DISTRIBUTED SYSTEMS WITH PYTHON AND RABBITMQ

DESIGNING RELIABLE, SCALABLE MESSAGING ARCHITECTURES FOR PRODUCTION

PRACTICAL PATTERNS | REAL-WORLD PROJECTS | PRODUCTION READY | COMPREHENSIVE OBSERVABILITY

MIKE DONOVAN

Distributed Systems with Python and RabbitMQ

Designing Reliable, Scalable Messaging Architectures for Production

Steve Publications

This book is available at
https://leanpub.com/distributedsystemswithpythonandrabbitmq

This version was published on 2026-08-04

Contents

Designing Reliable, Scalable Messaging Architectures for Production

This book teaches intermediate to advanced software engineers how to design, implement, deploy, operate, scale, and maintain robust distributed systems using Python and RabbitMQ. You will move from fundamental concepts of distributed computing through practical messaging patterns to production-ready architectures with complete, runnable code examples. Every chapter builds on the previous one, culminating in real-world projects that demonstrate event-driven microservices, distributed transactions via sagas, and comprehensive observability. By the end, you will have both the theoretical foundation and hands-on experience to build systems that stay reliable under load and recover gracefully from failure.

Chapter 1: Why Distributed Systems Matter

The Inevitability of Distribution

In 2010, most production applications ran as monoliths on a handful of servers. Requests flowed through a single codebase that talked to one or two databases. Deployment meant updating all the servers at once. Scaling meant buying bigger machines. That model worked reasonably well for systems serving thousands or even tens of thousands of concurrent users.

Today, it rarely does. Modern applications handle millions of requests per minute, process gigabytes of data in real time, serve users across continents with sub-second latency requirements, and must remain available through hardware failures, network partitions, and deployment cycles that happen dozens of times per day. A single monolithic process cannot reliably do all of this.

Distribution is not a choice anymore. It is the only way to build systems that meet modern demands for scale, availability, and resilience. When you distribute your system across multiple machines, processes, and services, you gain the ability to scale individual components independently, isolate failures so one bug does not take down everything, deploy updates without global downtime, and use specialized technologies for different parts of your workload.

The trade-off is complexity. Distributed systems introduce problems that simply do not exist in monolithic applications. Network calls fail. Messages arrive out of order or get duplicated. Two services disagree about the state of shared data because a replication delay left one replica behind. A deployment to one service breaks an assumption another service relied on. Clocks drift. Timeouts expire at different moments on different machines.

This book exists to help you navigate that complexity with discipline and clarity. We will focus on one of the most practical and widely adopted

approaches to distributed communication: message brokers, specifically RabbitMQ, combined with Python for application logic. This combination powers production systems at companies ranging from startups to financial institutions because it provides reliable asynchronous communication between services without forcing you to reinvent fundamental infrastructure problems.

Before we dive into specific technologies, let us understand why distribution matters and what happens when you get it wrong. The lessons in this chapter apply regardless of whether you use RabbitMQ, Kafka, gRPC, or any other distributed systems tool.

What Goes Wrong (and Often Does)

Distributed systems fail in ways that feel almost alien if you are used to monolithic programming. Understanding these failure modes is not academic; it directly informs every design decision you will make throughout this book.

Consider a simple scenario: Service A sends an order to Service B for processing via a message queue. Service B processes the order and updates inventory. In a monolith, this is two method calls in sequence. If the second call fails, you roll back with a database transaction. Clean, predictable, well understood.

In a distributed system, dozens of things can go wrong:

The message from A to B never arrives because the network connection between them drops mid-send. Service A thinks the order was sent; Service B never sees it. The customer paid but nothing happened.

The message arrives but Service B crashes after receiving it and before acknowledging it. Depending on your configuration, the message might be lost or redelivered later, causing duplicate processing. The customer gets charged twice.

Service B processes the message but crashes before persisting its result to the database. The message was acknowledged as processed, but the work was not completed. Data is inconsistent and there is no way to detect it automatically.

The message arrives at Service B while another instance of Service B is already processing the same order because a previous delivery timed out incor-

rectly. Two instances update inventory simultaneously without coordination. Stock counts become wrong.

