



EXPERT INSIGHT

Software Architecture Patterns for Serverless Systems

Architecting for innovation with event-driven
microservices and micro frontends

Second Edition

Foreword by:

Memi Lavi

CEO and Chief Cloud Architect at Polar Technologies

John Gilbert

<packt>

Software Architecture Patterns for Serverless Systems

Second Edition

Architecting for innovation with event-driven microservices and micro frontends

John Gilbert



BIRMINGHAM—MUMBAI

Software Architecture Patterns for Serverless Systems

Second Edition

Copyright © 2024 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Senior Publishing Product Manager: Denim Pinto

Acquisition Editor – Peer Reviews: Saby Dsilva

Project Editor: Amisha Vathare

Content Development Editor: Shikha Parashar

Copy Editor: Safis Editing

Technical Editor: Aneri Patel

Proofreader: Safis Editing

Indexer: Pratik Shirodkar

Presentation Designer: Ganesh Bhadwalkar

Developer Relations Marketing Executive: Vipanshu Parashar

First published: June 2021

Second edition: February 2024

Production reference: 1230224

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul's Square

Birmingham

B3 1RB, UK.

ISBN 9781803235448

www.packt.com

Foreword

Serverless.

This magic word has captured the imagination of countless cloud developers in recent years. What can be better than a serverless runtime, where you just slap your code into it, and it simply runs?

Well, as it turns out, having it “simply” run is not that simple. Serverless systems, while introducing new levels of simplicity, also brought with them several new architectural paradigms.

How do you secure serverless functions? How do you monitor them? How much is it going to cost? How does it affect the client code?

In fact, the serverless paradigm, which began as an IT-related paradigm, allowing running lightweight pieces of code on fully managed infrastructure, taking care of scaling, monitoring, updating, and more, formed into something much larger. How can we lightweightize (yep, that’s a word now) almost everything in our software? The microservices we used last year to make our software modular and lightweight suddenly became enormous chunks of code that can be further dismantled into lightweight functions, running on serverless platforms.

This led to some of the interesting developments in the software architecture field. For example, why should we aspire to run lightweight code in the backend, but stick to a massive monolith in the frontend? And because of this, micro frontends were born.

Why expose synchronous, strongly coupled web APIs, when we can use small, loosely coupled events instead?

All these questions led to a clear insight – a new software architecture paradigm is necessary, one that takes advantage of all these developments.

But how do you learn a new architecture? How can software architects, one that worked on dozens of microservices-based systems, become accustomed to the new paradigm and feel comfortable enough to try it on the new system they are working on?

Well, it turns out that there is more than one way of doing that.

In microservices, we had the seminal “Microservices” article by Martin Fowler that laid the foundations for what is now the most widely used architecture pattern out there. But even this article, quoted in every microservices book and lecture, is not a practical, walkthrough guide. After all, there are quite a few microservices books...

And for that, we have this book.

The book you hold in your hands (or maybe reads from a screen) is the ultimate one-stop-shop for the new architecture style, epitomized as “serverless.” It describes the new lightweight-oriented architecture that can be created using serverless cloud engines such as AWS Lambda, all the way to the frontend using micro frontends.

But it doesn’t stops there.

It also touches on the foundational topics such as Domain Driven Design and the SOLID principles, which form the basis of modern software design, explains the concept behind events, and also explains database-related patterns such as CQRS.

Finally, with the addition of micro frontends, you will be able to design your own serverless, event-driven systems that are flexible enough to allow you to deliver value at the rate that your business requires. This was really just a small taste of what you can find in it.

In short – this is the perfect book for the new wave of software architecture patterns.

Reading it, and then visiting again the parts you’re working on, will definitely make you a better software architect, fully adapted to the modern world of serverless architecture.

Memi Lavi

CEO and Chief Cloud Architect at Polar Technologies

Contributors

About the author

John Gilbert is a CTO with over 30 years of experience in architecting and delivering software systems across many industries. His cloud journey has spanned all the levels of cloud maturity, from lift and shift and software-defined infrastructure to microservices and continuous deployment. He was an early serverless adopter and put his first serverless workloads into production just months after AWS Lambda's introduction. He has also authored *Cloud Native Development Patterns and Best Practices* and *JavaScript Cloud Native Development Cookbook*. He finds delivering serverless solutions to be most fun and satisfying, as they force us to reconsider how we reason about systems and enable us to accomplish far more with much less effort and risk.

To my wife, Sarah, and our children, for their endless love and support on this journey

About the reviewers

Marco Lenzo is an experienced technical leader who has designed and implemented multiple highly available distributed systems. Currently, he holds the position of Lead Architect for an **Infrastructure-as-a-Service (IaaS)** product developed by several agile teams.

His love for IT started at a young age and is not limited to software development alone, but it also encompasses security, systems, and networks. He is a strong supporter of Agile, DevOps, GitOps and cloud-native technologies. He was an early adopter of Kubernetes and holds the **Certified Administration (CKA)** and **Kubernetes Certified Security Specialist (CKS)** certifications.

Marco enjoys coaching and mentoring and is an active contributor to the software development community. He currently runs a blog and YouTube channel focusing on software architecture, DevOps, and leadership.

I wish to thank John and Packt for this great opportunity. It was a pleasure reviewing a book which leverages skillfully the pillars of software development, such as DDD and the SOLID principles, to deal with complexity of the cutting-edge architecture of serverless systems.

Re Alvarez Parmar is a Principal Specialist Solutions Architect at **Amazon Web Services (AWS)**, where he advises Fortune-100 companies on building cloud-native systems. He has over 20 years of experience with architecting, building, and operating enterprise systems. He currently focuses on modern application development patterns, cloud architecture, and Kubernetes.

Being completely self-taught, Re is passionate about knowledge sharing and believes in giving back to the community that has given him so much. His works have been featured on the AWS Containers and Architecture blogs. He has given talks on cloud-native architecture at KubeCon US, and has been a regular speaker at AWS Re:Invent for the past 6 years.

As I sit down to write these acknowledgments, I am filled with gratitude and humility, knowing that no book comes to fruition through the efforts of one person alone. Behind the pages you hold in your hands is a network of support, encouragement, and unwavering love that I am privileged to acknowledge.

First and foremost, I must express my deepest gratitude to my wife, Nadia. Throughout the arduous journey of editing this book, she was my steadfast anchor, providing unfaltering support and understanding. She selflessly managed our home and took on countless responsibilities so that I could dedicate long hours to this book. Without her, none of this would have been possible, and for that, I am eternally grateful.

