Figure 24-7. Brooks's Law holds that a team's size affects its actual productivity

Process loss is a good factor in determining the correct team size for a particular project, and an effective software architect observes the development team to look for indications of process loss. For instance, if team members frequently run into merge conflicts when pushing code to a repository, this is an indication that they are working on the same code and possibly getting in each other’s way.

To avoid process loss, we recommend looking for areas of parallelism and having team members working on separate services or areas of the application. Anytime a project manager proposes adding a new team member to a project, an effective architect will look for opportunities to create parallel work streams. If they find none, they notify the project manager that a new addition could have a negative impact on the team.

Pluralistic Ignorance

Pluralistic ignorance is when everyone privately rejects a norm, but agrees to it because they think they are missing something obvious. For example, suppose the majority of people on a large team agree that using messaging between two remote services is the best solution. One person thinks this is a silly idea, because of a secure firewall between the two services. However, that person publicly agrees along with everyone else to use messaging. Although they privately reject the idea, they are afraid that they might be missing something obvious if they speak up. The larger the group, the less willing people are to confront others. On a smaller team, they might have spoken up to challenge the original solution, prompting the team to land on another protocol (such as REST) for a better solution.

The concept of pluralistic ignorance was made famous by the Danish children's story "[The Emperor's New Clothes](#)", by Hans Christian Andersen. In the story, two con artists, posing as tailors, convince the king that the new clothes they've "made" for him are invisible to anyone who is unworthy to see them. The king, who can't see the clothes but is unwilling to admit that he himself must be unworthy, struts around totally nude, asking all of his subjects how they like his new clothes. The subjects, afraid of being considered unworthy, assure the king that his new clothes are the best thing ever. This folly continues until a child finally calls out to the king that he isn't wearing any clothes at all.

During meetings, an effective software architect observes people's facial expressions and body language, alert for the possibility that some might mask their skepticism because of pluralistic ignorance. If the architect senses this happening, they act as a facilitator—perhaps interrupting to ask a skeptic what they think about the proposed solution and supporting them when they speak up, even if the person is wrong. The point here is for the architect, as a facilitator, to make sure everyone feels it is a safe enough environment to speak up.

The third factor that indicates appropriate team size is called *diffusion of responsibility*. As teams get bigger, their growth has a negative impact on communication. If team members are confused about who is responsible for what and things are getting dropped, those are good signs that the team is too large.

[Figure 24-8](#) shows someone standing next to a broken-down car on the side of a country road. In this scenario, how many people might stop and ask the motorist if everything is OK? Because it's a small road, it's probably in a small community, so maybe everyone who passes by will stop. But what if that same motorist is stranded on the side of a busy highway in a large city? Thousands of cars might simply drive by without anyone stopping to ask if everything is OK. This is a good example of the diffusion of responsibility. As cities get busier and more crowded, people assume the motorist has already called for help, or that some other member of the crowd

witnessing the event will help. However, in most of these cases, help is not on the way, and the motorist is stuck with a dead or forgotten cell phone.



Figure 24-8. Diffusion of responsibility

An effective architect not only guides the development team through implementing the architecture, but ensures that its members are healthy, happy, and working together to achieve a common goal. Looking for these three warning signs and helping to correct the problems they indicate is a good way to ensure an effective development team.

Leveraging Checklists

Airline pilots use checklists on every flight. Even the most experienced, seasoned veteran pilots have checklists for takeoff, landing, and thousands of other situations—both common ones and unusual edge cases. They use checklists because one missed aircraft setting or procedure (such as forgetting to set the flaps to 10 degrees before takeoff) can mean the difference between a safe flight and a disaster.

In Dr. Atul Gawande's excellent book *The Checklist Manifesto* (Picador, 2011), he describes the power of checklists to make surgical procedures safer. Alarmed at the high rate of staph infections in hospitals, Dr. Gawande created surgical checklists. Infection rates in hospitals using the checklists fell to near zero, while rates in control hospitals not using the checklists continued to rise.

Checklists work. They're excellent vehicles for making sure every task is covered and addressed. So why doesn't the software development industry leverage them too? Having worked for many years in this industry, we firmly believe that checklists make a big difference in the effectiveness of development teams. Of course, most software developers aren't handling life-and-death matters like flying airliners or performing

open-heart surgery. In other words, software developers don't require checklists for everything. The key is knowing when to leverage them and when not to.

Figure 24-9 is not a checklist—it's a set of procedural steps for creating a new database table, so it should not be in checklist form. Some of its tasks have dependencies; for example, the database table cannot be verified if the form has not yet been submitted. Any process with a procedural flow of dependent tasks should not be in a checklist. Nor should simple, familiar processes that are executed frequently without error.

| Done | Task description |
|--------------------------|---|
| <input type="checkbox"/> | Determine database column field names and types |
| <input type="checkbox"/> | Fill out database table request form |
| <input type="checkbox"/> | Obtain permission for new database table |
| <input type="checkbox"/> | Submit request form to database group |
| <input type="checkbox"/> | Verify table once created |

Figure 24-9. Example of a bad checklist

Good candidates for checklists include processes that don't have a set procedural order or dependent tasks, as well as those where people frequently skip steps or make errors. Don't go overboard and start making everything a checklist. Architects often do this once they find that checklists do, in fact, make development teams more effective. They risk invoking what is known as the *Law of Diminishing Returns*. The more checklists the architect creates, the less likely developers are to use them. It's also wise to make checklists as small as possible while still capturing all the necessary steps. Developers generally will not follow overly long checklists. If any of the tasks listed can be automated, automate them and remove them from the checklist.



Don't worry about stating the obvious in a checklist. The obvious stuff is what usually gets missed.

Three of the checklists we've found most helpful cover developer code completion, unit and functional testing, and software releases. Each checklist is discussed in the following sections.

The Hawthorne Effect

The hardest part of introducing checklists to a development team is getting developers to actually use them. It's all too common for some developers to run out of time and simply check off all the items in a checklist without actually performing the tasks.

One way to address this issue is by talking with the team about the difference using checklists can make and having them read *The Checklist Manifesto* by Atul Gawande. Make sure each team member understands the reasoning behind each checklist. Consider having them decide collaboratively what procedures should and shouldn't be on a checklist—creating a sense of ownership also helps.

When all else fails, there's the *Hawthorne effect*: the tendency for people who know they are being observed or monitored to change their behavior, generally to do the right thing. This effect doesn't require actual monitoring so much as a perception; for example, many employers mount non-functioning cameras in highly visible areas, while others install website monitoring software that they rarely check. (How many of those reports are actually viewed by managers?)

To use the Hawthorne effect to govern checklists, let the team know that because using checklists is critical to the team's productivity, all checklists will be verified to make sure the task was actually performed. In reality, occasional spot-checks are all that's needed; developers will be much less likely to skip items or falsely mark them as completed.

Developer Code-Completion Checklist

The developer code-completion checklist is a useful tool, particularly when a developer states that they are “done” with the code. It also is useful for arriving at a “definition of done”: if everything in the checklist is complete, the developer can say they are actually done with the code they were working on.

Here are some things to include in a developer code-completion checklist:

- Coding and formatting standards not included in automated tools
- Frequently overlooked items (such as absorbed exceptions)
- Project-specific standards
- Special team instructions or procedures

Figure 24-10 shows an example developer code-completion checklist. There are some obvious tasks here, like “Run code cleanup and code formatting” and “Make sure there are no absorbed exceptions.” How often do developers in a hurry forget to run code cleanup and formatting from the IDE? Plenty often. In *The Checklist Manifesto*, Gawande finds the same phenomenon with respect to surgical procedures—the obvious tasks are often the ones missed.

| Done | Task description |
|--------------------------|--|
| <input type="checkbox"/> | Run code cleanup and code formatting |
| <input type="checkbox"/> | Execute custom source validation tool |
| <input type="checkbox"/> | Verify the audit log is written for all updates |
| <input type="checkbox"/> | Make sure there are no absorbed exceptions |
| <input type="checkbox"/> | Check for hardcoded values and convert to constants |
| <input type="checkbox"/> | Verify that only public methods are calling setFailure() |
| <input type="checkbox"/> | Include @ServiceEntry point on service API class |

Figure 24-10. Example of a developer code-completion checklist

The architect should always review the checklist to see if any items can be automated or written as plug-ins for a code-validation checker. While the project-specific tasks in the checklist (such as executing the custom validator, verifying the audit log is written, calling the `setFailure()` method, and including the `@ServiceEntry` annotation) are good to have in a checklist, some of these could be automated. For example, while an automated check might not be feasible for “Include `@ServiceEntry` point on service API class,” it would work for “Verify that only public methods are calling `setFailure()`”—a straightforward check to automate with any code-crawling tool. Checking for areas of automation shortens the checklist’s size and improves its ratio of signal to noise.

Unit and Functional Testing Checklist

Perhaps one of the best checklists is for unit and functional testing. This checklist contains some of the more unusual and edge cases that software developers tend to forget to test. Whenever someone from QA finds a code issue based on a particular test case, add that test case to this checklist.

This particular checklist is usually one of the longest, since it encompasses all the types of tests that can be run against code. Its purpose is to ensure the most complete testing possible so that when the developer is done with the checklist, the code is essentially production ready.

Here are some of the items found in a typical unit and functional testing checklist:

- Special characters in text and numeric fields
- Minimum and maximum value ranges
- Unusual and extreme test cases
- Missing fields

As with the developer code-completion checklist, if any items can be written as automated tests or are already included in the automated test suite, they should be removed from the checklist and automated.

Developers sometimes don't know where to start when writing unit tests or how many they should write. This checklist provides a way to make sure general and specific test scenarios are included in the development process. In organizations where testing and development are performed by separate teams, this checklist helps to bridge the gap. The more development teams perform complete testing, the easier they make the testing teams' jobs, freeing them up to focus on business scenarios not covered in the checklists.

Software-Release Checklist

Releasing software into production is perhaps one of the most error-prone points in the software development lifecycle, so it makes for a great checklist. This checklist helps avoid failed builds and deployments, and significantly reduces the risk associated with releasing software.

The software-release checklist is usually the most volatile of the checklists presented here, because it changes each time a deployment fails or has problems, to address new errors and shifting circumstances.

The software-release checklist typically includes:

- Configuration changes in servers or external configuration servers
- Third-party libraries added to the project (JAR, DLL, etc.)
- Database updates and corresponding database-migration scripts

Anytime a build or deployment fails, the architect should analyze the root cause of the failure and add a corresponding entry to the software-release checklist. This way, the item will be verified during the next build or deployment, preventing the problem from happening again.

Providing Guidance

Another way software architects can make teams effective is by using design principles to provide guidance. This also helps form the constraint “room” in which developers work to implement the architecture. Communicating design principles is one of the keys to creating a successful team.

To illustrate this point, imagine you’re guiding a development team in using what is typically called the *layered stack*—the collection of third-party libraries that make up the application. Development teams usually have lots of questions about the layered stack, including which libraries are OK, which are not, and whether or when they can make their own decisions about libraries.

You might begin by having the developers answer the following questions about a library they want to use:

- Are there any overlaps between the proposed library and the system’s existing functionality?
- What is the justification for using the proposed library?

The first question guides developers to check if the functionality provided by the new library can be satisfied through an existing library or existing functionality. When developers ignore this activity (which sometimes happens), they can end up creating lots of duplicate functionality, particularly in large projects and teams.

The second question prompts the developer to ask if the new library or functionality is truly needed. We recommend asking for both a technical justification and a business justification, in part because this technique helps make developers aware of the need to provide business justifications.

The Impact of Business Justifications

One of your authors was the lead architect on a particularly complex Java-based project with a large development team. One team member was obsessed with the Scala programming language and desperately wanted to use it on the project. Their desire to use Scala ended up becoming so disruptive that two key team members declared their intention to leave the project for other, “less toxic” environments. Your author convinced them to hold off. Then he told the Scala enthusiast that he would support using Scala within the project *if* the enthusiast provided a business justification for the costs of the training and rewriting involved. The Scala enthusiast was ecstatic and said they would get right on it. They left the meeting yelling, “Thank you—you’re the best!”

The next day, the Scala enthusiast came into the office completely transformed and asked to speak with your author. They began by immediately (and humbly) saying,

“Thank you.” The Scala enthusiast explained that they had come up with all the technical reasons in the world to use Scala, but none of those technical advantages had any business value in terms of the cost, budget, and timeline. In fact, the Scala enthusiast had realized two things: first, that the increase in cost, budget, and timeline would provide no benefit whatsoever; and, second, that they’d been disrupting the team. Before long, the Scala enthusiast had transformed into one of its best and most helpful members. Being asked to provide a business justification for something they wanted increased their awareness of the business’s needs, making them a better software developer and making the team stronger and healthier. The two key developers who’d been planning on leaving stayed on the team.

Another good way to communicate design principles is through graphical explanations about which decisions the development team can make and which they can’t. The graphic in [Figure 24-11](#) shows what this might look like for controlling the layered stack.

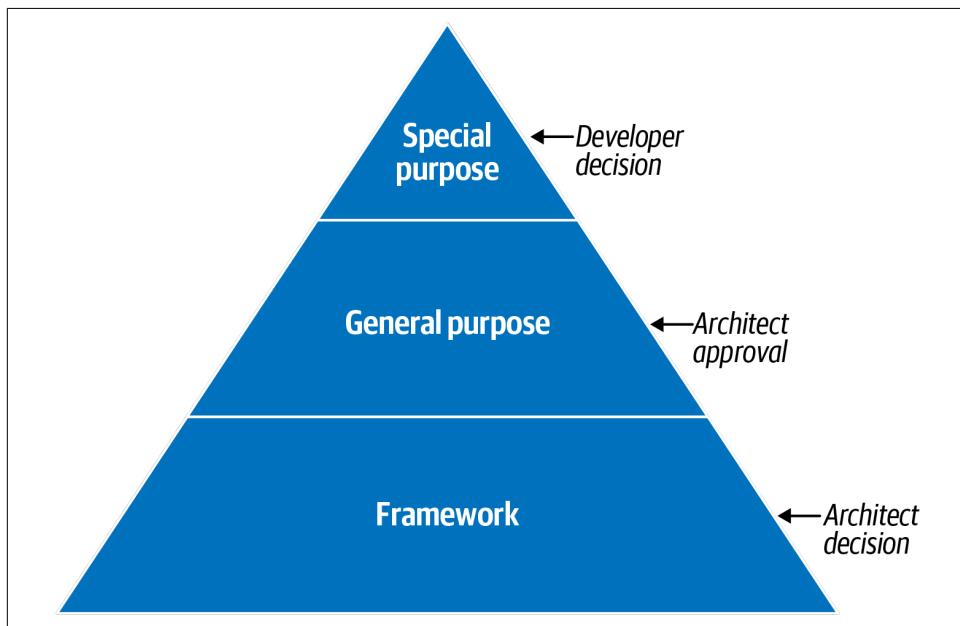


Figure 24-11. Providing guidance for the layered stack

These categories are only examples, but many more can be defined. In presenting [Figure 24-11](#) to the team, the architect should apply each category to the third-party library proposed for use:

Special purpose

These are specific libraries used for things like rendering PDFs, scanning barcodes, and other circumstances that do not warrant writing custom software.

General purpose

These libraries are wrappers on top of the language API and include things like Apache Commons and Guava for Java.

Framework

These libraries are used for things like persistence (such as Hibernate) and inversion of control (such as Spring). In other words, they make up an entire layer or structure of the application and are highly invasive.

Next, the architect creates the “room” around this design principle. Notice, in [Figure 24-11](#), that the architect has authorized developers to make decisions about special-purpose libraries without consulting the architect. For general-purpose libraries, however, while developers can analyze overlap analysis, provide justifications, and make a recommendation, this category of library requires architect approval. Finally, framework libraries are entirely the architect’s responsibility—the development teams shouldn’t even perform analysis for these types of libraries.

Summary

Making development teams effective is hard work. It requires lots of experience and practice, as well as strong people skills (which we will discuss in subsequent chapters). That said, the simple techniques in this chapter for elastic leadership, leveraging checklists, and providing guidance by communicating design principles do in fact work. We’ve seen how useful they prove to be in making development teams work more intelligently.

Some question architects’ role in such activities, insisting that this work should be assigned to the development manager or project manager. We strongly disagree. Software architects guide the team on technical matters and lead them through implementing the architecture. Building a close collaborative relationship with a development team allows the architect to observe team dynamics and facilitate changes to make the team more productive.

Negotiation and Leadership Skills

Negotiation and leadership skills are hard skills, obtained only through many years of learning, practice, and mistakes—but they are critical in becoming an effective software architect. This book can't make anyone an expert in negotiation and leadership overnight, but the techniques we introduce in this chapter are a good starting point. To dive deeper into this topic, we suggest Tanya Reilly's book *The Staff Engineer's Path: A Guide for Individual Contributors* (O'Reilly, 2022) and the classic negotiation book *Getting to Yes: Negotiating Agreement Without Giving In* (Penguin Books, 2011) by Roger Fisher, William L. Ury, and Bruce Patton.

Negotiation and Facilitation

In Chapter 1, we listed core expectations for software architects. The last expectation we discussed was understanding and being able to navigate your organization's office politics. The reason this is such an important expectation is that almost every decision you make as a software architect will be challenged: by developers who think they know more, by other architects who think they have a better idea or way of approaching the problem, and by stakeholders who think your solution is too expensive or will take too much time.

Negotiation is one of an architect's most important skills. Effective software architects understand the politics of the organization, have strong negotiation and facilitation skills, and can overcome disagreements to create solutions that all stakeholders agree on.

Negotiating with Business Stakeholders

Consider the following scenario in which you, as the lead architect, must negotiate with a key business stakeholder:

Scenario 1

Parker, the senior vice president and product sponsor for this project, insists that the new global trading system must support “five nines” of availability (99.999%). However, based on the amount of time between global markets when trading doesn’t occur (two hours), you know that three nines of availability (99.9%) would be sufficient to meet the project requirements. The problem is, you’ve seen that Parker does not like to be wrong and doesn’t respond well to being corrected, especially if they perceive it as condescending. Parker isn’t technically knowledgeable, but thinks they are, so they tend to get involved in the non-functional aspects of projects. As the architect, your goal is to convince Parker, through negotiation, that three nines (99.9%) of availability would be enough.

You’ll have to be careful to not be too egotistical and forceful in your analysis, but you also need to make sure you’re not missing anything that might prove you wrong during the negotiation. There are several key negotiation techniques you can use to help with this sort of stakeholder negotiation. The first is:



Pay attention to the buzzwords and jargon people use, even if they seem meaningless. They often contain clues that can help you better understand and negotiate the situation.

