REDIS for AI Applications

A PRACTICAL GUIDE TO CACHING, VECTOR SEARCH, RAG, LLM INTEGRATION, AND PRODUCTION DEPLOYMENT WITH PYTHON

VECTOR SEARCH
SEMANTIC SEARCH AT SCALE WITH REDIS

RAG PIPELINES
BUILD POWERFUL RETRIEVAL-AUGMENTED GENERATION SYSTEMS

AI AGENT MEMORY
CREATE PERSISTENT MEMORY FOR INTELLIGENT AI AGENTS

SEMANTIC CACHING
SLASH LLM COSTS WITH SMART, SEMANTIC CACHING

REAL-TIME INFERENCE
ORCHESTRATE REAL-TIME AI WORKFLOWS WITH STREAMS AND PUB/SUB

DMITRI ORLOV

Redis for AI Applications: Building Fast, Intelligent Systems

A Practical Guide to Caching, Vector Search, RAG, LLM Integration, and Production Deployment with Python

Steve Publications

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

This version was published on 2026-08-08

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

A Practical Guide to Caching, Vector Search, RAG, LLM Integration, and Production Deployment with Python

About this book: This book teaches you how to use Redis as the backbone of production-grade AI applications. You will learn to build vector search indexes for semantic retrieval, implement Retrieval-Augmented Generation pipelines, create persistent memory systems for AI agents, deploy semantic caching that slashes LLM costs, and orchestrate real-time inference with streams and pub/sub. Every concept is illustrated with complete, runnable Python code using redis-py, FastAPI, sentence-transformers, LangChain, and other modern libraries. By the end, you will be able to architect, implement, deploy, and operate Redis-powered AI systems at scale.

Introduction: Why Redis for AI?

A Day in the Life of an AI Application

It is 9:14 AM on a Tuesday. Your company's customer support platform receives a question from a user in Tokyo: "How do I reset my two-factor authentication without losing access to my account?" The query arrives at your FastAPI server, which immediately needs to answer it accurately and quickly.

Behind the scenes, several things happen within milliseconds. First, the system checks whether this exact question or something semantically similar has been asked before. If a cached response exists with high enough similarity, it is returned instantly without ever calling an LLM. This semantic cache saves both money and time.

For a new question that misses the cache, the system embeds the query using a sentence-transformers model to produce a 384-dimensional vector. That vector is used to search your knowledge base of support documents stored in Redis with an HNSW index. Within five milliseconds, the top three relevant document chunks are retrieved: one about two-factor authentication setup, another about account recovery procedures, and a third about security best practices.

These chunks are combined into a prompt template that includes the user's question along with metadata about their account tier and region. The enriched prompt is sent to an LLM API, which generates a grounded, accurate response citing the retrieved documents. Meanwhile, Redis tracks this interaction: the conversation history is stored for context in future turns, usage metrics are updated for billing and monitoring, and the new Q/A pair is cached for potential future reuse.

The entire flow takes under one second from query to answer. The user gets a correct response with cited sources. Your company saves money through caching, maintains quality through retrieval augmentation, and scales gracefully as traffic grows because Redis handles all the hot-path operations in sub-millisecond time.

This is not a hypothetical scenario. This is what production AI applications look like in 2026 when built on solid infrastructure. And Redis has become one of the most critical pieces of that infrastructure.

The Performance Problem with LLMs

Large Language Models are powerful but they come with fundamental performance challenges that no amount of model optimization fully solves. Understanding these challenges is essential to understanding why Redis matters.

First, LLM inference is slow. Even optimized models on dedicated hardware typically produce tokens at rates between 20 and 100 tokens per second. A response of 500 tokens takes five to 25 seconds. This latency is unacceptable for real-time applications like chat interfaces where users expect responses within one or two seconds.

Second, LLM inference is expensive. API pricing varies by provider and model size, but typical costs range from $0.10 to $60 per million input tokens and $0.30 to $180 per million output tokens for major cloud providers as of 2025-2026 [1]. For a high-traffic application processing thousands of queries per minute, these costs compound rapidly. A single enterprise deployment might spend tens of thousands of dollars per month on LLM API calls alone.

Third, LLMs hallucinate. Studies show that even state-of-the-art models generate factually incorrect information in roughly 30 percent of responses, with rates climbing to 60 percent or higher for specialized domains like mathematics or medicine [2]. This unreliability makes raw LLM outputs dangerous for applications where accuracy matters: customer support, healthcare triage, legal research, financial analysis.

Fourth, LLMs have limited context windows. While modern models support contexts of 128K tokens or more, there are practical limits to how much information can be effectively included in a single prompt. For applications that need to reference large knowledge bases, entire codebases, or long conversation histories, you cannot simply dump everything into the context window and expect coherent results.

These four problems define the architecture of modern AI applications. You need caching to reduce latency and cost. You need retrieval augmentation to improve accuracy. You need memory management to handle context limits. You need real-time coordination for multi-step workflows. And all of these

requirements demand infrastructure that operates at sub-millisecond speeds because any bottleneck in your supporting systems gets magnified by the already-slow LLM calls.