To my children, Jayden, Norah, and Neil. Your immeasurable energy, curiosity, and creativity is a constant source of inspiration. And finally, my mother, whose unwavering support and belief in me have been a guiding light in my life.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

<https://discord.gg/B8TubSxc35>

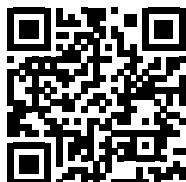


Table of Contents

Preface	xxvii
Chapter 1: Architecting for Innovation	1
Continuously delivering business value	1
By the skin of our teeth • 2	
Through high-velocity teamwork • 2	
Dissecting lead time	3
Risk mitigation • 4	
Decision making • 4	
Software development life cycle methodology • 5	
Hardware provisioning • 6	
Deployments • 6	
Software structure • 7	
Testing and confidence • 8	
Dependencies and inter-team communication • 8	
Dissecting integration styles	9
Batch integration • 10	
Spaghetti integration • 10	
Real-time integration • 11	
Enterprise application integration • 11	
Shared database • 12	
Service-oriented architecture • 13	
Microservices • 13	

Enabling autonomous teams with autonomous services	14
Autonomous services – creating bulkheads • 15	
<i>Asynchronous inter-service communication • 17</i>	
<i>Fortified boundaries • 17</i>	
Event-first – valuing facts • 18	
<i>Inversion of responsibility • 19</i>	
<i>Events as first-class citizens • 19</i>	
<i>Idempotence and ordered tolerance • 20</i>	
Serverless-first – creating knowledge • 20	
<i>Self-service • 20</i>	
<i>Disposable architecture • 21</i>	
Data life cycle – fighting data gravity • 22	
Micro frontends – equalizing tiers • 23	
Observability – optimizing everything • 24	
Organic evolution – embracing change • 25	
Summary	26
 Chapter 2: Defining Boundaries and Letting Go	 27
Learning the hard way	28
Building on proven concepts	29
Domain-driven design • 29	
<i>Bounded context • 30</i>	
<i>Domain aggregate • 30</i>	
<i>Domain event • 30</i>	
SOLID principles • 30	
<i>Single Responsibility Principle • 31</i>	
<i>Open-Closed Principle • 32</i>	
<i>Liskov Substitution Principle • 32</i>	
<i>Interface Segregation Principle • 33</i>	
<i>Dependency Inversion Principle • 34</i>	
Hexagonal Architecture • 35	

<i>Function-level (nano)</i> • 38	
<i>Service-level (micro)</i> • 38	
<i>Subsystem-level (macro)</i> • 38	
Thinking about events first	38
Start with event storming • 39	
Focus on verbs instead of nouns • 40	
Treat events as facts instead of ephemeral messages • 40	
Turn APIs inside out by treating events as contracts • 41	
Invert responsibility for invocation • 42	
Connect services through an event hub • 43	
Dividing a system into autonomous subsystems	44
By actor • 44	
By business unit • 45	
By business capability • 46	
By data life cycle • 46	
By legacy system • 47	
Creating subsystem bulkheads	48
Separate cloud accounts • 48	
External domain events • 49	
Dissecting an autonomous subsystem	50
Context diagram • 50	
Micro frontend • 51	
Event hub • 52	
Autonomous service patterns • 53	
<i>Backend For Frontend</i> • 53	
<i>External Service Gateways</i> • 54	
<i>Control services</i> • 54	
Dissecting an autonomous service	55
Repository • 55	
CI/CD pipeline and GitOps • 56	
Tests • 56	

- Stack • 56
- Persistence • 57
- Trilateral API • 58
 - Events • 58
 - API Gateway • 58
- Functions • 58
 - Nano architecture • 59
 - Micro architecture • 60
- Shared libraries • 61
- Governing without impeding 62**
 - Providing automation and cross-cutting concerns • 62
 - Promoting a culture of robustness • 63
 - Harnessing the four key team metrics • 64
- Summary 65**
- Chapter 3: Taming the Presentation Tier 67**
- Presentation tier innovation – zigzagging through time 67**
 - Client-side versus server-side rendering • 68
 - Build-time versus runtime rendering • 69
 - Web versus mobile • 70
- Breaking up the frontend monolith 71**
 - By subsystem • 71
 - By user activity • 71
 - By device type • 72
 - By version • 73
- Dissecting micro frontends 73**
 - The main app • 75
 - The index file • 76
 - The importmap file • 76
 - Micro-app registration • 77
 - Micro-app • 79

- The entry file • 79*
- Root component • 80*
- Micro-app activation • 80
 - Micro-app life cycle • 81*
 - Route-based activation • 81*
 - Manual activation • 82*
- Mount points • 83
- Manifest deployer • 85
- Inter-application communication • 85
- Designing for offline-first 86**
 - Transparency • 87
 - Status indicators • 87*
 - Outbox • 88*
 - Inbox • 88*
 - Local cache • 88
 - Caching code • 88*
 - Caching data reads • 91*
 - Caching data writes • 92*
 - Live updates • 94
 - WebSocket • 95*
 - Long polling • 96*
- Summary 97**
- **Chapter 4: Trusting Facts and Eventual Consistency 99**

- Living in an eventually consistent world 100
 - Staging • 100
 - Cooperative • 101*
 - Atomic • 101*
 - Consistency • 102
 - Transparency • 102*
 - Facts • 102*

<i>Chain reaction</i> • 103	
Concurrency and partitions • 103	
Order tolerance and idempotence • 104	
Parallelism • 104	
Publishing to an event hub	105
Event bus • 107	
Domain events • 107	
<i>Event envelope</i> • 107	
<i>Event-carried state transfer</i> • 109	
<i>Substitution</i> • 110	
<i>Internal versus external</i> • 110	
Routing and channel topology • 111	
Dissecting the Event Sourcing pattern	113
System-wide event sourcing • 114	
Event lake • 116	
<i>Perpetual storage</i> • 116	
<i>Indexing events</i> • 117	
<i>Replaying events</i> • 117	
Event streams	118
Temporal storage • 118	
Stream-first event sourcing • 119	
Concurrency control • 121	
Micro-event stores • 121	
Processing event streams	122
Micro batching • 122	
Choosing a programming paradigm • 124	
<i>Imperative programming</i> • 124	
<i>Functional reactive programming</i> • 125	
<i>Stream processing</i> • 125	
Creating a stream • 127	
Unit of work • 128	

Filtering and multiplexing • 128	
Mapping • 129	
Connectors • 130	
Designing for failure	131
Backpressure and rate limiting • 132	
Poison events • 133	
Fault events • 134	
Resubmission • 136	
Optimizing throughput	136
Batch size function parameter • 137	
Asynchronous non-blocking I/O • 137	
Pipelines and multiplexing • 139	
<i>Pipeline patterns • 141</i>	
Sharding • 143	
Batching and grouping • 143	
<i>Batching • 144</i>	
<i>Grouping • 144</i>	
Summary	145
Chapter 5: Turning the Cloud into the Database	147
Fighting data gravity	148
Competing demands • 148	
Insufficient capacity • 149	
Intractable volumes • 150	
Embracing the data life cycle	150
Create phase • 151	
Use phase • 152	
Analyze phase • 153	
Archive phase • 154	
Turning the database inside out	155
The transaction log • 156	
Derived data • 157	

Dissecting the CQRS pattern	158
System wide CQRS • 159	
Materialized views • 160	
Inbound bulkheads • 162	
Live cache • 164	
Capacity per reader, per query • 165	
Keeping data lean	165
Projections • 166	
Time to live • 167	
Implementing idempotence and order tolerance	168
Deterministic identifiers • 169	
Inverse optimistic locking • 170	
Immutable event triggers • 172	
Modeling data for operational performance	174
Nodes, edges, and aggregates • 174	
Sharding and partition keys • 176	
Single table design examples • 177	
<i>Restaurant service • 178</i>	
<i>Customer service • 179</i>	
<i>Cart service • 180</i>	
<i>Delivery service • 181</i>	
Leveraging change data capture	182
Database-first event sourcing • 183	
Soft deletes • 186	
Latching • 188	
Summary	190
 Chapter 6: A Best Friend for the Frontend	 191
Focusing on user activities	192
A BFF service is responsible for a single user activity • 192	
A BFF service is owned by the frontend team • 194	