Corporate buzzwords are generally meaningless, but can nevertheless provide valuable information when you’re about to enter into negotiations. Say you ask when a particular feature is needed and Parker responds, “I needed it yesterday.” You can’t literally provide that, but it should tell you that time to market is important to this stakeholder. Similarly, the phrase “this system must be lightning fast” means performance is a big concern, and “zero downtime,” rather than being literal, means that availability is critical in this application. Effective software architects read between the lines of such exaggerated statements to identify the stakeholder’s real concerns. This leads to a second negotiation technique:



Gather as much information as possible *before* entering into a negotiation.

Parker’s use of the phrase “five nines” indicates that they want the system to have high availability (specifically, 99.999% of uptime). However, it might be the case that Parker is unaware of what “five nines” of availability actually means in terms of

amount of downtime per year. **Table 25-1** shows the downtimes based on the number of nines of availability.

Table 25-1. The nines of availability

| Percentage uptime | Downtime per year (per day) |
|----------------------|-----------------------------|
| 90.0% (one nine) | 36 days 12 hrs (2.4 hrs) |
| 99.0% (two nines) | 87 hrs 46 min (14 min) |
| 99.9% (three nines) | 8 hrs 46 min (86 sec) |
| 99.99% (four nines) | 52 min 33 sec (7 sec) |
| 99.999% (five nines) | 5 min 35 sec (1 sec) |
| 99.9999% (six nines) | 31.5 sec (86 ms) |

“Five nines” of availability comes out to 5 minutes and 35 seconds of downtime per year, or an average of 1 second a day of unplanned downtime. That’s ambitious, costly, and, as it turns out, unnecessary for Scenario 1 based on the amount of time between global trading when trading doesn’t occur. Stating these goals in hours and minutes (or, in this case, seconds) is a much better way to have the conversation than sticking with the “nines” vernacular because it brings some actual metrics and quantified numbers into the discussion.

Negotiating Scenario 1 would include validating Parker’s concerns (“I understand that availability is very important for this system”), then steering the negotiation from the nines vernacular to one of reasonable hours and minutes of unplanned downtime. Three nines (which you deem good enough) comes to an average of 86 seconds of unplanned downtime per day—certainly a reasonable number, given the context of this global trading system. Stating it this way might be enough to convince Parker that this amount of unplanned downtime is satisfactory. However, if it doesn’t, here’s another tip:



When all else fails, state things in terms of *qualified* cost and time.

We recommend saving this negotiation tactic for last. We’ve seen too many negotiations start off on the wrong foot due to opening statements like “That’s going to cost a lot of money” or “We don’t have time for that.” Money and time (which includes the effort involved) are certainly key factors in any negotiation, but try other justifications and rationalizations that matter more before you bring them up. They should be a last resort. Once you reach an agreement with the stakeholder, you can consider cost and time if they are important.

Another important negotiation technique for such situations:



When you need to qualify demands or requirements, “divide and conquer.”

In *The Art of War*, the ancient Chinese warrior Sun Tzu writes of one’s opponent, “If his forces are united, separate them.” You can use this divide-and-conquer tactic during negotiations as well. In Scenario 1, Parker is insisting on five nines (99.999%) of availability for the new trading system. But does the *entire system* needs five nines of availability? You can qualify the requirement, narrowing it down to any specific area(s) of the system that genuinely need five nines of availability. This reduces the scope of difficult (and costly) requirements, not to mention the scope of the negotiation.

Negotiating with Other Architects

It’s not unusual for architects to have to negotiate with other architects on a project. Consider Scenario 2, where two architects disagree on which protocol to use:

Scenario 2

You’re the senior of two architects on a project, and you believe that asynchronous messaging would be the right approach for communication between a group of services, to increase both performance and scalability. However, Addison, the other architect on the project, strongly disagrees. They insist that REST would be a better choice, because it’s always faster than messaging and can scale just as well. They cite their research, which consists of a Google search and the output of a prompt to a popular generative AI tool. This is not the first heated debate you’ve had with Addison, nor will it be the last. You want to convince them that messaging is the right solution.

Now, as the senior architect, you certainly could tell Addison that their opinion doesn’t matter and ignore it. However, this will only intensify the animosity between you, and an unhealthy, noncollaborative relationship between the project’s two architects would almost certainly have a negative impact on the development team.



Always remember that *demonstration defeats discussion*.

Rather than arguing over REST versus messaging, you should *demonstrate* to Addison why messaging would be a better choice in this specific environment. Because every environment is different, simply Googling it rarely yields the correct answer. But if you compare the two options in a production-like environment and show Addison the results, you might be able to avoid an argument entirely.



Avoid being overly argumentative or letting things get too personal. Calm leadership, combined with clear and concise reasoning, will almost always win a negotiation.

This is a very powerful technique for dealing with adversarial relationships. Once things get personal or heated, the best thing to do is stop the negotiation. Re-engage later, when both parties have calmed down. Architects argue from time to time, but if you stay calm and project leadership, the other person will usually back down.

Negotiating with Developers

Effective software architects gain the team's respect through working together. That way, when they request something of the development team, it's far less likely to spark an argument or resentment.

As you saw in [Chapter 24](#), working with development teams can be difficult at times. When development teams feel disconnected from the architecture (or the architect), they often feel left out of decisions. This is a classic example of the *Ivory Tower* architecture antipattern. Ivory tower architects dictate from on high, giving development teams orders without regard for their opinions or concerns. This usually leads to the team losing respect for the architect and can eventually cause the team's dynamics to break down altogether. One negotiation technique that can help address this situation is always providing a justification for your decisions.



When convincing developers to adopt an architecture decision or to do a specific task, provide a justification rather than "dictating from on high."

If you provide a reason *why* something needs to be done, developers are more likely to agree. For example, consider the following conversation between an architect and a developer about making a simple query within a traditional n-tiered layered architecture:

Architect: "You must go through the Business layer to make that call."

Developer: "I disagree. It's much faster just to call the database directly."

There are several things wrong with this conversation. First, notice the architect's use of the words "you must." Not only is this type of commanding voice demeaning, it's one of the worst ways to begin a negotiation (or a conversation). The developer's response includes a reason: going through the Business layer will be slower and take more time.

Now consider an alternative approach:

Architect: "Since change control is most important to us, we have formed a closed-layered architecture. This means all calls to the database need to come from the Business layer."

Developer: "OK, I get it—but in that case, how am I going to deal with these performance issues for simple queries?"

Here, the architect is *justifying* the demand that all calls go through the Business layer. Providing the justification *first* is always a good approach. Most people tend to stop listening as soon as they hear something they disagree with. Stating the reason before the demand ensures that the developer will hear the architect's justification.

The architect has also taken a less personal approach to phrasing this demand. By saying "this means" rather than "you must," they've turned the demand into a simple statement of fact. Now take a look at the developer's response: instead of disagreeing with the layered-architecture restrictions, the developer asks a question about improving performance for simple calls. Now the two can engage in a collaborative conversation to find ways to make simple queries faster while still preserving the closed layers in the architecture.

Another effective negotiation tactic is to have the developer arrive at the solution on their own. For example, suppose that you, as an architect, are choosing between two frameworks, Framework X and Framework Y. Framework Y doesn't satisfy the security requirements for the system, so you naturally choose Framework X. A developer on your team strongly disagrees, insisting that Framework Y would still be the better choice. Rather than argue the matter, you tell the developer that if they can show you how Framework Y addresses the security concerns, the team will use Framework Y.

One of two things will happen next. Option 1 is that the developer tries to demonstrate that Framework Y satisfies the security requirements, but fails. In failing, they come to understand firsthand why the team can't use this framework. And because they arrive at the solution on their own, you automatically get their buy-in for the decision to use Framework X. You've essentially made it the developer's decision. This is a win. Option 2 is that the developer finds a way to address the security requirements with Framework Y and demonstrates it to you. This is a win as well. In this case, you missed something in your assessment of Framework Y, and now you have a better solution to the problem (and the developer still feels involved in the decision).



If a developer disagrees with a decision you make, have them arrive at the solution on their own.

Developers are smart people with useful knowledge. Collaborating with the development team is how architects gain their respect—and their assistance in finding better solutions. The more developers respect an architect, the easier it will be for the architect to negotiate with them.

The Software Architect as a Leader

A software architect is also a leader, guiding a development team as they implement the architecture. We maintain that about 50% of being an effective software architect is having good people skills, including facilitation and leadership. In this section, we discuss leadership techniques for software architects.

The 4 Cs of Architecture

We've said many times that change is a constant—and that's never more true than when the change in question involves things getting more and more complex, whether it's business processes, technology, or even architecture itself. As you saw in [Part III](#) of this book, some architectures are *very* complex. If an architecture needs to support six nines of availability (99.9999%), that's equivalent to unplanned downtime of about 86 milliseconds a day, or 31.5 seconds of downtime per *year*. This sort of complexity is known as *essential complexity*—in other words, “we have a hard problem.”

It's easy for architects (and developers) to fall into the trap of adding unnecessary complexity to solutions, diagrams, and documentation. To quote Neal:

Developers are drawn to complexity like moths to a flame—frequently with the same result.

[Figure 25-1](#) diagrams the major information flows of the backend processing systems at a very large global bank. Is this system *necessarily* complex—in other words, does it *have* to be complex? No one knows, because the architect has *made* it complex. This sort of complexity is called *accidental complexity*: in short, “we have made a problem hard.” Architects sometimes add complexity to prove their worth when things seem too simple, or to guarantee that they are always kept in the loop on decisions or even to maintain job security. Whatever the reason, introducing accidental complexity into something that does not have to be complex is one of the best ways to lose respect and become an ineffective leader and architect.

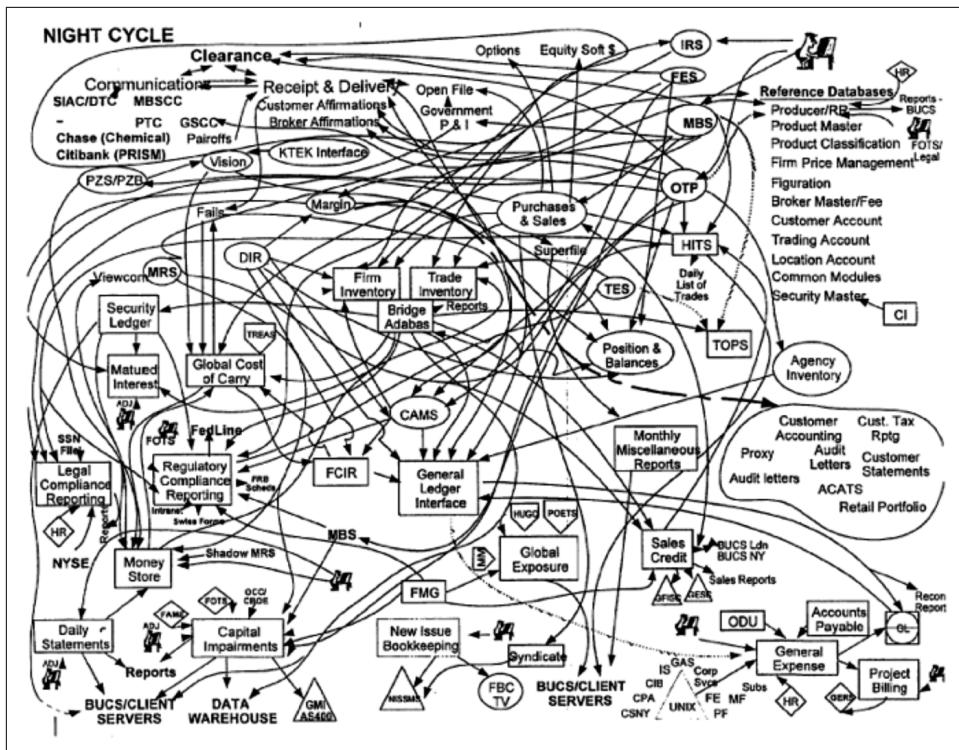


Figure 25-1. Introducing accidental complexity into a problem

To avoid accidental complexity, we use what we call the 4 Cs of architecture leadership: *communication*, *collaboration*, *clear*, and *concise*. These factors, as shown in Figure 25-2 (not to be confused with the 4 Cs within the C4 Model of diagramming), work together to create an effective communicator and collaborator on the team.

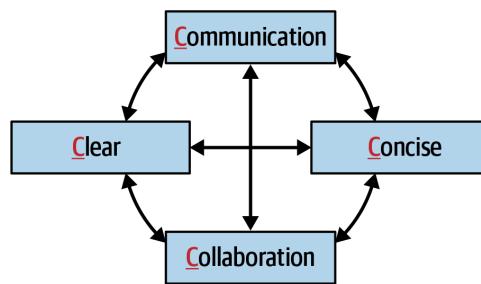


Figure 25-2. The 4 Cs of architecture

As a leader, facilitator, and negotiator, it's vital for a software architect to be able to communicate clearly and concisely. It is equally important to be able to collaborate with

developers, business stakeholders, and other architects. Focusing on the 4 Cs helps an architect gain the team's respect and helps make them the go-to person on the project for questions, advice, mentoring, coaching, and leadership.

Be Pragmatic, Yet Visionary

An effective software architect must be pragmatic, yet visionary. Striving for this balance is not as easy as it sounds; it takes a fairly high level of maturity and significant practice to accomplish.

A *visionary* is someone who thinks about or plans the future with imagination or wisdom. Being a visionary means applying strategic thinking to a problem, which is exactly what architects do. Architects plan for the future and make sure the architecture they build remains vital (valid and useful) for a long time. However, architects often become too theoretical in their planning and designs, creating solutions that are too difficult to implement—or even understand.

Now consider the other side of the coin, which is being *pragmatic*. Being pragmatic means dealing with things sensibly and realistically in a way that is based on *practical* rather than *theoretical* considerations. While architects need to be visionaries, they also need to apply practical, realistic solutions. Being pragmatic when creating an architectural solution means accounting for:

- Budget constraints and other cost-based factors
- Time constraints and other time-based factors
- The development team's skill set and skill level
- The trade-offs and implications of each architecture decision
- The technical limitations of any proposed design or solution

Good software architects strive to balance pragmatism with imagination and wisdom in solving problems (see [Figure 25-3](#)).

For example, consider an architect faced with sudden, significant increases in concurrent user load for reasons that are unclear. A visionary might come up with an elaborate way to improve the system's elasticity by breaking apart the databases and building a complex *data mesh*: a collection of distributed, domain-partitioned databases used to separate analytical data concerns from transactional ones. In theory, this approach *might* be possible, but being pragmatic means applying reason and practicality to the solution. For example, has the company ever used a data mesh before? What are the trade-offs of using a data mesh? Would this solution really solve the problem?

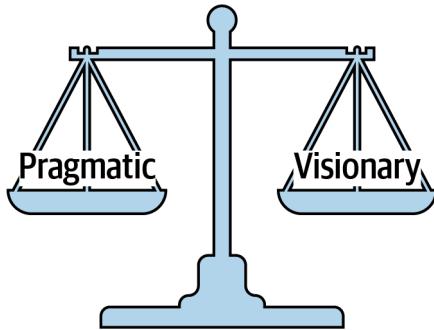


Figure 25-3. Good architects find the balance between being pragmatic and being visionary

Maintaining a good balance between being pragmatic and being visionary is an excellent way to gain respect as an architect. Business stakeholders appreciate visionary solutions that fit within a set of constraints, and developers appreciate having a practical (rather than theoretical) solution to implement.

A pragmatic architect would first look at what factors are limiting the system's elasticity. Finding and isolating any bottlenecks would be a practical first approach to this problem. Is the database creating a bottleneck with respect to some of the services invoked or other external sources needed? If so, could some of the data be cached to reduce calls to the database?

Leading Teams by Example

Bad software architects “pull rank,” using their title to get people to do things. Effective software architects get people to do things by leading through example. Again, this is all about gaining the respect of development teams, business stakeholders, and other people throughout the organization (such as the head of operations, development managers, and product owners).

The classic “lead by example, not by title” story involves a captain in command of a sergeant during a military battle. The captain, who is largely removed from the troops, commands all of the troops forward to take a particularly difficult hill. The soldiers, full of doubt, look for confirmation from the lower-ranking sergeant. The sergeant, understanding the situation, nods his head slightly, and the soldiers immediately move forward with confidence to take the hill.

The moral of this story is that rank and title mean very little when it comes to leading people. The computer scientist **Gerald Weinberg** is famous for saying, “No matter what the problem is, it’s a people problem.” Most people think that solving technical issues has nothing to do with people skills and everything to do with *technical*

knowledge. While technical knowledge is certainly necessary, it's only a part of solving any problem.

Suppose, for example, that an architect is meeting with a team of developers to solve an issue that's come up in production. One of the developers makes a suggestion, and the architect responds, "Well, *that's* a dumb idea." Not only will that developer not make any more suggestions, but now none of the other developers dare to say anything. The architect has just shut down collaboration across the entire team.

Let's look at another snippet of dialogue between an architect and a developer:

Developer: "So how are we going to solve this performance problem?"

Architect: "What you need to do is use a cache. That would fix the problem."

Developer: "Don't tell me what to do."

Architect: "What I'm telling you is that it would fix the problem."

This is a good example of communicating without *collaborating*. By using the words "what you need to do is," the architect is shutting down collaboration. Now consider a revised approach:

Developer: "So how are we going to solve this performance problem?"

Architect: "Have you considered using a cache? That might fix the problem."

Developer: "Hmmm, no, we didn't think about that. What are your thoughts?"

Architect: "Well, if we put a cache here..."

The words "have you considered" turn the command into a question, placing control back in the developer's hands so that they can have a collaborative conversation with the architect. How you use language is vitally important to building a collaborative environment.

Leading collaboration as an architect isn't just about how you personally collaborate with others—it's also about facilitating collaboration among the team members. Try to observe team dynamics and notice situations like the one in this dialogue. If you witness a team member using demanding or condescending language, take them aside and coach them on using collaborative language. Not only will this create better team dynamics, it will also help the team members respect one another.

If you need to get someone to do something they otherwise might not want to do, sometimes it's best to turn a request into a favor. In general, human beings dislike being told what to do, but want to help others. Consider the following conversation between an architect and developer regarding an architecture refactoring effort during a busy iteration:

Architect: "I'm going to need you to split the payment service into five different services, with each service containing the functionality for each type of payment we accept, such as store credit, credit card, PayPal, gift card, and reward points. That will provide better fault tolerance and scalability in the website. It shouldn't take too long."

Developer: "Sorry, I'm way too busy this iteration for that. I really can't do it."

Architect: “Listen, this is important and it needs to be done this iteration.”

Developer: “Sorry, I can’t. Maybe one of the other developers can do it. I’m just too busy.”

The developer immediately rejects the task, even though the architect has justified it as providing better fault tolerance and scalability. The architect is *telling* the developer to do something they are simply too busy to do—and their demand doesn’t even include the person’s name! Now consider the technique of turning the request into a favor:

Architect: “Hi, Sridhar. Listen, I’m in a real bind. I really need to have the payment service split into separate services for each payment type to get better fault tolerance and scalability, and I waited too long to do it. Is there any way you can squeeze this into this iteration? It would really help me out.”