What Redis Actually Is Today

Redis started as a simple in-memory key-value store created by Salvatore Sanfilippo in 2009. Its original design was elegant in its simplicity: keep everything in RAM, use efficient data structures, process commands with an event loop, and respond as fast as possible. This approach made Redis exceptionally fast for caching workloads, which is how most people first encountered it.

But over more than fifteen years of development, Redis has evolved far beyond its original scope while retaining the performance characteristics that made it popular in the first place. Today's Redis is a multi-model data platform that combines several capabilities into a single system:

In-memory data store and cache: Redis still excels at its original role. It provides sub-millisecond read and write latency for strings, hashes, lists, sets, sorted sets, streams, JSON documents, and other data structures. For AI applications, this speed matters because every millisecond saved in your infrastructure layer is a millisecond not added to already-lengthy LLM inference times.

Vector database: With Redis Stack (which bundles the open-source Redis server with modules like RediSearch), you can create vector indexes using HNSW or FLAT algorithms and perform similarity search on high-dimensional embeddings. This capability transforms Redis into a viable vector store for RAG systems, semantic caching, recommendation engines, and AI agent memory. At billion-vector scale, Redis 8 achieves 90 percent precision at median latencies around 200 milliseconds [3].

Search engine: RediSearch provides full-text search with BM25 ranking, prefix matching, fuzzy search, and faceted aggregation. When combined with vector search, you get hybrid retrieval that leverages both semantic similarity and keyword matching for higher quality results than either approach alone.

Streaming engine: Redis Streams (introduced in Redis 5.0) provides a log-like data structure with consumer groups, message acknowledgment, and delivery tracking. This makes Redis suitable for event-driven architectures

where you need to process events reliably across multiple consumers without losing data.

Message broker: Redis pub/sub enables real-time broadcasting of messages to subscribers. While simpler than streams (no persistence or acknowledgment), it is ideal for use cases where low-latency notifications matter more than guaranteed delivery, such as live updates in collaborative AI tools or real-time chat systems.

Lock manager and coordination service: Redis supports distributed locking via SET NX with expiration, which enables safe concurrent access to shared resources across multiple application instances. For AI applications running batch processing jobs, model training tasks, or resource-intensive inference requests, distributed locks prevent race conditions and duplicate work.

The key insight for AI engineers is that Redis consolidates many of the infrastructure needs of an AI application into a single platform. Instead of maintaining separate systems for caching (Redis), vector search (Pinecone or Weaviate), streaming (Kafka or RabbitMQ), session management (another Redis instance), and rate limiting (a custom service), you can run all of these workloads on one Redis deployment. This consolidation reduces operational complexity, eliminates data synchronization overhead between systems, and improves performance by keeping all relevant data in the same memory space.

That said, consolidation is not always the right choice. Specialized vector databases may offer better recall at certain scales, dedicated message brokers handle higher throughput for streaming workloads, and managed services reduce operational burden. The decision depends on your specific requirements around scale, latency, cost, and team expertise. This book will help you make those decisions with concrete information about trade-offs and real-world performance data.

How This Book Is Organized

This book is structured to take you from Redis fundamentals through advanced AI-specific patterns to production deployment. Each chapter builds on the previous ones, so reading in order is recommended if you are new to Redis. If you already know Redis well, you can skip ahead to the chapters most relevant to your needs.

The Introduction you are reading now establishes why Redis matters for AI applications and sets up the book's journey.

Chapter 1 covers Redis fundamentals: installation, connecting with redis-py, core data structures, transactions, pipelines, and persistence. This chapter brings you up to speed on the basics without getting lost in minutiae. It focuses only on what matters for building AI systems.

Chapter 2 explores caching patterns specifically tuned for AI workloads. You will learn how to cache LLM responses, embeddings, and prompts using both exact matching and semantic similarity. The chapter covers TTL strategies, eviction policies, rate limiting for AI endpoints, and cache invalidation approaches that prevent stale data from undermining your application's quality.

Chapter 3 dives into Redis Stack's search and vector capabilities. You will learn how to create vector indexes with HNSW and FLAT algorithms, store and query embeddings, combine vector search with metadata filters, and tune index parameters for optimal performance. This chapter provides the foundation for all the retrieval-based patterns that follow.

Chapter 4 is a complete guide to building RAG systems with Redis. You will implement an end-to-end pipeline: document ingestion, chunking strategies, embedding generation with sentence-transformers, indexing in Redis, retrieval via vector search, and integration with LLMs through FastAPI. The chapter includes evaluation techniques for measuring and improving retrieval quality.

Chapter 5 covers practical patterns for building production LLM applications: conversation history management, streaming responses through pub/sub, prompt template storage and versioning, tool use coordination, multi-step reasoning workflows, and usage tracking for cost monitoring.