A BFF service is decoupled, autonomous, and resilient • 194	
Dissecting the Backend for Frontend pattern	195
Datastore • 196	
API gateway • 197	
Query and command functions • 198	
Listener function • 198	
Trigger function • 199	
Dissecting function-level nano architecture	200
Models • 201	
Connectors • 201	
Handlers • 202	
Choosing between REST and GraphQL	203
REST • 204	
GraphQL • 205	
Implementing different kinds of BFF services	207
CRUD BFF services • 207	
List-of-values (Lov) BFF services • 208	
Task BFF services • 209	
Search BFF services • 209	
Action BFF services • 211	
Dashboard BFF services • 212	
Reporting BFF services • 213	
Archive BFF services • 214	
Summary	215
 Chapter 7: Bridging Intersystem Gaps	 217
Creating an anti-corruption layer	217
Macro-level ports and adapters • 218	
Substitution principle • 219	
External bulkhead • 220	

Dissecting the External Service Gateway pattern	220
Connectivity • 220	
Semantic transformation • 221	
<i>Ingress</i> • 222	
<i>Egress</i> • 223	
Packaging • 224	
Separate cloud accounts • 224	
Integrating with third-party systems	225
Egress – API call • 225	
Ingress – webhook • 227	
Asynchronous request response • 227	
Synchronous ESG/BFF hybrid • 229	
Integrating with other subsystems	229
Egress – upstream subsystem • 230	
Ingress – downstream subsystem • 232	
Integrating across cloud providers	234
Integrating with legacy systems	235
Ingress – Change Data Capture • 236	
Egress – Direct SQL • 238	
Egress – circuit breaker • 239	
Ingress – relay • 240	
Providing an open API and SPI	242
Ingress API – event • 242	
Ingress API – command • 243	
Egress SPI – webhook • 244	
Egress API – query • 245	
Tackling common data challenges	246
Idempotence • 246	
Enriching data • 247	

Latching and cross-referencing • 248	
Slow data resync • 249	
Summary	250
 Chapter 8: Reacting to Events with More Events	 253
<hr/>	
Promoting inter-service collaboration	254
Choreography's pros and cons • 254	
Dissecting the Control Service pattern	256
collect • 258	
correlate • 258	
collate • 259	
evaluate • 260	
emit • 262	
expire • 263	
Orchestrating business processes	264
Entry and exit events • 265	
Parallel execution • 269	
Employing the Saga pattern	271
Compensating transactions • 271	
Abort events • 272	
Calculating event-sourcing snapshots	274
What is ACID 2.0? • 275	
Snapshot events • 276	
Implementing complex event processing (CEP) logic	279
Decision tables • 279	
Missing events • 281	
Leveraging machine learning (ML) for control flow	283
Models • 283	
Predictions • 284	
Summary	284

Chapter 9: Running in Multiple Regions	287
Justifying multi-regional deployment	288
Regional disruptions • 288	
Nominal cost • 289	
Higher user satisfaction • 289	
Choosing a regional topology	290
Primary/hot-secondary • 290	
Active/active • 290	
<i>Balanced users • 291</i>	
<i>Balanced data processing • 291</i>	
Preparing for regional failover	292
Offline-first • 293	
Protracted eventual consistency • 293	
Checking regional health	294
Traditional health checks • 294	
Regional health checks • 295	
<i>Metrics • 295</i>	
<i>Regional health check service • 297</i>	
Configuring regional routing	299
Content Delivery Network (CDN) • 299	
API Gateway • 301	
<i>Regional endpoints • 302</i>	
<i>Global endpoints • 304</i>	
CDN plus API Gateway • 306	
Global bus • 308	
Replicating across regions	310
Change event vs domain event replication • 310	
Multi-master replication • 311	
Round-robin replication • 313	

Dissecting regional failover	315
Query failover • 315	
Command failover • 317	
Trigger failure points • 318	
Listener failure points • 319	
Addressing intersystem differences	320
External global endpoints • 321	
Legacy regional endpoints • 322	
Implementing multi-regional cron jobs	323
Inserting job records and events • 324	
Relying on replication and idempotence • 324	
Summary	325
 Chapter 10: Securing Autonomous Subsystems in Depth	 327
 Shared responsibility model	 328
Securing cloud accounts	329
Accounts-as-code • 329	
Identity access management • 330	
Securing CI/CD pipelines	332
Pipeline permissions • 332	
Deployment service permissions • 333	
Permission boundaries • 334	
Securing the perimeter	336
Global/edge infrastructure • 336	
Encryption in transit • 337	
Federated identity • 337	
Securing the frontend	337
OpenID Connect • 338	
Conditional rendering • 340	
Protected routes • 341	
Passing the JWT to BFF services • 342	

Securing BFF services	342
JWT authorizer • 342	
JWT assertion • 343	
JWT filter • 344	
Last modified by • 345	
Least privilege • 346	
Redacting sensitive data	346
Envelope encryption • 347	
Crypto-shredding • 349	
Securing ESG services	350
Securing shared secrets • 350	
Using API keys • 352	
Auditing continuously	352
Build-time and runtime auditing • 352	
Change event audit • 353	
Summary	354
 Chapter 11: Choreographing Deployment and Delivery	 357
Optimizing testing for continuous deployment	358
Continuous discovery • 358	
Continuous testing • 359	
Focusing on risk mitigation	360
Small batch size • 360	
Decoupling deployment from release • 361	
Feature flags • 362	
<i>Natural</i> • 362	
<i>Keystone</i> • 362	
<i>Artificial</i> • 363	
Fail forward fast • 363	
Achieving zero-downtime deployments	364
The Robustness principle • 364	

Between the frontend and its backend • 365	
Between producers and consumers • 366	
Between the backend and its data store • 367	
Planning at multiple levels	368
Experiments • 369	
Story backlog • 370	
<i>Story mapping</i> • 371	
<i>Story planning</i> • 372	
Task roadmaps • 372	
Turning the crank	373
Task branch workflow • 373	
<i>Create task branch</i> • 374	
<i>Create draft pull request</i> • 375	
<i>Ready for review</i> • 375	
<i>Merge to master</i> • 375	
<i>Accept canary deployment</i> • 376	
Feature flipping • 376	
<i>Exploratory testing</i> • 377	
<i>Beta users</i> • 377	
<i>General availability</i> • 377	
Dissecting CI/CD pipelines	378
Continuous integration pipeline • 378	
<i>Serverless testing honeycomb</i> • 380	
<i>Unit testing</i> • 383	
<i>Integration testing</i> • 385	
<i>Contract testing</i> • 387	
<i>Transitive end-to-end testing</i> • 390	
Continuous deployment pipeline • 391	
<i>Regional canary deployments</i> • 392	
<i>Synthetics</i> • 393	
Summary	395

