

LOCAL FIRST SOFTWARE DEVELOPMENT

BUILDING SOFTWARE THAT WORKS WITHOUT THE CLOUD



FULLY FUNCTIONAL OFFLINE

Works without an
internet connection



OPTIONAL SYNCHRONIZATION

Cloud is only a
sync layer



SECURE BY DESIGN

Privacy and security
built in



BUILT TO SCALE

From personal tools
to large collaborative
systems

ALEXEI KOWALSKI

Local First Software Development

Building Software That Works Without the Cloud

Steve Publications

This book is available at <https://leanpub.com/localfirstsoftwaredevelopment>

This version was published on 2026-08-30



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.

© 2026 Steve Publications

Contents

Building Software That Works Without the Cloud	1
Introduction: The End of Always-On Software	2
When the Cloud Goes Away	2
What Local First Actually Means	3
Why This Book Exists	4
Chapter 1: Foundations and the Evolution from Offline-First to Local-First	8
The Evolution of Offline Support	8
Offline First Versus Local First	9
Edge Computing and Distributed Systems	11
Peer-to-Peer Architectures	12
Why Now: Latency, Privacy, and Resilience	13
Chapter 2: Local Data Ownership and Embedded Databases	16
The Primacy of Local Data	16
SQLite: The World’s Most Deployed Database	17
Alternatives and Specialized Stores	20
Browser Storage and IndexedDB	22
Designing Your Local Schema	24
Chapter 3: Building the Core Application Locally	28
Project Setup and Architecture Decisions	28
The Local Data Model	28
CRUD Operations Without a Server	28
Optimistic Updates and Immediate Feedback	28
Testing the Offline Experience	28
Chapter 4: Synchronization Fundamentals	29
What Synchronization Actually Is	29

CONTENTS

The Conflict Problem	29
Consistency Models in Practice	29
Designing a Sync Protocol	29
Chapter 5: Conflict Detection and Resolution	30
Types of Conflicts	30
Last-Write-Wins and Its Pitfalls	30
Version Vectors and Causal Ordering	30
Tombstones and Logical Deletion	30
Idempotent Operations	30
Chapter 6: Conflict-Free Replicated Data Types (CRDTs)	31
The Mathematics of Convergence	31
State-Based Versus Operation-Based CRDTs	31
Common CRDT Types	31
Yjs: A Practical CRDT Engine	31
Automerge and Document-Oriented CRDTs	31
When CRDTs Help and When They Do Not	31
Chapter 7: Event Sourcing, Operational Transformation, and Alternatives	33
Event Sourcing Basics	33
Operational Transformation Explained	33
OT Versus CRDTs	33
Hybrid Approaches	33
Choosing Your Concurrency Model	33
Chapter 8: Multi-Device Synchronization in Practice	34
Architecture of a Sync Backend	34
Change Streams and Incremental Sync	34
Durable Queues for Reliable Delivery	34
Background Synchronization	34
Connectivity Detection and Retry Strategies	34
Handling Partial Failures	34
Chapter 9: Peer-to-Peer Communication and Real-Time Collaboration	36
Direct Device-to-Device Communication	36
WebRTC for Local-First Applications	36
Mesh Networks and Gossip Protocols	36
Real-Time Presence and Cursors	36
Trade-offs of Pure P2P	36

Chapter 10: Security, Encryption, and Privacy 37

Authentication Without a Central Authority 37

End-to-End Encryption for Local Data 37

Key Management and Rotation 37

Secure Sharing and Access Control 37

Privacy by Design 37

Chapter 11: Platform-Specific Implementation Details 38

Web Applications and Service Workers 38

Desktop Applications 38

Mobile Platforms 38

Edge Runtimes and WASM 38

Cross-Platform Data Sharing 38

Chapter 12: Testing, Debugging, and Observability 39

Testing Offline Behavior 39

Simulating Network Partitions 39

Testing Synchronization Correctness 39

Debugging Distributed State 39

Observability Without the Cloud 39

Chapter 13: Operations, Performance, and Production Deployment . . . 40

Storage Management and Cleanup 40

Caching Strategies 40

Schema Evolution and Migrations 40

Backup and Recovery 40

Performance Optimization 40

Deployment Patterns 40

Chapter 14: Hybrid Architectures and When the Cloud Is Optional . . . 42

The Hybrid Philosophy 42

Cloud-Assisted Discovery and Coordination 42

Optional Backup Layers 42

Technology Landscape Overview 42

Architectural Decision Framework 42

Conclusion: Designing for Resilience 43

The Local First Checklist 43

What We Have Learned 43

The Future of Offline-Native Software 43

References 44

 Official Documentation 44

 CRDT Libraries and Projects 44

 Local-First Frameworks and Sync Engines 44

 Foundational Papers and Articles 44

 Books 44

 Distributed Systems Concepts 44

 Technical Articles and Resources 45

 Verification and Testing Research 45

 Industry and Community 45

Building Software That Works Without the Cloud

This book teaches you how to design, build, and operate software that treats the user's device as the primary source of truth. The application is fully functional offline, with the cloud serving only as an optional synchronization layer. Through complete working examples, deep technical explanations, and production-ready patterns, you will learn the distributed-systems fundamentals, data architecture, synchronization protocols, security mechanisms, and engineering practices required to build local-first applications that scale from personal tools to collaborative systems used by thousands of users across devices.

Introduction: The End of Always-On Software

When the Cloud Goes Away

It is a Tuesday afternoon when your team's project management tool goes dark. Not gradually, not with degraded performance. Just gone. Someone in your organization clicked "save" on a configuration change that brought down the authentication service, and now nobody can access their tasks, notes, or timelines. You stare at a spinning loading indicator that will never complete. The work stops.

This is not hypothetical. Major cloud outages happen regularly enough that engineers have started tracking them as a metric: AWS has suffered over 50 significant incidents since 2014, Google Cloud and Azure have similar records, and each outage costs affected organizations hundreds of thousands or millions of dollars in lost productivity [1]. But the real damage is not just financial. It is the erosion of trust: the growing awareness that your ability to do your work depends on infrastructure you do not control, operated by companies whose priorities may not align with yours.