Chapter 6 explores AI agent memory architectures. You will learn how to implement short-term working memory, long-term episodic memory with vector search, semantic memory retrieval, cross-agent shared memory, and session continuity using Redis data structures and the LangGraph checkpointing integration.

Chapter 7 covers real-time AI with Redis Streams and pub/sub. Topics include event-driven inference pipelines, consumer groups for scalable processing, real-time chat systems with LLM backends, and streaming data enrichment patterns.

Chapter 8 addresses session management and state: user authentication, feature flags for AI features, personalization profiles, distributed locks for concurrency control, and failure recovery patterns.

Chapter 9 dives into horizontal scaling with Redis Cluster: sharding strategies for vector workloads, replication and read replicas, failover and high availability, capacity planning, and scaling from a single node to global deployment.

Chapter 10 focuses on performance optimization: memory management, data structure selection, vector search tuning, pipeline usage, async Python patterns, and profiling techniques for diagnosing bottlenecks.

Chapter 11 covers security, observability, and reliability: authentication with ACLs, TLS encryption, monitoring with Prometheus and Grafana, alerting strategies, backup and disaster recovery planning.

Chapter 12 provides a practical guide to production deployment: managed versus self-hosted options, Docker and Kubernetes deployment with Helm charts, configuration management, CI/CD integration, and comparison of major vendor offerings including Redis Enterprise, AWS ElastiCache, Azure Cache for Redis, and GCP Memorystore.

Chapter 13 synthesizes everything into complete reference architectures: a production RAG platform, a real-time AI chat system, a multi-agent orchestration system, and an AI-powered search engine. Each architecture shows how all the pieces fit together with explanations of design decisions and trade-offs.

The Conclusion brings the book to a close by summarizing key lessons, looking at emerging trends in Redis and AI infrastructure, and offering final recommendations for building production systems.

Throughout the book, Python is the primary programming language. Every major concept includes complete, runnable code examples using redis-py, FastAPI, Pydantic, NumPy, sentence-transformers, LangChain, and other modern libraries. The examples are designed to be production-oriented with proper error handling, configuration management, connection pooling, and async patterns where appropriate.

Let's begin.

Chapter 1: Redis Fundamentals for AI Engineers

This chapter brings you up to speed on the core Redis concepts you need for building AI applications. We will cover installation, client connection patterns with redis-py, the essential data structures, transactions and pipelines, and persistence options. The focus is on practical knowledge that directly applies to the patterns we will build in later chapters.

Installing and Running Redis Locally

The fastest way to get Redis running locally is through Docker. This approach works consistently across operating systems and lets you switch between standard Redis and Redis Stack (which includes vector search) with a single command change.

For development and learning, use Redis Stack because it includes the RediSearch module needed for vector indexes and full-text search:

```
# docker-compose.yml for local AI development
version: "3.9"

services:
  redis-stack:
    image: redis/redis-stack:latest
    container_name: ai-redis
    ports:
      - "6379:6379"     # Redis client port
      - "8001:8001"     # Redis Insight web UI
    volumes:
      - redis-data:/data
    environment:
      - REDIS_ARGS=--appendonly yes --maxmemory 512mb --maxmemory-policy
      ↪  allkeys-lru

volumes:
  redis-data:
```

Start the container with:

```
docker compose up -d
```

Verify it is running by connecting with redis-cli:

```
docker exec -it ai-redis redis-cli ping
# Expected output: PONG
```

If you need standard Redis without modules (for example, to compare behavior or for production deployments that do not require vector search), use the official redis image instead:

```
services:
  redis:
    image: redis:7.2-alpine
    ports:
      - "6379:6379"
    volumes:
      - redis-data:/data
    command: redis-server --appendonly yes --maxmemory 512mb --maxmemory-policy
    ↪  allkeys-lru
```

For production, you will configure Redis differently with proper authentication, TLS encryption, and resource limits. Those details are covered in Chapters 11 and 12. For now, the local development setup above is sufficient to follow along with all code examples.

Connecting with redis-py (sync and async)

redis-py is the official Python client for Redis. It provides both synchronous and asynchronous APIs that mirror each other closely. Install it with:

```
pip install redis
```

Starting with version 8.0, redis-py uses RESP3 protocol by default while maintaining backward compatibility with existing code [4]. This means you get richer response types out of the box without changing your application logic.