Chapter 12: Optimizing Observability	397
Failing forward fast	398
Turning observability inside out	398
Leveraging FinOps	399
Cost is a metric • 399	
Dissecting monthly cloud cost • 400	
Worth-based development • 401	
Collecting resource metrics	402
The USE method • 402	
Capacity and concurrency limits • 403	
Throttling, iterator age, and queue depth • 403	
Errors and fault events • 404	
Tracking system events	405
Alerting on work metrics	406
The RED method • 407	
BFF service metrics • 408	
Domain event metrics • 408	
Anomaly detection • 409	
Observing real user activity	410
Real User Monitoring (RUM) • 410	
Synthetics • 411	
Tuning continuously	412
Function memory allocation • 412	
Cold starts • 413	
Timeouts and retries • 413	
Cache-control • 414	
Summary	415

Chapter 13: Don't Delay, Start Experimenting	417
<hr/>	
Gaining trust and changing culture	417
Establishing a vision • 418	
Building momentum • 419	
Constructing an architectural runway • 420	
Seeding and splitting teams • 420	
Funding products, not projects	421
Architecture-driven funding • 422	
Team capacity-driven funding • 422	
Dissecting the Strangler pattern	423
Event-first migration • 424	
Micro frontend – headless mode • 425	
Retirement • 425	
Addressing event-first concerns	426
System of record versus source of truth • 426	
Duplicate data is good • 427	
Avoid false reuse • 428	
Poly everything	429
Polyglot programming • 429	
Polyglot persistence • 430	
Polycloud • 431	
Summary	432
<hr/>	
Other Books You May Enjoy	435
<hr/>	
Index	439

Preface

Welcome to the book *Software Architecture Patterns for Serverless Systems*! Our industry has been searching for a silver bullet that allows us to quickly implement high-quality software. While there has been no single development that helps us do this, there has been the culmination of many developments that, when brought together in the right combination, help us to drive down lead time and continuously deliver business value.

Lean methods, DevOps practices, and the cloud have played a significant role in helping us to continuously deliver high-quality solutions. Now it is our software itself that has become the bottleneck and thus impedes our ability to deliver. So, my goal for this book is to show how to architect systems that enable change and system evolution. This is a comprehensive reference guide to designing architecture and implementing proven practices for delivering business value in a dynamic environment.

This book shows how the **serverless-first** mindset empowers **autonomous teams** and how the **event-first** approach creates an **inversion of responsibility** that allows us to create enterprise-scale serverless systems. You will learn how to apply the SOLID principles to define a system architecture with well-defined boundaries and then fortify these boundaries with autonomous services and autonomous subsystems. Add in micro frontends and finally, you'll be able to architect your own event-driven, serverless systems that are ready to adapt and change, so that you can deliver value at the pace needed by your business.

To get the most out of this book, be prepared with an open mind to discover why serverless is different. Serverless forces us to rewire, how we reason about systems. It tests all our preconceived notions of software architecture. I have personally found delivering serverless solutions to be, the most fun and satisfying. So, be prepared to have a lot of fun building serverless systems.

Who this book is for

This book is for software architects and aspiring software architects who want to learn about different patterns and best practices to design better software. Intermediate-level experience in software development and design is required. Beginner-level knowledge of the cloud will be beneficial.

What this book covers

Chapter 1, Architecting for Innovation, teaches architects that their job is to facilitate change and that their architecture must enable change. We survey the forces that impact lead time and look at the history of different integration styles to see how this influences the architectural decisions behind all the design patterns. Then we introduce autonomous services and the CPCQ flow and enumerate the set of concepts that enable teams to drive down lead times and continuously deliver business value.

Chapter 2, Defining Boundaries and Letting Go, is where architects begin to divide and conquer the problem domain. An architect will learn how to apply SOLID principles and event-first thinking to define architectural boundaries by dividing a system into autonomous subsystems and those subsystems into autonomous services. They will also learn the importance of continuous governance and how to rely on automation, observability, and key performance indicators to assert the performance of individual teams and to perpetuate an architectural feedback loop.

Chapter 3, Taming the Presentation Tier, addresses the challenges at the presentation tier and provides modern solutions, such as micro frontends, offline-first techniques, and more.

Chapter 4, Trusting Facts and Eventual Consistency, covers the foundational patterns that turn events into facts and support asynchronous inter-service communication, such as the event hub, systemwide event sourcing, and event stream processing.

Chapter 5, Turning the Cloud into the Database, covers the core data patterns that fight data gravity by turning the database inside out and spreading it out along the phases of the data life cycle, including **Command Query Responsibility Segregation (CQRS)** and **Change Data Capture (CDC)**.

Chapter 6, A Best Friend for the Frontend, covers the Backend for Frontend pattern that supports end-user activities and micro applications throughout the data life cycle.

Chapter 7, Bridging Intersystem Gaps, covers the External Service Gateway pattern that supports interactions with external systems and creates an anti-corruption layer that protects the system from everything outside the system, including third-party systems, legacy systems, other autonomous subsystems, and more.

Chapter 8, Reacting to Events with More Events, covers the Control Service pattern that supports business process orchestration, sagas, complex event processing, machine learning, and more.

Chapter 9, Running in Multiple Regions, describes how to run serverless systems in multiple regions and prepare for regional failover with regional health checks, latency-based routing, data replication and more.

Chapter 10, Securing Autonomous Subsystems in Depth, introduces the shared responsibility model for serverless computing and covers topics such as securing CI/CD pipelines, data reaction and more.

Chapter 11, Choreographing Deployment and Delivery, describes how to decouple deployment from release so that teams can continuously deploy and deliver with zero downtime. It covers the core concepts of backward compatibility and quality assurance.

Chapter 12, Optimizing Observability, describes how to turn serverless metrics into actionable information so that teams can experiment with confidences and fail forward fast.

Chapter 13, Don't Delay, Start Experimenting, provides thoughts on how to start making forward progress toward the architectural vision provided throughout the book. It addresses some of the myths and anti-patterns that keep teams from getting off the ground and covers various hurdles and impediments that architects must confront.

To get the most out of this book

To follow along and experiment with the templates provided with this book, you will need to configure your development environment according to the following steps:

1. Install Node Version Manager (<https://github.com/nvm-sh/nvm>) or (<https://github.com/coreybutler/nvm-windows>)
2. Install the latest version of Node.js with `nvm install node`
3. Create an AWS account (<https://aws.amazon.com/free>) and configure your credentials for the Serverless Framework (<https://www.serverless.com/framework/docs/providers/aws/guide/credentials>)
4. Download the templates: (<https://github.com/jgilbert01/templates>) and (<https://github.com/jgilbert01/micro-front-end-template>)
5. Run the project scripts, such as:

```
$ npm ci
$ npm test
$ npm run test:int
$ npm run dp:np:w
```

Download the example code files

The code bundle for the book is hosted on GitHub at <https://github.com/jgilbert01/templates> and <https://github.com/jgilbert01/micro-frontend-template>. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots and diagrams used in this book. You can download it here: <https://packt.link/gbp/9781803235448>

Conventions used

There are a number of text conventions used throughout this book.

`Code in text`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: “We can implement the same role-based assertions in GraphQL by decorating the model with directives, such as `@hasRole`.”