Consider a different scenario: a field technician working on an aircraft maintenance inspection in a hangar with poor cellular coverage. The cloud-based inspection app loads slowly, times out on every form submission, and eventually displays "no connection" errors. The technician switches to paper forms (the backup that should never have been necessary). Later, when connectivity returns, someone has to manually re-enter all those paper records into the system.

Or think about a writer using a cloud-based note-taking application during a long-haul flight. They can read their existing notes in cached form, but every new thought they type sits in limbo, unsaved locally, queued for upload that will not happen for six hours. The interface hints at this uncertainty with subtle warning icons and "sync pending" badges that create cognitive friction with every keystroke.

These scenarios share a common root cause: the software was designed around an assumption that is increasingly false. It assumes continuous, reliable network connectivity to centralized servers. This is what we call cloud-first architecture, and it has become so dominant that many developers no longer question it. They build applications as if the internet were always on, always fast, and always available. In reality, it is none of those things, consistently and predictably.

What Local First Actually Means

Local-first software rejects this assumption at the architectural level. In a local-first application, the primary copy of all data resides on the user's device. The application reads from and writes to local storage first, providing immediate responsiveness regardless of network conditions. The cloud, when used at all, serves only as an optional synchronization layer, a mechanism for sharing data across devices or with other users, not a mandatory dependency for basic functionality.

This is not merely “offline support” tacked onto a cloud-dependent application. Offline-first and local-first are related but distinct concepts. Offline-first applications cache data locally to tolerate temporary disconnections, but they still treat the server as the authoritative source of truth: the system that decides what data is correct, who can access it, and what happens when conflicts arise. Local-first applications invert this relationship: the local device is always the source of truth for its user, and synchronization is a secondary concern handled through carefully designed protocols rather than simple overwrite semantics.

The term “local-first” was coined in 2019 by researchers at Ink & Switch, an industrial research lab co-founded by Adam Wiggins (who previously co-founded Heroku), along with Martin Kleppmann from the University of Cambridge [2]. Their manifesto identified seven ideals that local-first software should strive for:

Fast. Operations respond without network round-trips. When you type a character, click a button, or save a document, the feedback is immediate because the data lives on your device, not in a data center hundreds of miles away.

Multi-device. Your data synchronizes across all your devices seamlessly.

Start writing on your phone during your commute, continue on your laptop at work, review on your tablet at home, without manual export or import.

Offline. The application works fully when disconnected from the network. This is not a degraded mode with limited features; it is the primary mode. Connectivity is an enhancement, not a prerequisite.

Collaboration. Multiple people can work together on shared data in real time, just as they would in cloud-based tools like Google Docs or Figma. Local-first does not mean working alone.

Longevity. Your data remains accessible even if the service provider goes out of business. Because you own the primary copy locally, you are not held hostage by vendor lock-in or arbitrary shutdown decisions.

Privacy. End-to-end encryption ensures that only you and your chosen collaborators can read your data, not the service providers, not advertisers, not government agencies conducting bulk surveillance. Your data is encrypted before it leaves your device.

User control. You decide where your data lives, who can access it, and how long it persists. You are the administrator of your own information, not a tenant in someone else's system.

These ideals form a gradient rather than a strict checklist. No existing application achieves all seven perfectly, and different applications will prioritize them differently [3]. But they provide a north star for design decisions that fundamentally reshape how we think about software architecture.

Why This Book Exists

The local-first movement has gained significant momentum since 2019. The first Local-First Conference took place in Berlin in May 2024, attracting researchers and engineers from around the world [4]. Major technology companies have begun adopting local-first principles: Apple Notes uses CRDT-based synchronization for cross-device editing [5], Figma's multiplayer features rely on sophisticated conflict-free data structures [6], and emerging frameworks like ElectricSQL, PowerSync, RxDB, Yjs, and Automerge are making it easier to build local-first applications without reinventing the wheel.

Yet despite this growing interest, there is no comprehensive technical guide that takes a professional software engineer from first principles through

production deployment of a local-first system. Existing resources fall into three categories: academic papers that explain CRDTs formally but provide little practical guidance, product documentation for specific frameworks that assumes you already understand the underlying concepts, and scattered blog posts that cover individual pieces without showing how they fit together.

This book fills that gap. It is designed for professional software engineers who have general programming knowledge but may come from cloud-centric backgrounds: developers who are used to thinking in terms of REST APIs, centralized databases, and server-side authorization, and who need to learn distributed-systems concepts as applied to local-first design.

You will not find isolated code snippets here. Instead, you will follow a running example application called TaskSync through the entire book. It starts as a simple local-only task manager using SQLite, then progressively adds multi-device synchronization, real-time collaboration with CRDTs, peer-to-peer communication via WebRTC, end-to-end encryption, and cross-platform deployment for web, desktop, and mobile. Each chapter builds on the previous ones, so by the end you will have seen a complete system evolve from concept to production-ready implementation.

The book is organized in a logical progression:

- **Foundations** (Chapters 1–2) establish terminology and historical context, distinguishing local-first from related paradigms like offline-first, edge computing, and peer-to-peer architectures. You will learn why local data ownership matters and explore embedded database options with deep focus on SQLite as the industry standard.
- **Core Implementation** (Chapter 3) begins the running example with a fully local-only implementation, demonstrating proper data modeling, optimistic updates, and offline functionality before any synchronization is considered.
- **Synchronization Fundamentals** (Chapters 4–5) introduce the challenges of keeping data consistent across devices, what it means to synchronize, how conflicts arise when multiple users edit the same data offline, and strategies for detecting and resolving those conflicts using version vectors, causal ordering, tombstones, and idempotent operations.
- **Conflict-Free Data Structures** (Chapters 6–7) cover CRDTs as a mathematical foundation for conflict-free collaboration, explaining the theory behind state-based and operation-based approaches, surveying practical implementations like Yjs and Automerge, and comparing these with

complementary techniques like event sourcing and operational transformation.