Synchronous client: For simple scripts or applications that do not need concurrent I/O, the synchronous client is straightforward:

```python
import redis
from redis.exceptions import RedisError, ConnectionError as RedisConnectionError

def create_redis_client(
    url: str = "redis://localhost:6379/0",
    max_connections: int = 20
) -> redis.Redis:
    """Create a Redis client with connection pooling and sensible defaults."""
    pool = redis.ConnectionPool.from_url(
        url,
        max_connections=max_connections,
        decode_responses=True,          # Return strings instead of bytes
        retry_on_timeout=True,          # Retry on socket timeouts
        socket_keepalive=True,          # Keep TCP connections alive
        socket_keepalive_options={
            redis.SOL_TCP: 1,
           redis.TCP_KEEPIDLE: 300,   # Start sending keepalives after 5 minutes
            redis.TCP_KEEPINTVL: 60,    # Interval between keepalives: 1 minute
            redis.TCP_KEEPCNT: 9        # Max number of failed keepalives before drop
        },
        health_check_interval=30        # Periodically validate connections from pool
    )

    client = redis.Redis(connection_pool=pool)

    # Verify connectivity on startup
    try:
        client.ping()
    except RedisError as e:
        raise RuntimeError(f"Failed to connect to Redis at {url}: {e}") from e

    return client


# Usage in a FastAPI application
from fastapi import FastAPI, Depends
from contextlib import asynccontextmanager

app = FastAPI()

redis_client: redis.Redis | None = None


@asynccontextmanager
async def lifespan(app: FastAPI):
    global redis_client
    # Startup
```

```
    redis_client = create_redis_client(
        url="redis://localhost:6379/0",
        max_connections=20
    )
    yield
    # Shutdown: close the connection pool
    if redis_client:
        redis_client.connection_pool.disconnect()


app.router.lifespan_context = lifespan


def get_redis() -> redis.Redis:
    """Dependency injection for Redis client."""
    if redis_client is None:
        raise RuntimeError("Redis client not initialized")
    return redis_client


@app.get("/health")
def health_check(redis: redis.Redis = Depends(get_redis)) -> dict[str, str]:
    redis.ping()
    return {"status": "ok", "redis": "connected"}
```

Asynchronous client: For high-concurrency applications like AI APIs that handle many simultaneous requests, use the async client. It integrates naturally with asyncio-based frameworks like FastAPI:

```
from redis.asyncio import Redis as AsyncRedis, ConnectionPool
from redis.exceptions import RedisError

async def create_async_redis_client(
    url: str = "redis://localhost:6379/0",
    max_connections: int = 20
) -> AsyncRedis:
    """Create an async Redis client with connection pooling."""
    pool = ConnectionPool.from_url(
        url,
        max_connections=max_connections,
        decode_responses=True,
        retry_on_timeout=True,
        socket_keepalive=True,
        health_check_interval=30
    )

    client = AsyncRedis(connection_pool=pool)

```

```
    try:
        await client.ping()
    except RedisError as e:
        await client.aclose()
        raise RuntimeError(f"Failed to connect to async Redis at {url}: {e}")
        ↪  from e

    return client


# FastAPI lifespan with async Redis
async_redis_client: AsyncRedis | None = None


@asynccontextmanager
async def async_lifespan(app: FastAPI):
    global async_redis_client
    async_redis_client = await create_async_redis_client()
    yield
    if async_redis_client:
        await async_redis_client.aclose()


app.router.lifespan_context = async_lifespan


async def get_async_redis() -> AsyncRedis:
    """Async dependency for Redis client."""
    if async_redis_client is None:
        raise RuntimeError("Async Redis client not initialized")
    return async_redis_client


@app.get("/health-async")
async def health_check_async(redis: AsyncRedis = Depends(get_async_redis)) ->
↪  dict[str, str]:
    await redis.ping()
    return {"status": "ok", "redis": "connected"}
```

Key points about connection management:

- Create a single connection pool at application startup and share it across all requests. Do not create new pools or clients per request.
- Set max_connections based on your expected concurrency. A typical range is 5 to 20 for most applications. Too high wastes file descriptors; too low causes blocking when all connections are in use.
- Use health_check_interval to periodically validate pooled connections and remove stale ones.

- Always close the connection pool on application shutdown to release resources cleanly.

Connection Pooling and Best Practices

Connection pooling is essential for production Redis clients because establishing new TCP connections has overhead: DNS resolution, TCP handshake, TLS negotiation (if enabled), and authentication. For high-throughput AI applications making dozens or hundreds of Redis calls per request, this overhead adds up quickly.

redis-py manages connection pooling automatically when you create a Redis client. The pool creates connections lazily as needed, reuses them across operations, and maintains them in a ready state for immediate use.

Here are practical guidelines:

1. **One pool per application process:** Do not share pools across processes. Each Python process should have its own pool. In web frameworks like FastAPI or Django, create the pool once at startup as shown above.
2. **Size pools appropriately:** The optimal pool size depends on your concurrency model and Redis capacity. A good starting formula is: pool_size = number_of_application_workers * 2 to 5. Monitor connected_clients in Redis with INFO clients and adjust if you see connection exhaustion errors.
3. **Use BlockingConnectionPool under heavy load:** By default, redis-py raises an error when the pool is exhausted. BlockingConnectionPool instead blocks waiting for a free connection:

```python
from redis.asyncio import BlockingConnectionPool

pool = BlockingConnectionPool.from_url(
    "redis://localhost:6379/0",
    max_connections=20,
    timeout=5  # Wait up to 5 seconds for a free connection
)
```