Developer (*pausing*): “I’m really busy this iteration, but I guess I’ll see what I can do.”

Architect: “Thanks, Sridhar, I really appreciate the help. I owe you one.”

Developer: “No worries. I’ll see that it gets done this iteration.”

First, using the person’s name makes the conversation more personal and familiar, rather than an impersonal professional demand. Using a person’s name and proper pronouns during conversations or negotiations can help build respect and healthy relationships. Not only do people like hearing their own names, but it creates a sense of familiarity as well. Practice remembering people’s names by using them frequently. If you find a name hard to pronounce, research the correct pronunciation, then practice it until it is perfect. When someone tells you their name, we like to repeat it back and ask if we’re pronouncing it correctly. If it’s not correct, we repeat this process until we get it right.

Second, the architect in the prior conversation admits they are in a “real bind” and that splitting the services would really “help them out a lot.” This doesn’t always work, but playing off the basic human urge to help others has a better probability of success than the first conversation. Try it the next time you face this sort of situation.

Another effective leadership technique when you meet someone for the first time or greet someone you only see occasionally is to always shake the person’s hand and make eye contact. The handshake is an important people skill that goes back to medieval times. It lets both people know they are friends, not foes, and forms a bond between them. That said, be aware of the cultural aspect of handshakes. For example, in the US, the UK, Europe, and Australia, a handshake is an acceptable means of greeting someone, whereas other cultures use different greetings (such as Japan, where it’s about bowing and the timing of the bow). Nevertheless, sometimes it can be hard to get a simple handshake right.

A handshake should be firm, but not overpowering. Look the person in the eye; looking away while shaking someone’s hand is a sign of disrespect, and most people will notice that. Also, don’t keep the handshake going too long. Two or three seconds

are all you need. Don't go overboard with handshakes either. If you come to the office every morning and start shaking everyone's hand, it's weird enough to make people uncomfortable. However, a monthly meeting with the head of operations is the perfect opportunity to stand up, say "Hello, Ruth, nice seeing you again," and give Ruth a quick, firm handshake. Knowing when to shake hands and when not to is part of the complex art of people skills.

As a leader, be careful to respect and preserve personal boundaries between people at all levels. The handshake is a professional way of forming a physical bond. Since that's a good thing, some people assume that it's even better and more bonding to hug someone. It's not. Hugging in a professional setting, regardless of the environment, can make people uncomfortable and could potentially even become a form of workplace harassment. The same holds true for general conduct in the workplace or when traveling with a coworker. Skip the hugs, adhere to commonsense professional conduct, and stick with handshakes.

A good leader should become the "go-to person" on the team—the person developers go to with questions and problems. An effective software architect will seize the opportunity and take the initiative to lead the team, regardless of their title or role on the team. If someone is struggling with a technical issue, take initiative: step in and offer help or guidance. The same is true for nontechnical situations. If a team member comes into work looking sort of depressed and bothered, as though something is definitely up, an effective software architect might offer to talk. "Hey, Antonio, I'm heading over to get some coffee. Why don't we head over together?" Then during the walk, they can ask if everything is OK. This provides an opening for a more personal discussion and maybe even, in the best case, a chance to mentor and coach. However, an effective leader will also pay attention to verbal and nonverbal signs (like facial expressions and body language) and recognize when it's time to back off.

Another way to establish yourself as the go-to person on the team is to host periodic brown-bag "lunch and learn" sessions about a specific technique or technology, like design patterns or some new features of the latest programming-language release. If you are reading this book, you have some particular skill or knowledge that others don't have. Hosting these events isn't just about providing valuable information to developers or exhibiting your technical prowess—it's also an opportunity to practice mentoring and public-speaking skills, and identifies you as a leader and mentor.

Integrating with the Development Team

An architect's calendar is usually filled with overlapping meetings, like the calendar shown in [Figure 25-4](#). So how do architects find the time to integrate with the team, guide and mentor them, and be available for questions or concerns?

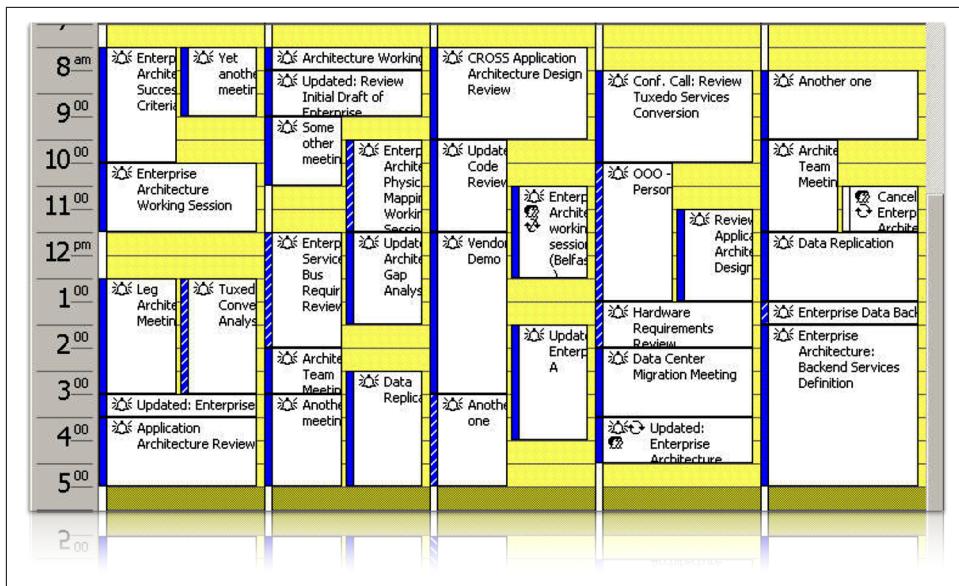


Figure 25-4. A typical calendar of a software architect

Unfortunately, frequent meetings are a necessary evil. The key is controlling them so you can make time for the team. As Figure 25-5 shows, there are two types of meetings: those *imposed upon* you (someone invites you to a meeting), and those *you impose* (you call the meeting).

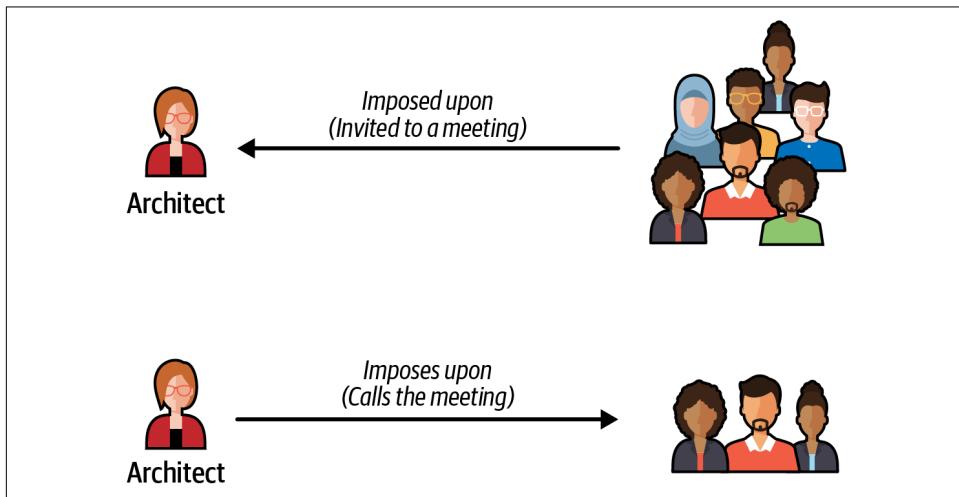


Figure 25-5. Meeting types

The meetings others call are the hardest to control. Because software architects must communicate and collaborate with lots of different stakeholders, they are invited to almost every meeting—even if their presence isn’t really needed. When invited to a meeting, ask the organizer why you’re needed. If inviting you is just a way to keep you in the loop, that’s what meeting notes are for. Asking *why* you should be at a meeting can help you decide which meetings to attend and which you can skip. Looking at the meeting agenda is also useful for this. Also, are you needed at the *whole* meeting, or just the part that will discuss a particular topic? Could you leave after that agenda item? Don’t waste time in a meeting that you could be spending helping the development team solve problems.



Ask for the meeting agenda ahead of time to help qualify if you are really needed at the meeting or not.

When developers, or the tech lead, are invited to a meeting, consider going in their place, particularly if both you and the tech lead are invited, so the team can stay focused on the tasks at hand. While deflecting meetings away from useful team members might increase the time *you* spend in meetings, it increases the development team’s productivity and their respect for you.

As an architect you will sometimes have to impose meetings on others, but since this is where you have control, keep them to an absolute minimum. Set an agenda and stick to it. Don’t let some other issue that isn’t relevant to everyone disrupt the meeting. Ask yourself whether the meeting you are calling is more important than the work it will pull the team away from. For example, if it’s a matter of communicating some important information, could you simply send an email rather than call a meeting? However, if you do need to schedule a meeting with a development team, try to schedule meetings first thing in the morning, right after lunch, or toward the end of the day so as to minimize the disruption to the developers during central work hours.

Developer Flow State

Flow is a state of mind developers frequently get into where the brain gets 100% engaged in a particular problem, allowing full attention and maximum creativity. For example, when a developer might be working on a particularly difficult algorithm or piece of code, hours can feel like minutes. Pay close attention to your team’s *productivity flow* and be sure not to disrupt it. You can read more about flow state in the book *Flow: The Psychology of Optimal Experience* by Mihaly Csikszentmihalyi (Harper Perennial, 2008).

Another good way to integrate with the development team, if you work on-site, is to sit with them. Sitting in a cubicle away from the team sends the message “I am special and should not be disturbed.” Sitting alongside the team sends the message “I’m an integral part of the team and available for questions or concerns.” When you’re on-site but can’t sit with your development team, the best thing to do is walk around and be seen as much as possible. An architect who’s never visible, stuck on a different floor or always in their office, cannot possibly guide the team. Block off time in the morning, after lunch, or late in the day to converse, help with issues, answer questions, and do basic coaching and mentoring. Developers appreciate this type of communication and will respect you for making time for them during the day. The same holds true for other stakeholders: stopping in to say hi to the head of operations while on a coffee run is an excellent way to keep the lines of communication open.

Sometimes, such as if you work in a remote environment, it’s not possible to sit with the development team or walk around and be seen. This makes collaboration much more difficult. For more information on managing remote teams, we highly recommend Jacqui Read’s book *Communication Patterns* (O’Reilly, 2023), Part 4 of which is entirely devoted to remote teams.

Summary

The negotiation and leadership tips we present in this chapter are meant to help software architects form better collaborative relationships with the development team and other stakeholders. These are necessary skills. But don’t just take it from us; take it from [Theodore Roosevelt](#), the 26th president of the United States:

The most important single ingredient in the formula of success is knowing how to get along with people.

—Theodore Roosevelt

Architectural Intersections

So far in this book, we've shown how to identify the critical characteristics an architecture must support, how to select the most appropriate architectural style for those characteristics and for the business problem, how to make effective architecture decisions, and how to lead and guide development teams through implementing the architecture. However, for an architecture to work, it must also be aligned with other facets of the technical and business environment. We call these alignments the *intersections of architecture*.

In this chapter, we discuss several important intersections that arise when creating or validating a software architecture:

Implementation

Is the implementation aligned with architectural concerns surrounding operational characteristics, architectural constraints, and the internal structure of the architecture?

Infrastructure

Do the infrastructure and the way the architecture is deployed align with the operational concerns of the architecture, such as scalability, responsiveness, fault tolerance, and availability?

Data topologies

One widely ignored alignment is that between the intersection of architecture and data topologies and data type. The data topology (monolithic, domain databases, and database per service) must properly align with the architectural style for the system to work.

Engineering practices

Does the way the development team creates, maintains, and tests software match the corresponding architecture? Does the deployment pipeline match the architectural style?

Team topologies

The way teams are organized can significantly impact the architecture, and vice versa. If the team structure is not properly aligned with the architecture, development teams will usually struggle, finding even the simplest of changes challenging.

Systems integration

What other systems or services does the architecture need to communicate with? Not paying attention to this particular intersection can have devastating results in terms of maintenance, reliability, and operational characteristics such as scalability, responsiveness, and availability.

The enterprise

Is the architecture aligned with the frameworks, practices, guiding principles, and standards across the organization and the enterprise?

The business environment

Is the architecture properly aligned with the business environment and the problem domain? Too often architects ignore this important intersection, and as a result the architecture fails to meet the goals or needs of the business.

Generative AI

How does the increasing use of large language models (LLMs) impact the architecture? This intersection is quickly becoming an important one as more companies leverage generative AI within their systems.

The following sections describe these intersections in more detail.

Architecture and Implementation

The First Law of Software Architecture also happens to be software architects' most common response to any question: "It depends." Perhaps the second most common response is, "That's an implementation detail." When a software architecture fails to achieve its goals, this second response is often to blame.

For an architecture to work properly, its *implementation*—that is, its source code—must be aligned with its design to address three things: the architecture’s operational concerns (such as fault tolerance, responsiveness, scalability, and so on), internal structure, and constraints. This section addresses each of the three in turn.

Operational Concerns

Operational concerns are the architectural characteristics we focused on in [Part I](#) of the book, which form the foundation of any software architecture, and which the architecture must support in order to solve the business problem at hand. Architectural characteristics are what drive architectural decisions (as discussed in [Chapter 21](#)).

So what does it mean for the architecture and the implementation to fall out of alignment in how they address the system’s operational concerns? Let’s say you’re an architect working on a new order-entry system that needs to support anywhere from several thousand to half a million concurrent customers. You choose a microservices architecture style, based on the star ratings found in “[Style Characteristics](#)” on page 354 in [Chapter 18](#), which is appropriate based on this system’s high scalability and elasticity needs. However, during *implementation*, the development team observes that, because the services’ bounded contexts are so tightly formed (see “[Bounded Context](#)” on page 331 in [Chapter 18](#)), the Order Placement service cannot access the inventory database directly. Instead, it must synchronously call the Inventory service to get the current inventory for any item a customer is interested in buying. Not only does this synchronous call tightly couple the two services, it greatly slows the system’s responsiveness.

The development team decides to use an *in-memory replicated cache* between these services, as shown in [Figure 26-1](#): data resides in the *internal memory* of each service instance, and is always kept in sync behind the scenes through caching. Caching products for this purpose include [Apache Ignite](#) and [Hazelcast](#). Here, the Inventory service would have a writable in-memory cache containing the item IDs and current inventory counts; each instance of the Order Placement service would have a replicated read-only version of that cache in its internal memory. Using an in-memory replicated cache decouples the services and *significantly* improves responsiveness.

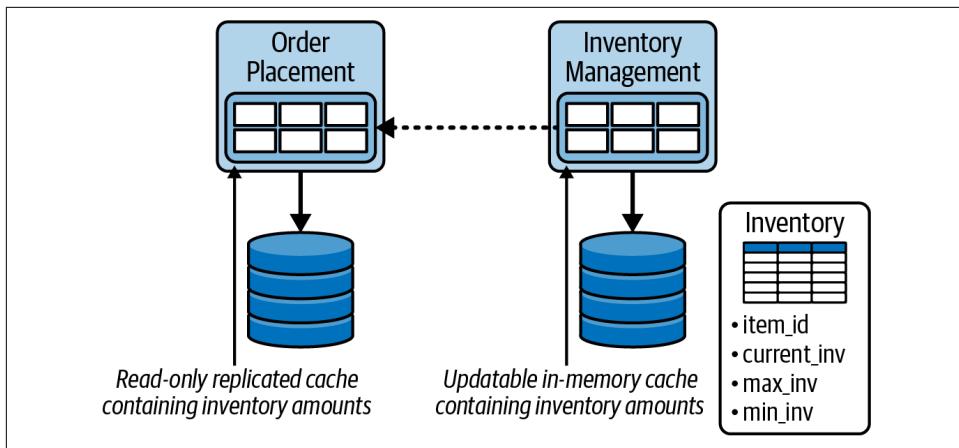


Figure 26-1. The development team decided to use in-memory replicated caching between services, resulting in out-of-memory conditions when scaling the services

After production release, as the number of concurrent users grows, more instances of each service are needed to handle the load. The system crashes when the load reaches about 80,000 concurrent customers because the internal cache's memory requirements are too high, causing out-of-memory conditions in all of the virtual machines.

In this scenario, the architecture and its implementation are misaligned. While the architecture focuses on supporting high levels of *scalability* and *elasticity*, the implementation focuses on *responsiveness* and *service decoupling*. Both teams made good decisions, but in service of different goals.

Structural Integrity

You learned in [Chapter 8](#) that logical components are the building blocks of any system and form its *logical architecture*. They are usually represented through the directory structures in the source-code repository (or namespaces, depending on the programming language). Since the logical architecture describes how the system works and what parts of the system interact with other parts, it's critical that the structure of the source code matches that of the logical architecture.

Without proper guidance, knowledge, and governance, it's easy for developers to ignore the system's logical architecture and start creating directory structures and namespaces as they please, without regard for their impact on the system's integrity. This misalignment results in architectures that are difficult to maintain, test, and deploy, and as a result become less reliable and harder to evolve or adapt to new features, such as the logical architecture illustrated in [Figure 26-2](#).

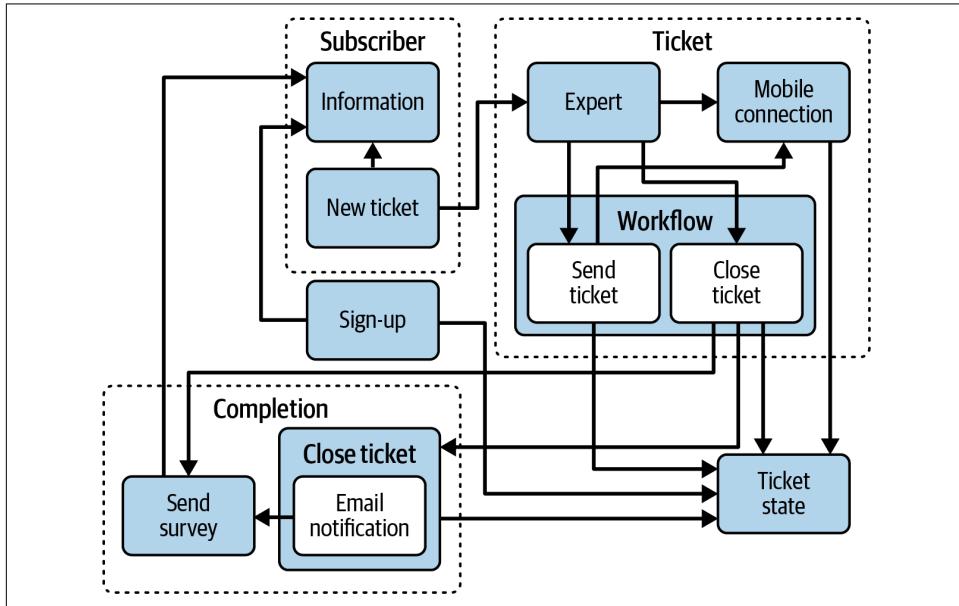


Figure 26-2. An example of an internal logical architecture that lacks governance and alignment

To ensure that the structure of the source code matches the logical architecture, we recommend using automated governance tools, such as [ArchUnit](#) for the Java platform, [ArchUnitNet](#) and [NetArchTest](#) for the .NET platform, [PyTestArch](#) for Python, or [TSArch](#) for TypeScript and JavaScript. These automated tools, coupled with good communication and collaboration between the architect and the development team, create implementations that are properly aligned with the architecture, as shown in [Figure 26-3](#).