- **Multi-Device Systems** (Chapter 8) extends the running example to support synchronization across devices, implementing a sync backend with durable queues, background synchronization, connectivity detection, retry strategies, and partial failure handling.
- **Peer-to-Peer Communication** (Chapter 9) covers direct device-to-device communication using WebRTC and gossip protocols, enabling real-time collaboration without routing all traffic through a central server.
- **Security** (Chapter 10) addresses authentication, authorization, end-to-end encryption, key management, and privacy in local-first systems: how to share data securely between users when the data lives on their devices, not your servers.
- **Platform Implementation** (Chapter 11) covers specific details for building local-first applications across web browsers with service workers and IndexedDB, desktop platforms with Tauri and Electron, mobile platforms with native SQLite integration, and edge runtimes.
- **Testing and Operations** (Chapters 12–13) explain how to test local-first applications for correctness under partition conditions, debug distributed behavior, manage storage and schema evolution, optimize performance, and deploy sync infrastructure in production.
- **Hybrid Architectures** (Chapter 14) shows how to integrate cloud services as optional enhancements for discovery, backup, and coordination without making them mandatory dependencies. It surveys the current technology landscape including ElectricSQL, PowerSync, libSQL/Turso, PouchDB/CouchDB, and others.

Throughout the book, I will distinguish stable principles from rapidly changing implementation details. The mathematical properties of CRDTs will not change next year; the specific JavaScript libraries implementing them might. Understanding the former makes you resilient to changes in the latter. Similarly, the trade-offs between consistency and availability are fundamental constraints of distributed systems that no amount of clever engineering can eliminate. Recognizing these constraints early saves months of debugging later.

Every code example is designed to be complete and runnable, with explicit version assumptions for languages, frameworks, and libraries. You will see project structures, configuration files, database schemas, migration scripts,

synchronization protocols, test suites, error handling patterns, security mechanisms, and deployment instructions: not as abstract concepts but as concrete implementations you can study, modify, and use as a foundation for your own work.

If you are tired of building applications that break when the network fails, tired of users losing data because a service shut down, or simply curious about how to build software that respects users' control over their information, this book is for you. Let us begin.

Chapter 1: Foundations and the Evolution from Offline-First to Local-First

The Evolution of Offline Support

To understand local-first architecture, we need to trace how our thinking about offline capability has evolved over the past two decades. This is not just a technical history; it is a story about shifting assumptions regarding connectivity, user agency, and the role of centralized infrastructure in software design.

In the early days of personal computing before the internet became ubiquitous, all software was inherently local-first by necessity. Word processors stored documents on your hard drive. Spreadsheet applications kept data in local files. Email clients like Outlook Express downloaded messages to your machine and managed them locally. There was no question about where data lived. It lived on the device you were using because that was the only place it could live.

When web-based applications emerged in the late 1990s and early 2000s, they brought a radical departure from this model. Applications like Gmail (launched in 2004) convinced users to store their email in browsers rather than local clients. Google Docs (originally Writely, acquired by Google in 2006) demonstrated that collaborative editing could happen through a web interface without any local software installation. The trade-off was clear: you gained the ability to access your data from anywhere and collaborate easily with others, but you lost direct control over where that data physically resided.

As cloud computing infrastructure matured throughout the 2010s, this model became dominant. Services like Dropbox (launched in 2007), Salesforce, Slack, Trello, and countless others stored all data on their servers. Local devices held only temporary caches, views into a centrally managed source of truth that existed in data centers operated by service providers. This

architecture simplified development dramatically: there was one authoritative copy of the data, one place to implement business logic, one location for security controls and backups.

The problems emerged gradually at first. Users noticed that their applications became unusable on airplanes or in areas with poor connectivity. They discovered that when services shut down (like Google Reader in 2013, or more recently numerous smaller SaaS products), their data disappeared along with the platform. They learned that “free” cloud storage often came with implicit costs: advertising profiles built from their usage patterns, data mining for product training, or compliance requirements that forced disclosure to government agencies.

By the mid-2010s, a counter-movement began forming. Developers started asking whether there was a way to have both worlds: the convenience of cloud-based collaboration and accessibility, combined with the data ownership and reliability of traditional desktop software. This led first to the offline-first movement, and eventually to local-first as a more comprehensive architectural philosophy.

Offline First Versus Local First

The terms “offline-first” and “local-first” are frequently used interchangeably in casual conversation, but they represent meaningfully different design philosophies with distinct technical implications. Understanding this distinction is crucial for making the right architectural choices.

Offline-first is primarily concerned with resilience to network failures. An offline-first application assumes that connectivity will be intermittent or unreliable and designs accordingly. It caches data locally so users can continue working when disconnected, queues write operations for later upload, and handles reconnection gracefully by synchronizing pending changes with the server. The key insight of offline-first design is that treating offline as an afterthought, adding caching only after the core online functionality works, produces poor user experiences and fragile applications.

Offline-first architecture still treats the server as the authoritative source of truth. Local storage is a cache, not the primary copy. When conflicts arise between local changes and server state, the server’s version typically wins, or conflicts are resolved using server-side rules that the client must accept.

Examples include progressive web apps (PWAs) that use service workers to cache API responses, mobile applications that store data in SQLite for offline access but sync with a central backend, and frameworks like Angular Material's offline support patterns.

Local-first goes further than offline-first by making local storage the authoritative source of truth for each user. The server is not the system of record; it is at most a relay or coordination point that helps synchronize data between devices and users. This shift has profound implications:

1. **Data ownership:** Because the primary copy lives on the user's device, the user genuinely owns their data. If the service provider shuts down, the data remains accessible locally. Exporting data is trivial because it never left: you already have it.
2. **Immediate responsiveness:** All operations are local first by definition. There is no network round-trip for reads or writes, so latency is measured in microseconds rather than milliseconds. The UI responds instantly to user input regardless of connectivity status.
3. **Conflict resolution philosophy:** Instead of the server deciding which version wins when conflicts occur, local-first systems use mathematical properties (like CRDTs) or negotiated protocols to ensure all replicas converge to a consistent state without arbitrary decisions by a central authority.
4. **Security model:** End-to-end encryption becomes natural rather than bolted on. If data is encrypted locally before synchronization, the sync infrastructure never sees plaintext; it only relays ciphertext between devices. This provides zero-knowledge security where even the service provider cannot read user data.
5. **Architecture complexity shifts:** Offline-first moves some complexity from the client to the server (the server handles conflict resolution). Local-first moves complexity into the synchronization protocol itself (designing convergence guarantees, managing encryption keys, handling peer discovery), but often simplifies the backend significantly because it does not need to be the authoritative data store.