Service A sends a message but the queue is full because consumers are slow. The publisher blocks or fails. From the user perspective, their action appears to hang or error even though the system will eventually process it if they retry.

Each of these scenarios illustrates a fundamental truth: in distributed systems, nothing is instantaneous, nothing is guaranteed, and failures are not exceptional; they are normal operating conditions. Your code must be designed for failure from the first line you write, not as an afterthought added when someone asks about production readiness.

This book will show you how to handle each of these scenarios using RabbitMQ features like publisher confirms, consumer acknowledgements, dead-letter exchanges, and idempotent processing combined with Python best practices for connection management, retry logic, and error handling. But first, you need the theoretical foundation to understand why these patterns exist and when to apply them.

The Cost of Getting It Right

Building distributed systems correctly is expensive in terms of engineering effort, operational overhead, and sometimes performance. Every reliability mechanism adds latency. Every replication factor consumes storage. Every retry loop risks amplifying problems instead of solving them. Every additional service increases your deployment surface and the number of things that can break.

The question is not whether you should invest in doing it right; the question is how much to invest given your constraints. A startup processing ten orders per hour does not need the same infrastructure as a payment processor handling ten thousand transactions per second. But both face the same fundamental challenges, just at different scales.

Here is the practical guidance: design for reliability from day one even if you start small. Use durable messages instead of transient ones because switching later requires schema migrations and downtime. Implement idempotent consumers early because adding idempotency to existing code that assumes single execution is painful. Add monitoring before you need it because debugging a

production incident without metrics is like searching for a needle in a haystack while blindfolded.

The patterns and practices in this book are designed to be proportional to your scale. A single RabbitMQ node running on one machine with durable queues, publisher confirms, and basic monitoring provides dramatically better reliability than the same setup without those features, at minimal cost. As you grow, you can add clustering, quorum queues, advanced retry strategies, and sophisticated observability without changing fundamental architectural decisions.

The alternative approach of building fast first and adding reliability later has caused more production outages and data loss incidents than almost any other single factor in software engineering history. Do not be one of those stories.

How This Book Is Structured

This book is organized as a progression from fundamentals to mastery, with each chapter building on concepts introduced earlier. You should read it in order because later chapters assume familiarity with earlier material. The structure follows three phases:

Phase one establishes foundations. Chapter 2 covers the theoretical underpinnings of distributed systems: the CAP theorem, consistency models, fault tolerance principles, and reliability concepts that inform every design decision. Chapter 3 explores messaging and communication patterns including point-to-point queues, publish/subscribe, request/reply, and event-driven architecture. These chapters are technology-agnostic because the concepts apply regardless of which message broker you choose.

Phase two focuses on RabbitMQ and Python specifically. Chapters 4 through 8 dive deep into RabbitMQ architecture, core concepts, client libraries, reliability features, advanced capabilities, and design patterns for building robust distributed systems. Each chapter includes complete, production-ready Python code examples using modern best practices with type hints, async programming, proper error handling, and connection management. You will learn not just how to use RabbitMQ features but why they exist and when each one matters.

Phase three applies everything to real-world projects and operational concerns. Chapter 9 walks through building a complete production-ready task

queue system from scratch, explaining every design decision. Chapter 10 demonstrates event-driven microservices architecture with CQRS patterns, event sourcing integration concepts, and saga-based distributed transactions. Chapters 11 through 14 cover observability, security, deployment at scale with Docker and Kubernetes, high availability clustering, performance tuning, and systematic troubleshooting of production issues.

Throughout the book, code examples are complete and runnable. There are no placeholders like "implement this yourself" or omitted sections marked as exercises. Every dependency is listed, every configuration file is provided, and every design trade-off is explained. You can copy these examples directly into your own projects as starting points for production systems.

The goal is not to produce RabbitMQ users who can follow tutorials but engineers who understand distributed systems deeply enough to design reliable architectures, debug complex failures, and make informed trade-offs under pressure. Let us begin with the theory that makes all of this possible.

Chapter 2: Foundations of Distributed Systems

The CAP Theorem (and What It Really Means)

The CAP theorem is the most cited and most misunderstood result in distributed systems theory. Eric Brewer originally formulated it as a conjecture in 2000, and Seth Gilbert and Nancy Lynch provided a formal proof two years later at MIT. The theorem states that in the presence of a network partition, a distributed system must choose between consistency and availability.