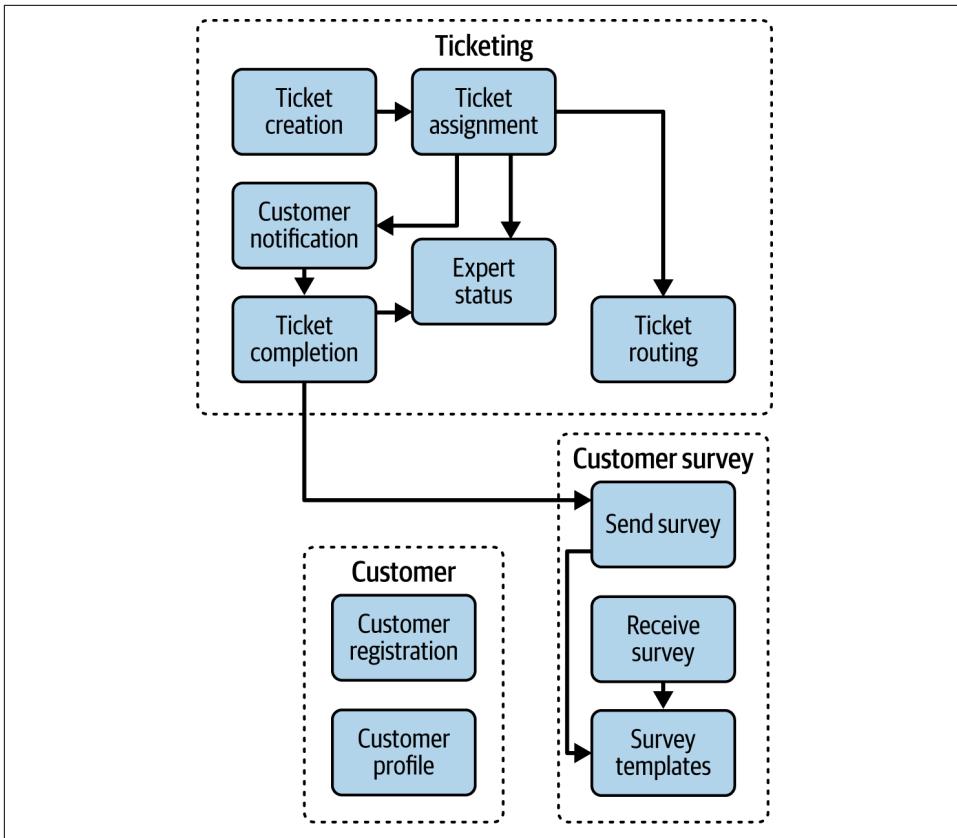


Figure 26-3. An example of an internal logical architecture based on proper governance and alignment

Compare [Figure 26-2](#) with [Figure 26-3](#). Notice how the architecture in [Figure 26-3](#) is much more maintainable, testable, deployable, reliable, adaptable, and extensible than that of the misaligned architecture shown in [Figure 26-2](#). This comparison illustrates the importance of this type of implementation alignment.

Architectural Constraints

A *constraint* is a governing rule or principle describing some sort of restriction within the architecture (such as limiting communications to only REST or using a specific type of database) required to achieve its goals. If the system's implementation doesn't adhere to its constraints, the architecture will fail. Thus, part of a software architect's job is to identify and communicate the *constraints* of an architecture.

To illustrate what we mean by this particular intersection, consider a business trying to release a new system with a very limited budget and a tight deadline. This business

expects lots of structural changes to the database and needs to get those changes done as quickly as possible. In this case, a traditional layered architecture (see [Chapter 10](#)) would be an excellent fit due to its simplicity, cost-effectiveness, and technical partitioning. Because this style separates the layers, database changes can be isolated to only one layer, making them easier and faster.

For the layered architecture to work in this particular business problem, the architect would need to define the following constraints:

- All database logic must reside in the Persistence layer.
- The Presentation layer cannot access the Persistence layer directly, but must go through all of the layers, even for simple queries.

These constraints are necessary to prevent spreading the database logic throughout the entire architecture, and so that changes to the physical database structure (such as dropping a table or changing a column name) won't affect any code outside the Persistence layer.

Now let's say the UI developers decide it's faster to call the database directly and implement the architecture that way instead. In addition, the backend developers realize it would be much easier to maintain and test the code if the Business logic and database logic are together, so they also ignore the constraints and couple these concerns in the business layer of the architecture. This implementation is not aligned with the constraints of the architecture, which means that database changes will impact all of the code in every layer, take too long, and the system will not meet the business goals.

Architectural tools are also useful for governing architectural constraints.

Architecture and Infrastructure

The scope of software architecture has grown over the last couple of decades to encompass more and more responsibility and perspective. Around the mid-2000s, the typical relationship between architecture and operations was contractual and formal, with lots of bureaucracy. Most companies outsourced operations to a third party to avoid the complexity of hosting their own operations, with SLAs for uptime, scale, responsiveness, and other important characteristics. Today, however, architecture styles such as microservices freely leverage characteristics that used to be solely operational. For example, elastic scale was once built into space-based architectures (see [Chapter 16](#)), but today, microservices handles it less painfully through tighter collaboration between architects and DevOps.

History: How Pets.com Gave Us Elastic Scale

People often assume that our current technical capabilities (like elastic scale) are just invented one day by some clever developer. In reality, though, the best ideas are often born of hard lessons. Pets.com is an early example. This ecommerce site appeared around 1998, hoping to become the Amazon.com of pet supplies. Its brilliant marketing department created a compelling mascot: a sock puppet with a microphone that said irreverent things. The mascot became a superstar, appearing in public at parades and national sporting events.

Unfortunately, Pets.com's management apparently spent all the money on the mascot, not on infrastructure. Once orders started pouring in, they weren't prepared. The website was slow, transactions were lost, deliveries delayed...it was pretty much the worst-case scenario. Shortly after a disastrous Christmas rush, Pets.com closed down, selling its only remaining valuable asset—the mascot.

What Pets.com needed was *elastic scale*: the ability to spin up more instances of resources when they became needed. Cloud providers now offer this feature as a commodity, but early ecommerce companies had to manage their own infrastructure, and many fell victim to a previously unheard-of phenomenon: too much success can kill a business. The fall of Pets.com, and other similar horror stories, led architects to pay more attention to such intersections when creating software architectures.

The intersection of architecture and infrastructure is important because it facilitates operational architectural characteristics. Just because an *architecture* can support high scalability doesn't mean it will—if the corresponding *infrastructure* doesn't support it, it won't (as demonstrated by Pets.com). At client sites, we've all too often witnessed architects and developers being blamed for architectural failures that were really caused by a misalignment between architecture and infrastructure.

In most cases, this misalignment occurs because of a lack of communication and collaboration between the architect and those responsible for the infrastructure and operations. Architects often fail to realize infrastructure's influence on characteristics such as scalability, responsiveness, fault tolerance, performance, availability, elasticity, and so on. This misalignment gave rise to the field of DevOps.

For many years, many companies considered operations to be separate from software development, often outsourcing them to a third-party company as a cost-saving measure. In the 1990s and 2000s, many architectures were designed defensively around the assumption that operations would be outsourced and thus outside of architects' control. (For a good example of this, see [Chapter 16](#).) However, in the mid-2000s, companies started experimenting with new forms of architecture that combined many operational concerns. For example, older architecture styles such as orchestration-driven SOA required elaborate tools and frameworks to support

capabilities like scalability and elasticity, which greatly complicated implementation. So architects built architectures that could handle scale, performance, elasticity, and a host of other capabilities *internally*. The side effect was that these architectures were vastly more complex.

The creators of the microservices architecture style realized that operational concerns are better handled by operations. By creating a collaborative relationship between architecture and operations, the architects realized they could simplify their designs and rely on the operations people to handle the things they handle best. They teamed up with operations to create microservices, and to lay the foundations of what would become the DevOps movement. While DevOps has helped, this intersection of architecture and infrastructure is nevertheless still problematic for most companies.

While infrastructure is less of a concern, cloud environments can still get misaligned with an architecture. For example, deploying services across regions or even availability zones can decrease or even cancel out the performance and data integrity benefits of in-memory replicated caches and distributed caches. Similarly, co-locating services, containers, or even Kubernetes Pods on the same virtual machine will significantly increase performance, but it will also adversely impact scalability, fault tolerance, availability, and elasticity.

Aligning architecture with infrastructure involves close communication and collaboration between architects and infrastructure team members, or even adopting DevOps practices, so that all stakeholders understand the critical operational concerns. Only then can architects truly realize the operational benefits of the architecture they selected—the ones that led us to grant those wonderful five-star ratings.

Architecture and Data Topologies

The intersection of architecture and data topologies is often overlooked. Choosing the wrong database type or topology can harm an architecture and negate its best architectural characteristics. For example, monolithic databases, while providing good data consistency and transactional support, can detract from scalability and fault tolerance. Similarly, distributed database topologies, while good at scalability and change control, can decrease a system's data integrity, data consistency, and performance.

The following sections describe the intersection of architecture and data topologies.

Database Topology

A database's *topology*, as we discussed in [Chapter 15](#), refers to how its physical databases are configured within the architecture. Three basic topologies include a monolithic database, distributed domain-based databases, or the distributed database-per-service topology. [Figure 26-4](#) illustrates these basic database topology types.

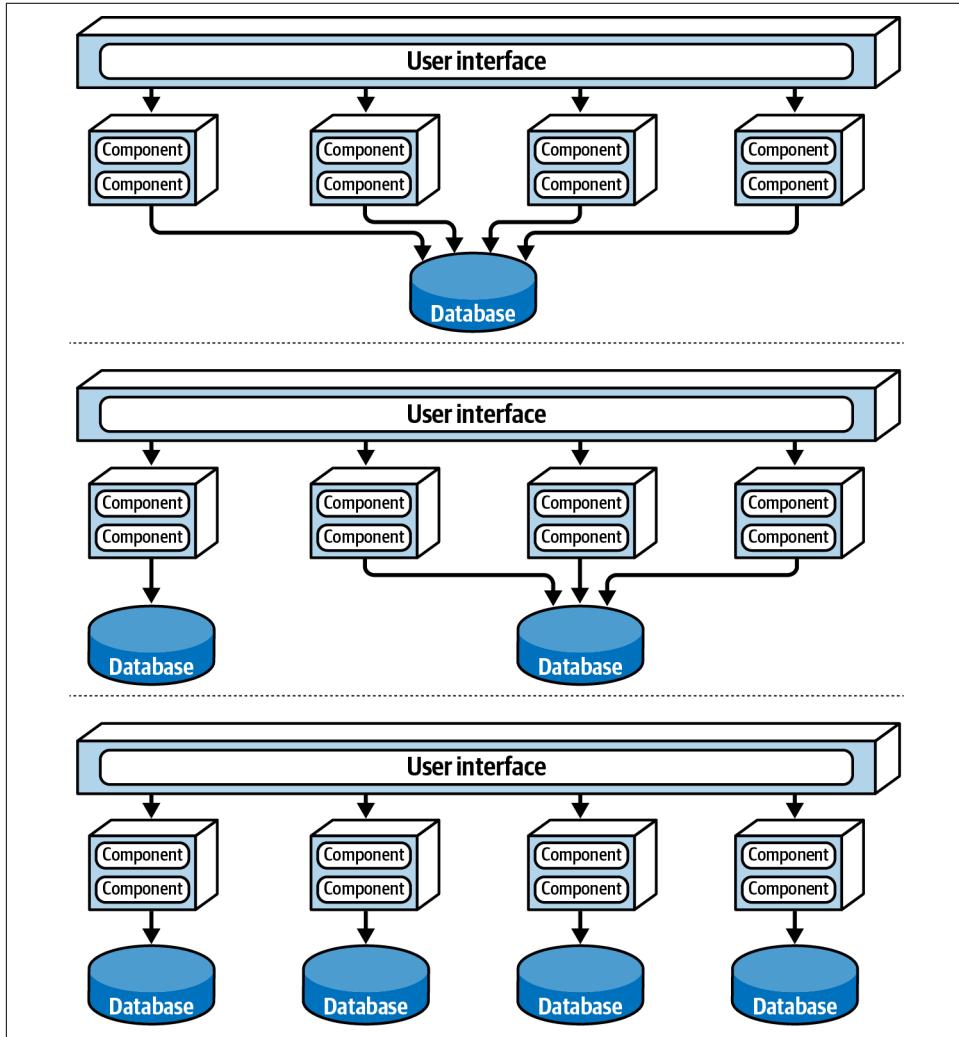


Figure 26-4. Common types of database topologies

The database topology must be aligned with the architecture in order to work correctly. For example, microservices architectures typically use a database-per-service

pattern to maintain a strict bounded context. Without this proper alignment, it would be extremely challenging for architects to control change, and the system's operational characteristics, such as fault tolerance, scalability, elasticity, maintainability, testability, and deployability, would all suffer. That said, some styles, such as the service-based architecture (see [Chapter 14](#)), are more flexible with regard to the physical database topology.

Architectural Characteristics

In [Part II](#) of this book, we showed that every architectural style has its superpowers (rated at 4 to 5 stars) and its weaknesses (1 to 2 stars). So do database types—and it's important to align a system's architectural superpowers with the corresponding superpowers of its database type. In our book *Software Architecture: The Hard Parts*, your authors rated the characteristics of six different database types: relational, key-value, document, columnar, graph, and NoSQL. You might recall that scalability and elasticity are superpowers for microservices, event-driven, and space-based architectures. They are also superpowers for key-value and columnar databases, which means that these database types are good choices to amplify those architectural characteristics.

Data Structure

The structure of data being stored and accessed is also a factor in this intersection. If the *data structure* is relational—that is, built upon a hierarchy of interdependent relationships—then a relational database will align well. However, storing key-value pairs in a relational database is a misalignment that can lead to inefficiencies in both the database and the architecture. Not all data carries with it the same structure, so keep an eye on this. Some data may be relational, other data may be document based (particularly when storing JSON-based event or request payloads), and still other data may be key-value driven. Given the potential diversity of data structures within any given architecture, we recommend leveraging polyglot databases whenever feasible.

Read/Write Priority

If the business problem involves high read or write volumes, that's important information for aligning the database topology with the architecture. For example, if the architecture requires high write volumes as a priority over infrequent reads, then a columnar database would be a good fit. However, if the reverse is true and high read volumes are the priority, then a key-value database, document database, or graph database would be more appropriate. If the system prioritizes reads and writes about equally, then relational and NewSQL databases would be good choices. Misaligning this factor can lead to poorly performing systems.

Architecture and Engineering Practices

In the late 20th century, dozens of software-development methodologies became popular, including Waterfall and many flavors of Agile (such as Scrum, Extreme Programming, Lean, and Crystal). At the time, most architects believed that none of this affected software architecture, treating development as an entirely separate process. However, over the last several years, advances in engineering have thrust process concerns upon software architecture. It's useful to separate software-development *processes* from *engineering practices*. By *processes*, we mean how teams are formed and managed, how meetings are conducted and workflows organized—in short, the mechanics of how people organize and interact. *Engineering practices*, on the other hand, refer to the process-agnostic techniques and tools teams use to develop and release software. For example, Extreme Programming (XP), continuous integration (CI), continuous delivery (CD), and test-driven development (TDD) are all proven engineering *practices* that don't rely on a particular process. Thus, the term *software engineering* encompasses both software development and these practices.

Focusing on engineering practices is important. Software development lacks many of the features of more mature engineering disciplines. For example, civil engineers can predict structural change with much more accuracy than software engineers can predict similar aspects of software structure. This means that the Achilles' heel of software development is estimation: how much time, how many resources, how much money? Part of what contributes to this difficulty is that traditional practices of estimation don't accommodate the exploratory nature of software development and the unknowns that typically arise when developing software.

While process is mostly separate from architecture, iterative processes fit its nature better. Trying to build a modern system like microservices using an antiquated process like Waterfall will create a great deal of friction. One aspect of architecture where Agile methodologies really shine is in migrating from one architectural style to another. Agile methodologies support such changes better than planning-heavy processes because they have tight feedback loops and encourage techniques like the **Strangler Pattern** and **feature toggles**.

Architects often also serve as project technical leaders, which means determining what engineering practices the team uses. Just like carefully considering the problem domain before choosing an architecture, architects must also ensure that their architectural style and engineering practices mesh. For example, the microservices architectural philosophy assumes that teams will automate things like machine provisioning, testing, and deployment. Trying to build a microservices architecture with an antiquated operations group, manual processes, and little testing would likely lead to failure. Just as different problem domains lend themselves toward certain architectural styles, so do different engineering practices.

The evolution of thought continues, from Extreme Programming to continuous delivery and beyond, as advances in engineering practices make new architecture capabilities possible. Neal's book, *Building Evolutionary Architectures*, highlights new ways to think about the intersection of engineering practices and architecture that can improve how we automate architectural governance. The book offers an important new nomenclature and way of thinking about architectural characteristics, and covers techniques for building architectures that change gracefully over time.

In the software development world, nothing remains static. Architects might design a system to meet certain criteria, but to ensure that their designs survive both implementation and the inevitable march of change, what we need is an *evolutionary architecture*.

Building Evolutionary Architectures introduces the concept of using *architectural fitness functions* to protect (and govern) architectural characteristics as change occurs over time. Recall from [Chapter 6](#) that architectural fitness functions are a way to get an objective integrity assessment of some architectural characteristic(s). This assessment may include a variety of mechanisms, such as metrics, unit tests, monitors, and chaos engineering.

To see how fitness functions can be used to help align this intersection, consider the business problem of needing a fast time to market. Time to market translates to *agility*—the system's ability to respond quickly to change. Agility is a composite architectural characteristic consisting of maintainability, testability, and deployability (see [Chapter 6](#)). All three of these architectural characteristics are influenced by engineering practices and procedures, and as such can be measured and tracked through fitness functions. For example, both microservices and service-based architectures support high levels of agility. However, if the engineering practices surrounding these architectural characteristics are not aligned with the architecture, the system won't meet those agility goals and requirements. Fitness functions can be used to help identify a misalignment, prompting the architect to take action to realign the engineering practices with the architecture (or vice versa).