A block of code is set as follows:

```
export const forRole = (role) => (req, res, next) => {
  const groups = getUserGroups(req);
  if (groups.includes(role)) return next();
  else res.error(401, 'Unauthorized');
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
toUpdateRequest: (uow) => ({
  Key: {
    pk: uow.event.thing.id,
    sk: 'Thing',
  },
  ...updateExpression({
    name: uow.event.thing.name,
    ttl: uow.event.timestamp + (60*60*24*33) // 33 days
  }),
}),
```

Bold: Indicates a new term, an important word, or words that you see on the screen. For instance, words in menus or dialog boxes appear in the text like this. For example: “**Software as a Service (SaaS)** products typically provide a well-documented open API for invoking their functionality.”



Warnings or important notes appear like this.



Tips and tricks appear like this.

There is a concise diagramming convention used throughout this book. Serverless services have multiple resources, which can clutter diagrams with a lot of connecting arrows. The following sample diagram demonstrates how we minimize the number of arrows by placing related resources adjacent to each other so that they appear to touch. The nearest arrow implies the flow of execution or data. In this sample diagram, the arrow on the left indicates that the flow moves through the API gateway to a function and into the database, while the arrow on the right indicates the flow of data out of the database stream to a function and into the event bus. These diagrams are created using Cloudcraft (<https://cloudcraft.co>).

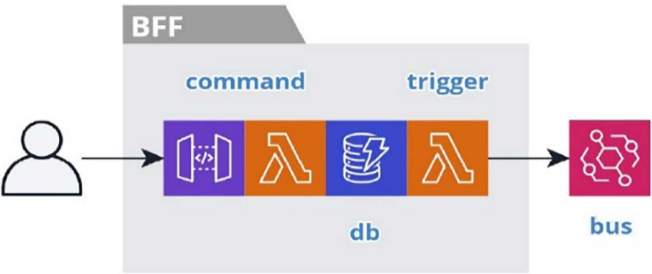


Figure: Demonstration of flow

The following table provides a legend of the most common icons used in the diagrams throughout the book.







Icon	Resource	AWS	Azure	GCP
	Event Bus	Event Bridge	Event Grid	Eventarc
	Event Stream	Kinesis	Event Hubs	Pub/Sub
	Function	Lambda	Functions	Functions
	Datastore	Dyanmo DB	Cosmos DB	Firestore
	Object Store	S3	Blob Storage	Cloud Storage
	API Gateway	API Gateway	API Mgt	API Gateway

Figure: Resource icon legend

The software architecture patterns in this book are cloud provider agnostic. However, I use AWS to implement the examples and AWS-specific icons in the diagrams. The preceding table enumerates various cloud provider resources that you could use to implement these patterns on the major cloud providers.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, email us at customer-care@packtpub.com and mention the book title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/support/errata and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Share your thoughts

Once you've read *Software Architecture Patterns for Serverless Systems, Second Edition*, we'd love to hear your thoughts! Please [click here](#) to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



<https://packt.link/free-ebook/9781803235448>

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

1

Architecting for Innovation

Change. Change is the only constant in the software industry. There are countless reasons for this change. All too often, the needs for our software change faster than we can write and we cannot adapt fast enough.

This is our quest in this book and in our industry. We need to architect our software for *change*. We must *innovate* at the speed of business. We need to adapt and change our software in real time. We need to *experiment* to find the best solutions in a dynamic environment. And we need the *confidence* to move fast and innovate, without breaking what works.

To achieve this, we first need to know what holds us back. We need to understand our software development past so that we improve going forward. So, in this first chapter, we will start with an analogy of many software projects and then consider an analogy that we can aspire to. Then we will survey our software development practices and integration styles to see how they increase *lead time* and slow us down. Finally, we will take this knowledge and sketch out the serverless event-driven architecture that we will unfold throughout the book: an architecture that gives *autonomous* teams the confidence to continuously *innovate* and deliver *business value*.

In this chapter, we're going to cover the following main topics:

- Continuously delivering business value
- Dissecting lead time
- Dissecting integration styles
- Enabling autonomous teams with autonomous services

Continuously delivering business value

Why do we write software? It's a simple—but important—question. The answer may seem obvious, yet, as with so many things, the answer is much more nuanced than it seems at first.

For me, the answer depends on which role I am playing. Am I playing the role of developer or architect? As a developer, I expect that my answer is the same as that of most developers. My short answer is: “Because I can, because it is fun, and it’s what I do.” I can create things, seemingly out of thin air. The creative process itself is its own reward. I enjoy what I do!

I have spent most of my career building custom software solutions. When someone who is not in the industry asks me what I do, I like to use an analogy; the typical custom motorcycle shop reality TV show serves as a great analogy for most software development projects.

By the skin of our teeth

Most of us have seen this type of show, and they all follow a similar formula. The customer wants a cool, custom motorcycle but only has a vague idea of what the final product should be. The shop owner is a wizard at turning the customer’s idea into an amazing design that they cannot resist. They agree on a price and a totally unrealistic schedule. Now, it is up to the team to turn the design into reality, no matter what it takes.

They do it on a compressed schedule and a shoestring budget because they enjoy creating things and seeing their customers love what they create. Never mind the fact that the process was filled with enough drama to keep the audience entertained. Mistakes were made, workarounds were devised, the requirements changed, rework was needed, and the team barely held it together along the way. Working around the clock, the team delivers, and the customer loves the results.

The problem with this analogy is that it hits a little too close to home. This really sums up the kind of experience many—if not most—software development teams live through. But the ends don’t justify the means. This is *unsustainable*.

Through high-velocity teamwork

As a **chief technical officer (CTO)** and chief architect, my goal is to do better than delivering by the skin of our teeth. We build software to help achieve the vision and mission of the organizations we work for. Continuously delivering business value must be our primary focus. Yet, we need to create *sustainable* practices and an architecture that supports them.

To this end, I much prefer the analogy of a Grand Prix motorcycle racing team. Manufacturers assemble these teams with the core purpose of driving *innovation* and outperforming the competition. They take the current product and push it to its limits and beyond. They collect copious *metrics* and scrutinize rider *feedback*. They formulate new experiments, rebuild the bike to the new specs, and run the tests again, pushing the limits ever farther. It is an amazing example of *high-velocity teamwork and continuous improvement*.

These teams achieve an inspiring level of sustainability as they continue to produce value, lap after lap, race after race, and season after season. There is drama, but it comes from their desire to outdo the competition by delivering the best-performing product.

This is an example of what we are striving for with our craft. We write software to relentlessly and *continuously deliver business value*. As software development teams, we accept that we work in a *dynamic environment* of fickle customers and stiff competition. We expect our business strategies to evolve and shift. Our methodologies and practices have become much more accommodating of constant change, and the cloud enables us to react more quickly.

However, we need more than just techniques and technology to succeed and flourish in this environment. We must *innovate* to stay ahead and deliver *value*. In other words, we must *experiment* with different *changes* to our software. **Our architecture must enable change**—this is our focus in this book. Our *methodology* and our *architecture* have to work *together* to support the sustainable pace of innovation that we need to achieve. The bottom line is this: we must consciously **architect for innovation**.

Let's start by looking at the forces that impact our ability to deliver timely solutions.