Here is what each term means precisely:

Consistency means that every read receives the most recent write or an error. If you write value X to a key and then read that key from any node in the system, you get X back, not some stale previous value. All nodes agree on the current state at all times.

Availability means that every request receives a response without guarantee that it contains the most recent write. The system always responds; it never says "I am down" or times out because of internal coordination issues. Every node accepts reads and writes independently.

Partition tolerance means that the system continues to operate despite arbitrary message loss or failure between nodes. The network may split into isolated groups that cannot communicate with each other, and the system must keep working within each group.

The critical insight that confuses many people is that CAP is not a choice among three options where you pick two. Partition tolerance is mandatory in any real distributed system because networks fail; there is no such thing as a production system without partitions. The actual trade-off is between consistency and availability when a partition occurs. You cannot have both during a partition.

When a network splits into two groups, the system faces an impossible choice. If it prioritizes consistency, one group must stop accepting writes

because it cannot guarantee those writes will be replicated to the other group before responding. Any write that succeeds on only one side would create divergent data, violating consistency. This is a CP system: consistent but unavailable during partitions.

If it prioritizes availability, both groups continue accepting reads and writes independently. Clients always get responses. But now the two groups have divergent data because each processed writes the other group did not see. When the partition heals, the system must resolve conflicts somehow. This is an AP system: available but inconsistent during partitions.

RabbitMQ as a message broker sits in an interesting position relative to CAP. For queue operations, it is primarily availability-oriented: publishers can send messages and consumers can receive them even when some nodes are unreachable, assuming messages are routed to reachable queues. However, RabbitMQ uses quorum queues based on the Raft consensus algorithm for replicated queues, which provides strong consistency guarantees at the cost of requiring a majority of nodes to be available. A three-node quorum queue tolerates one failure but becomes unavailable if two nodes go down simultaneously because there is no longer a majority to reach consensus.

The practical lesson from CAP is not that you must choose between consistency and availability for your entire system. Modern systems often make different choices for different data or different operations. Your user session cache can be eventually consistent while payment records require strong consistency. Read operations can tolerate stale data while write operations demand linearizability. The key is understanding the trade-offs explicitly rather than implicitly accepting defaults that may not match your requirements.

Consistency Models from Strong to Eventual

Consistency models define the guarantees a system provides about when writes become visible to readers. These models exist on a spectrum from strongest to weakest, with each level trading some guarantee for better performance or availability.

Strong consistency, also called linearizability, is the model most developers intuitively expect. It requires that every read sees the most recent completed write as if all operations happened in a single global order. If client A writes value X and the write returns success, then any subsequent read from any client

must return X or a later value, never an older one. This is what a single-process application with a local variable provides. In distributed systems, achieving strong consistency requires coordination between nodes, typically through consensus algorithms like Raft or Paxos, which adds latency and reduces availability during failures.

Sequential consistency relaxes strong consistency slightly by requiring that all operations appear to execute in some total order that respects the program order of each individual process. Different processes may observe operations in different orders as long as each process sees its own operations in the order it issued them. This is weaker than linearizability but still provides strong guarantees about operation ordering.

Causal consistency requires that causally related operations are seen by all processes in the same order, while unrelated operations may be observed in different orders. If write B depends on write A (because B reads the value written by A), then any process that sees B must also see A. Writes that are independent of each other can be seen in any order. This model provides good intuitive guarantees for many applications while allowing more concurrency than stronger models.

Session consistency guarantees strong consistency within a single client session but allows weaker consistency across sessions. Reads performed by a client after its own write will always see that write, but other clients may see stale data until replication completes. Web applications commonly use this model: your shopping cart updates are immediately visible to you as you add items, but inventory counts seen by other users may lag slightly behind actual changes.

Eventual consistency is the weakest common model and the one most relevant to messaging systems like RabbitMQ. It guarantees that if no new updates are made to a data item, eventually all accesses to that item will return the last updated value. "Eventually" is not precisely defined; it depends on replication lag, network conditions, and system load. Under normal operation with healthy nodes, eventual consistency typically means milliseconds or seconds of lag. During failures or high load, it can mean minutes or longer.