Architecture and Team Topologies

As we've discussed throughout [Part II](#) of the book, team topologies can have a direct impact on software architecture, and vice versa. This alignment is so important that we've included a section on team topologies for each architecture style we introduce in this book.

One of the most basic ways team topologies can align with architecture is by type of partitioning. Like architectures, teams can be domain partitioned or technically partitioned. Domain-partitioned teams are organized by domain area and are typically cross-functional, with specialization throughout the team. For example, a particular

domain-partitioned team might focus on the customer-facing portion of a system, and as such would be responsible for the end-to-end processing of customer-related functionality, from the UI to the database. Technically partitioned teams, in contrast, each focus on one particular technical function of the architecture and are usually organized by technical categories: for example, UI teams, backend-processing teams, shared-services teams, and database teams would align with the layered architecture style very nicely. Alternatively, these teams might be technically partitioned into business-function teams and data-synchronization teams, which would align well with the space-based architectural style.

Understanding how teams are organized is vital for ensuring the success of the system. If the organization's team topology is misaligned with the architecture, the teams will struggle to implement and maintain the architecture, which will be unlikely to meet its business goals.

Architecture and Systems Integration

Systems rarely live in isolation. Most require additional processing and data from other systems—and that brings us to the intersection of architecture and systems integration. When one system needs to communicate with another system to perform additional processing or retrieve data, its architect faces a host of challenges and implications. For example, is the system that is being called available? Does it scale and perform to the same level as the requirements of the calling system?

When architects don't focus enough on systems integration, the systems' static and dynamic coupling often results in architectures that can't scale, aren't responsive, and lack agility. When integrating with other systems, consider which communication protocols to use, what types of contracts to have between systems, whether the systems' architectural characteristics are compatible, and if the integration preserves each system's architectural quantum.

Architecture and the Enterprise

Every enterprise has a set of standards and guiding principles. By *enterprise*, we mean the collection of all systems and products within a company (or a department or division within the company). For example, many companies force certain security standards, practices, or procedures on architectural solutions, regardless of the type of system. Enterprise standards can also involve platforms, technologies, documentation standards, and diagramming standards, to name a few. Be aware of enterprise-level standards and practices, and ensure the architecture is properly aligned with them.

We have experienced many situations where the architect ignored enterprise-level practices, standards, and procedures. The usual result has been that the architectural

solution, however technically effective, is deemed a failed “one-off” solution and scrapped. We can’t stress enough the importance of aligning the architecture with enterprise practices to ensure its success.

Architecture and the Business Environment

The business environment has a significant and *direct* influence on the architecture of its systems (and vice versa), and the business environment never stops changing. Is the company undergoing severe cost-cutting measures to stay afloat, or aggressively expanding? Is the business pivoting and repositioning every quarter to find its niche in a highly volatile and competitive market, or does it find itself in a position of stability? An effective software architect understands the position and direction of the company, and aligns the architectures of critical systems to match the business environment.

We call this alignment *domain-to-architecture isomorphism*. For example, companies undergoing extreme cost-cutting measures would not align well with microservices or space-based architectures, which are very costly to create and maintain. Conversely, businesses that are aggressively expanding through mergers and acquisitions would not be served well by monolithic architecture styles that lack the ability to evolve and adapt.

One issue architects typically face with this intersection is business change, particularly *unknown change*. Former US Secretary of Defense Donald Rumsfeld once famously said that:

There are known knowns; there are things we know we know. We also know there are known unknowns; that is to say we know there are some things we do not know. But there are also unknown unknowns—the ones we don’t know we don’t know.

Many products and systems start with a list of *known unknowns*: things developers must learn about the domain and technology they know will change. However, these same systems also fall victim to *unknown unknowns*: things no one knew were going to crop up, yet have appeared unexpectedly. “Unknown unknowns” are the nemeses of software systems. This is why all “Big Design Up Front” software efforts suffer: architects cannot design for unknown unknowns. To quote Mark:

All architectures become iterative because of *unknown unknowns*. Agile just recognizes this and does it sooner.

Planning for change in software architecture is *hard*. Evolutionary architecture practices help address an ever-changing business landscape, as does iterative architecture. Embracing architectural characteristics such as *portability*, *scalability*, *evolvability*, and *adaptability* also helps make a software architecture more flexible and adaptable to change.

Barry O'Reilly, an experienced software architect specializing in complexity theory and software design, came up with a new way of thinking about constant business change, called *residuality theory*. In his book *Residues: Time, Change, and Uncertainty in Software Architecture* (Leanpub, 2024), O'Reilly describes techniques for treating business change as *stressors*, and the corresponding architectural changes as *residues*. His theory is that as the architect responds to change by applying more and more residues to the architecture, these residues will eventually start addressing *unknown* changes the architect cannot possibly predict, creating an architecture that has reached a critical state within complexity theory. It's an interesting theory, one we are watching closely.

Architecture and Generative AI

As we write the second edition of this book in early 2025, *generative artificial intelligence* (Gen AI) and large language models (LLMs) have infiltrated the world of software development and software design. Many companies are incorporating them into systems to perform tasks that previously were only performed manually by humans. Not surprisingly, Gen AI intersects with software architecture too: architects are incorporating LLMs into software architectures, and some are even using Gen AI tools to assist them in thinking through hard problems.

Incorporating Generative AI into Architecture

One approach we recommend for incorporating Gen AI into an architecture is to leverage *abstraction* and *modularity*. It's important to be able to replace one LLM with another quickly, and to allow for guardrails (rails) and evaluate results (evals) from various LLMs.

For example, suppose a job search company wants to leverage Gen AI to anonymize résumés, with the goal of reducing bias and placing focus on job seekers' skills rather than their demographics or other such factors. This task is normally done by humans, but could easily be done by an LLM. But are the LLM's results accurate? Does it remove too much information from the résumé? Does it keep too much demographic information in place? For this sort of system, it's vital to be able to gather samples and metrics and compare LLM engines. Tools such as [Langfuse](#) help create this kind of observability within the architecture.

Generative AI as an Architect Assistant

Given a proper prompt, an LLM (such as [Copilot](#)) can produce source code that saves developers a lot of time and effort. They're great for solving very specific, deterministic problems, such as "write source code in the C# programming language that generates a unique four-digit PIN number that has no repeating digits." But can

LLM technology assist software architects with typical tasks? Here are some general examples of architecture-related prompts:

- Risk assessment: “Are there risk areas within this architecture?”
- Risk mitigation: “How should I address this risk?”
- Antipatterns: “Are there any common antipatterns in this architecture?”
- Decisions: “Should I use orchestration or choreography for this workflow?”

As of the time of the writing of this second edition (early 2025), we haven’t had a tremendous amount of success with this endeavor. Asking an LLM whether microservices or space-based architecture would be most appropriate for a given situation rarely (if ever) yields the right answer. Why? Because, as we’ve demonstrated in this book, *everything in software architecture is a trade-off*. LLMs are great for understanding *knowledge*, but to this day, they still lack the *wisdom* necessary to make appropriate decisions. That wisdom includes so much context that it’s much faster for the architect to solve a business problem by themselves than to teach an LLM all about the problem and its extended environment and context. The fact that we’ve included eight other intersections to be concerned about should be evidence enough that this is a daunting task.

That said, we’ve seen some tools with promise. For instance, [Thoughtworks Haiven](#) can interpret an architecture diagram and fully describe a software architecture, saving the work of having to export a diagram into a machine-readable format such as XML and use it to prompt the LLM. Once they’ve imported that information, users can ask Haiven simple questions about the architecture, such as whether it can identify any bottlenecks or issues. Other efforts have included using an LLM to translate a PlantUML diagram or a pseudolanguage description of an architecture into executable ArchUnit code to govern the structure of a system. A lot of activity is happening in this area, so expect rapid change in the coming years in how Gen AI can assist architects.

Summary

Software architecture is a holistic activity that involves many facets of an organization. Effective software architects realize that creating and maintaining an architecture is much more than just selecting a particular architectural style and moving forward with implementation. It’s also about making sure that the architecture is aligned with other aspects of the environment, and about using the communication and collaboration skills described in [Part III](#) to make that alignment happen.

The Laws of Software Architecture, Revisited

Way back in [Chapter 1](#), we described the three laws of software architecture:

- Everything in software architecture is a trade-off.
- *Why* is more important than *how*.
- Most architecture decisions aren't binary but rather exist on a spectrum between extremes.

Just about every example in this book has illustrated these laws, which points to their origin story. When we wrote the first edition, we hoped to find numerous things that seemed universally true about software architecture and codify them as *laws*. To our surprise, we ended up identifying just two laws in the first edition, then uncovered one more while writing the second. True to our original intent, these three laws seem pretty universal and inform many important perspectives for working software architects.

In this brief chapter, we'll revisit these laws in light of the examples we've shown and point out some nuances about trade-off analysis.

First Law: Everything in Software Architecture Is a Trade-Off

Our first law is one of the defining characteristics of software architecture—everything is a trade-off. Many people think that the job of a software architect is to find silver bullet solutions to sticky problems and become a hero, but that rarely happens.

(Architects rarely get credit for good decisions, but always get blamed for bad ones.) No, the real job of software architecture is analyzing trade-offs.

We believe, for a couple of reasons, that every architect should hone their reputation as an objective arbiter of trade-offs, rather than evangelizing for a particular approach.

First, evangelism in architecture is dangerous in the long term, because yesterday's best practice tends to become tomorrow's antipattern. Architects make trade-off decisions based on current factors and situations as best they can, with incomplete knowledge. However, even if a decision is sound when it's made, the software development ecosystem exists in a constant state of evolution and churn, which means that the circumstances slowly change and are likely to eventually weaken (or invalidate) the decision. If the architect has invested social capital in evangelizing for that solution, their reputation may suffer when that decision must change later. Always stay clear-eyed and objective about technology choices to avoid attaching your credibility to a decision that didn't age well.

Second, decision makers in organizations aren't really looking for enthusiastic advocacy so much as sober objectivity. An architect who develops a reputation as the go-to person for an objective trade-off analysis becomes a valuable asset to their organization. When the decision is critical, decision makers want someone whose judgment they can trust; that should be you.

Let's look at some examples of trade-off analysis for a couple of software architecture decisions, and then discuss a common gap in understanding.

Shared Library Versus Shared Service

A common conundrum for architects involves shared behavior in distributed architectures, such as microservices or EDA: should we use a shared library that is compiled into each service at build time, or use a shared service that other services call at runtime, as illustrated in [Figure 27-1](#)?

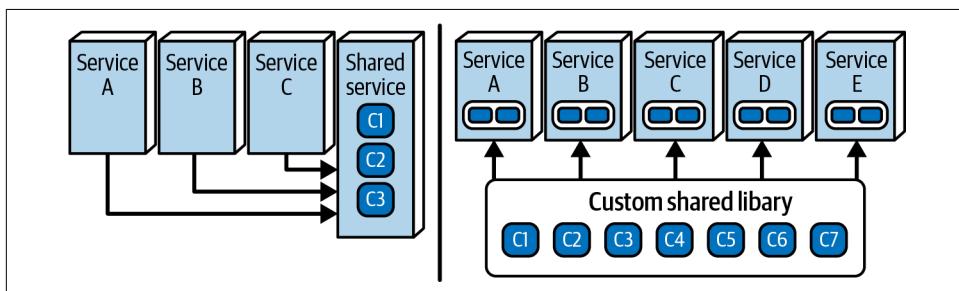


Figure 27-1. Should the solution use a shared library or shared service for common functionality?

In [Figure 27-1](#), the service on the left encapsulates the shared behavior, and other services call to access it. On the right, we have a shared library, which is compiled into each service before deployment. Which is better?

By now, readers know the answer to all such questions: “It depends!” But you need to answer the inevitable follow-up question: “Depends on what?” Like many nontrivial decisions in software architecture, the answer to this question isn’t immediately obvious, so it’s time for you, as the architect, to undertake trade-off analysis.

First, you must determine all the contextualized trade-offs that make a difference for this solution. This list will be highly specific to both the organization and the solution, so you’ll need to rely on your knowledge of the organization, technology landscape, team capabilities, budgets, and everything else that informs your trade-off options. For this solution, we’ve created the following list of pertinent factors:

Heterogeneous code

If solutions are written in multiple platforms, then the service will be easier to work with because, regardless of the technology stack, callers access the service via the network, making the implementation platform irrelevant. For the library, the team will need a version of the code for each technology stack and will have to keep the versions in sync, greatly adding to the project’s overall complexity.

High code volatility

Recall that code volatility measures how fast the code changes (commonly known as *churn*). In a service, callers access new functionality as soon as they deploy the updated service, so if you need to change the library, you’ll have to recompile and deploy each service to take advantage of the new features.

Ability to version changes

Versioning is much easier in the library than in the service. When the library needs to update, the team can resolve the version differences at compile time and build just what they need. For the service, version information must be determined at runtime, which complicates the interaction with the service.

Overall change risk

Change risk is better for the library. Once you change the library code and successfully compile it into the service, you can have high confidence that it will work. The service, on the other hand, may change without any compile-time verification, which raises the potential likelihood of a runtime fault at invocation time.

Performance

The library will clearly have better performance, as the calls to shared functionality are in-process calls, compared to the network calls required for the service. These calls will be much slower than in-process calls because of network latency and other factors.

Fault tolerance

As we discussed in [Chapter 9](#), runtime access to services always suffers potential network issues, including the service in question here. The library provides better fault tolerance: once you compile, test, and deploy the service, you can have high confidence that it is stable.

Scalability

Just like performance, calls between services suffer from latency, which diminishes scalability. Thus, the library will offer better scalability, as access to the shared behavior is a very efficient in-process call.

Once you've determined your list of important factors, you can create a matrix to compare how well each does for each criterion, as shown in [Table 27-1](#).

Table 27-1. Trade-offs between shared service and shared library

| Factor | Shared library | Shared service |
|----------------------------|----------------|----------------|
| Heterogeneous code | - | + |
| High code volatility | - | + |
| Ability to version changes | + | - |
| Overall change risk | + | - |
| Performance | + | - |
| Fault tolerance | + | - |
| Scalability | + | - |

The accumulation of positives for the shared library makes it the winner...at least, for these factors and in this context. It may or may not be the solution to the problem at hand—you may need to apply additional weighting (see “[Second Corollary: You Can’t Do It Just Once](#)” on page 494 for more details)—but now you and your team have a good idea of the forces at play.

Synchronous Versus Asynchronous Messaging

Consider the following trade-off analysis: your team is building a distributed architecture to send trade information to both the `Notification` and `Analytics` services, and you’re trying to decide whether to use a *queue* or *topic* to implement this behavior.

The first option is a queue, or a point-to-point communication protocol. As you learned in [Chapter 2](#), the publisher knows who is receiving the message. To reach multiple consumers, the publisher needs to send a message to one queue for each consumer. If the `Trading` service wants to use queues to tell the `Analytics` service and the `Reporting` service about trades, the implementation will appear as in [Figure 27-2](#).

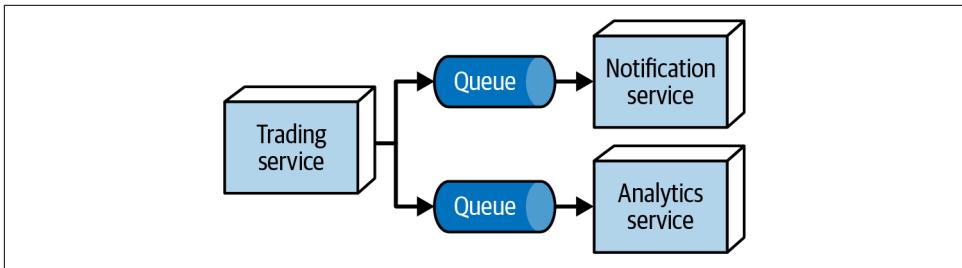


Figure 27-2. Using a queue to pass trade information to Notification and Analytics

In [Figure 27-2](#), the Trading service issues a message to each consumer via a queue. Each consumer has its own queue, and the sender posts a message to each.

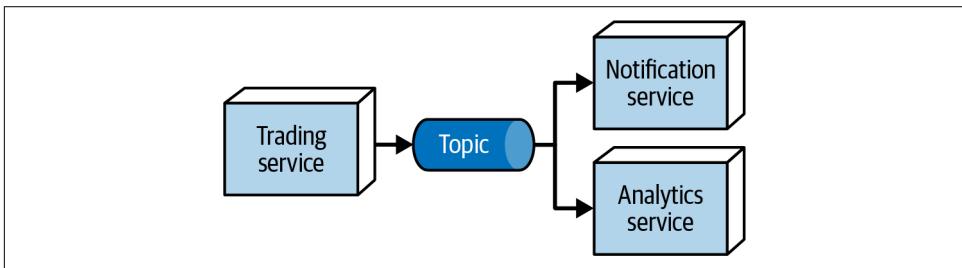


Figure 27-3. Using a topic to pass trade information to Notification and Analytics

The alternative topology uses a *topic* instead, which implements a broadcast rather than point-to-point communication. In [Figure 27-3](#), the Trading service posts a single message to a topic. Each consumer subscribes to the topic, and they all receive a notification when a message appears. In this case, the publisher of the message doesn't know (or care) who the consumers are—in fact, the team can add new consumers anytime without changing the other consumers or the producer.

Both options are viable, so how should you decide? By performing a trade-off analysis, of course.

With queues, every service that the Trading service needs to notify will need a separate queue. This is nice if the Notification service and the Analytics service need different information, since you can send different messages to each queue. The Trading service is aware of every system to which it communicates, which makes it harder for another (potentially rogue) service to “listen in.” That’s especially beneficial if security is high on the priority list. Because each queue is independent, you can monitor them separately and even scale them independently if needed. The Trading service is tightly coupled to its consumers—it knows exactly how many there are. However, if you need to send messages to the Compliance service, you’ll have to rework the Trading service to start sending messages to a third queue.

Table 27-2 summarizes the trade-offs of using queues for this project.

Table 27-2. Trade-offs when using a queue

| Advantage | Disadvantage |
|---|---|
| Supports heterogeneous messages for different consumers | Higher degree of coupling |
| Allows independent monitoring of queue depth | Trading service must connect to multiple queues |
| More secure | Requires additional infrastructure |
| Less extensible (must add queues for more consumers) | Good support for extensibility and evolvability |

Let's do the same exercise for topics. One advantage is clear: topics are extremely extensible. Any new consumer interested in, for example, compliance, can subscribe to that topic without touching any of the existing behavior. However, this advantage has downsides: each consumer must consume the same message from the topic, which can lead to *stamp coupling* (shown in [Figure 9-9](#)). And, because every consumer can read the entire message, there are security concerns: *should* everyone be able to read the entire message?

With this perspective in hand, you can make your trade-off table, shown in [Table 27-3](#).