The distinction matters most when considering what happens in edge cases. In an offline-first application where the server is down for three days, users can read cached data and queue writes, but they cannot collaborate effectively with each other because all coordination goes through the unavailable server. In a

local-first application, users might use peer-to-peer channels to share updates directly, or at minimum each user continues working on their own authoritative copy without any degradation in functionality.

Some practitioners argue that offline-first is a subset of local-first: any truly local-first application must also be offline-first by definition, since if the primary copy is local, the application must work when disconnected. Others maintain that you can build useful offline-first applications without the full complexity of local-first synchronization protocols. In practice, the boundary is fuzzy, and many applications fall somewhere on a spectrum between fully cloud-dependent and fully local-first.

Edge Computing and Distributed Systems

Edge computing, processing data closer to where it is generated rather than in centralized data centers, shares some goals with local-first architecture but addresses different problems. Understanding both helps clarify what makes local-first unique.

Traditional cloud computing centralizes computation and storage in large data centers, optimizing for economies of scale and operational efficiency. Edge computing pushes some computation and storage closer to users or devices: to regional data centers, cellular towers, IoT gateways, or even the devices themselves, primarily to reduce latency and bandwidth costs. Content delivery networks (CDNs) are perhaps the most familiar form of edge computing: they cache static content at locations geographically close to users so that a request for an image does not need to traverse the globe to reach the origin server.

Edge computing architectures typically maintain a centralized source of truth in the cloud while replicating subsets of data or computation to edge nodes. The edge serves as an optimization layer: faster reads, reduced latency for real-time processing, lower bandwidth costs. But the primary system remains centralized. When an edge node fails or goes offline, its cached data may become stale, and it must resynchronize with the central system when it comes back online.

Local-first architecture can be viewed as taking edge computing to its logical extreme: each user's device is an edge node that holds not a cache but the primary copy of their data. The distinction is subtle but important. In edge

computing, the edge optimizes access to centralized data. In local-first design, the “edge” (the user’s device) owns the data and uses centralized infrastructure only for coordination when needed.

This relationship has interesting implications for hybrid architectures. A system might use local-first principles on user devices (each device has its own authoritative copy of relevant data), edge computing for regional sync relays (reducing latency for synchronization traffic), and centralized cloud services for optional features like global search, analytics, or backup. Each layer serves a specific purpose without being a mandatory dependency for core functionality.

The broader context of distributed systems theory is also relevant here. Local-first applications are fundamentally distributed systems. Multiple replicas of data exist across devices; they must maintain consistency despite network partitions and concurrent updates, and they must handle failures gracefully. Concepts from distributed systems research: the CAP theorem, eventual consistency, consensus algorithms, vector clocks, failure detectors. All apply directly to local-first design.

The CAP theorem states that in a distributed system, you can guarantee at most two of three properties: consistency (all nodes see the same data at the same time), availability (every request receives a response), and partition tolerance (the system continues operating despite network failures) [7]. Local-first systems typically choose availability and partition tolerance (AP), accepting eventual consistency rather than strong consistency. This is not a limitation but a deliberate design choice: users need their applications to work reliably even when disconnected, and they can tolerate brief periods where different devices see slightly different versions of shared data as long as those differences resolve automatically over time.

Peer-to-Peer Architectures

Peer-to-peer (P2P) architectures provide another useful comparison point. In P2P systems, there is no central server; all participants are equal peers that communicate directly with each other. File-sharing networks like BitTorrent, distributed hash tables like Chord, and blockchain networks are all P2P systems.

P2P architectures excel at resilience and decentralization. There is no single point of failure because there is no central authority. Nodes can join and leave

freely without disrupting the overall system. Data is distributed across many participants, making it harder to censor or destroy.

However, pure P2P has significant practical challenges:

- **Discovery:** How do peers find each other without a central directory? Bootstrap nodes help but introduce partial centralization.
- **NAT traversal:** Most devices sit behind network address translation (NAT) routers that block incoming connections, making direct peer communication difficult without techniques like STUN/TURN or hole punching.
- **Access control:** Enforcing fine-grained permissions in a decentralized system is complex. Who decides who can read or write what?
- **Reliability:** Peers are ephemeral: they go offline, lose data, or disconnect unexpectedly. Ensuring durable storage requires redundancy across many nodes.
- **Security:** Without central authority, detecting and mitigating malicious peers is harder. Sybil attacks (creating many fake identities) become a concern.

Local-first applications can incorporate P2P communication as an optional channel for synchronization (WebRTC data channels enable direct browser-to-browser communication, for example). But most practical local-first systems use a hybrid approach: primarily client-server communication for simplicity and reliability, with P2P as an enhancement for specific scenarios like real-time collaboration sessions or low-latency sync between devices on the same network.

The key insight is that local-first does not require pure P2P architecture. What it requires is that each user's device maintains its own authoritative copy of data, regardless of how synchronization happens. The communication topology (client-server, mesh, hybrid) is an implementation detail orthogonal to the local-first principle.

Why Now: Latency, Privacy, and Resilience

The local-first movement has gained traction now (five years after its 2019 manifesto) for several converging reasons that make the timing particularly favorable.

Latency expectations have increased dramatically. Users are accustomed to instant feedback from their applications. A 2018 study by Bailis et al. measured server-to-server round-trip times between AWS data centers worldwide and found latencies ranging from 15 milliseconds within a region to over 300 milliseconds across continents [8]. For interactive applications, even 150 milliseconds of delay is perceptible and frustrating. Local-first architecture eliminates this latency for all local operations because the data lives on the device. Writes are acknowledged in microseconds, reads return immediately, and UI updates happen without waiting for network responses.

Privacy regulations have created new constraints. The General Data Protection Regulation (GDPR) in Europe, California Consumer Privacy Act (CCPA), and similar laws worldwide impose strict requirements on how personal data is collected, stored, processed, and shared. Local-first architecture with end-to-end encryption makes compliance easier because the service provider never sees plaintext user data; they only handle encrypted blobs. This reduces regulatory risk and liability while giving users genuine privacy guarantees rather than policy-based promises.