RabbitMQ's behavior relates to consistency models in subtle ways. The broker itself does not store application data; it transports messages between producers and consumers. Consistency concerns arise at the boundaries: when a message is published, when it is delivered to a queue, and when it is

consumed and processed.

With publisher confirms enabled and messages written to durable queues, RabbitMQ provides strong consistency about whether a message was accepted by the broker. The confirm is sent only after the message is safely stored, so if you receive a confirm, the message will not be lost due to broker restart. This is essentially a write acknowledgement with durability guarantees.

Message delivery to consumers can be configured for at-least-once or at-most-once semantics depending on acknowledgement mode. At-least-once delivery (manual acknowledgements) means every published message will be delivered to at least one consumer, but some messages may be delivered multiple times if a consumer crashes before acknowledging. At-most-once delivery (auto-acknowledgements) means each message is delivered exactly once or not at all, but messages can be lost if a consumer crashes after receiving but before processing.

The choice between these modes directly maps to the consistency trade-off: at-least-once favors durability and correctness at the cost of potential duplicates that your application must handle idempotently; at-most-once favors simplicity and single execution at the cost of potential message loss.

Fault Tolerance and Failure Modes

Fault tolerance is the ability of a system to continue operating correctly in the presence of component failures. In distributed systems, failures are not rare edge cases; they are guaranteed to occur regularly. Hard drives fail, network cables get unplugged, processes crash, data centers lose power, and software bugs cause unexpected behavior under unusual conditions. A fault-tolerant system anticipates these failures and continues providing service with degraded performance if necessary but without catastrophic data loss or extended downtime.

Understanding failure modes is essential for designing fault tolerance because different types of failures require different mitigation strategies. Here are the primary failure modes in distributed systems:

Crash failures occur when a process or node stops functioning entirely. The node becomes unresponsive and does not send or receive messages. This is the simplest failure mode to handle because the node is clearly down; other nodes

can detect its absence through timeouts or heartbeats and take corrective action like redirecting traffic to healthy nodes or promoting a replica to leader.

Network failures include packet loss, delays, reordering, duplication, and complete partitioning between groups of nodes. Network failures are more insidious than crash failures because a node may be running correctly but unable to communicate with other nodes. From the perspective of other nodes, it appears crashed. The node itself may continue processing requests locally, creating divergent state that must be reconciled when connectivity is restored.

Byzantine failures are the most general and difficult to handle: a node behaves arbitrarily or maliciously, sending incorrect data, contradicting itself, or colluding with other faulty nodes to compromise system integrity. Most distributed systems assume crash-stop failures rather than Byzantine failures because Byzantine fault tolerance requires complex algorithms and significant overhead. However, software bugs can cause behavior that looks Byzantine from the outside even when no node is intentionally malicious.

Omission failures occur when a component fails to perform an expected action: a message is not sent, a response is not received, or a timeout expires without completion. Omission failures are common in messaging systems and require explicit handling through timeouts, retries, and monitoring. The challenge is distinguishing between "the operation is slow but will complete" and "the operation has failed and will never complete."

Timing failures occur when operations take longer than expected, causing timeouts to expire prematurely or deadlines to be missed. Timing failures are particularly problematic because they can cascade: a slow response from one service causes its caller to timeout and retry, generating additional load that makes the slow service even slower, creating a feedback loop that can bring down an entire system.

RabbitMQ provides several mechanisms for handling these failure modes. Heartbeat messages detect dead connections faster than operating system TCP timeouts. Publisher confirms verify that messages were received and persisted by the broker before considering them safely delivered. Consumer acknowledgements ensure that messages are not lost if a consumer crashes mid-processing. Dead-letter exchanges capture messages that cannot be processed successfully so they can be inspected and handled rather than silently disappearing. Clustering with quorum queues replicates data across multiple nodes so the system continues operating when individual nodes fail.

The key principle is defense in depth: no single mechanism protects against all failures, but combining multiple mechanisms provides robust protection against a wide range of failure scenarios while keeping any individual component simple enough to understand and verify.

Reliability, Availability, Durability

Reliability, availability, and durability are three distinct properties that production systems must balance. Confusing them leads to designs that excel at one while failing catastrophically at another.