4. **Handle transient failures with retries:** Network blips happen. Configure retry_on_timeout and implement application-level retry logic for critical operations:

```python
from tenacity import retry, stop_after_attempt, wait_exponential,
↪  retry_if_exception_type
from redis.exceptions import ConnectionError as RedisConnectionError,
↪  TimeoutError as RedisTimeoutError

@retry(
    stop=stop_after_attempt(3),
    wait=wait_exponential(multiplier=0.1, max=2),
    retry=retry_if_exception_type((RedisConnectionError, RedisTimeoutError))
)
async def resilient_get(redis: AsyncRedis, key: str) -> str | None:
    """Get a value with automatic retry on transient failures."""
    return await redis.get(key)
```

5. **Monitor your connections:** In production, track connection pool utilization and Redis client count as part of your observability strategy. Sudden spikes in connections often indicate a problem: a leak where connections are not being returned to the pool, or an unexpected traffic surge.

Core Data Structures: Strings, Hashes, Lists, Sets, Sorted Sets

Redis provides several data structures that map well to AI application needs. Understanding when to use each one is critical for building efficient systems.

Strings: The simplest Redis type. Stores binary-safe data up to 512 MB. In Python with decode_responses=True, you work with regular strings:

```python
# Basic string operations
await redis.set("user:1001:preferences", '{"language": "en", "model":
↪  "gpt-4"}')
value = await redis.get("user:1001:preferences")  # Returns the JSON string

# Atomic increment/decrement (useful for counters)
await redis.incr("api:request_count")            # Increment by 1
await redis.incrby("api:token_usage", 1500)      # Increment by specific amount
await redis.decr("llm:remaining_calls")          # Decrement by 1

# Set with expiration (TTL) for caching
await redis.setex("cache:embedding:doc_42", 3600, embedding_bytes)  # Expires
↪  in 1 hour

# Conditional set (useful for distributed locks and cache-aside patterns)
acquired = await redis.set("lock:process_batch", "worker_1", nx=True, ex=60)
if acquired:
    # We got the lock, do work
    pass
```

Naming conventions matter in Redis. Use colons as namespace separators to organize keys by domain: `service:entity:id` or `cache:type:key`. This makes it easier to manage keys, run pattern-based operations, and reason about your data model.

Hashes: Store field-value pairs under a single key. Ideal for objects with multiple attributes like user profiles, document metadata, or cached LLM responses:

```python
# Store a complete object as a hash
await redis.hset("user:1001", mapping={
    "name": "Alice Chen",
    "email": "alice@example.com",
    "tier": "premium",
    "last_active": "2026-08-08T14:30:00Z",
    "preferences": '{"model": "claude-3.5", "temperature": 0.7}'
})

# Get individual fields without deserializing the whole object
name = await redis.hget("user:1001", "name")  # "Alice Chen"

# Get multiple fields at once (single round trip)
fields = await redis.hmget("user:1001", "name", "tier", "last_active")

# Update specific fields without affecting others
await redis.hset("user:1001", "last_active", "2026-08-08T15:45:00Z")

```

```
# Check if a field exists
has_email = await redis.hexists("user:1001", "email")

# Delete specific fields
await redis.hdel("user:1001", "last_active")

# Get all fields (use carefully for large hashes)
all_fields = await redis.hgetall("user:1001")
```

For AI applications, hashes are particularly useful for storing document metadata alongside vector data. When you create a vector index in Redis Stack, the underlying documents are typically stored as hashes with fields for content, embedding (as binary), and searchable metadata.

Lists: Ordered collections of strings supporting push/pop from both ends. Useful for queues, recent activity tracking, and conversation history:

```
# Use lists as FIFO queues (push right, pop left)
await redis.rpush("queue:inference_requests", request_id_1)
await redis.rpush("queue:inference_requests", request_id_2)
request_id = await redis.lpop("queue:inference_requests")  # Get oldest

# Use lists as stacks (push and pop from same end)
await redis.rpush("stack:recent_searches:user_1001", "Redis caching patterns")
await redis.rpush("stack:recent_searches:user_1001", "HNSW tuning guide")

# Keep only the most recent N items (useful for conversation history)
for message in conversation_messages:
    await redis.rpush(f"chat:history:{session_id}", json.dumps(message))
await redis.ltrim(f"chat:history:{session_id}", -100, -1)  # Keep last 100
↪  messages

# Get a range of items (e.g., last 20 messages in a conversation)
messages = await redis.lrange(f"chat:history:{session_id}", -20, -1)
parsed_messages = [json.loads(m) for m in messages]

# Block until an item is available (for consumer patterns)
result = await redis.blpop("queue:inference_requests", timeout=5)
if result:
    queue_name, request_id = result
```