Cloud costs have become unsustainable for many organizations. The “pay-per-use” model of cloud computing sounds attractive but often leads to runaway costs as applications scale. Database queries, API calls, data transfer, compute time, everything accumulates charges. Local-first architecture shifts computation and storage to user devices (which users already own and operate), dramatically reducing backend infrastructure costs. As Martin Kleppmann noted at Local-First Conf 2024, local-first design eliminates the need for large backend engineering teams, 24/7 on-call rotations, and expensive cloud computing bills [9].

Resilience concerns have grown. The COVID-19 pandemic demonstrated both the value and fragility of cloud-dependent infrastructure. Organizations that relied entirely on centralized services experienced disruptive outages when those services failed under unprecedented load or suffered configuration errors. Local-first applications provide inherent resilience: each device continues functioning independently regardless of what happens to central infrastructure.

Technology maturity has reached a tipping point. Five years ago, building local-first applications required deep expertise in distributed systems and custom implementations of synchronization protocols. Today, production-ready libraries and frameworks make it accessible to more developers:

- CRDT libraries like Yjs (with over 14,000 GitHub stars) and Automerge provide battle-tested conflict-free data structures [10][11].
- Sync engines like ElectricSQL, PowerSync, and RxDB handle the complexity of keeping local databases in sync with backends [12][13][14].
- Web APIs like IndexedDB, Service Workers, and WebRTC enable sophisticated offline and peer-to-peer capabilities directly in browsers.
- Desktop frameworks like Tauri make it easy to bundle local-first web applications into native executables with access to SQLite and system resources.

None of these technologies alone makes local-first possible; the concept existed before any of them. But together they form an ecosystem that lowers the barrier to entry significantly, making local-first a practical choice rather than a research curiosity.

The convergence of these factors suggests that local-first is not a passing trend but a fundamental shift in how we think about software architecture. As users become more aware of their data rights, as cloud costs continue rising, and as technology tools mature further, the proportion of applications that benefit from local-first design will only grow. The question for engineers is no longer whether local-first matters but how to apply its principles effectively to real-world problems.

In the next chapter, we will dive into the technical foundation of local-first architecture: local data ownership and embedded databases. We will explore why SQLite has become the de facto standard for client-side storage, examine alternatives and their trade-offs, and learn how to design schemas that work well in a distributed, eventually-consistent environment.

Chapter 2: Local Data Ownership and Embedded Databases

The Primacy of Local Data

Local data ownership is not merely a philosophical principle; it is the architectural foundation upon which everything else in local-first software rests. Every other characteristic: speed, offline capability, multi-device sync, privacy, longevity. All flow from the decision to store the primary copy of data on the user's device rather than on a remote server.

In cloud-first architecture, the flow of data follows a simple pattern: the user interacts with an interface that sends requests to a server, the server reads or modifies data in its database, and returns results to the client. The client is essentially a thin view layer with minimal local state. This works well when connectivity is reliable and latency is acceptable, but it creates fundamental dependencies: the application cannot function without the server, and the user has no direct access to their data outside the service provider's ecosystem.

Local-first architecture inverts this pattern. The flow becomes: the user interacts with an interface that reads and writes directly to local storage on the device, and synchronization with other devices or users happens asynchronously in the background through a separate mechanism. The server is not part of the critical path for any operation. This shift requires fundamentally different thinking about data management.

The implications are substantial:

Performance. Local storage access is orders of magnitude faster than network round-trips. Reading from an SSD takes microseconds; reading from a remote database typically takes tens to hundreds of milliseconds depending on distance and load. For interactive applications, this difference transforms the user experience from “waiting for the server” to “instant response.”

Reliability. Local storage is always available as long as the device is powered on. It does not depend on network connectivity, server uptime, or API

availability. Applications that read and write locally can continue functioning during outages, in airplane mode, or in areas with poor coverage.

Data sovereignty. When data lives on the user's device, the user genuinely controls it. Exporting is trivial because the data already exists in a local file that the user can copy. Importing to another application requires only reading that file. There is no vendor lock-in through API restrictions or export format limitations.

Security. Local-first architecture enables end-to-end encryption naturally. Data can be encrypted on the device before any synchronization occurs, ensuring that intermediate infrastructure (sync servers, CDNs, ISPs) never sees plaintext. This provides zero-knowledge security where even the service provider cannot read user data.

Scalability. Each user's device handles its own storage and computation, distributing the load across millions of devices instead of concentrating it in centralized infrastructure. This dramatically reduces backend costs for read-heavy workloads because most reads are served locally rather than through API calls to a central database.

The trade-off is complexity in synchronization. When every device has its own copy of data, keeping those copies consistent requires careful protocol design: handling concurrent edits, detecting conflicts, ensuring eventual convergence, managing encryption keys across devices. This complexity is the price we pay for resilience and user control, and subsequent chapters will address it systematically.

SQLite: The World's Most Deployed Database

Among embedded database options, SQLite stands as the overwhelming favorite for local-first applications. Created by D. Richard Hipp in 2000, SQLite has quietly become the most widely deployed database engine in history. More instances of SQLite run on devices worldwide than any other database, primarily because it is bundled with virtually every web browser, mobile operating system, desktop application framework, and embedded device [15].

SQLite's design philosophy explains its success: it is public domain (no licensing restrictions), serverless (no separate process to manage), self-contained (a single C library file), zero-configuration (no installation or setup required), and ACID-compliant (guarantees data integrity even during crashes

or power failures). For local-first applications, these properties are not incidental conveniences; they are essential requirements.

Let us examine why SQLite is particularly well-suited to local-first architecture:

Single file storage. An entire SQLite database lives in a single file on disk. This makes backup trivial (copy the file), migration straightforward (move the file), and distribution simple (ship the file with your application). For local-first applications where users need to export or share their data, this is invaluable.

Cross-platform compatibility. The same SQLite file format works identically on Windows, macOS, Linux, iOS, Android, web browsers (via WebAssembly), and embedded systems. A database created on a mobile phone can be opened on a desktop computer without any conversion. This enables seamless multi-device workflows where users switch between devices naturally.

SQL interface. SQLite uses standard SQL for querying, which means developers can leverage decades of knowledge about relational data modeling, query optimization, and transaction management. Unlike custom document stores or key-value databases that require learning proprietary APIs, SQLite's interface is familiar to virtually every backend developer.