Table 27-3. Trade-offs when using a topic

| Advantage | Disadvantage |
|--|---|
| Low coupling | Homogeneous message for each consumer |
| Trading service generates just one message | Can't monitor or scale individual consumers |
| More extensible/evolvable | Less secure |
| | Less scalability options |

Now that you've done the trade-off analysis, you need to return to the organizational goals for this solution and see which option is the better fit. If security is more important, you should probably select queues. If the organization is growing rapidly and has other services interested in trades, then extensibility might be a priority, which would lead you to use topics.

First Corollary: Missing Trade-Offs

Our first law has two corollaries. The first is:

If you think you've discovered something that *isn't* a trade-off, more likely you just haven't *identified* the trade-off...yet.

—Corollary 1

The essence of a software architecture decision is trade-off analysis, but what happens when you encounter a decision that seems to not have any trade-offs? Our advice: keep looking!

Consider code reuse. Surely this is a purely beneficial practice, right? The more code the organization can reuse, the less code it must write, saving time and duplication.

Two factors dictate how effective code reuse will be. Architects often discover the first but miss the second. The first factor is *abstraction*: if you can abstract this code and use it from multiple call points, it's a good candidate for reuse. The second factor is *low volatility*. When a team reuses a module of code that's always changing, this creates churn in the entire system. Every time the shared code changes, all callers of that code must coordinate around that change. Even if it isn't a breaking change, the team must still verify that the change hasn't broken anything. When architectures reuse code inappropriately, teams end up chasing breaking changes all over the architecture. This was one of the key lessons architects as a field learned from the orchestration-driven SOA architecture style (covered in [Chapter 17](#)), one of the underlying philosophies of which was to try to reuse as much code as possible. From a practical standpoint, teams working in those architectures were swimming through quicksand: every change had the potential to send unpredictable side effects rippling out through the system.

That's the hidden trade-off: reusing code effectively requires good abstraction *and* low volatility. This is why the most successful reuse targets in architecture are "plumbing": technology frameworks, libraries, platforms, and so on. The portion of most applications that changes the fastest is the domain (the motivation for writing the software in the first place), so domain concepts are terrible candidates for reuse. (Notice that this underlies the DDD principle of bounded context—no bounded contexts can reuse any of the implementation details of another bounded context.)

Why We Can't Have Nice Things—Trade-Offs!

As professional consultants, our clients often make requests like the following: "We like the idea of microservices and distributed architectures that feature high degrees of decoupling because they allow high degrees of agility and fast deployment. However, we also want a high degree of institutional reuse so that teams aren't constantly rewriting code."

We have to be the bearers of bad news here: you cannot have both these things, because the way a system implements reuse is via coupling. No organization can have both decoupling *and* a high degree of reuse. The two things are fundamentally incompatible. This is a prime example of how organizations can misunderstand key trade-offs.

Second Corollary: You Can't Do It Just Once

It would be nice if an architect could just perform one trade-off analysis—just think *really hard* and decide once and for all to use choreography for all workflows. But there are two problems with trade-off analysis. First, there are often dozens or even hundreds of variables (technical and otherwise) that contribute to the decision: complexity, team experience, budget, team topology, schedule pressure...the list is endless. Subtle differences in those variables can push a particular analysis in one direction or another, making this an ongoing exercise. It's dangerous for architects to make sweeping, semipermanent decisions based on assumptions that might not be valid for future applications of this solution.

Think of this corollary as job security for architects. We have to keep doing trade-off analysis over and over, even for seemingly similar situations. This reinforces the idea that the real job is trade-off analysis, not making permanent, perfect decisions.

Second Law: Why Is More Important Than How

Our second law emphasizes the importance of *why* over *how*. As experienced architects, we can look at an existing system and tell someone *how* it works. However, there will be some decisions where it isn't clear *why* the previous architect chose this option over another, because they didn't record all of the decision criteria with their ultimate solution.

This is why we emphasize the importance of using both architecture diagrams *and* ADRs (covered in [Chapter 21](#)). Every trade-off analysis generates a tremendous amount of context that doesn't appear in the solution. It's critical to document that analysis (along with the known compromises and limitations of the solution) to prevent future architects (who might be you) from having to redo the analysis just to understand *why*.

Out of Context Antipattern

One antipattern that's common in trade-off analysis is the *Out of Context* antipattern. This antipattern occurs when the architect understands the trade-offs but not how to *weight* all of them based on the current context.

Consider the trade-off analysis we did in "[Shared Library Versus Shared Service](#)" on [page 488](#). Taken objectively, the shared library appears to be the preferred solution, since it achieved more positive than negative assessments. However, this trade-off analysis has a potential flaw: do all of its criteria have equal weight?

Imagine that a team has code in several platforms. They aren't overly concerned with performance or scale, but want a clean way to manage shared behavior. In this case, the first two trade-off criteria carry a much higher priority, which should lead them

to choose the shared service. As a bonus, the team has already figured out what issues they will have to mitigate.

Architects rely on experience to build trade-off criteria, but must also weigh those criteria to find the correct fit for the solution. Generic trade-off analysis isn't very useful—it only becomes valuable when applied in a specific context.

The Spectrum Between Extremes

In our first edition, we identified only two laws. However, we have gradually realized that a third law exists:

Most architecture decisions aren't binary but rather exist on a spectrum between extremes.

—Third Law of Software Architecture

It would be nice to live in a world with nice, clean, binary decisions—but software architecture doesn't exist in that world. People often observe how difficult it is to come up with comprehensive definitions of important concepts in software architecture: architecture versus design, orchestration versus choreography, topics versus queues, and so on. The underlying reason is that the decision criteria aren't binary; they lie along a rather messy spectrum.

In “[Architecture Versus Design](#)” on page 17, we discussed the spectrum between architecture and design and how to determine where a particular decision lies within that spectrum. In fact, this law provides a useful way to think about decisions that are closer to the software architecture side of the spectrum than the design side:

A software architecture decision is one where each of the options has significant trade-offs.

If everything in software architecture is a trade-off, then an architectural decision must involve trade-offs for each option.

As an architect, don't try to reduce every decision to a binary—few of those exist in our world. That's one of the reasons that every answer in software architecture is “it depends”—it depends on where on the spectrum of possible solutions the criterion in question falls.

As architects, we make our decisions in a swamp of uncertainty. Accommodating this is annoying, but necessary. Not only do we sometimes have to base important decisions on incomplete information, but often the decision wouldn't be clear-cut, even if we had the full scoop, because it resides somewhere on a spectrum between two extremes.

Welcome to software architecture!

Parting Words of Advice

How do we get great designers? Great designers design, of course.

—Fred Brooks

So how are we supposed to get great architects, if they only get the chance to architect fewer than a half-dozen times in their career?

—Ted Neward

Practice is the proven way to build skills and become better at anything in life, including architecture. We encourage new and existing architects to keep honing their skills in the craft of designing architecture as well as widening their individual technology breadth. To that end, we've created some **architecture katas** modeled after the examples in this book, which you can find on the companion website. We encourage you to use them to practice and build your skills in architecture.

People who use our katas often ask: is there an answer guide somewhere? Unfortunately, no. To quote Neal:

There are not right or wrong answers in architecture—only *trade-offs*.

When your authors started using architecture katas during our live training classes, we initially kept the drawings the students produced, with the goal of creating an answer repository. We quickly gave up, though, because we realized that these were incomplete artifacts. The teams had captured *how* they implemented their solutions with drawings of topologies. The *why* was much more interesting—but while they explained the trade-offs they considered in class, they didn't have the time to create architecture decision records. Keeping just the *how* meant we had only half of the story.

So, our parting words of advice: always learn, always practice, and *go do some architecture!*

Discussion Questions

Chapter 1: Introduction

1. What are the four dimensions that define software architecture?
2. What is the difference between an architecture decision and a design principle?
3. List the eight core expectations of a software architect.
4. What is the First Law of Software Architecture?

Chapter 2: Architectural Thinking

1. Name three criteria for determining whether a decision is more about architecture or more about design.
2. List the three levels of knowledge in the knowledge triangle and provide an example of each.
3. Why is it more important for an architect to focus on technical breadth rather than technical depth?
4. What are some of the ways of maintaining your technical depth and remaining hands-on as an architect?

Chapter 3: Modularity

1. Describe the difference between modularity and granularity, and provide an example of each.
2. What is the difference between coupling and cohesion?
3. What does the term *connascence* mean?

4. What is the difference between static and dynamic connascence?
5. What is the strongest form of connascence?
6. What is the weakest form of connascence?
7. Which is preferable within a code base: static or dynamic connascence?

Chapter 4: Architecture Characteristics Defined

1. What three criteria must an attribute meet to be considered an architecture characteristic?
2. What is the difference between an implicit characteristic and an explicit one? Provide an example of each.
3. Provide an example of an operational characteristic.
4. Provide an example of a structural characteristic.
5. Provide an example of a cross-cutting characteristic.
6. Why is it impossible to create an industry-wide standard list of architecture characteristics?

Chapter 5: Identifying Architectural Characteristics

1. Give a reason why it is a good practice to limit the number of characteristics (“-ilities”) an architecture should support.
2. True or false? Most architecture characteristics come from business requirements and user stories.
3. If a business stakeholder states that time to market (that is, getting new features and bug fixes pushed out to users as fast as possible) is the most important business concern, which architecture characteristics would the architecture need to support?
4. What is the difference between scalability and elasticity?
5. What is the definition of a *composite* architectural characteristic? Provide one example.

Chapter 6: Measuring and Governing Architecture Characteristics

1. Why is cyclomatic complexity such an important metric to analyze for architecture?
2. What are architecture fitness functions? How can they be used to analyze an architecture?
3. Provide an example of an architecture fitness function to measure the scalability of an architecture.
4. What is the most important criterion for an architecture characteristic to allow architects and developers to create fitness functions?

Chapter 7: The Scope of Architectural Characteristics

1. What is an architectural quantum, and why is it important to architecture?
2. Assume a system consisting of a single user interface with four independently deployed services, each containing its own separate database. Would this system have a single quantum or four quanta? Why?
3. What is the difference between *static* and *dynamic* coupling? Provide an example of each.
4. Why does *synchronous* communication have a bigger potential impact on operational architectural characteristics than *asynchronous* ones?

Chapter 8: Component-Based Thinking

1. We define the term *component* as a building block of an application—something the application does. A component usually consists of a group of classes or source files. How do components typically manifest within an application or service?
2. What is the difference between technical partitioning and domain partitioning? Provide an example of each.
3. What is the advantage of domain partitioning?
4. Under what circumstances would technical partitioning be a better choice than domain partitioning?
5. What is the Entity Trap? Why is it not a good approach for component identification?

- When might you choose the Workflow approach over the Actor/Action approach when identifying core components?

Chapter 9: Foundations

- List the eight fallacies of distributed computing.
- Name three challenges that distributed architectures have that monolithic architectures don't.
- What is stamp coupling? What are some ways of addressing stamp coupling?
- What is the difference between technical versus domain partitioning?
- List three characteristics that distinguish an architectural style from a pattern.

Chapter 10: Layered Architecture Style

- What is the difference between an open layer and a closed layer?
- Describe the concept of *layers of isolation* and its benefits.
- What is the Architecture Sinkhole antipattern?
- What are some of the main architecture characteristics that would drive you to use a layered architecture?
- Why isn't testability well supported in the layered architecture style?
- Why isn't agility well supported in the layered architecture style?

Chapter 11: Modular Monolith Architecture Style

- How does the modular monolith differ from the n-tiered layered architecture?
- Describe the difference between the peer-to-peer and mediator approaches when communicating between modules.
- What are the three common risks associated with the modular monolith architectural style?
- Name three primary strengths of the modular monolith and when you should consider using it.
- Given its modularity, why aren't operational characteristics such as scalability and fault tolerance good in the modular monolith?

Chapter 12: Pipeline Architecture Style

1. Can pipes be bidirectional in a pipeline architecture?
2. Name the four types of filters and their purpose.
3. Can a filter send data out through multiple pipes?
4. Explain why the pipeline architecture is well suited for cloud-based environments.
5. Is the pipeline architecture style technically partitioned or domain partitioned?
6. In what way does the pipeline architecture support modularity?

Chapter 13: Microkernel Architecture Style

1. What is another name for the microkernel architecture style?
2. Under what situations is it OK for plug-in components to be dependent on other plug-in components?
3. What are some of the tools and frameworks that can be used to manage plug-ins?
4. What would you do if you had a third-party plug-in that didn't conform to the standard plug-in contract in the core system?
5. Provide two examples of the microkernel architecture style.
6. What characteristics of the core system determine its degree of "microkern-ality"?
7. Why is the microkernel architecture always a single architecture quantum?
8. What is *domain/architecture isomorphism*?

Chapter 14: Service-Based Architecture Style

1. Why are services in service-based architecture called *domain services*?
2. Name two common risks associated with service-based architecture.
3. Which database topologies (monolithic, domain, and dedicated) can you use with service-based architecture?
4. What techniques can you use to manage database changes within a service-based architecture?
5. Do domain services require a container (such as Docker) to run?
6. Which architecture characteristics does the service-based architecture style support well?
7. Why isn't elasticity typically well supported in a service-based architecture?

8. How can you increase the number of architecture quanta in a service-based architecture?

Chapter 15: Event-Driven Architecture Style

1. Name four differences between events and messages.
2. What's the difference between an initiating event and a derived event?
3. What is a *poison event*?
4. Can an event processor trigger multiple derived events? If so, why would you want to do this?
5. Why would you trigger an event from an event processor that no other event processor responds to?
6. What are some of the negative trade-offs in using asynchronous processing?
7. Describe the Swarm of Gnats antipattern and explain why you should avoid it.
8. Give two primary reasons why event-driven architecture is highly responsive and has great performance characteristics.
9. What are some of the techniques for preventing data loss when sending and receiving messages from a queue?

Chapter 16: Space-Based Architecture Style

1. Where does space-based architecture get its name from?
2. What is a primary aspect of space-based architecture that differentiates it from other architecture styles?
3. Name the four components that make up the virtualized middleware within a space-based architecture.
4. What is the role of a data writer in space-based architecture?
5. Under what conditions would a service need to access data in a database through the data reader?
6. Does a small cache size increase or decrease the chances for a data collision?
7. What is the difference between a replicated cache and a distributed cache? Which one is typically used in space-based architecture?
8. List the three most strongly supported architecture characteristics in space-based architecture.
9. Why does testability rate so low for space-based architecture?

Chapter 17: Orchestration-Driven Service-Oriented Architecture

1. What was the main driving force behind service-oriented architecture?
2. What are the four primary service types within a service-oriented architecture?
3. List some of the factors that led to the downfall of service-oriented architecture.
4. Is SOA technically partitioned or domain partitioned?
5. How is domain reuse addressed in SOA? How is operational reuse addressed?
6. What is the primary use for this architecture style in modern systems?

Chapter 18: Microservices Architecture

1. Describe the *bounded context* concept in microservices. Why is it so critical for microservices architectures?
2. Why is it so important to isolate data in a microservices architecture?
3. Describe what is meant by *protocol-aware heterogeneous interoperability* and how microservices supports it.
4. What is the difference between orchestration and choreography in microservices? Provide examples where each of these communication styles would be an appropriate choice.
5. Why does microservices use the database-per-service topology? Can it use a monolithic or domain database topology?
6. What are two of the biggest risks in the microservices architecture?
7. Why are agility, testability, and deployability so well supported in microservices?
8. What are three reasons that performance is usually an issue in microservices?
9. Describe a topology where a microservices ecosystem might have only a single quantum.

Chapter 19: Choosing the Appropriate Architecture Style

1. In what way does a system's data architecture (the structure of the logical and physical data models) influence your choice of architecture style?
2. Delineate the steps an architect uses to determine a system's architecture style, data partitioning, and communication styles.
3. What factor leads an architect toward a distributed architecture?

4. What two input analyses does an architecture need to perform to choose an appropriate architectural style?

Chapter 20: Architectural Patterns

1. Name two patterns that implement the architectural concern of separation of operational and domain concerns.
2. What component does an orchestrated workflow have that is absent from choreographed ones?
3. What are some advantages of using a single broker in an event-driven architecture? What are some disadvantages?
4. What two data operations does CQRS break apart?

Chapter 21: Architectural Decisions

1. What is the Covering your Assets antipattern?
2. What are some techniques for avoiding the Email-Driven Architecture antipattern?
3. What are the five factors Michael Nygard defines for identifying something as architecturally significant?
4. What are the five basic sections of an architecture decision record?
5. In which section of an ADR do you typically add the justification for an architecture decision?
6. Assuming you don't need a separate Alternatives section, in which section of an ADR would you list the alternatives to your proposed solution?
7. What are three basic scenarios in which you would mark the status of an ADR as *Proposed*?

Chapter 22: Analyzing Architecture Risk

1. What are the two dimensions of the risk assessment matrix?
2. Describe the three main activities in risk storming.
3. Why does risk storming need to be a collaborative exercise?
4. Why does the identification activity within risk storming need to be an individual activity and not a collaborative one?
5. What would you do if three participants identified risk as high (6) for a particular area of the architecture, but another participant identified it as only medium (3)?

6. What risk rating (1–9) would you assign to unproven or unknown technologies?

Chapter 23: Diagramming Architecture

1. What is *irrational artifact attachment*, and why is it significant with respect to documenting and diagramming architecture?
2. What are the 4 Cs of the C4 modeling technique?
3. When diagramming architecture, what do dotted lines between components mean?
4. Why is it always important to include a title and key in an architecture diagram?

Chapter 24: Making Teams Effective

1. What are the three types of architect personalities? What type of boundary does each personality create?
2. What five factors should you consider in determining your level of involvement with the team?
3. What are three warning signs that your team is getting too big?
4. List three basic checklists that would be good for a development team to use.

Chapter 25: Negotiation and Leadership Skills

1. Why is negotiation such an important skill for an architect?
2. Name some negotiation techniques for a scenario when a business stakeholder insists on five nines of availability, but only three nines are really needed.
3. What can you derive from a business stakeholder telling you, “I needed it yesterday”?
4. Why is the *demonstration defeats discussion* technique so effective?
5. What is the divide-and-conquer rule? How can it be applied when negotiating architecture characteristics with a business stakeholder? Provide an example.
6. List the 4 Cs of architecture.
7. Explain why it is important for an architect to be both pragmatic and visionary.
8. What are some techniques for managing and reducing the number of meetings you are invited to?

Chapter 26: Architectural Intersections

1. What techniques can you use to ensure structural alignment with the architecture during implementation?
2. Explain why aligning architecture and infrastructure is so important, and provide an example of this alignment.
3. Give an example of when a team's topology might not be aligned with the architecture.
4. Why should an architect in charge of a particular system be concerned about the intersection of architecture and systems integration? Provide an example of where architecture might get misaligned with system integration.
5. What is *domain/architecture isomorphism*, and why is it so important when considering the intersection of architecture and the business?
6. Provide an example of the intersection of architecture and the enterprise.