Sets: Unordered collections of unique strings. Useful for tracking distinct values, tagging, and membership checks:

```python
# Add items to a set (duplicates are ignored automatically)
await redis.sadd("tags:document_42", "machine-learning", "python", "redis")
await redis.sadd("tags:document_42", "machine-learning")  # No effect, already exists

# Check membership in O(1) time
is_tagged = await redis.sismember("tags:document_42", "python")  # True

# Get all members
all_tags = await redis.smembers("tags:document_42")

# Set operations (useful for recommendation and filtering)
# Users who liked both document A and document B
mutual_likes = await redis.sinter("likes:doc_a", "likes:doc_b")

# All users who liked either document
all_likes = await redis.sunion("likes:doc_a", "likes:doc_b")

# Remove items
await redis.srem("tags:document_42", "machine-learning")
```

Sorted Sets: Like sets but each member has an associated score that determines its position. This makes sorted sets ideal for leaderboards, priority queues, time-series data, and ranked results:

```python
# Add members with scores
await redis.zadd("leaderboard:api_calls", {
    "user:1001": 1500,
    "user:1002": 890,
    "user:1003": 2340
})

# Get top N (highest scores first)
top_users = await redis.zrevrange("leaderboard:api_calls", 0, 9, withscores=True)

# Get range by score (useful for time-based queries)
# Store timestamps as scores to query by time window
now = time.time()
recent_events = await redis.zrangebyscore(
    "events:user_activity",
    min=now - 3600,  # Last hour
    max=now,
    withscores=True
)

# Use sorted sets as priority queues (lower score = higher priority)
await redis.zadd("queue:inference_priority", {
```

```
    f"request:{uuid4()}": time.time()  # Score is submission time for FIFO
})
next_request = await redis.zpopmin("queue:inference_priority")  # Get highest
↪  priority

# Count members in a score range
active_users_last_hour = await redis.zcount(
    "events:user_activity",
    min=now - 3600,
    max=now
)
```

For AI applications, sorted sets are frequently used for:

- Tracking API usage and rate limiting (store request timestamps as scores)
- Ranking retrieved documents by relevance score
- Maintaining time-ordered event logs for audit trails
- Implementing priority queues for inference requests

Transactions, Pipelines, and Lua Scripting

Redis provides three mechanisms for executing multiple commands efficiently: transactions, pipelines, and Lua scripts. Understanding their differences is important for building correct and performant AI applications.

Pipelines: Batch multiple commands into a single network round trip without atomicity guarantees. This dramatically reduces latency when you need to execute many commands:

```
# Without pipeline: N round trips
for i in range(100):
    await redis.set(f"cache:item:{i}", f"value_{i}")

# With pipeline: 1 round trip for all 100 commands
async with redis.pipeline() as pipe:
    for i in range(100):
        pipe.set(f"cache:item:{i}", f"value_{i}")
    results = await pipe.execute()  # All commands sent at once

# Pipelines also work for reads
async with redis.pipeline() as pipe:
    pipe.get("user:1001:name")
    pipe.get("user:1001:tier")
    pipe.hgetall("user:1001:preferences")
    name, tier, prefs = await pipe.execute()
```

For AI applications, pipelines are essential when:

- Storing batches of embeddings during document ingestion
- Updating multiple cache entries in a single request
- Fetching user preferences and session data together
- Bulk operations on rate-limit counters

Transactions (MULTI/EXEC): Group commands so they execute sequentially without interleaving from other clients. Redis transactions do not provide rollback (if one command fails, others still execute), but they do guarantee atomicity in the sense that no other client's commands will run between MULTI and EXEC:

```
async with redis.pipeline(transaction=True) as pipe:
    await pipe.multi()
    await pipe.incr("counter:total_requests")
    await pipe.incr(f"counter:user:{user_id}")
    await pipe.expire(f"counter:user:{user_id}", 86400)
    results = await pipe.execute()
```

Transactions are useful when you need to ensure that a group of operations completes as a unit, such as updating counters and metadata together. For most AI application use cases, pipelines provide sufficient performance improvement without needing transactional guarantees.