Dissecting lead time

Our ability to continuously deliver business value is a function of *lead time*. If we measure our lead times in months or even weeks, then our ability to deliver is not continuous. We are wasting valuable time while we assume that we are building the right solution. Much like the Grand Prix motorcycle analogy, we want to put our software in users' hands as fast as possible so that they can test drive it and provide *feedback*. Then, we tune the solution and quickly send it back out for another test drive.

To become a high-velocity team, we need to build the *muscle memory* that allows us to produce this seemingly uninterrupted flow of ideas and experiments. However, we need to get out of our own way. The typical software product delivery pipeline is full of potential *bottlenecks* that increase lead time. We need to understand these bottlenecks before we can produce an architecture that enables change.

Let's survey the forces that influence our lead time. We will start at the beginning of the pipeline and work our way through to see what optimizations we need so that we can take control of our lead times.

Risk mitigation

Risk is a powerful motivator. Traditionally, we mitigated risk by slowing down and increasing lead time. We believed this allowed us to double- and triple-check our results before releasing software to users. Yet, in today's dynamic environment, the biggest risk is not delivering the right solution to users or not delivering it in time to be effective. More often than not, the solution we set out to deliver is not the solution we land on, yet when we slow down, it takes that much longer to learn that we are off the mark. Then, it is all the more likely that we will not deliver a timely solution.

Instead, we need to frame our original idea as a hypothesis that needs validation. Then, we put the software in the hands of real users as quickly as possible so that we can find out if we are heading in the right direction and adjust the course before it is too late. Driving down lead times and moving at this velocity allows us to respond quickly to user feedback and iterate to the right solution. Along the way, we are actually delivering more and more value.

However, going faster introduces the risk of breaking things that already work. Users rely on the existing features and expect them to work. Users always want more, but not at the expense of losing what they already have. So, we need to *move fast without breaking things*.

The fear of breaking things naturally causes teams to slow down, so we need an architecture that gives us confidence to move fast. Our architecture must mitigate the risk of breaking things by limiting the blast radius of any mistakes to just what is changing.

But we need buy-in from decision-makers.

Decision making

Decision-making and a willingness to make decisions is another factor that influences lead times. Traditionally, companies manage software products as projects. To move forward, a project must have a plan, a budget, and approval. The process of achieving this is time-consuming and expensive, which produces an incentive to create fewer but larger and more expensive projects. This, in turn, creates a vicious cycle that further increases lead time. Ultimately, the risk of approving such a large budget limits the effectiveness of the project-based approach.

Fortunately, companies are beginning to understand that these traditional methods do not work in today's dynamic environment. They are learning that it is more effective to define the desired outcomes and then give teams the autonomy to make smaller, targeted decisions and work toward the desired outcomes. As architects, we need to promote this change by providing leadership and guiding the business to more effective management practices for software products. We will cover this topic in *Chapter 13, Don't Delay, Start Experimenting*.

In the meantime, we will put a methodology and an architecture in place that enables us to react quickly as the business shifts priorities.

Software development life cycle methodology

The **Software Development Life Cycle (SDLC)** methodology a team follows is the most obvious factor that impacts lead time. **Waterfall** is the classic methodology and the king of long lead times, with software releases measured in *months to years*. Its most significant false assumption is that we can define all the requirements upfront. However, in a dynamic environment, this is not possible. We do not have the luxury of time. We must *discover* the requirements as we go.

Agile methods are a major improvement, with lead times measured in *weeks*. However, a 2-week Agile sprint, as an example, only produces production-ready software. It does not necessarily put the software into the hands of the actual end users in production. It is not uncommon for several sprints to pass before we actually release (that is, deliver) the software to real users in production. The net result is that we still have real lead times measured in *months*, which is too long.

Kanban and **Lean** methods do the best job at helping us control *batch size* so that we can put working software in the hands of real beta users with lead times measured in *hours to days*. However, to move at this pace, we need to learn how to mitigate the risk to working software by *decoupling deployments from releases*, and we need to build the muscle memory that allows us to work at this pace. The following diagram depicts this practice, and we will cover this in detail in *Chapter 11, Choreographing Deployment and Delivery*:

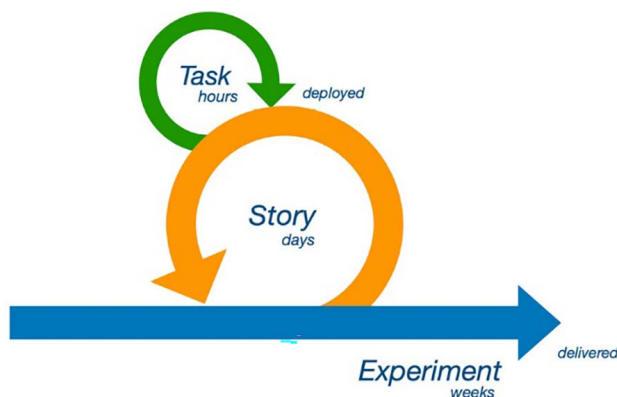


Figure 1.1: Decoupling deployment from release

But these practices require an architecture that enables change and allows us to work at this pace. So, let's continue down the product delivery pipeline to uncover more bottlenecks that impact lead time and influence our new architecture.

Hardware provisioning

Hardware provisioning is the cornerstone of lead time. Historically, we purchased these hard assets (that is, computers) as a capital expense. This required accounting, budgeting, and approval processes with long lead times. Then, we needed to purchase the hardware and have it delivered. After that, we needed to schedule resources to install and configure the hardware. The lead time for this process could easily take many months.

With hardware as the long pole in the tent, there was little incentive to optimize other parts of the software product delivery pipeline. The Waterfall method was well suited for these conditions, and Agile processes worked well if we provisioned hardware proactively. There was no reason to go faster.

Then, the cloud changed everything. We could provision hardware *on demand*. Hardware became an operating expense and no longer required long approval processes. The context changed, and this set a *domino effect* in motion that triggered the optimization of all downstream activities in the software product delivery pipeline. The next domino to fall was *manual* deployment.

Deployments

Before the cloud, there was little reason to *automate* hardware provisioning and software deployments. In my experience, deploying a new release of software was an error-prone, manual process that could easily take days, with all hands on deck to resolve seemingly endless configuration issues. At this pace, there was no incentive to deploy more often than every 2 to 3 weeks or so—it would simply be a waste of effort to deploy more frequently. But none of this mattered when the lead time for hardware was longer.

But the cloud turned *manual* deployments into the new long pole in the tent since hardware was now available on demand. We had an incentive to optimize this step in the delivery pipeline. It became imperative to automate the deployment of the entire stack—not just the software, but the infrastructure as well. We started treating the entire stack as **infrastructure-as-code (IaC)**.

Deployment lead times plummeted. The quality and predictability of deployments increased dramatically. A deployment was no longer a major event—it was just routine. We began to develop the muscle memory that allowed us to deploy at will, but we ran headlong into the next bottleneck. Our software could not get out of its own way. We needed to change how we structured and architected software.

Software structure

How we deploy software has a natural impact on how we structure and architect software. Deploying software *manually* is a tedious process that creates an incentive to do fewer deployments. As a result, we gravitated toward a monolithic software architecture, which is completely understandable. Combining many features into a single unit of deployment eased the burden of performing deployments. However, as we began to automate software deployments, it became abundantly clear that combining features into monoliths was the next *bottleneck* in the software product delivery pipeline.