Chapter 27: The Laws of Software Architecture, Revisited

1. List three generic advantages of a shared library over a shared service. Do these advantages mean an architect should *always* choose a shared library when this trade-off appears?
2. What is the first corollary of the First Law of Software Architecture?
3. Why does the second corollary to the First Law of Software Architecture insist that trade-off analysis must occur over and over?
4. Why are so many software architecture decisions best understood on a spectrum rather than as convenient binaries?

Index

A

abstractionness (modularity metric), 45
accidental complexity, 48, 459
Accidental SOA antipattern, 324
ACID (atomicity, consistency, isolation, durability) database transactions, 212, 224
active/active state, 304
ADRs (see Architectural Decision Records)
Advanced Message Queueing Protocol (AMQP), 253
afferent coupling, 44, 121
Agile methodologies, 480
agility, 481
Ambulance pattern, 420
AMQP (Advanced Message Queueing Protocol), 253
Analysis Paralysis antipattern, 388
anemic events, 245
antipatterns
 Accidental SOA, 324
 Analysis Paralysis, 388
 architectural decision-related, 387-390
 Architecture Sinkhole, 158
 Big Ball of Mud, 133-134
 Bottleneck Trap, 34
 Covering Your Assets, 387-389
 defined, 387
 DLL Hell, 169
 Email-Driven Architecture, 389
 Entity Trap, 115-116, 333
 Frozen Caveman, 24
 generic architecture, 77
 Grains of Sand, 351
 Groundhog Day, 389

Irrational Artifact Attachment, 425
Out of Context, 494
Swarm of Gnats, 246-249
Apache Kafka, 191-192
application services, 318
application silos, 12
ArchiMate technical diagramming standard, 428
architects (see software architect)
architectural characteristics
 analyzing when creating logical components, 120
 defined, 55-65
 cloud characteristics, 60
 cross-cutting architectural characteristics, 61-64
 non-functional requirements, as term, 56
 operational architectural characteristics, 59
 partial listing of characteristics, 59-64
 self-assessment questions, 498
 structural architectural characteristics, 60
 system design, 56-58
 defining, 3
 fitness functions, 87-93
 identifying, 67-79
 architecture katas origins, 70
 composite architectural characteristics, 68-69
 design versus architecture and trade-offs, 76

- extracting architectural characteristics, 69
extracting architectural characteristics from domain concerns, 67
kata: Silicon Sandwiches, 71-77
limiting/prioritizing architectural characteristics, 77-79
self-assessment questions, 498
working with katas, 70
- intersection of architecture with data topologies, 479
- ISO definitions, 62-64
- measuring, 81-86
- cyclomatic complexity, 83-85
 - multiple definitions of performance, 83
 - operational measures, 82-82
 - process measures, 86
 - self-assessment questions, 499
 - structural measures, 83-85
- overspecificity in Vasa case, 78
- scope of, 95-106
- architectural quanta and granularity, 96-99
 - impact of scoping, 100-105
 - kata: Going Green, 103
 - scoping and architectural style, 102-103
 - scoping and the cloud, 105
 - self-assessment questions, 499
 - synchronous communication, 99
- trade-offs and least worst architecture, 64-65
- worksheet for determining characteristics critical to success, 78-79
- architectural constraints, 474
- Architectural Decision Records (ADRs), 391-402
- basic structure, 392-398
 - Compliance section, 396-397
 - Consequences section, 396
 - Context section, 394
 - Decision section, 395
 - Notes section, 397
 - status section, 393-394
 - title section, 393
 - documentation provided by, 401
 - example: Going, Going, Gone, 398-399
 - storing, 400-401
 - using for standards, 402
 - using with existing systems, 402
- architectural decisions, 387-403
- antipatterns, 387-390
- Covering Your Assets, 387-389
 - Email-Driven Architecture antipattern, 389
 - Groundhog Day antipattern, 389
- Architectural Decision Records, 391-402
- as documentation, 400-401
 - basic structure, 392-398
 - example: Going, Going, Gone, 398-399
 - storing ADRs, 400-401
 - using ADRs for standards, 402
 - using with existing systems, 402
- architecturally significant decisions, 390-391
- defining, 5
- leveraging generative AI and LLMs in, 402
- self-assessment questions, 504
- architectural extensibility, 31, 230
- architectural fitness function, 87, 481
- (see also fitness functions)
- architectural intersections (see intersections of architecture)
- architectural katas (see katas)
- architectural patterns, 371-384
- architectural styles versus, 131-132, 371
 - communication, 375-378
 - CQRS, 378
 - infrastructure, 379-384
 - Broker-Domain pattern, 380-382
 - Domain-Broker pattern, 383-384
 - orchestration versus choreography, 375-378
 - reuse, 372-375
- separating domain and operational coupling, 372-375
- architectural quanta, 96-99
- domain-driven design's bounded context, 97
 - scope and, 100-105
- architectural risk, analyzing, 405-422
- risk assessments, 406-409
 - risk matrix, 405-406
 - risk storming, 410-416
 - phase 1: identification, 411
 - phase 2: consensus, 412-414
 - phase 3: mitigation, 415
 - use case, 416-422
 - user-story risk analysis, 416
 - self-assessment questions, 504

architectural styles, 2, 131-152
(see also specific styles, e.g.: event-driven architecture style)
architecture partitioning, 137-142
architecture patterns and (see architectural patterns)
characteristics described by, 131-132
choosing the appropriate style, 359-370
 decision criteria, 361-364
 distributed case study: Going, Going, Gone, 367-370
 factors in shifting fashions in architecture, 359-360
 monolith case study: Silicon Sandwiches, 364-367
 self-assessment questions, 503
defining, 2
fundamental patterns, 133-136
kata: Silicon Sandwiches, 140-142
self-assessment questions, 500
architectural thinking, 17-36
 analyzing trade-offs, 30-33
 architecture versus design, 17-20
 level of effort, 19
 significance of trade-offs, 19
 strategic versus tactical decisions, 18
 balancing architecture and hands-on coding, 33-35
 self-assessment questions, 497
 technical breadth, 20-29
 20-minute rule, 24-25
 developing a personal radar for technological change, 25-29
 understanding business drivers, 33
architecturally significant decisions, 390-391
architecture isomorphism, 362
architecture risk-assessment matrix, 405-406
Architecture Sinkhole antipattern, 158
architecture vitality, 9
architecture, design versus, 17-20, 76
 level of effort, 19
 significance of trade-offs, 19
 strategic versus tactical decisions, 18
ArchUnit, 90
asynchronous communication, 364
auto acknowledge mode, 255
availability, as implicit characteristic, 75
AWS Step Functions, 184

B

Backends for Frontends (BFF) pattern, 367
best practices, architecture patterns versus, 371
Big Ball of Mud antipattern, 133-134
Bottleneck Trap antipattern, 34
bounded context, 331
 domain-driven design and, 97
 microservices and, 329
BPEL (Business Process Execution Language), 259
BPM (Business Process Management), 260
brittle architecture, 99
business drivers, as part of architectural thinking, 33
business environment, intersection of architecture with, 483-484
Business Process Execution Language (BPEL), 259
Business Process Management (BPM), 260
business services, 317
business stakeholders, negotiating with, 453-456
business-domain expertise, 11

C

C4 (context, container, component, class) technical diagramming standard, 427
caching
 near-cache considerations, 295
 replicated and distributed caching, 291-294
CC (see Cyclomatic Complexity)
chaos engineering, 92
checklists
 fitness functions as, 93
 leveraging, 445-449
 developer code-completion checklist, 447-448
 Hawthorne effect, 447
 software release checklist, 449
 unit and functional testing checklist, 448
Chidamber and Kemerer Object-Oriented Metrics Suite, 43
choreography
 in microservices, 341-344
 orchestration versus, 375-378
CID (correlation ID), 257
client acknowledge mode, 255
client/server architecture, 135-136
 browser and web server, 135

desktop and database server, 135
single-page JavaScript applications, 135
three-tier architecture, 136
closed layers, 155
cloud environments
 architectural characteristics, 60
 event-driven architecture, 275
 intersection of architecture and infrastructure, 477
 layered architecture, 159
 microkernel architecture, 203
 microservices architecture, 351
 modular monolith architecture, 171
 orchestration-driven service-oriented architecture, 323
 pipeline architecture, 184-185
 scoping and, 105
 service-based architecture, 219
 space-based architecture, 302
code reviews, 35
code smell, 85
coding, balancing architecture work with, 33-35
cohesion, defined, 97
communication
 diagramming architecture (see diagramming architecture)
 in microservices, 339-340
 synchronous, 99
compatibility, ISO definition, 62
compensating transaction framework, 347
compensating updates, 150
compile-based plug-in components, 198
complexity, cyclomatic, 83-85
compliance with design principles/architecture decisions, 10
complicated-subsystem teams, defined, 151
component coupling, 121-125
 Law of Demeter, 123-125
 static coupling, 121
 temporal coupling, 122
component identification, 113-116
 Actor/Action approach, 114-115
 Entity Trap antipattern, 115-116
 Workflow approach, 113
component-based thinking, 107-128
 case study: Going, Going, Gone (discovering components), 125-128
 component coupling, 121-125
 Law of Demeter, 123-125
 static coupling, 121
 temporal coupling, 122
 analyzing architectural characteristics, 120
 analyzing roles/responsibilities, 118-120
 assigning user stories to components, 117-118
 identifying core components, 113-116
 restructuring components, 120
 defining logical components, 107-109
 logical versus physical architecture, 110-111
 self-assessment questions, 499
components, modules and, 54
composite architectural characteristics, 68-69
concert ticketing system use case, 313
connascence, 49-53
 dynamic, 50-51
 properties, 51-53
 static, 49-50
connascence of values, 345
constraints, defined, 474
construction techniques, defined, 391
consumer filters, 183
Conway's Law, 139
correlation ID (CID), 257
coupling, 121-125
 architectural quanta and, 98-99
 component coupling, 121-125
 Law of Demeter, 123-125
 static coupling, 121
 temporal coupling, 122
 orthogonal, 374
 separating domain coupling and operational coupling, 372-375
 Hexagonal architecture, 373-374
 Service Mesh, 374-375
 static, 121
 temporal, 122
Covering Your Assets antipattern, 387-389
CQRS (Command–Query–Responsibility–Segregation) pattern, 378
cross-cutting architectural characteristics, 61-64
customizability, 76
Cyclic Dependencies antipattern, 88-89
Cyclomatic Complexity (CC), 83-85
 defined, 84

- formula for calculating, 84
limitations of metric, 48
threshold value for, 85
- ## D
- data abstraction layer, 301
data collisions, 304-307
data isolation, 333
data latency, 355
data pumps, 297-298
data topologies, intersection of architecture with, 477-479
architectural characteristics, 479
data structure, 479
database topologies, 478-479
read/write priority, 479
data-based event payload, 240-243
Database-per-Service pattern, 273-274, 349-351
DDD (see domain-driven design)
decisions, strategic versus tactical, 18
declarative transactions, 323
dedicated database topology, 273-274
dependencies, defined, 391
dependency-free plug-ins, 204
derived event, 228
design principles
 architect's role in defining, 8
 architect's role in ensuring compliance with, 10
 communicating to team, 450-452
design, architecture versus, 17-20, 76
developer code-completion checklist, 447-448
developers
 flow state, 467
 negotiating with, 457-459
 technical depth versus technical breadth, 23
development teams, making effective, 433-452
appropriate level of involvement for software engineer, 439-442
architect personality types, 436-439
 armchair architect, 437-438
 control-freak architect, 437
 effective architect, 438
collaboration, 433-435
constraints and boundaries, 435-436
impact of business justifications, 450
leading by example, 462-465
leveraging checklists, 445-449
developer code-completion checklist, 447-448
Hawthorne effect, 447
software release checklist, 449
unit and functional testing checklist, 448
negotiating with teams, 457-459
providing guidance, 450-452
self-assessment questions, 505
team topologies (see team topologies)
warning signs of too-large team, 443-445
 diffusion of responsibility, 444
 pluralistic ignorance, 444
 process loss, 443
DevOps, 476
diagramming architecture, 423-431
 guidelines, 428-430
 self-assessment questions, 505
 standards, 426-428
 ArchiMate, 428
 C4, 427
 UML, 427
 tools, 424-426
 using layers semantically, 426
diffusion of responsibility, 444
direction of risk, 408-409
Distance from the Main Sequence metric
 basics, 46-49
 verification using fitness functions, 90-93
distributed architectures
 Going, Going, Gone case study, 367-370
 microservices as, 330
 monolithic architectures versus, 143-150
 fallacy #1: reliable network, 144
 fallacy #2: zero latency, 144
 fallacy #3: infinite bandwidth, 145
 fallacy #4: secure network, 146
 fallacy #5: unchanging topology, 147
 fallacy #6: single administrator, 148
 fallacy #7: zero transport cost, 148
 fallacy #8: homogeneous network, 149
 fallacy #9: easy versioning, 149
 fallacy #10: reliable compensating updates, 150
 fallacy #11: observability as optional, 150
 scoping and, 103
distributed caching, 293-294
DLL Hell antipattern, 169
domain coupling, separating operational coupling from, 372-375

domain database topology, 271-272
domain partitioning, 138-140, 142
 microkernel architecture and, 205
 modular monolithic topology, 165
 service-based architecture and, 221
 teams and, 481
domain services, 209
domain-driven design (DDD)
 bounded context and, 97
 domain partitioning in, 138
 microservices and, 329
 modular monolithic architecture and, 165
domain-to-architecture isomorphism, 483
domains, in modular monolithic architecture,
 166
durable subscriber, 253
dynamic connascence, 50-51, 345
dynamic coupling, 99
 communication as, 99
 in microservices, 353
dynamic quantum entanglement, 237

E

EasyMeals use case, 177-180
EDA (see event-driven architecture (EDA) style)
EDI (electronic data interchange) tools, 191
effective architect personality type, 438
efferent coupling, 44, 122
Elastic Leadership, 439
elastic scale, 476
elasticity, 72-73
electronic data interchange (EDI) tools, 191
Email-Driven Architecture antipattern, 389
enabling teams, defined, 151
enforced heterogeneity, 340
engineering practices
 intersection of architecture with, 480-481
 software engineering and, 480
ensuring compliance, 10
enterprise service bus (ESB), 319, 324-325
enterprise services, 318
enterprise, intersection of architecture with,
 482
Entity Trap antipattern, 115-116, 333
essential complexity, 48
ETL (extract, transform, and load) tools, 191
event mediator, 259
event payloads, 240-246

anemic events, 245
data-based, 240-243
key-based, 243-244
trade-off summary, 244
event-based application, 227
event-driven architecture (EDA) style, 227-281
 choosing between request-based and event-based models, 280
 cloud considerations, 275
 common risks, 275
 data topologies, 268-274
 dedicated data topology, 273-274
 domain database topology, 271-272
 monolithic database topology, 269-270
 examples and use cases, 280-281
 governance, 276
 style characteristics, 277-280
 style specifics, 232-268
 asynchronous capabilities, 235-238
 broadcast capabilities, 239
 derived events, 233-234
 error handling, 249-252
 event payload, 240-246
 events versus messages, 232-233
 mediated event-driven architecture,
 258-268
 preventing data loss, 253-255
 request-reply processing, 256-258
 Swarm of Gnats antipattern, 246-249
 triggering extensible events, 235
 team topology considerations, 276-277
 topology, 228-231
eventual consistency, 224
explicit architectural characteristics, 72-75
extensible derived event, 235
extract, transform, and load (ETL) tools, 191

F

facilitation, as requirement for software architect, 11
fallacies of distributed computing
 fallacy #1: reliable network, 144
 fallacy #2: zero latency, 144
 fallacy #3: infinite bandwidth, 145
 fallacy #4: secure network, 146
 fallacy #5: unchanging topology, 147
 fallacy #6: single administrator, 148
 fallacy #7: zero transport cost, 148
 fallacy #8: homogeneous network, 149

- fallacy #9: easy versioning, 149
fallacy #10: reliable compensating updates, 150
fallacy #11: observability as optional, 150
fan-in coupling (see afferent coupling)
fan-out coupling (see efferent coupling)
federated broker, 229
filters, in pipeline architectural style, 182
fire-and-forget communication, 228, 235
First Law of Software Architecture, 6, 487-494
 architecture partitioning and, 137
 example: shared library versus shared service, 488-490
 example: synchronous versus asynchronous messaging, 490-492
 first corollary: missing trade-offs, 492-493
 second corollary: trade-off analysis as primary job for architects, 494
fitness functions
 architectural characteristics, 87-93
 cyclic dependencies, 88-89
 Distance from the Main Sequence, 90-93
flow state, 467
4 Cs of architecture leadership (communication, collaboration, clear, concise), 459-461
Front Controller pattern, 343
frontends, 338-339
Frozen Caveman antipattern, 24
full-time equivalency (FTE) rate, 394
function point analysis, 56
functional cohesion, 96, 97
functional suitability, 63
- G**
- generative AI
 inability to evaluate trade-offs, 1
 intersection of architecture with, 484-485
 Gen AI as architect assistant, 484-485
 incorporating Gen AI into architecture, 484
 leveraging in architectural decisions, 402
generic architecture antipattern, 77
Going Green (kata)
 architecture quantum as scope for architectural characteristics, 103-105
 microkernel architecture for, 194-196
 service-based architecture, 221-226
Going, Going, Gone (GGG)
 ADR example, 398-399
- discovering components, 125-128
distributed architectures, 367-370
EDA style, 280-281
governance, 86
 architectural characteristics, 86
 EDA style, 276
 layered architectural style, 159
 microkernel architectural style, 204
 microservices architectural style, 352-353
 modular monolith architectural style, 172-174
orchestration-driven service-oriented architecture, 324-325
pipeline architectural style, 186-188
service-based architectural style, 219
space-based architecture style, 307-310
Grains of Sand antipattern, 351
granularity
 architectural quanta and, 96-99
 modularity versus, 38
Groundhog Day antipattern, 389
guaranteed delivery, 254
- H**
- handshakes, 464
happy path, 194
Hawthorne effect, 447
Hexagonal architecture, 373-374
hugging, 465
- I**
- implementation coupling, 98
implementation, intersection of architecture with, 470-475
 architectural constraints, 474
 operational concerns, 471-472
 structural integrity, 472-474
implicit architectural characteristics
 importance of, 58
 modularity and, 37
 Silicon Sandwiches (kata example), 75-77
in-memory replicated cache, 471
incoming coupling (see afferent coupling)
infrastructure services, 319
infrastructure, intersection of architecture with, 475-477
initiating event, 228
instability (modularity metric), 46
insurance claims processing use case, 207