Lua scripting: Execute scripts atomically on the Redis server. Lua scripts are the foundation for many advanced patterns including rate limiting, distributed locks with proper expiration handling, and complex conditional operations:

```
# Token bucket rate limiter implemented as a Lua script
TOKEN_BUCKET_SCRIPT = """
local key = KEYS[1]
local capacity = tonumber(ARGV[1])
local refill_rate = tonumber(ARGV[2])
local now = tonumber(ARGV[3])
local requested = tonumber(ARGV[4])

-- Get current state
local bucket = redis.call('HMGET', key, 'tokens', 'last_refill')
local tokens = tonumber(bucket[1])
local last_refill = tonumber(bucket[2])
```

```
13
14  if not tokens then
15      -- Initialize bucket
16      tokens = capacity
17      last_refill = now
18  end
19
20  -- Refill tokens based on elapsed time
21  local elapsed = now - last_refill
22  local new_tokens = elapsed * refill_rate
23  tokens = math.min(capacity, tokens + new_tokens)
24
25  local allowed = 0
26  if tokens >= requested then
27      tokens = tokens - requested
28      allowed = 1
29  end
30
31  -- Save state
32  redis.call('HMSET', key, 'tokens', tokens, 'last_refill', now)
33  redis.call('EXPIRE', key, math.ceil(capacity / refill_rate) + 60)
34
35  return {allowed, math.floor(tokens)}
36  """
37
38  # Register the script for efficient reuse
39  token_bucket_sha = redis.register_script(TOKEN_BUCKET_SCRIPT)
40
41  async def check_rate_limit(
42      redis: AsyncRedis,
43      client_id: str,
44      capacity: int = 100,
45      refill_rate: float = 10.0,  # tokens per second
46      requested: int = 1
47  ) -> tuple[bool, int]:
48      """Check if request is allowed under token bucket rate limit."""
49      now = time.time()
50      key = f"ratelimit:{client_id}"
51
52      result = await token_bucket_sha(
53          keys=[key],
54          args=[capacity, refill_rate, now, requested]
55      )
56
57      allowed = bool(result[0])
58      remaining = int(result[1])
59      return allowed, remaining
60
61
62  # Usage in a FastAPI middleware
```

```python
from fastapi import Request, HTTPException
from starlette.middleware.base import BaseHTTPMiddleware

class RateLimitMiddleware(BaseHTTPMiddleware):
    async def dispatch(self, request: Request, call_next):
        client_id = request.client.host if request.client else "unknown"

        allowed, remaining = await check_rate_limit(
            async_redis_client,
            client_id,
            capacity=100,      # 100 tokens in bucket
            refill_rate=10.0   # Refill 10 per second
        )

        if not allowed:
            raise HTTPException(
                status_code=429,
                detail="Rate limit exceeded. Try again later.",
                headers={"Retry-After": "5"}
            )

        response = await call_next(request)
        response.headers["X-RateLimit-Remaining"] = str(remaining)
        return response
```

Lua scripts execute atomically on the Redis server, which means they are immune to race conditions. This makes them ideal for patterns that require read-modify-write operations across multiple keys or complex conditional logic. The trade-off is that long-running scripts block other commands, so keep scripts short and efficient.

Persistence Options: RDB and AOF

Redis is an in-memory database, which means all data lives in RAM by default. If the process stops or the machine loses power, everything is lost unless persistence is configured. Redis offers two persistence mechanisms: RDB snapshots and AOF (Append Only File) logging.

RDB (Redis Database): Creates point-in-time snapshots of the dataset at configurable intervals. The snapshot is a compact binary file that can be used to restore the database.

```python
# Trigger an RDB snapshot manually
await redis.bgsave()  # Background save: does not block Redis

# Check last save status
info = await redis.info("persistence")
last_save = info.get("rdb_last_save_time", 0)
print(f"Last RDB snapshot: {datetime.fromtimestamp(last_save)}")
```

RDB configuration in redis.conf:

```
# Save when at least N keys changed within M seconds
save 900 1      # Save after 900 seconds (15 min) if at least 1 key changed
save 300 10     # Save after 300 seconds (5 min) if at least 10 keys changed
save 60 10000   # Save after 60 seconds if at least 10,000 keys changed

# Disable RDB entirely (not recommended for production)
# save ""
```

Pros of RDB:

- Compact binary format suitable for backups and disaster recovery
- Fast database restarts because loading a single file is efficient
- Minimal performance impact since snapshots run in background processes

Cons of RDB:

- Data loss risk: you can lose up to the last snapshot interval (e.g., 15 minutes) if Redis crashes
- Snapshot creation forks the process, which can cause brief pauses on large datasets

AOF (Append Only File): Logs every write operation to a file. On restart, Redis replays the log to reconstruct the dataset.

```
# Enable AOF persistence
appendonly yes

# Fsync policy: how often to flush to disk
# always = sync after every write (safest but slowest)
# everysec = sync once per second (recommended default)
# no = let OS decide when to flush (fastest but riskiest)
appendfsync everysec

# Use RDB preamble for faster startup (Redis 7+ recommended)
aof-use-rdb-preamble yes
```

Pros of AOF:

- Better durability: with everysec fsync, you lose at most one second of data on crash
- Human-readable log format (unless compressed) makes debugging easier
- No fork overhead like RDB snapshots

Cons of AOF:

- Files grow larger than RDB snapshots over time
- Slightly slower writes due to disk I/O
- Requires periodic rewriting (automatic with auto-aof-rewrite-percentage and auto-aof-rewrite-min-size settings)

Recommendation for AI applications: Use both RDB and AOF together. This is the default in modern Redis configurations. RDB provides fast restarts and compact backups, while AOF minimizes data loss. The hybrid mode (aof-use-rdb-preamble yes) stores an RDB snapshot as a preamble in the AOF file, combining fast startup with incremental logging [5].

For pure caching workloads where data can be regenerated, persistence may not matter. But for AI applications storing embeddings, conversation history, agent memory, and rate-limit state, you typically want at least some durability to avoid costly recomputation on restart.