Reliability is the probability that a system performs its intended function without failure over a specified period. A reliable system does what it is supposed to do, consistently and correctly. In messaging terms, reliability means messages are delivered to their intended recipients with their content intact, in the right order when ordering matters, and processed exactly as designed. Reliability depends on both correct implementation (no bugs that cause incorrect behavior) and fault tolerance (graceful degradation when components fail).

Availability is the proportion of time a system is operational and able to serve requests. It is typically expressed as a percentage over a given period: 99 percent availability means the system is down for about 3.65 days per year; 99.9 percent allows about 8.76 hours of downtime per year; 99.99 percent allows about 52 minutes. Achieving higher availability requires redundancy: multiple instances of each component so that if one fails, others take over seamlessly. Availability also depends on how quickly the system can recover from failures and whether recovery happens automatically or requires manual intervention.

Durability is the guarantee that once data has been successfully written, it will not be lost even in the face of system crashes, power failures, or hardware malfunction. In messaging systems, durability means that once a message has been acknowledged as received by the broker, it will survive broker restarts and remain available for consumption. Durability requires persisting data to non-volatile storage like disk rather than keeping it only in memory.

These three properties interact in important ways. A system can be highly available but unreliable if it responds to every request but sometimes returns incorrect results or loses data silently. A system can be reliable and durable but unavailable if it refuses to respond during maintenance windows or

when components are being replaced. Designing production systems requires understanding which property matters most for each operation and optimizing accordingly.

For RabbitMQ-based systems, the typical configuration prioritizes durability and reliability over raw availability because losing messages is usually worse than briefly delaying them. Durable queues with persistent messages ensure that data survives broker restarts. Publisher confirms provide assurance that published messages were safely stored. Manual consumer acknowledgements guarantee that messages are not considered processed until the consumer completes its work successfully. These mechanisms add latency compared to transient, auto-acknowledged messaging but prevent silent data loss that is often impossible to detect or recover from after the fact.

Availability in RabbitMQ is achieved through clustering: multiple broker nodes sharing state so that if one node fails, others continue serving clients. Quorum queues provide replicated storage across nodes so queue data survives individual node failures. Client libraries with automatic reconnection transparently switch to healthy nodes when a connection drops. Together, these features enable RabbitMQ deployments that achieve 99.9 percent or higher availability in production environments.

Idempotency as a Design Principle

Idempotency is arguably the single most important concept for building reliable distributed systems with message queues. An operation is idempotent if applying it multiple times has the same effect as applying it once. Mathematically, f(f(x)) equals f(x) for all valid inputs x. In practical terms, if a consumer receives and processes the same message twice due to a network glitch or retry, the system state after both deliveries should be identical to the state after a single delivery.

Idempotency is essential because at-least-once delivery guarantees that messages will not be lost but does not guarantee they will not be duplicated. When a consumer crashes after receiving a message but before sending its acknowledgement, RabbitMQ redelivers the message to another consumer (or the same consumer after it restarts). Without idempotent processing, this redelivery causes duplicate effects: charging a customer twice, sending two confirmation emails, creating two database records for the same order.

Implementing idempotency requires tracking which operations have already been applied and ignoring or safely reapplying duplicates. The specific approach depends on your domain:

For operations that create resources, use unique identifiers derived from the business context rather than auto-generated IDs. If an order message contains an order ID generated by the ordering service, the fulfillment service checks whether it has already processed that order ID before creating a fulfillment record. If a record exists, it returns success without doing anything; if not, it creates the record and proceeds.

For operations that update resources, design updates as conditional writes that depend on expected current state. For example, incrementing an inventory count should check that the current count matches the expected value before updating, preventing duplicate increments from applying twice. Alternatively, use append-only patterns where each operation adds a new entry to a log rather than modifying existing data; deduplication then becomes a matter of filtering out known entries when reading.

For operations that send notifications or side effects, track sent notifications in a database keyed by the triggering event ID. Before sending, check whether a notification for this event was already sent. This approach requires storing state about completed operations, which adds complexity but prevents duplicate customer-facing actions.

A practical pattern for implementing idempotency with RabbitMQ is to include a unique message ID or correlation ID in every message and have consumers store processed IDs in a fast lookup structure like a database table with a unique constraint or an in-memory cache with expiration. When a message arrives, the consumer checks whether its ID has been seen before. If yes, it acknowledges and discards the message as a duplicate. If no, it processes the message, records the ID as processed, and sends acknowledgement.