- interfaces, defined, 391
- International Organization for Standards (ISO)
definitions for architectural characteristics, 62-64
- interpersonal skills, 11
- intersections of architecture, 469-485
with data topologies, 477-479
 architectural characteristics, 479
 data structure, 479
 database topologies, 478-479
 read/write priority, 479
with engineering practices, 480-481
with generative AI, 484-485
 Gen AI as architect assistant, 484-485
 incorporating Gen AI into architecture, 484
 with implementation, 470-475
 architectural constraints, 474
 operational concerns, 471-472
 structural integrity, 472-474
with infrastructure, 475-477
- with systems integration, 482
with team topologies, 481
with the business environment, 483-484
with the enterprise, 482
- Inverse Conway Maneuver, 139
- Irrational Artifact Attachment antipattern, 425
- ISO (International Organization for Standards)
definitions for architectural characteristics, 62-64
- J**
- Jakarta Messaging API, 253
- Java, 136
- Java 1.0, 40
- JDepend, 89
- K**
- katas, 70
(see also Silicon Sandwiches; Going Green)
origins, 70
working with, 70
- key-based event payloads, 243-244
- knowledge pyramid, 20-23
- known unknowns, 483
- L**
- Lack of Cohesion in Methods (LCOM), 43-44
- large language models (LLMs), 484-485
- last participant support (LPS), 255
- last responsible moment, 387
- Law of Demeter (Principle of Least Knowledge), 123-125
- Law of Diminishing Returns, 446
- laws of software architecture, 487-495
defining, 6-8
First Law: everything in software architecture is a trade-off, 6, 487-494
example: shared library versus shared service, 488-490
example: synchronous versus asynchronous messaging, 490-492
first corollary: missing trade-offs, 492-493
second corollary: trade-off analysis as primary job for architects, 494
- Second Law: why is more important than how, 7, 494
Out of Context antipattern, 494
- Third Law: decisions exist on a spectrum, 7, 495
- layered architectural style, 153-164
cloud considerations, 159
common risks, 159
data topologies, 159
examples and use cases, 163
governance, 159
self-assessment questions, 500
style characteristics, 161-163
style specifics, 155-159
 adding layers, 157-159
 layers of isolation, 155-156
team topology considerations, 160
topology, 153-155
when not to use, 163
when to use, 162
- layered stack, 450
- layers of isolation, 156
- layers, using semantically when diagramming, 426
- LCOM (Lack of Cohesion in Methods), 43-44
- leadership, 459-465
(see also team topologies)
4 Cs of architecture and, 459-461
as requirement for software architect, 11, 459-465
balancing pragmatism and vision, 461-462

integrating with the development team, 465-468
leading teams by example, 462-465
leadership skills
self-assessment questions, 505
leaf nodes, 109
least worst architecture, 64-65
LLMs (large language models), 402, 484-485
logical architecture, creating
analyzing architectural characteristics, 120
analyzing roles/responsibilities, 118-120
assigning user stories to components, 117-118
creating, 112-121
identifying core components
Actor/Action approach, 114-115
Entity Trap antipattern, 115-116
Workflow approach, 113
physical architecture versus, 110-111
restructuring components, 120
logical components, 4
(see also component-based thinking)
defining, 107-109
restructuring, 120
LPS (last participant support), 255

M

maintainability, ISO definition, 63
mean time to recover (MTTR), 159, 190
mediated event-driven architecture, 258-268
medical monitoring system use case, 356-357
meetings, when integrating with development team, 465-468
message bus, 319
microfrontends, 338
microkernel architectural style, 193-208
architecture characteristics ratings, 205-207
cloud considerations, 203
common risks, 203
contracts, 201
core system, 194-196
data topologies, 202
examples and use cases, 207-208
plug-in components, 198-200
registry, 201
self-assessment questions, 501
Silicon Sandwiches case study, 366-367
spectrum of microkernelity, 200
team topology considerations, 204

topology, 193
microservices architectural style, 132, 329-357
cloud considerations, 351
common risks, 351-352
data topologies, 348-351
enforced heterogeneity, 340
examples and use cases, 356-357
governance, 352-353
intersection of architecture and infrastructure, 477
self-assessment questions, 503
style characteristics, 354-356
style specifics, 331-348
API layer, 334
bounded context, 331
choreography and orchestration, 341-344
communication, 339-340
data isolation, 333
frontends, 338-339
granularity, 332
operational reuse, 334-337
transactions and sagas, 345-348
team topology considerations, 353-354
topology, 330-331
modular monolith architectural style, 165-180
cloud considerations, 171
common risks, 171
data topologies, 170
domain partitioning in, 138
examples and use cases, 177-180
governance, 172-174
Silicon Sandwiches case study, 365-366
style characteristics, 175-177
when not to use, 177
when to use, 176
style specifics, 166-170
modular structure, 168
module communication, 168-170
monolithic structure, 167
team topology considerations, 174
topology, 165
modular programming languages, 39
modularity, 37-54
defining, 38-40
granularity versus, 38
Java package namespace, 40
metrics, 40-53
abstractness, 45

cohesion, 41-44
connascence, 49-53
coupling, 44
Distance from the Main Sequence, 46-49
instability, 46
limitations of, 48
modular reuse before classes, 39
modules to components, 54
self-assessment questions, 497
module communication, 168-170
mediator approach, 169
peer-to-peer, 169
monolithic architectures
case study: Silicon Sandwiches, 364-367
microkernel architecture, 366-367
modular monolith, 365-366
distributed architectures versus, 143-150
scoping and, 102
monolithic database topology, 269-270
monolithic frontend, 338
MTTR (mean time to recover), 159, 190

N

n-tiered architectural style (see layered architectural style)
namespaces, 39
near-cache model, 295
negotiation skills, 453-459
negotiating with business stakeholders, 453-456
negotiating with developers, 457-459
negotiating with other architects, 456-457
self-assessment questions, 505
Netflix, Chaos Monkey, 92
non-functional characteristics, 391
non-functional requirements, longevity of term, 56
nondeterministic workflows, 279

O

object-oriented programming languages, 39
observability, 150
online auction system use case, 313
(see also Going, Going, Gone (GGG))
open layers, 156
operational architectural characteristics, 59
operational concerns, 471-472
operational coupling, separating domain coupling from, 372-375

operational measures, of architectural characteristics, 82-82
orchestration
choreography versus, 375-378
in microservices, 341-344
orchestration engine, 319
orchestration-driven service-oriented architecture, 315-328
cloud considerations, 323
common risks, 323
data topologies, 322
declarative transactions, 323
examples and use cases, 327-328
governance, 324-325
self-assessment questions, 503
style characteristics, 325-327
style specifics, 316-322
application services, 318
business services, 317
enterprise services, 318
infrastructure services, 319
message flow, 319
orchestration engine and message bus, 319
reuse and coupling, 320-322
taxonomy, 317-320
team topology considerations, 325
topology, 315
orthogonal coupling, 374
outgoing coupling (see efferent coupling)

P

package namespace (Java 1.0), 40
packaging, modularity and, 38
patient monitoring system use case, 356-357
patterns in architecture, architectural styles versus, 131-132
performance efficiency, ISO definition, 62
performance, multiple definitions of, 83
persisted queues, 304
personal boundaries, 465
Pets.com, 476
physical architecture
defined, 110
logical architecture versus, 110-111
pipeline architectural style, 181-192
cloud considerations, 184-185
common risks, 185
data topologies, 184

examples and use cases, 191-192
filters, 182
governance, 186-188
pipes, 183
self-assessment questions, 501
style characteristics, 189-191
when not to use, 190
when to use, 190
team topology considerations, 188
topology, 181
platform teams, defined, 151
plug-in components
dependencies, 204
microkernel architectural style, 198-200
plug-in registry, 201
pluralistic ignorance, 444
POCs (proofs-of-concept), 34
point-to-point communication, 198
poison event, 231
politics, understanding/navigating, 12
portability, ISO definition, 63
pragmatism, defined, 461
Principle of Least Knowledge (Law of Demeter), 123-125
process loss, 443
process, defined, 480
producer filters, 182
product-based applications, microkernel architecture for, 193
productivity flow, 467
proofs-of-concept (POCs), 34
protocol-aware heterogeneous interoperability, 339
pseudosynchronous communications (request-reply messaging), 256-258

R

read/write priority, 479
reliability, ISO definition, 62
replicated caching, 291
replication latency (RL), 304
representational consistency, 423
Request for Comments (RFC), 394
request-based application, 227
request-reply messaging, 256-258
residuality theory, 484
risk (see architecture risk)
risk assessments, 406-409
risk storming, 410-416

phase 1: identification, 411
phase 2: consensus, 412-414
phase 3: mitigation, 415
use case, 416-422
availability risk, 418-419
elasticity risk, 420
security risk, 421-422
user-story risk analysis, 416
risk-assessment matrix, 405-406
RL (replication latency), 304
rules engine, 207
runtime plug-in components, 198

S

Saga pattern, 345
scalability, 72
scoping (see architectural characteristics, scope of)
security
as implicit characteristic, 75
ISO definition, 62
self-assessment questions, 497
semantic coupling, 98
semantic decoupling, 239
separation of concerns, 155
separation of technical concerns, 139
serialization, three-tier architecture and, 136
serverless architectural style, 351
service discovery, 337
Service Mesh pattern
microservices and, 336
separating domain coupling and operational coupling, 374-375
service plane, 336
service-based architectural style, 209-226
cloud considerations, 219
common risks, 219
data topologies, 215-218
examples and use cases, 224-226
governance, 219
self-assessment questions, 501
style characteristics, 221-224
style specifics, 211-215
API Gateway options, 215
service design and granularity, 213
user interface options, 213-214
team topology considerations, 220
topology, 209-210
service-level agreements (SLAs), 419

- service-level objectives (SLOs), 419
service-oriented architecture (SOA), 316
(see also orchestration-driven service-oriented architecture)
services, origins and evolution of term, 317
Sidecar pattern, 334-337, 374
Silicon Sandwiches (kata example), 71-77
 explicit characteristics, 72-75
 implicit characteristics, 75-77
monolithic architecture case study, 364-367
 microkernel architecture, 366-367
 modular monolith, 365-366
partitioning, 140-142
 domain partitioning, 142
 technical partitioning, 142
single monolithic database topology, 269-270
Single-Broker pattern, 383
SLAs (service-level agreements), 419
SLOs (service-level objectives), 419
SOA (service-oriented architecture), 316
(see also orchestration-driven service-oriented architecture)
soft skills
 architectural decisions (see architectural decisions)
 diagramming architecture (see diagramming architecture)
 leadership (see leadership)
 making teams effective (see development teams, making effective)
 negotiation (see negotiation skills)
software architects
 collaboration with development team, 433-435
 integrating with the development team, 465-468
 leadership by (see leadership)
 negotiations between, 456-457
 personality types, 436-439
 armchair architect, 437-438
 control-freak architect, 437
 effective architect, 438
 role in making teams effective (see development teams, making effective)
 role/expectations of, 8-12
 continually analyzing architecture, 9
 ensuring compliance with decisions/design principles, 10
 interpersonal skills, 11
 keeping current with latest technology/trends, 9
 knowing the business domain, 11
 making architecture decisions, 8
 political skills, 12
 understanding of diverse technologies, 10
 technical breadth as requirement for, 20-29
 technical depth versus technical breadth, 23
 tips and techniques for deepening technical abilities, 34-35
software architecture (basics)
 defining, 2-6
 four dimensions of, 2-6
 laws of software architecture, 6-8
 role/expectations of software architect, 8-12
 self-assessment questions, 497
software developers (see developers; development teams, making effective)
software release checklist, 449
solutions, architecture patterns versus, 371
source code (see implementation)
space-based architectural style, 283-314
 cloud considerations, 302
 common risks, 303-307
 data collisions, 304-307
 data synchronization and consistency, 304
 frequent reads from database, 303
 high data volumes, 304
 data grid, 288-295
 near-cache considerations, 295
 replicated and distributed caching, 291-294
 data pumps, 297-298
 data readers, 300-302
 data topologies, 302
 data writers, 298-299
 deployment manager, 297
 examples and use cases, 313-314
 concert ticketing system, 313
 online auction system, 313
 governance, 307-310
 messaging grid, 287
 processing grid, 296
 processing unit, 286
 self-assessment questions, 502
 style characteristics, 311-312
 team topology considerations, 310-311

topology, 284-286
virtualized middleware, 287
stability, as implicit characteristic, 75
stakeholders, negotiating with, 453-456
stale expertise, 23
stamp coupling, 146, 242-243, 276
static connascence, 49-50
static coupling, 98, 121, 352
strategic decisions, tactical decisions versus, 18
stream-aligned teams, defined, 151
structural architectural characteristics, 60
structural integrity, 472-474
structured programming languages, 39
Swarm of Gnats antipattern, 246-249
synchronous communication, 99
 as factor in choosing architectural style, 364
 in microservices, 339
synchronous send, 254
system design, architectural characteristics and, 56-58
systems integration, intersection of architecture with, 482

T

TAB (Technology Advisory Board), 26
tactical decisions, strategic decisions versus, 18
tags, in pipeline architecture, 186-188
tax preparation software use case
 microkernel architecture and, 207
 registry for, 201
team topologies, 151
 EDA style and, 276-277
 intersection of architecture with, 481
 Inverse Conway Maneuver and, 139
 layered architectural style and, 160
 microservices architectural style and, 353-354
 modular monolith architectural style and, 174
 orchestration-driven service-oriented architectural style and, 325
 pipeline architectural style and, 188
 service-based architectural style and, 220
 space-based architectural style and, 310-311
team types that intersect with software architecture, 151
teams, making effective (see development teams, making effective)

teamwork, as requirement for software architect, 11
technical breadth, 20-29
 20-minute rule, 24-25
 developing a personal radar for technological change, 25-29
 technical depth versus, 10, 20
 Thoughtworks Technology Radar, 26-29
technical debt, 35
technical depth, technical breadth versus, 10, 20
technical leaders, architects as, 480
technical partitioning, 139, 142
 microkernel architecture and, 205
 teams and, 481
technical top-level partitioning, 137
technically partitioned architecture
 layered architecture as, 155
 pipeline architecture style as, 189
Technology Advisory Board (TAB), 26
technology bubbles, 26
technology radar (see Thoughtworks Technology Radar)
temporal coupling, 122
temporary queue, 257-258
tester filters, 183
Thoughtworks Haiven, 485
Thoughtworks Technology Radar, 26-29
 four quadrants, 26
 four rings, 27
three-tier architecture, 136
time to market, agility and, 481
top-level partitioning, 137-138
trade-off analysis, 76
trade-offs
 analyzing, as part of architectural thinking, 30-33
 architecture versus design, 19
 First Law of software architecture, 487-494
 in architectural characteristics, 64-65
transactions, in microservices, 345-348
transformer filters, 182
tuple space, 284
20-minute rule, 24-25

U

UML technical diagramming standard, 427
unit and functional testing checklist, 448
unitary architecture, 134
unknown unknowns, 483

unstructured monolith, 171

usability, ISO definition, 62

user stories

 assigning to logical components, 117-118

 user-story risk analysis, 416

V

Vasa (Swedish warship), 78

visionary, defined, 461

W

Workflow Event pattern, 249-252

Z

Zone of Pain, 48

Zone of Uselessness, 48

About the Authors

Mark Richards is an experienced hands-on software architect involved in the architecture, design, and implementation of microservices and other distributed architectures. He is the founder of DeveloperToArchitect.com, a website devoted to assisting developers in the journey from developer to software architect.

Neal Ford is director, software architect, and meme wrangler at Thoughtworks, a global IT consultancy with an exclusive focus on end-to-end software development and delivery. Before joining Thoughtworks, Neal was the chief technology officer at The DSW Group, Ltd., a nationally recognized training and development firm.

Colophon

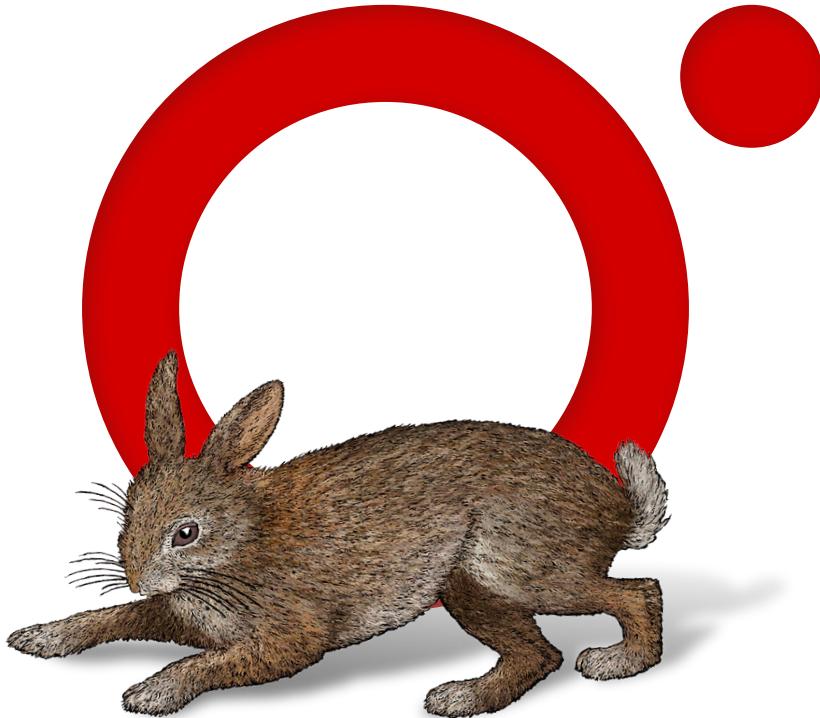
The animal on the cover of *Fundamentals of Software Engineering* is the red-fan parrot (*Deroptyus accipitrinus*), a native of South America, where it is known by several names such as *loro cacique* in Spanish, or *anacã, papagaio-de-coleira*, and *vanaquíá* in Portuguese. This New World bird makes its home up in the canopies and tree holes of the Amazon rainforest, where it feeds on the fruits of the *Cecropia* tree, aptly known as “snake fingers,” as well as the hard fruits of various palm trees.

As the only member of the genus *Deroptyus*, the red-fan parrot is distinguished by the deep red feathers that cover its nape. Its name comes from the fact that those feathers will “fan” out when it feels excited or threatened and reveal the brilliant blue that highlights each tip. The head is topped by a white crown and yellow eyes, with brown cheeks that are streaked in white. The parrot’s breast and belly are covered in the same red feathers dipped in blue, in contrast with the layered bright green feathers on its back.

Between December and January, the red-fan parrot will find its lifelong mate and then begin laying two to four eggs a year. During the 28 days in which the female is incubating the eggs, the male will provide her with care and support. After about 10 weeks, the young are ready to start fledging in the wild and begin their 40-year life span in the world’s largest tropical rainforest.

While the red-fan parrot’s current conservation status is designated as of Least Concern, many of the animals on O’Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on a black and white engraving from *Lydekker’s Royal Natural History*. The series design is by Edie Freedman, Ellie Volckhausen, and Karen Montgomery. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag’s Ubuntu Mono.



O'REILLY®

**Learn from experts.
Become one yourself.**

60,000+ titles | Live events with experts | Role-based courses
Interactive learning | Certification preparation

Try the O'Reilly learning platform free for 10 days.