Our software itself was impeding our ability to change it and reduce lead times. We could iterate on the functionality and automate the deployments, but we were afraid to actually pull the deployment trigger for fear of the *unintended consequences* of redeploying the whole monolith for a small change in just one piece of the monolith. We could not control the *blast radius* if something went wrong. A mistake could bring down the entire monolith. So, we would wait until we could batch up multiple changes, which forced lead times back up and brought us full circle.

We needed to break up the monolith. *Automated deployments* made it practical to deploy many *microservices*. We could make small changes to individual services and redeploy them independently. We could finally drive down lead times for individual services, and the lead times of each service would not impede each other—or so we thought. As we will see in the *Dissecting integration styles* section, dividing the monolith into microservices is not without its own side effects. But we were on the right track.

Yet prevailing questions remained. How would we slice and dice the monolith into smaller deployable units? How would we retain the development *simplicity* and *productivity* of the monolith? For example, how could a developer work on one service without having to run all the services? How would we maintain the *understandability* of the system as a whole? How would we avoid the inevitable *big ball of mud* that would become known as the Microlith? Worst yet, how would we avoid the dreaded microservices death star? Most importantly, how would we control the *blast radius* when we broke a microservice?

We will start to unravel these questions in the *Enabling autonomous teams with autonomous services* section. Then, we will begin our deep dive into the patterns and building blocks of our architecture in *Chapter 2, Defining Boundaries and Letting Go*. But first, let's continue our survey of the factors that impact lead time. Testing is the next bottleneck in our software product delivery pipeline.

Testing and confidence

Traditional testing methods have a mixed relationship with lead time and a rich history of marginalization. As projects approached their deadlines, time would simply run out, and testing teams had to leverage what limited time remained. Yet to go faster and drive down lead times, we actually need *more testing*. We need to test continuously to mitigate the risk of breaking working software. This gives teams the confidence to go faster because, without confidence, teams will naturally put on the brakes and drive up lead time.

The problem is that our traditional testing methods are predicated on the same false assumption that we can define all the requirements upfront. Those methods no longer make sense because they require the software to proceed through a set of gates that drive up lead time when we need to knock it down.

Instead, our architecture will allow us to experiment. We will increase testability and observability. We will automate *continuous testing* and shift it to the left in the delivery pipeline to ensure there are no regressions in working software. We will leverage *feature flags* to control access to the new software. Then, we will put the new software in the hands of beta users so that they can validate our hypothesis. We will see how all these pieces fit together in *Chapter 11, Choreographing Deployment and Delivery*.

But to err is human. We will make mistakes, so we need to limit the *blast radius*. To do this, we need to understand the dependencies between the software that is changing and the working software, and this requires *inter-team communication*, which has a negative impact on lead time.

Dependencies and inter-team communication

Dependencies and inter-team communication have a significant impact on lead time. When one team is dependent on another, we cannot compress lead times any more than the time it takes for other teams to respond to our requests. When teams share resources, one team cannot make a change without the consent of the others. We need to allocate additional time and effort to coordinate and align team schedules and priorities. Plus, teams will naturally put on the brakes when they are not confident that they will receive prompt assistance when they are in a pinch.

Conway’s Law states that *organizations are constrained to produce designs that are copies of their communication structure*. In other words, to drive down lead times, our team structure and the structure of our software architecture need to work together. Teams cannot be at the mercy of other teams. They cannot share resources. They cannot rely on matrixed teams for needed skill sets. They must be completely *self-sufficient*. They must own the full stack. They must be *cross-functional*. Teams must be **autonomous**, and our software architecture must enable autonomous teams. In the *Serverless-first – creating knowledge* section, we will see how **self-service** cloud capabilities make this practical.

However, we cannot eliminate all dependencies in a system. A software system is a collection of many features, and those features are related to each other in natural ways. To maximize team autonomy and take control of lead times, we need an approach for creating autonomous features. We need an *inter-feature communication mechanism* that helps us mitigate risks and gives teams the confidence to experiment.

So let’s look at the different *integration styles* that allow features to communicate. In this section, we optimized our software delivery pipeline by removing the bottlenecks. In the next section, we will see how our choice of integration style influences our architecture and our lead times.

Dissecting integration styles

Our software systems are composed of many features that must work together to deliver value for the end users. How these features communicate has been one of the most significant topics in our industry. The communication pathways within a system impact the timely flow of information, but they also impact our ability to deliver timely solutions. A highly integrated system will maximize the flow of information. However, if we integrate features too tightly, then it impedes our ability to adapt to changing requirements. It can even put the stability of a system at risk as we make changes.

Our architecture must balance these opposing concerns. Let’s survey the different integration styles to see how they impact our systems, so we can take the best parts from each and bring them together in our new architecture.

Batch integration

When I started my career in software development, the major pain point for the industry was information silos. Companies had multiple systems throughout their organizations, and they were not well integrated. Valuable business information was isolated in these silos (that is, systems). Data in one system was not shared with the others. Duplicate manual data entry was the order of the day.

The integrations that did exist were batch-oriented. At best, these jobs ran on a nightly schedule. This meant that the data was often stale and even in conflict. The reconciliation of data between the silos was often a nightmare. The dependencies between the silos were an afterthought. The flow of information between them was too slow, which diminished the value of the overall system.

But the silos were *autonomous*. They could change on their own schedule, and downtime in one did not cause downtime in another. The silos were eventually consistent. It was just a matter of when, and at what cost.

Spaghetti integration

To make matters worse, the batch integrations were *point-to-point*. Each job only took a single source system and a single target system into consideration. This resulted in what we commonly refer to as *spaghetti integration*. The following diagram depicts a worst-case scenario where every system (that is, node) integrates with every other system in both directions. This results in $N(N-1)/2$ integrations, where N = number of nodes. In reality, this depiction is much cleaner and more predictable than what we see in practice:

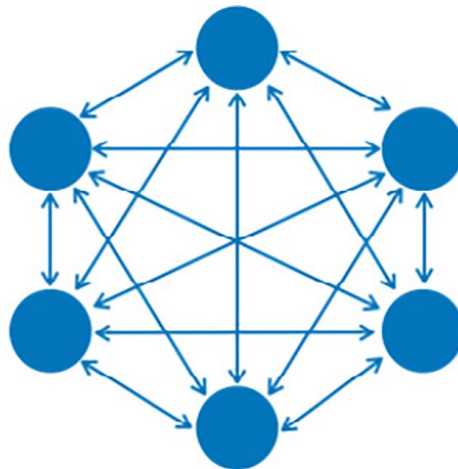


Figure 1.2: Point-to-point integration style

This approach creates a maintenance nightmare. A change to one system can result in weeks—or even months—of rework (that is, lead time) to upgrade all the integrations. Procrastination is a common way to avoid this problem. Meanwhile, the lead time for needed improvements grows and grows as teams *resist change*. Without a clean integration approach, the dependencies between the systems are a barrier to change.