Transaction support. SQLite provides full ACID transactions with proper isolation levels. When multiple operations must succeed or fail together (like transferring money between accounts or updating related records), SQLite guarantees consistency without requiring external coordination mechanisms. This is critical for maintaining data integrity in local-first applications where there is no central server to enforce constraints.

Write-Ahead Logging (WAL) mode. Introduced in version 3.7.0 in 2010, WAL mode dramatically improves SQLite's concurrency characteristics [16]. In traditional rollback journal mode, writers block all readers and vice versa: a single write transaction locks the entire database file. WAL mode allows multiple concurrent readers alongside a single writer by logging changes to a separate write-ahead log file before applying them to the main database. This reduces lock contention significantly and improves performance for mixed read/write workloads by orders of magnitude.

Here is how to enable WAL mode:

```
1 PRAGMA journal_mode=WAL;
```

Once set, this mode persists across connections for that database file. The trade-offs are worth understanding: WAL creates additional files (database-name.db-wal and database-name.db-shm) that must remain alongside the main database file or corruption occurs. It also has slightly different behavior for read-only access and backup operations compared to rollback journal mode.

Embedded in every platform. SQLite requires no installation because it is already present on virtually every computing platform:

- Web browsers: Available via WebAssembly builds like `sql.js`, WASM-SQLite, or through frameworks that bundle SQLite
- iOS/macOS: Built into the operating system as `libsqlite3.dylib`
- Android: Included in the platform since early versions
- Windows/Linux desktop: Easily bundled as a static library or dynamic link library
- Node.js: Accessible through bindings like `better-sqlite3`, `sql.js`, or `node-sqlite3`

This ubiquity means you can use SQLite consistently across all platforms without worrying about availability or compatibility.

Performance characteristics. For typical local-first application workloads (thousands to millions of rows, mixed reads and writes, interactive query patterns), SQLite performs exceptionally well. A properly configured SQLite database in WAL mode can handle hundreds of concurrent read operations with sub-millisecond latency and thousands of write operations per second on modern hardware. Benchmarks show WAL mode achieving 70,000 reads per second and 3,600 writes per second compared to only 5,600 reads and 291 writes in default rollback journal mode [17].

The primary limitation is concurrent write throughput. SQLite allows only one writer at a time, even in WAL mode. For most local-first applications this is not a problem because user interactions generate writes sequentially rather than concurrently. However, if your application has multiple background processes or threads all writing simultaneously, you may encounter lock contention that requires careful transaction management.

Size and resource usage. SQLite's C library is approximately 800 kilobytes when compiled, making it lightweight enough for even the most constrained embedded environments. Database files grow proportionally to stored data without significant overhead: a database with one million rows of moderate complexity typically occupies tens of megabytes rather than gigabytes.

For local-first applications, SQLite's combination of reliability, performance, portability, and simplicity makes it the default choice unless you have specific requirements that demand something else. The rest of this chapter examines those alternatives briefly before we dive into practical schema design.

Alternatives and Specialized Stores

While SQLite is the general-purpose champion for local-first storage, certain use cases benefit from specialized alternatives. Understanding these options helps you make informed choices rather than defaulting to SQLite blindly.

IndexedDB. This is the standard browser API for client-side structured data storage [18]. IndexedDB is a NoSQL document store that runs entirely in the browser without plugins or WebAssembly overhead. It supports transactions, indexes, cursors for efficient iteration, and can store large binary objects (blobs) including files and media.

IndexedDB excels in web applications where you need to store significant amounts of structured data directly in the browser without relying on WebAssembly bridges to SQLite. Its asynchronous API works naturally with JavaScript's event loop, and it integrates well with modern frameworks through wrapper libraries like Dexie.js or RxDB.

However, IndexedDB has notable limitations: its callback-based API (despite promise wrappers) is verbose and error-prone compared to SQL; queries are less expressive than SQL, you can filter by indexed fields but cannot perform joins or complex aggregations easily; and browser storage policies may evict data under disk pressure without explicit user consent. Safari, in particular, has an aggressive policy of clearing IndexedDB data for sites not visited within seven days unless the user explicitly grants persistent storage [19].

For local-first web applications, IndexedDB is appropriate when your data model fits a document-oriented pattern (hierarchical JSON-like structures with simple queries) and you want to avoid WebAssembly overhead. For

relational data models or complex query requirements, SQLite via WASM remains superior.

RxDB. Reactive Database for JavaScript is a local-first NoSQL database built on top of IndexedDB or other storage backends [20]. RxDB adds reactive querying (subscribe to query results that update automatically when underlying data changes), JSON Schema validation, encryption, compression, and replication plugins for syncing with various backends.

RxDB shines in applications where you need real-time reactive updates throughout your UI, when a document changes locally or via sync, all subscribed views update automatically without manual refresh logic. Its plugin architecture allows customization of storage backend, synchronization protocol, and security features.

The trade-off is abstraction overhead: RxDB adds complexity on top of IndexedDB that may not be necessary for simpler applications. If your application has straightforward data access patterns without heavy reactive requirements, direct IndexedDB usage or SQLite might be cleaner.

PouchDB/CouchDB. PouchDB is a JavaScript database designed to work seamlessly with CouchDB servers through a compatible replication protocol [21]. It stores documents as JSON with automatic revision tracking for conflict detection and resolution. PouchDB chooses the best available storage mechanism on each platform: IndexedDB in browsers, LevelDB or similar in Node.js environments.

The CouchDB replication protocol is mature and well-understood, supporting continuous bidirectional sync, document filtering, and conflict handling through revision trees. This makes it suitable for applications that need robust offline sync with a central CouchDB server while maintaining full functionality when disconnected.

However, PouchDB's document-oriented model limits query expressiveness compared to SQL, and its replication protocol assumes a client-server topology rather than true peer-to-peer equality. For pure local-first applications without a CouchDB backend, the replication features add complexity without benefit.

libSQL/Turso. libSQL is an open-source fork of SQLite maintained by Turso that adds distributed replication capabilities while maintaining full backward compatibility with SQLite file formats and APIs [22]. Turso Database (their

managed service) distributes libSQL databases to edge locations worldwide, providing sub-millisecond read latency through embedded replicas.

For local-first applications, libSQL's "embedded replicas" feature is particularly interesting: you can run a libSQL database directly inside your application process that replicates from a primary database hosted on Turso Cloud or elsewhere. This gives you local SQLite performance with optional cloud connectivity for backup and multi-device sync.

The trade-off is vendor consideration: while libSQL itself is open-source, using Turso's managed features creates some dependency on their service. For applications requiring complete independence from any provider, standard SQLite remains preferable.

Specialized options. Several other databases serve niche local-first use cases:

- **WatermelonDB:** An offline-first database for React Native and React that uses SQLite with an optimistic sync pattern
- **Realm:** A mobile-focused embedded database with object-oriented API and automatic change notifications
- **Dexie.js:** A lightweight wrapper around IndexedDB that simplifies its verbose API (not a database itself but a valuable tool)

For most local-first applications, the choice narrows to SQLite for relational data models requiring complex queries, or IndexedDB (possibly via RxDB) for document-oriented models in browser environments. The rest of this book assumes SQLite as our primary storage engine because it provides the most general-purpose foundation.

Browser Storage and IndexedDB

Because web applications are increasingly important targets for local-first development, browsers provide universal access without installation, automatic updates, and cross-platform consistency, understanding browser storage options is essential even if SQLite via WASM is your preferred backend.

Browsers offer several storage mechanisms, each with different characteristics:

localStorage. Synchronous key-value store limited to approximately 5-10 megabytes per origin. Data persists until explicitly deleted or the user clears browser data. Operations block the main thread, making it unsuitable for large datasets or frequent updates. Best used for small configuration values, theme preferences, or tokens, not primary application data.

sessionStorage. Similar to localStorage but scoped to a single browser tab and cleared when the tab closes. Useful for temporary state that should not persist across sessions.

IndexedDB. Asynchronous NoSQL database with no practical size limit (browsers negotiate storage quotas with the operating system, typically allowing gigabytes). Supports transactions, indexes, cursors, and binary data. Can be accessed from Web Workers to avoid blocking the main thread. This is the primary browser-native option for serious local-first web applications.

Cache Storage. Designed for service worker asset caching rather than application data. Stores request/response pairs for offline serving of static resources. Not suitable for structured application data but essential for PWA functionality.

File System Access API and OPFS (Origin Private File System). Newer browser APIs that provide file-level access within an origin's sandbox. OPFS offers a fast, persistent filesystem accessible from both main thread and workers with performance approaching native file I/O. Emerging as a promising option for large local datasets, though currently less mature than IndexedDB.

For our running example application TaskSync, we will use SQLite via WASM in the browser to maintain consistency with desktop and mobile implementations. This means bundling a WebAssembly build of SQLite that runs inside the browser process, slightly larger initial download but identical data model and query language across all platforms. Libraries like WASM-SQLite or sql.js provide this capability, though WASM-SQLite is generally preferred for better performance through more efficient memory management.

The project structure will look like this:

```
1  tasksync/  
2  └─ src/  
3     └─ db/  
4         └─ index.ts           # Database initialization and schema  
5         └─ migrations.ts      # Schema migration logic  
6         └─ queries.ts         # Typed query functions  
7     └─ sync/                  # Synchronization (added later)  
8     └─ components/            # UI components  
9     └─ app.ts                 # Application entry point  
10 └─ public/  
11     └─ index.html  
12     └─ sqlite.wasm           # SQLite WebAssembly binary  
13 └─ package.json  
14 └─ tsconfig.json
```

This structure separates database concerns from application logic and UI, making it easier to evolve the synchronization layer independently in later chapters.

Designing Your Local Schema

Designing a database schema for local-first applications requires considerations beyond traditional relational design. Because data will be replicated across devices and edited concurrently without central coordination, your schema must support conflict detection and resolution, efficient incremental synchronization, and eventual consistency guarantees.

Here are key principles for local-first schema design:

Every record needs an immutable identifier. Use a universally unique identifier (UUID) or similar globally unique ID rather than auto-incrementing integers as primary keys. Auto-increment IDs created independently on different devices will collide when synchronized, causing data loss or corruption. UUID version 4 (random) or version 7 (time-ordered with randomness) are common choices.

```

1  -- Good: UUID primary key that never changes
2  CREATE TABLE tasks (
3      id TEXT PRIMARY KEY,                -- UUID v4
4      title TEXT NOT NULL,
5      description TEXT,
6      completed INTEGER NOT NULL DEFAULT 0, -- Boolean as integer
7      created_at TEXT NOT NULL,           -- ISO 8601 timestamp
8      updated_at TEXT NOT NULL            -- ISO 8601 timestamp
9  );
10
11 -- Bad: Auto-increment ID that will collide across devices
12 CREATE TABLE tasks (
13     id INTEGER PRIMARY KEY AUTOINCREMENT, -- Collision risk!
14     title TEXT NOT NULL
15 );

```

Track changes with timestamps or version counters. To support incremental synchronization (sending only changed records rather than full database dumps), each record should have metadata indicating when it was last modified. An `updated_at` timestamp is the simplest approach; a version counter provides additional guarantees for conflict detection.

```

1  CREATE TABLE tasks (
2      id TEXT PRIMARY KEY,
3      title TEXT NOT NULL,
4      description TEXT,
5      completed INTEGER NOT NULL DEFAULT 0,
6      created_at TEXT NOT NULL DEFAULT (datetime('now')),
7      updated_at TEXT NOT NULL DEFAULT (datetime('now')),
8      version INTEGER NOT NULL DEFAULT 1
9  );

```

The version counter increments with each modification. During synchronization, comparing versions helps detect concurrent edits: if two devices both modified the same record since last sync, their versions will diverge from what either expects, signaling a conflict that needs resolution.

Use tombstones for deletions. In distributed systems with eventual consistency, deleting a record requires marking it as deleted rather than removing it immediately. This “tombstone” ensures the deletion propagates to other replicas before the record is permanently removed. Without tombstones, a deleted record on one device might be resurrected when an outdated copy syncs from another device that does not yet know about the deletion.

```

1 CREATE TABLE tasks (
2     id TEXT PRIMARY KEY,
3     title TEXT NOT NULL,
4     description TEXT,
5     completed INTEGER NOT NULL DEFAULT 0,
6     created_at TEXT NOT NULL DEFAULT (datetime('now')),
7     updated_at TEXT NOT NULL DEFAULT (datetime('now')),
8     deleted_at TEXT,           -- NULL = active, timestamp = deleted
9     version INTEGER NOT NULL DEFAULT 1
10 );
11
12 -- Query only active tasks
13 SELECT * FROM tasks WHERE deleted_at IS NULL;
14
15 -- "Delete" a task by marking it with tombstone
16 UPDATE tasks SET deleted_at = datetime('now'), updated_at = datetime('now'),
    → version = version + 1 WHERE id = 'some-uuid';

```

Tombstones accumulate over time and require periodic garbage collection, deleting records whose tombstones have been synchronized to all known replicas for a sufficient period. We will cover tombstone management in detail in Chapter 5.

Design for incremental sync. Your schema should make it easy to query “what has changed since timestamp T” efficiently. This typically means indexing the `updated_at` column and designing sync queries that fetch only modified records:

```

1 -- Create index for efficient change detection
2 CREATE INDEX idx_tasks_updated_at ON tasks(updated_at);
3
4 -- Fetch all changes since last sync
5 SELECT * FROM tasks WHERE updated_at > '2024-01-15T10:30:00Z';

```

Avoid complex cross-record constraints. Foreign key constraints and triggers that enforce relationships across tables complicate synchronization because violations might occur temporarily during concurrent edits on different devices. For local-first schemas, prefer application-level consistency checks over database-enforced constraints, especially for relationships that span synchronized data.

This does not mean abandoning referential integrity entirely, use foreign keys where they protect critical invariants and the related records are unlikely to be edited concurrently from different devices. But recognize that strict

constraint enforcement assumes a single authoritative copy of data, which local-first architecture explicitly rejects.

Plan for encryption. If your application will encrypt data end-to-end (recommended for privacy-sensitive applications), design your schema to store ciphertext rather than plaintext. This typically means storing binary blobs or base64-encoded strings for sensitive fields:

```
1 CREATE TABLE tasks (  
2     id TEXT PRIMARY KEY,  
3     title_encrypted BLOB NOT NULL,    -- Encrypted title  
4     description_encrypted BLOB,      -- Encrypted description  
5     completed INTEGER NOT NULL DEFAULT 0,  
6     created_at TEXT NOT NULL DEFAULT (datetime('now')),  
7     updated_at TEXT NOT NULL DEFAULT (datetime('now')),  
8     deleted_at TEXT,  
9     version INTEGER NOT NULL DEFAULT 1  
10 );
```

Encryption adds complexity to querying, you cannot search or filter encrypted fields efficiently without special techniques like searchable encryption. Design your schema to keep frequently queried fields unencrypted when possible, encrypting only truly sensitive data. We will cover encryption strategies in Chapter 10.

With these principles established, we are ready to begin building our running example application. In the next chapter, we will create TaskSync as a fully local-only task management application using SQLite, implementing proper data modeling, optimistic updates, and offline functionality before considering any synchronization between devices.

Chapter 3: Building the Core Application Locally

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

Project Setup and Architecture Decisions

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

The Local Data Model

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

CRUD Operations Without a Server

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

Optimistic Updates and Immediate Feedback

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

Testing the Offline Experience

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

Chapter 4: Synchronization

Fundamentals

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

What Synchronization Actually Is

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

The Conflict Problem

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

Consistency Models in Practice

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

Designing a Sync Protocol

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

Chapter 5: Conflict Detection and Resolution

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

Types of Conflicts

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

Last-Write-Wins and Its Pitfalls

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

Version Vectors and Causal Ordering

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

Tombstones and Logical Deletion

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

Idempotent Operations

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

Chapter 6: Conflict-Free Replicated Data Types (CRDTs)

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

The Mathematics of Convergence

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

State-Based Versus Operation-Based CRDTs

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

Common CRDT Types

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

Yjs: A Practical CRDT Engine

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

Automerge and Document-Oriented CRDTs

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

When CRDTs Help and When They Do Not

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

Chapter 7: Event Sourcing, Operational Transformation, and Alternatives

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

Event Sourcing Basics

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

Operational Transformation Explained

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

OT Versus CRDTs

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

Hybrid Approaches

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

Choosing Your Concurrency Model

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

Chapter 8: Multi-Device Synchronization in Practice

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

Architecture of a Sync Backend

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

Change Streams and Incremental Sync

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

Durable Queues for Reliable Delivery

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

Background Synchronization

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

Connectivity Detection and Retry Strategies

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

Handling Partial Failures

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

Chapter 9: Peer-to-Peer Communication and Real-Time Collaboration

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

Direct Device-to-Device Communication

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

WebRTC for Local-First Applications

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

Mesh Networks and Gossip Protocols

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

Real-Time Presence and Cursors

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

Trade-offs of Pure P2P

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

Chapter 10: Security, Encryption, and Privacy

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

Authentication Without a Central Authority

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

End-to-End Encryption for Local Data

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

Key Management and Rotation

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

Secure Sharing and Access Control

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

Privacy by Design

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

Chapter 11: Platform-Specific Implementation Details

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

Web Applications and Service Workers

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

Desktop Applications

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

Mobile Platforms

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

Edge Runtimes and WASM

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

Cross-Platform Data Sharing

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

Chapter 12: Testing, Debugging, and Observability

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

Testing Offline Behavior

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

Simulating Network Partitions

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

Testing Synchronization Correctness

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

Debugging Distributed State

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

Observability Without the Cloud

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

Chapter 13: Operations, Performance, and Production Deployment

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

Storage Management and Cleanup

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

Caching Strategies

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

Schema Evolution and Migrations

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

Backup and Recovery

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

Performance Optimization

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

Deployment Patterns

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

Chapter 14: Hybrid Architectures and When the Cloud Is Optional

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

The Hybrid Philosophy

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

Cloud-Assisted Discovery and Coordination

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

Optional Backup Layers

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

Technology Landscape Overview

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

Architectural Decision Framework

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

Conclusion: Designing for Resilience

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

The Local First Checklist

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

What We Have Learned

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

The Future of Offline-Native Software

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

References

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

Official Documentation

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

CRDT Libraries and Projects

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

Local-First Frameworks and Sync Engines

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

Foundational Papers and Articles

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

Books

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

Distributed Systems Concepts

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

Technical Articles and Resources

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

Verification and Testing Research

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

Industry and Community

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