Summary

This chapter covered the Redis fundamentals essential for building AI applications: installation with Docker, connecting via redis-py with proper connection

pooling, the core data structures (strings, hashes, lists, sets, sorted sets), and mechanisms for efficient multi-command execution (pipelines, transactions, Lua scripts). We also discussed persistence options and their trade-offs.

These foundations support everything that follows. In the next chapter, we will apply these concepts to caching patterns specifically designed for AI workloads: reducing LLM latency, cutting costs through response caching, and implementing semantic caching that understands natural language queries.

Chapter 2: Caching Patterns for AI Workloads

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

The Cost of Not Caching: Latency and Economics

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

Basic Caching with TTLs and Eviction Policies

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

Caching LLM Responses and Embeddings

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

Semantic Caching for Natural Language Queries

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

Rate Limiting AI Endpoints

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

Cache Invalidation Strategies for AI Systems

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

Summary

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

Chapter 3: Redis Stack, Search, and Vector Indexes

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

What Is Redis Stack?

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

Installing and Configuring Redis Stack

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

Creating Vector Indexes with HNSW and FLAT

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

Storing and Querying Embeddings

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

Hybrid Search: Combining Vectors with Metadata Filters

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

Performance Tuning Vector Queries

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

Summary

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

Chapter 4: Building RAG Systems with Redis

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

What Is RAG and Why It Matters

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

Document Ingestion Pipeline with Python

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

Chunking Strategies for Different Content Types

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

Generating Embeddings with sentence-transformers

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

Building the Retrieval Layer with Redis Vector Search

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

Integrating Retrieval with LLMs via FastAPI

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

Evaluating and Improving Retrieval Quality

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

Summary

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

Chapter 5: LLM Application Patterns with Redis

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

Managing Conversation History at Scale

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

Streaming Responses Through Redis Pub/Sub

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

Prompt Template Storage and Versioning

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

Tool Use and Function Calling Coordination

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

Multi-Agent Workflow Orchestration

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

Tracking Usage, Costs, and Tokens

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

Summary

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

Chapter 6: AI Agents, Memory, and Context Management

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

The Memory Problem in AI Agents

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

Short-Term vs Long-Term Memory Architectures

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

Episodic Memory with Vector Search

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

Semantic Memory and Knowledge Retrieval

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

Cross-Agent Shared Memory

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

Building an Agent Memory Layer in Python

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

Summary

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

Chapter 7: Real-Time AI with Redis Streams and Pub/Sub

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

Why Real-Time Matters for AI

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

Redis Pub/Sub vs Streams

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

Building Event-Driven Inference Pipelines

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

Consumer Groups for Scalable Processing

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

Real-Time Chat with LLM Backends

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

Streaming Data Enrichment with AI Models

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

Summary

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

Chapter 8: Session Management and State for AI Applications

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

Session Storage with Redis Hashes

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

Authentication and Token Management

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

Feature Flags and Experimentation for AI Features

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

Personalization and User Profiles

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

Distributed Locks for Concurrency Control

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

Handling State Failures and Recovery

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

Summary

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

Chapter 9: Scaling Redis for Production AI Workloads

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

Understanding Redis Cluster Architecture

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

Sharding Strategies for Vector Workloads

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

Replication and Read Replicas

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

Failover and High Availability

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

Capacity Planning for AI Applications

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

Scaling Patterns: From Single Node to Global Deployment

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

Summary

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

Chapter 10: Performance Optimization and Tuning

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

Memory Management and Optimization

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

Choosing the Right Data Structure for the Job

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

Optimizing Vector Search Performance

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

Pipelines and Batch Operations

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

Async Python with Redis (asyncio patterns)

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

Profiling and Diagnosing Performance Issues

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

Summary

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

Chapter 11: Security, Observability, and Reliability

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

Authentication and Access Control

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

Encryption in Transit and at Rest

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

Network Security and Isolation

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

Monitoring with Metrics, Logs, and Traces

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

Alerting on Critical Conditions

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

Backup Strategies and Disaster Recovery

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

Summary

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

Chapter 12: Production Deployment Patterns

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

Managed Redis vs Self-Hosted: Decision Framework

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

Deploying with Docker and Docker Compose

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

Kubernetes Deployment with Helm

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

Configuration Management for Environments

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

CI/CD Integration and Automated Testing

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

Vendor Landscape: Comparing Options in 2026

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

Summary

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

Chapter 13: End-to-End Architecture Patterns

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

Reference Architecture: Production RAG Platform

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

Reference Architecture: Real-Time AI Chat System

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

Reference Architecture: Multi-Agent Orchestration

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

Reference Architecture: AI-Powered Search Engine

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

Migration Patterns for Existing Systems

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

Choosing Your Stack: A Decision Guide

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

Summary

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

Conclusion: The Future of Redis in AI Infrastructure

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

What We Built

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

Where Redis Is Headed

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

The Evolving AI Infrastructure Landscape

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

Final Recommendations

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

References

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