The choice of storage for processed IDs depends on your requirements. In-memory caches like Redis provide fast lookups but lose data on restart, potentially allowing duplicates during recovery. Databases with unique constraints provide durable guarantees but add latency. A common compromise is to use an in-memory cache for recent messages (covering the typical redelivery window) combined with database checks for older messages that might be redelivered after extended outages.

Idempotency should be designed into your system from the beginning, not

added later when duplicates appear in production. Retrofitting idempotency often requires invasive changes to business logic and data models because existing code assumes single execution. Designing for idempotency upfront adds minimal overhead while providing essential protection against one of the most common sources of bugs in distributed systems.

Chapter 3: Messaging and Communication Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Synchronous vs Asynchronous Communication

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Point-to-Point and Queue-Based Messaging

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Publish/Subscribe and Event Streaming

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Request/Reply Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Event-Driven Architecture Fundamentals

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Chapter 4: RabbitMQ Architecture and Core Concepts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Installing and Configuring RabbitMQ

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Nodes, Clusters, and Topology

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Virtual Hosts and Namespaces

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Exchanges and Their Types

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Queues, Bindings, and Routing Keys

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

The Message Lifecycle

Chapter 5: Python Clients and Connection Management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Choosing Your Client Library

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Connection Parameters and Heartbeats

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Channel Management Best Practices

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Async vs Sync Clients

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Robust Reconnection Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Resource Cleanup and Graceful Shutdown

Chapter 6: Building Reliable Producers and Consumers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Publishing Messages Correctly

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Consumer Acknowledgements and Nacks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Publisher Confirms for Delivery Guarantees

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Message Durability and Persistence

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Prefetch Tuning and Flow Control

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Error Handling Patterns

Chapter 7: Advanced RabbitMQ Features

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Priority Queues for Critical Workloads

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Delayed and Scheduled Messaging

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Dead-Letter Exchanges and Retry Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Message Time-to-Live and Expiration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

RabbitMQ RPC Pattern Implementation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Queue Arguments and Advanced Configuration

Chapter 8: Designing Robust Distributed Systems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Retry Strategies with Exponential Backoff

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Circuit Breaker Pattern Implementation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

The Saga Pattern for Distributed Transactions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Idempotent Consumer Design

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Message Ordering Guarantees

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Handling Duplicate and Out-of-Order Messages

Chapter 9: Project One - Production Task Queue System

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Requirements and Architecture Decisions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Project Structure and Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Task Serialization and Schema Design

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Worker Implementation with Graceful Shutdown

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Retry Logic and Dead-Letter Handling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Testing the Task Queue System

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Docker and Deployment Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Chapter 10: Event-Driven Microservices with CQRS

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Microservice Communication via Events

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Implementing the CQRS Pattern

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Event Sourcing Integration Concepts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Inter-Service Saga Orchestration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Domain Events and Event Versioning

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Handling Schema Evolution

Chapter 11: Observability in Production

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Structured Logging for Distributed Systems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Metrics Collection with Prometheus

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Distributed Tracing with OpenTelemetry

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Health Checks and Readiness Probes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Debugging Message Flow Issues

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Correlation IDs Across Services

Chapter 12: Security, Authentication, and Authorization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

TLS/SSL Encryption End-to-End

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

User Authentication and Password Policies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Authorization and Access Control

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

VHost Isolation Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Secrets Management in Production

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Network Security and Firewalls

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Chapter 13: Deployment, Scaling, and High Availability

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Docker and Docker Compose Setup

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

RabbitMQ Clustering Fundamentals

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Quorum Queues for High Availability

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Federation and Shovel Plugins

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Kubernetes Deployment Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Performance Tuning and Capacity Planning

Chapter 14: Production Pitfalls and Troubleshooting

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Memory Pressure and Flow Control

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Disk Alarms and Storage Management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Diagnosing Consumer Lag

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Connection Storms and Resource Exhaustion

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Split-Brain Scenarios

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

Systematic Troubleshooting Methodology

Conclusion

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.

References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/distributedsystemswithpythonandrabbitmq.