Real-time integration

The industry was looking for a new integration approach, and the client-server model seemed like the solution. We started writing real-time integrations between the new **graphical user interface (GUI)** applications and the legacy systems. From the end user perspective, this seemed like a good thing because the latest information was immediately available. However, we had actually made things worse.

We had made everything *tightly coupled*. The *availability* of one system was now dependent on the availability of the systems it integrated with. A mistake in one system could now ripple through and incapacitate many systems. We had eliminated the reconciliation problem by shifting the burden to the end users. The system would remain consistent, but the users could not accomplish their work.

We had undervalued the *autonomy* that the information silos provided. We optimized one part of the equation, instead of *optimizing the whole*. Information was available in real time when everything was running smoothly, but it wasn't available at all when things went wrong.

The net result was an increase in lead time. The new real-time integration style was still point-to-point spaghetti integration. We needed more inter-team communication and coordination to keep those *brittle* integrations working.

Enterprise application integration

Along the way, a new approach to integration emerged. The deregulation of the 1990s resulted in many acquisitions and mergers. Companies found themselves with duplicate systems. Consolidating these systems was a lengthy process, so we needed to integrate them in the meantime. This led to the emergence of the **enterprise application integration (EAI)** industry and paradigm.

The duplicate systems needed to be integrated in such a way that they did not need to change. Each would still be the master of its own data, so we needed to support multiple heterogeneous masters. We implemented bi-directional synchronization between all the systems. Eventually, all but one of the duplicate systems would be retired by just turning them off once all users were transitioned.

The resulting approach was a near-real-time, eventually consistent, event-based synchronization with a *hub-and-spoke* model that supported *plug-and-play*. The following diagram depicts this optimized solution where each application (that is, node) integrates in both directions with a *canonical model*. This reduced the maintenance burden to $N - 1$ integrations, where N = number of nodes:

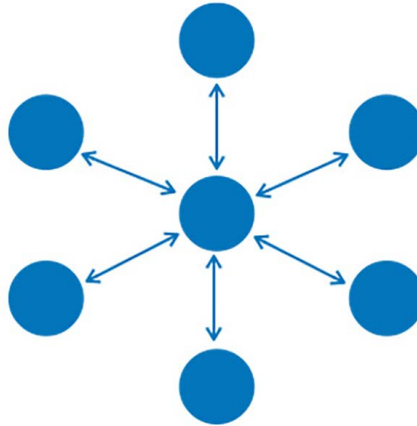


Figure 1.3: Hub-and-spoke integration style

This approach was a major improvement. The systems remained decoupled and autonomous, with no single point of failure. The low latency of near-real time significantly lessened the need for reconciliation to account for errors in eventual consistency. The technical debt and complexity of point-to-point spaghetti integration were eliminated, and lead time was kept in check since teams did not have to coordinate with every other team.

Yet this approach did not reach widespread adoption.

Shared database

Instead, the trend shifted toward the consolidation of data within a mammoth monolithic shared database. The relational database became the center of the system. All applications interacted with the same data. An update from one application was immediately visible in other applications. The development model was simple and straightforward.

But the database was now the single point of failure. Any downtime for the database meant downtime for the entire system. A change to the shared schema could require changes to many applications. A change for one application required approval by a committee to help ensure that it did not break other applications. The net result was an increase in lead time.

Service-oriented architecture

Meanwhile, **service-oriented architecture (SOA)** captured the industry's imagination, and **enterprise service bus (ESB)** middleware flooded the market. SOA added an additional abstraction layer between the applications and the database, which eased the burden of data model changes and limited the impact of a database schema change in the service layer. This was a move in the right direction.

However, the bloated middleware increased lead time and decreased performance. The ESB was another shared resource that we needed to procure, provision, configure, and manage. As with all shared resources, the central team that owned the ESB was overworked, so lead times increased, and performance suffered because every request had to compete for ESB resources.

Microservices

Eventually, the deficiencies of SOA, combined with the impedance mismatch of monolithic software deployments, led to the adoption of microservices. We were finally starting to move back in the direction of autonomous systems. Each microservice was independently deployable and had its own database. Microservices did not share resources. A single team owned a microservice and all its resources. This helped keep a lid on the need for inter-team communication.

However, we did not change our programming model and integration style. We took all our best practices for the monolith and superimposed them on microservices, creating **microliths** instead. Despite all its flaws, real-time synchronous integration was still the preferred approach, but the volume of microservices and their interdependencies magnified these flaws many times, as illustrated in the following image:

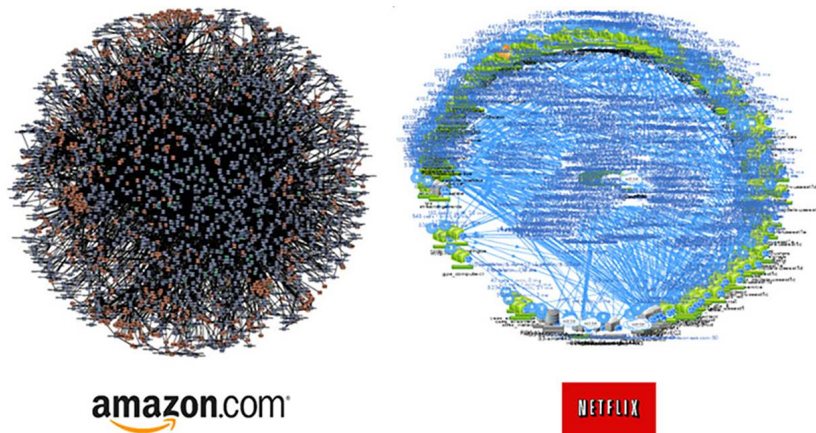


Figure 1.4: Microservices death star

Figure 1.4 depicts examples of the inevitable **microservices death star** that is created when we leave all these dependencies unchecked. It is the modern equivalent of spaghetti integration. The individual microservices themselves become the shared resources. An outage in a highly dependent microservice would ripple through the system, and a change to such a microservice could require reworks across many microservices.

Lead time did improve for small, self-contained changes, but a mistake in any change still represented a risk to the system. Lead time for any significant change remained high. We still needed extensive inter-team coordination to keep these brittle integrations working. All in all, the complexity of these systems skyrocketed.

To drive down lead times and continuously deliver business value, our architecture needs to bring together the best of all these integration styles. We need the **autonomy of information silos** and the **simplicity of the database programming model**. We need the **level of abstraction provided by SOA** and the **efficiency and loose coupling of EAI**, and we need the **independence of microservices without the complexity**. We need to balance the need for timely information with the need for timely solutions. We need to *optimize the whole*.

Now we are ready to establish our architectural vision and cover the core concepts of the architecture that enables change.

Enabling autonomous teams with autonomous services

As we have just seen, our ability to drive down lead times and iterate to innovative solutions is impeded by many forces. Lean methods and the cloud help us eliminate waste, but our software itself is an impediment to innovation. The complex dependencies within our systems necessitate inter-team communication and coordination, which consumes valuable time. The high coupling between components creates the risk of cascading failures when things go wrong. So, teams naturally slow down for fear of breaking things.

We need a new architectural vision. Our architecture must enable autonomous teams to move fast and limit the blast radius of honest mistakes. The following diagram depicts this architectural vision. Imagine a topology of autonomous services that we design for integration from the ground up: