Aalto-yliopisto
Aalto-universitetet
Aalto University

The Limits of Generalized Sync: A Taxonomy of Architectures, Trade-offs, and Decision Factors

Master's thesis

Mikael Siidorow

School of Science
Master's Programme in Computer, Communication and Information Sciences

Espoo, 25 May 2026

Author	Mikael Siidorow
Thesis title	The Limits of Generalized Sync: A Taxonomy of Architectures, Trade-offs, and Decision Factors
Date	25 May 2026
No. of pages	131
Major	Computer Science
Supervisor	Prof. Petri Vuorimaa
Advisors	University Teacher Juho Vepsäläinen (DSc) Ian Tuomi (MSc)
Collaborative partner	Teamspective Oy

Web applications increasingly compete on responsiveness, driving engineers to move data closer to the client. Sync engines promise reusable synchronization infrastructure, but whether that promise holds across application types remains an open question. This thesis investigates the extent to which synchronization can be generalized across web applications.

The study combines a literature review, analysis of 69 practitioner sources, semi-structured interviews with 13 practitioners, and a case study. The thesis classifies 14 sync engine instances across 13 projects along eight architectural dimensions and builds a decision framework on this taxonomy. Sync engines form four architectural clusters rather than a smooth spectrum. The online/offline boundary is the most discriminating dimension and sets the limit of generalization. For always-online applications, the read path generalizes well, and the write path generalizes when the engine defines how writes are applied, reconciled, and rolled back. For offline-capable applications, write-path reuse remains limited by authorization, conflict resolution, and application-specific business logic. The thesis identifies seven technical trade-offs and five fundamental limits.

Application fitness varies by domain: real-time B2B productivity tools are the broadest fit, while transactional and data-intensive applications fit poorly. Adoption depends less on synchronization than on speed, developer experience, and incremental integration with existing Postgres infrastructure. A case study at a B2B SaaS company illustrates the authorization limit and reveals deployment friction added by existing managed Postgres infrastructure. The thesis argues that sync engines function as general-purpose infrastructure within clear architectural limits, but not across all web application types.

Keywords	CRDTs, data synchronization, eventual consistency, local-first software, replicated databases, sync engines, web applications
Language	English

Aalto-yliopisto | Perustieteiden korkeakoulu
Computer, Communication and Information Sciences -maisteriohjelma

DIPLOMITYÖN TIIVISTELMÄ

Tekijä	Mikael Siidorow
Työn nimi	Yleiskäyttöisen synkronoinnin rajat: Arkkitehtuurien taksonomia, suunnitteluvalinnat ja päätöskriteerit
Päiväys	25. toukokuuta 2026
Sivumäärä	131
Pääaine	Computer Science
Työn valvoja	Prof. Petri Vuorimaa
Työn ohjaajat	Yliopisto-opettaja Juho Vepsäläinen (TkT) DI Ian Tuomi
Yhteistyötaho	Teamspective Oy

Selainsovellusten nopeudesta on tullut keskeinen kilpailutekijä, minkä seurauksena dataa pyritään tuomaan yhä lähemmäs käyttäjää. Synkronointimoottorit (engl. *sync engines*) tarjoavat datan synkronoinnin uudelleenkäytettävänä infrastruktuurina, mutta mallin soveltuvuus erilaisiin sovellustyyppeihin on avoin kysymys. Tässä diplomityössä selvitetään mallin yleistettävyyden rajat selainsovellusten kontekstissa.

Tutkimuksessa yhdistyvät kirjallisuuskatsaus, 69 alan ammattilaislähteen analyysi, 13 asiantuntijan puolistrukturoidut haastattelut sekä tapaustutkimus. Aineiston pohjalta rakentuu kahdeksan ulottuvuuden taksonomia 14 synkronointimoottorista 13 eri projektissa sekä viitekehys käyttöönoton tueksi.

Synkronointimoottorit jakautuvat neljään arkkitehtuuriluokkaan, joista offline-tuki erottuu keskeisimpänä jakolinjana. Verkkoyhteyden tarve määrittää arkkitehtuurin yleistettävyyden: aina verkossa toimiville sovelluksille luku- ja kirjoituspolku ovat yleistettävissä, kunhan moottori vastaa kirjoitusten suorittamisesta, yhteensovittamisesta ja kumoamisesta. Offline-tuetuissa sovelluksissa kirjoituspolun yleistämistä rajoittavat valtuutuksien hallinta, konfliktinratkaisu ja sovelluslogiikka. Työssä tunnistetaan seitsemän teknistä kompromissia ja viisi perustavanlaatuista rajoitetta.

Soveltuvuus vaihtelee sovellustyypeittäin: reaaliaikaiset B2B-tuottavuustyökalut edustavat laajinta soveltuvuusaluetta, transaktio- ja dataintensiiviset sovellukset sopivat malliin huonosti. Käyttöönottoa ohjaavat synkronoinnin sijaan nopeus, kehittäjäkokemus ja vaiheittainen integroitavuus olemassa olevaan Postgres-infrastruktuuriin. Tapaustutkimus B2B SaaS -yrityksessä havainnollistaa valtuutuksen rajoitetta ja paljastaa hallinnoitujen Postgres-palveluiden aiheuttaman käyttöönottokitkan. Työ osoittaa, että synkronointimoottorit toimivat yleiskäyttöisenä infrastruktuurina vain selkeiden arkkitehtuurirajojen sisällä, eivät kaikissa selainsovelluksissa.

Avainsanat	CRDT-tietorakenteet, datan synkronointi, hajautetut järjestelmät, local-first-ohjelmistot, ohjelmistoarkkitehtuuri, selainsovellukset, synkronointimoottorit
Kieli	Englanti

Acknowledgements

I would like to thank my supervisor, Professor Petri Vuorimaa, and my advisors, University Teacher Juho Vepsäläinen, and Ian Tuomi at Teamspective, for their guidance throughout this thesis. Petri's reviews at key milestones structured the higher-level academic framing and contribution focus. Juho's feedback every two weeks strengthened the academic structure and writing, while Ian helped refine the research topic and ground the case study in production implementation.

I am also grateful to the 13 practitioners who shared their time and experience in interviews. Their accounts of designing, adopting, and building sync engines made this study possible.

Espoo, 25 May 2026
Mikael Siidorow

Declaration of AI Use

During the preparation of this thesis, I used generative artificial intelligence (AI) tools as research and writing support. Most use was conversational: I asked questions, tested interpretations, compared source material, and refined drafts for clarity. AI also supported visualization work and review against evaluation criteria. For the practitioner content analysis described in Chapter 3, I used AI to support source processing before manual validation. The processing pipeline and Appendix A describe that use in detail.

I used Claude Code and Codex extensively during the case study implementation, including for code generation, debugging, schema and migration scripting, and documentation drafting. Furthermore, I used Claude Code to explore Zero's source code and example projects, looking for patterns to adapt in the case study. The integration time, friction, and adoption signals reported in Chapter 7 reflect AI-assisted development rather than solo engineering.

The final text and arguments are my own, including the decisions about evidence, analysis, and interpretation. I reviewed AI-assisted material before using it in the thesis. I take full responsibility for the originality, accuracy, and scientific integrity of this thesis.

Table of Contents

Symbols and abbreviations

ACID	atomicity, consistency, isolation, durability
AI	artificial intelligence
API	application programming interface
ARQ	additional research question
B2B	business-to-business
CAP	consistency, availability, partition tolerance
CD	continuous delivery
CI	continuous integration
CRDT	conflict-free replicated data type
CRM	customer relationship management
CRUD	create, read, update, delete
DAG	directed acyclic graph
DDL	data definition language
DX	developer experience
HA	high availability
HTTP	hypertext transfer protocol
IVM	incremental view maintenance
JSON	JavaScript Object Notation
LLM	large language model
LWW	last-write-wins
OPFS	origin private file system
ORM	object-relational mapper
OT	operational transformation
PACELC	partition, availability, consistency, else latency, consistency
REST	representational state transfer
RPC	remote procedure call
SaaS	software as a service
SPA	single-page application
SQL	structured query language
SWR	stale-while-revalidate
UI	user interface
WAL	write-ahead log
WASM	WebAssembly
ZQL	Zero Query Language

1. Introduction

Web applications compete on responsiveness. Users expect instant interactions: clicking, typing, and navigating without waiting for the server [1]. The traditional request-response model makes every interaction depend on the network, producing visible latency through spinners, loading states, and degraded user experience on slow connections [2].

Client-side caching reduces perceived latency but remains server-authoritative. Libraries, such as SWR [3] and TanStack Query [4], allow the client to optimistically display data, but the server can reject writes and the client has no durable local state. The local-first software movement proposes a more radical solution: treat the client's copy of the data as primary and synchronize in the background [5]. This approach eliminates spinners entirely, but introduces distributed systems complexity that most application teams are not equipped to handle.

A category of tools called sync engines addresses this problem. These systems promise to abstract synchronization so that engineers interact with local data while the engine handles the network. The promise is compelling: buy synchronization off the shelf instead of building it. Linear, a project management tool for software teams, spent half a year building foundational infrastructure including a custom sync engine before shipping their product [6]. A general-purpose sync engine that eliminates that investment saves months of engineering time.

Synchronization is not a simple problem, however. It intersects with consistency guarantees, conflict resolution, authorization, and domain-specific business logic. Whether these concerns can be generalized at all, or whether they are inherently application-specific, is an open question.

1.1. Research gap

Academic research covers the building blocks that sync engines use: optimistic replication [7], conflict-free replicated data types (CRDTs) [8], consistency models [9], and server architectures for collaborative editing [10, 11]. None of this work classifies sync engines as integrated products or evaluates when they are appropriate versus custom implementations.

Practitioner discourse on sync engines has grown quickly. Two dedicated conferences, Local-First Conf and Sync Conf, launched in 2024 and 2025, and a FOSDEM devroom for local-first software ran for the first time in 2026. Sync engine vendors publish architectural talks, blog posts, and podcasts explaining their design decisions. The

localfirst.fm series in particular hosts extended architectural discussions. Researchers contribute practitioner-facing syntheses such as Weidner's [12] catalog of central-server collaboration architectures. This discourse is unsystematic: labels such as "local-first" and "real-time" conflate distinct goals like local responsiveness, live updates, and collaboration [13].

No empirical study examines how sync engine architectural choices relate to fitness for specific application types, or where generalization breaks down. Answering this question requires systematic classification of the architectural patterns involved and empirical analysis of their trade-offs and fitness across application domains. This thesis contributes to that understanding.

1.2. Research questions

This thesis investigates the extent to which client-side data synchronization can be generalized across web applications.

Main RQ: To what extent can client-side data synchronization be generalized across web applications?

The main research question is addressed through three additional research questions (ARQs) that structure the investigation:

1. **ARQ1 (Taxonomy):** What architectural patterns exist across the synchronization spectrum from server-authoritative to local-first?
2. **ARQ2 (Trade-offs/Limits):** What trade-offs and limits do these architectural patterns introduce?
3. **ARQ3 (Fitness):** How do application requirements determine sync engine fitness?

1.3. Scope and definitions

This thesis defines a sync engine as a system that keeps client-side application state synchronized with one or more authoritative replicas. Caching layers such as SWR [3] and TanStack Query [14] lack a server-side sync component and fall outside this definition. CRDT libraries such as Yjs [15] and Loro [16] can satisfy the definition when paired with synchronization providers, but fall outside the scope of this study because they do not self-describe as sync engines and lack integrated delivery infrastructure. Automerge [17] is included because it self-describes as a sync engine and provides integrated sync infrastructure through automerge-repo. Full offline capability is not required for inclusion, as the research question concerns generalization limits, not adherence to local-first ideals. Chapter 2 covers the full definitions and boundary cases.

The study draws on four data sources. Academic literature provides theoretical foundations in distributed systems and CRDTs. Practitioner content analysis covers 69 conference talks, podcast episodes, and blog posts. Semi-structured interviews with 13 practitioners include sync engine vendors, engineers who adopted sync engines, and engineers who built custom synchronization. A case study examines sync engine adoption in production at Teamspective, a B2B SaaS platform for employee engagement, people analytics, and feedback. Chapter 3 details the research design.

1.4. Contributions

This thesis contributes a decision framework for sync engine adoption. The framework explains when generalized synchronization works for web applications, and where it breaks down.

The framework has four components. First, the thesis classifies 14 sync engine instances across 13 projects along eight architectural dimensions and finds four architectural clusters. Second, the trade-off analysis identifies seven technical trade-offs and five fundamental limits. Third, the fitness analysis explains where sync engines fit and what drives the adoption decision. Fourth, a Teamspective case study applies the framework to the adoption of Zero, a database-pluggable sync engine, in a B2B SaaS application on managed Postgres. The case study illustrates the authorization limit and reveals deployment friction added by managed Postgres infrastructure.

1.5. Thesis structure

The remainder of this thesis is organized as follows. Chapter 2 establishes theoretical foundations in distributed systems constraints, optimistic replication, and CRDTs, and surveys current sync engines. Chapter 3 details the research design combining literature review, practitioner content analysis, semi-structured interviews, and a case study. Chapter 4 presents the synchronization pattern taxonomy and addresses ARQ1. Chapter 5 analyzes trade-offs and documents limits of generalization, and addresses ARQ2. Chapter 6 evaluates application fitness and decision factors, and addresses ARQ3. Chapter 7 reports the Teamspective case study, which applies the framework to a production adoption. Chapter 8 synthesizes findings, discusses implications, acknowledges limitations, and identifies future work. Chapter 9 summarizes the contributions and answers the main research question.

2. From distributed systems to sync engines

This chapter covers the distributed systems theory and synchronization patterns needed to understand sync engines. Distributed systems constraints shape how all synchronization architectures work [18, 19]. Synchronization patterns range from server-authoritative request-response [20] to fully local-first architectures [5]. Sync engines package synchronization as reusable infrastructure and aim to make it a commodity rather than a custom build [21].

2.1. Distributed systems foundations

Synchronization architectures operate under distributed systems constraints. This section covers four foundational concepts: clocks for reasoning about event ordering, consistency models that define what guarantees a system provides, the CAP theorem and PACELC that formalize the trade-offs between these guarantees, and the end-to-end argument that suggests where sync engines should and should not enforce logic.

2.1.1. Clocks

Reasoning about events across replicas requires a way to talk about when events happen. Wall-clock time is unreliable for this purpose: client devices can have their clocks manually adjusted, hardware clocks drift, and skew between nodes is routinely on the order of tens of milliseconds [22]. Lamport [23] formalizes a wall-clock-independent alternative through the "happens-before" relation, which defines when one event causally precedes another. Logical clocks turn happens-before into concrete timestamps that respect causal ordering without requiring synchronized real-time clocks [23]. Hybrid logical clocks combine wall-clock time with logical tie-breaking and approximate a total real-time order without strict clock synchronization [22]. The happens-before relation and the clocks built on it underpin both consistency models and conflict detection in synchronization systems.

2.1.2. Consistency models

Distributed systems replicate data across multiple nodes for availability and performance [7]. Replication raises a basic question: when one replica is updated, what do other replicas see [24]? Consistency models describe the guarantees that a system provides about this divergence. Understanding these models matters because every sync engine makes a specific consistency choice.

Strong consistency, also known as linearizability, makes all replicas behave as if there is a single copy [18]. Every read returns the most recent write. This guarantee requires coordination between replicas, which costs latency [19].

Eventual consistency relaxes the coordination requirement: replicas may temporarily diverge, but converge to the same state if no new updates arrive [24]. Vogels [24] defines the "inconsistency window" as the period between an update and when all observers see it. The acceptable window depends on the domain. A one-second delay is tolerable for a chat application, but not for a financial transaction.

Strong eventual consistency (SEC), introduced by Shapiro et al. [8], offers a stronger form of eventual consistency: if two replicas have received the same set of updates in any order, they are in the same state. This guarantee requires no rollback or conflict resolution after the fact [8]. Conflict-free replicated data types (CRDTs) achieve SEC through mathematical properties of their operations [8]. Preguiça [25] provides an overview of CRDT designs that build on this foundation. Viotti and Vukolić [9] presented a hierarchy of over 50 consistency models and showed how these guarantees relate to each other.

Vogels [24] also identified **client-centric consistency** variations that matter for user experience. *Read-your-writes* ensures that a user always sees their own updates [24]. *Monotonic reads* guarantees that once a user sees a version, they never see an older version [24]. *Session consistency* provides read-your-writes guarantees within a session [24]. These client-centric guarantees matter for applications built on eventually consistent systems [24]. A user who submits a form and does not see the result has a broken experience, regardless of the server's consistency model.

Many sync engines write locally first and reconcile in the background, providing eventual consistency [7, 21]. CRDT-based engines provide strong eventual consistency at the CRDT layer through the data types defined above [8]. Convex, a managed backend platform with built-in sync, instead chooses strong consistency, trading local speed for transactional guarantees [26]. Local writes feel instant because they apply immediately to the client replica that handles subsequent reads [5]. Replicas converge through later reconciliation [7]. This design choice reflects a trade-off that the next section formalizes.

2.1.3. CAP theorem and PACELC

Brewer [27] conjectured that a distributed system cannot simultaneously guarantee consistency, availability, and partition tolerance (CAP). Gilbert and Lynch [18] formally proved this conjecture: in an asynchronous network where partitions can occur, a system must sacrifice either consistency or availability. During a network partition, the system must choose between stopping requests to maintain consistency or serving potentially stale data to maintain availability [18].

For web applications, partitions are not rare [28]. A user on a subway, a flaky mobile connection, or a server restart all constitute partitions. Any system that claims to work offline has already chosen availability over consistency during partitions [5]. However, Abadi [19] argues that CAP is insufficient because it only describes behavior during failures. PACELC extends CAP by adding the normal-operation trade-off. The acronym encodes the decision tree: if there is a Partition, choose between Availability and Consistency; Else (during normal operation), choose between Latency and Consistency [19]. This latency-consistency trade-off is present at all times during system operation, not just during failures.

PACELC provides a lens for classifying sync engines. PowerSync follows a PA/EL strategy at the client: writes apply to a local SQLite replica first and remain available during partitions, with authoritative commit deferred to the server [29, 30]. Convex follows a PC/EC strategy: it prioritizes consistency at all times, requiring a server roundtrip for transactional guarantees [26]. Under the same lens, traditional REST also follows PC/EC: the server is authoritative, and the client waits for a response [20]. This classification exposes a design choice. Sync engines that prioritize local speed (PA/EL) cannot guarantee strong consistency [19]. Engines that guarantee strong consistency (PC/EC) cannot provide instant local writes [19]. No engine can do both.

2.1.4. The end-to-end argument

Saltzer et al. [31] argue that functions placed in low-level system components are often redundant or insufficient. The application at the endpoints must verify correctness anyway, because only it understands the domain. The low-level system lacks the context to distinguish valid from invalid states [31]. Low-level mechanisms are therefore justified as *performance optimizations*, not as correctness guarantees. The network layer can reduce retries, but it cannot decide whether a duplicate "Buy" click is intentional or a network retry. This distinction between performance optimization and correctness guarantee defines where sync engines stop.

For sync engines, the end-to-end argument suggests a clear division of responsibility. A sync engine can handle data transport and caching as a performance optimization [31]. It cannot enforce that "users may only edit documents if their budget is under $500," because this is domain logic that requires endpoint knowledge [31]. Complex business invariants require application-level validation that a generic engine lacks [31].

This suggests a boundary: sync engines are most effective when they handle transport and least effective when they attempt to encode domain-specific validation. The end-to-end argument helps explain why organizations with complex domain logic, such as Figma [32, 33], build custom sync: their correctness requirements cannot be delegated to a generic engine. Chapter 6 examines this prediction against practitioner evidence.

2.2. Data synchronization

Applications do not face a binary cloud-versus-local choice. Instead, they sit on a spectrum of synchronization approaches, from fully server-authoritative to fully local-first [5]. Each position on this spectrum implies a different set of trade-offs between latency, availability, consistency, and complexity [19]. This section defines four positions on the spectrum, starting with the theoretical foundation of optimistic replication.

2.2.1. Optimistic replication

Saito and Shapiro [7] surveyed optimistic replication: allowing replicas to diverge temporarily and reconciling in the background. This is the operating principle behind all sync engines and local-first systems. Optimistic algorithms let data be accessed without prior synchronization, assuming that conflicts will occur only rarely [7].

The survey identified six design dimensions [7], three of which this thesis builds on. The first is *single-master versus multi-master* replication. Single-master is simpler but limits availability, because only one node can accept writes [7]. Multi-master replication, where any replica can accept writes, enables offline operation and collaboration but requires conflict resolution [7].

The second dimension is *state transfer versus operation transfer*. State transfer sends full snapshots, which is simple but bandwidth-heavy [7]. Operation transfer sends deltas or individual operations, which is more efficient but requires causal ordering [7, 23].

The third dimension is *conflict resolution strategy*, which ranges from syntactic approaches such as last-writer-wins to semantic approaches that use application-specific merge logic [7]. As Saito and Shapiro observed, "conflict resolution is usually highly application specific" [7], a finding that remains relevant two decades later. Terry et al. [34] demonstrated this early with Bayou, a weakly connected replicated storage system for mobile computing where each write carries an application-specific merge procedure. If conflict resolution is usually domain-specific, then sync engines that try to make it generic face hard limits. These three dimensions map directly to the taxonomy dimensions developed in Chapter 4.

2.2.2. Traditional request-response

In the traditional request-response model, the client sends a request, the server processes it, and responds [20]. The server holds the authoritative copy of all data [20]. REST, as defined by Fielding [20], remains the dominant pattern for web APIs: 93% of API practitioners reported using it in Postman's 2025 survey of over 5,700 respondents [35]. The client maintains no state beyond what the user interface (UI) needs to render,

and every user action that requires data triggers a network request [20]. This model offers a simple mental model, server-authoritative consistency, and well-understood tooling [20]. The limitation is that every interaction depends on the network: latency is visible to the user as loading spinners, and there is no offline capability [5].

2.2.3. Client-side caching

Client-side caching in web applications operates at two distinct layers. At the transport layer, the Hypertext Transfer Protocol (HTTP) defines caching mechanisms through headers such as `Cache-Control` and `ETag` [36]. The `stale-while-revalidate` directive allows the browser to serve stale cached responses immediately while revalidating in the background [37]. The server defines the header logic, and the browser manages the cache according to those headers [36].

At the application layer, libraries such as SWR [3] and TanStack Query [14] maintain an in-memory cache of application programming interface (API) responses in JavaScript. These libraries require application code to define cache keys, with refetching behavior configurable on top of built-in defaults [3, 14, 38, 39]. SWR adapts the *stale-while-revalidate* pattern from HTTP [37] to application code rather than the transport layer [3]. This application-level caching is an explicit architectural choice to manage server state on the client. Cache invalidation is application-managed: the libraries expose invalidation and revalidation primitives, but application code must decide when to invalidate, revalidate, or update cached data after writes [3, 40]. Changes made by other users remain invisible until the next network refetch or polling interval [3, 39]. Collaboration is not a design goal: the cache reflects one user's view of the server, not a shared state.

A related technique is *optimistic updates*: the UI reflects a user action before the server confirms it. If the server rejects the change, the UI rolls back. This is not local-first, because the server remains authoritative and the client is speculating that the write will succeed [5]. As Kleppmann et al. [5] note, optimistic UI does not provide the same guarantees as local-first because the request may still fail.

Both layers reduce perceived latency without changing the underlying architecture [3, 36]. HTTP-level caches are managed by the browser and typically survive page reloads [36]. Application-level caches are in-memory only by default, meaning they are lost on page refresh or tab close [41, 42]. Neither layer provides offline support or changes the server-authoritative model [5].

Persistent caching extends these patterns into browser storage APIs that survive page reloads and tab closures. `localStorage` provides simple synchronous key-value storage [43], while IndexedDB provides an asynchronous client-side database for larger volumes of structured data [44]. The Origin Private File System (OPFS) offers direct

file system access with better performance than IndexedDB for large datasets [45, 46]. Service Workers can intercept network requests and serve cached responses, enabling basic offline functionality for both static assets and API data [47]. Application-level caching libraries such as TanStack Query also support persisting their in-memory cache to IndexedDB or similar storage [42]. Saito and Shapiro [7] describe this pattern: the server remains the source of truth, while the client reads from the local cache and queues writes for later submission. Conflict resolution stays server-side, so when two devices modify the same data offline, the server must decide which version wins. Browser storage also has reliability concerns: clearing site data can delete IndexedDB contents [5].

This is where most modern production web applications likely sit today, as Kleppmann et al. [5] observe that offline support is "an afterthought in most web apps." This position on the spectrum adds resilience without the full complexity of CRDTs or multi-master replication [8]. The case study in Chapter 7 examines a system using in-memory client caching before sync engine adoption.

2.2.4. Offline-first, local-first, and CRDT-based synchronization

At the other end of the spectrum, local-first architecture inverts the traditional model [5]. Local-first builds on the earlier offline-first movement, which focused on applications working without connectivity [48]. It extends offline-first by adding collaboration as a core requirement and setting a more demanding goal: applications continuing to work indefinitely even if the company and all its servers disappear overnight [5]. In practice the term is used loosely, with most products adopting some local-first principles rather than the full ideological commitment. The copy of data on the local device becomes the primary copy, and the server acts as a secondary replica for backup and relay [5]. CRDTs enable this architecture by guaranteeing that concurrent modifications converge without coordination [8]. Any replica can accept writes at any time, and merging is automatic and deterministic [8]. This is multi-master optimistic replication [7] with mathematically guaranteed convergence [8]. Local-first delivers true offline capability, instant local writes, no loading spinners, and the possibility of peer-to-peer synchronization [5].

However, CRDTs introduce their own limitations. They accumulate change history, creating performance and memory overhead [5]. Global invariants such as uniqueness constraints are difficult to express [8]. Even a set CRDT requires choosing between add-wins and remove-wins semantics, with no universally correct choice [8]. The one case CRDTs cannot automatically resolve is when multiple users concurrently update the same property of the same object [5]. Figure 1 summarizes these four positions. The following section examines the local-first movement and CRDTs in more detail.

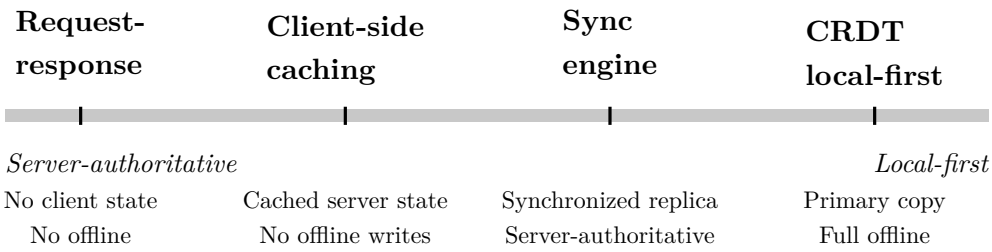

Figure 1: The synchronization spectrum from server-authoritative to local-first.

2.3. Local-first software

This section defines local-first software through Kleppmann et al.'s seven ideals and then introduces CRDTs, the data structures that make local-first architectures possible.

2.3.1. The local-first manifesto

Kleppmann et al. [5] define local-first software through seven ideals:

1. **No spinners:** the app responds instantly from local data.
2. **Your work is not trapped on one device:** data syncs across devices.
3. **The network is optional:** full functionality without internet.
4. **Seamless collaboration:** real-time and asynchronous editing.
5. **The long now:** data persists for decades, not tied to a service.
6. **Security and privacy by default:** end-to-end encryption.
7. **You retain ultimate ownership and control.**

These ideals describe a vision, not a binary classification [5]. No existing system fully satisfies all seven, and the authors acknowledge that web applications face inherent limitations in achieving full local-first properties. van Hardenberg and Kleppmann [49] demonstrate these challenges in practice: building PushPin, a peer-to-peer collaboration tool, revealed that achieving production quality with local-first principles requires solving problems in networking, access control, and data management that remain open research questions.

The ideals serve as a benchmark for evaluating sync engines. For example, Zero satisfies ideals 1 (no spinners) and 2 (multi-device), but explicitly excludes ideal 3 (offline) [50]. Convex satisfies ideals 2 and 4 (collaboration), but as a cloud service it does not satisfy ideals 6 (privacy) or 7 (user ownership). The paper's call for better tooling around CRDTs predicted the current wave of sync engines [5]. Whether these engines fulfill the local-first vision or merely borrow its language is a question this thesis explores.

2.3.2. Conflict-free replicated data types

Shapiro et al. [8] formalized CRDTs as data types that guarantee strong eventual consistency. If updates satisfy specific mathematical properties, such as commutativity for operations or monotonicity for states, replicas converge without coordination [8]. Preguiça [25] provides an overview of the CRDT research that has developed since then, and Almeida [51] provides a tutorial covering operation-based, state-based, and their delta-state and pure operation-based variants.

CRDTs come in two flavors [8]. *State-based CRDTs* (CvRDTs) have replicas send their full state, and receiving replicas merge via a join operation [8]. This approach is simple but bandwidth-heavy [25]. *Operation-based CRDTs* (CmRDTs) have replicas send individual operations [8]. This is more efficient but requires causal delivery: operations must arrive in a causally consistent order [8, 23].

Practical CRDT types include counters, sets, maps, registers, and text sequences [25]. Kleppmann and Beresford [52] extended CRDTs to JavaScript Object Notation (JSON) documents, enabling structured data replication. Litt et al. [53] addressed rich text editing, showing that extending CRDTs beyond plain text requires careful handling of formatting intent. Two notable CRDT libraries are Yjs and Automerge. Yjs [15] implements the Yet Another Transformation Approach (YATA) algorithm for peer-to-peer shared editing on arbitrary data types [54]. Automerge [55] builds on Kleppmann and Beresford's JSON CRDT research [52]. These libraries provide the building blocks that some sync engines use or are influenced by. The main alternative to CRDTs for collaborative editing is Operational Transformation (OT) [10]. While the original OT research supported decentralized architectures, production systems such as Google Wave relied on a central server to order operations [56].

CRDTs have several limitations that matter for sync engines. CRDTs handle *data merging*, not *business logic*: they guarantee convergence but not semantic correctness [8]. Global invariants are difficult to express: maintaining cross-record constraints in a CRDT requires application-specific trade-offs that do not generalize across domains [8]. Change history accumulates over time, creating memory and performance overhead [5]. Kleppmann et al. [5] hypothesized that "default merge semantics" are sufficient for most applications. This thesis examines whether this holds for complex business domains.

2.4. Sync engine landscape

This section defines sync engines, surveys their history, and maps the current landscape. The terminology is inconsistent across the community [57], which motivates the taxonomy in Chapter 4.

2.4.1. Defining sync engines

No standard definition of "sync engine" exists in the academic literature. The local-first community, vendors, and practitioners use overlapping terminology to describe systems with different architectures and capabilities [57]. Boodman and Chase [21] defines a sync engine as "software that keeps multiple copies of changing data consistent across devices and users." Kleppmann and Riccomini [58] frame offline-first and real-time collaboration as application-level multi-leader replication, situating sync engines within the broader replication design space. The authors then define the category by the replication process itself: a sync engine captures local changes, propagates them, receives remote changes, merges them into local state, and invokes conflict resolution when needed [58, p. 221]. The problem sync engines address is that traditional API-based architecture makes every interaction depend on the network, resulting in latency, stale data, and no offline capability [21]. The solution is keeping a local copy of data on each device, with changes pushed back and forth in the background [21].

Definition. A *sync engine* is a system that keeps client-side application state synchronized with one or more authoritative replicas. *Client-side application state* means the client holds a local representation of server or peer data, whether persistent or in-memory. *Synchronized* means the system provides server-side delivery logic or a peer protocol that propagates changes between client-side state and authoritative replicas. *Authoritative replicas* means the canonical state may reside in a centralized database, a managed backend, or a set of converging peers.

This definition is proposed by the author for this thesis. Like Boodman and Chase [21], it treats consistency across replicas as the defining requirement. PowerSync synchronizes durable local replicas while Convex maintains reactive in-memory state, and the definition covers both forms of client-side state [26, 59]. Practitioner sources distinguish pluggable sync engines, which integrate with an existing database, from managed systems that replace the backend datastore [60].[1] Since both address the same application-level problem, this thesis treats the distinction as integration depth instead of an exclusion criterion. The term *authoritative replicas* draws on the replication literature, where single-master versus multi-master systems are a primary design axis [7]. Chapter 4 examines empirically whether server-authoritative designs are the dominant pattern across current sync engines.

The definition's reach is clearest at four boundary cases. The first is *caching layers*. SWR [3], TanStack Query [14], and GraphQL clients such as Apollo Client [61] cache

[1] Aaron Boodman frames this as the distinction between sync engines and syncing datastores in an X thread from October 2024: https://x.com/aboodman/status/1843049082237170036, archived at https://perma.cc/Y3ZR-LKJF. Accessed 2026-05-02.

API responses on the client, but the server has no awareness of what each client holds and does not manage delivery. Client-side state managers such as TanStack DB are components rather than sync engines when the application or another product supplies the delivery protocol, server-side tracking, and authoritative state [62, 63]. A sync engine requires delivery logic, either in a server-side component or in a peer protocol, that tracks which data each replica should receive and delivers updates.

The second is *reactive databases without local persistence*, where the category boundary is contested. Boodman and Chase [50] categorizes Convex as "not a sync engine" but a "reactive database" where reads and writes are server-first. Under this narrower view, a sync engine requires a local writable replica, which Convex does not provide. This thesis includes Convex because its server manages what data each client holds and pushes updates through reactive subscriptions [26]. Convex's lack of client-side persistence and offline capability places it at the boundary of the category.

The third is *standalone CRDT libraries*. Yjs [15] and Loro [16] do not satisfy this definition on their own because they do not bundle delivery infrastructure, though a stack built on either of them with a synchronization provider can. The libraries themselves also do not self-describe as sync engines. Automerge [55] describes itself as a sync engine, and van Hardenberg [17] calls it "a local first sync engine." Automerge provides networking and storage through automerge-repo, which satisfies this definition. Automerge operates on CRDT documents rather than queryable relational state, but it is included in the taxonomy in Chapter 4 alongside the relational engines. Yjs and Loro therefore fall outside the scope of this study as standalone libraries.

The fourth is *real-time delivery primitives*. GraphQL with subscriptions [64] and PartyKit [65] provide real-time delivery without satisfying the definition. GraphQL subscriptions extend client-side caching with server-pushed query updates, but each subscription is independent and the server tracks no per-client state. Conflict resolution and offline writes remain the application's responsibility. PartyKit provides Party rooms with WebSocket delivery and optional persistence through Durable Objects, but it supplies coordination primitives rather than integrated sync logic. Sync engines can be built on top of either, but the application must provide the per-replica state tracking.

Full offline capability is not required for inclusion. Zero explicitly excludes offline writes but remains in scope because the research question concerns generalization limits, not adherence to local-first ideals [50].

2.4.2. History of sync engines

Sync engines are not new. Boodman and Chase [21] identified Lotus Notes (1989) [66] as probably the first mass-market sync engine. Microsoft Exchange (1996), Dropbox (2007), Figma (2016), and Linear (2019) all built custom sync engines [21].

Earlier general-purpose attempts include CouchDB's document database (2005) [67, 68], Firebase's real-time data sync (2012) [69, 70], and Replicache's server reconciliation (2020) [71, 72]. Boodman and Chase [21] observed that these earlier systems suffered from one or more limitations: no fine-grained authorization, limited partial sync, non-standard data models, or limited business logic on read and write paths. Recent sync engines, including Zero [73], ElectricSQL [74], PowerSync [59], and Jazz [75], target these limitations, and this thesis investigates whether they address these limits in practice.

2.4.3. Current landscape

Several sync engines exist today, but they take different architectural approaches. Table 1 previews the 14 instances classified in this thesis along two dimensions: integration with the application backend, and offline-write support. Chapter 4 develops the full classification across eight architectural dimensions.

Table 1: The 14 sync engine instances across 13 projects discussed in this thesis, with integration model, offline-write capability, first public release, and current development status.[2] Chapter 4 classifies these engines across eight dimensions.

Engine	**Integration**	**Offline writes**	**First release**	**Status**
Zero [73]	Pluggable	Online only	2024 [76]	Stable [76]
Electric-SQL [74]	Pluggable	Read-only	2024 [77]	Stable [77]
PowerSync [59]	Pluggable	Full	2023 [78]	Stable [78]
Convex [26]	Bundled	Online only	2022 [79]	Stable [80]
InstantDB [81]	Bundled	Full	2024 [82]	Stable [82]
Jazz v2 [83]	Bundled	Full	2026 [83]	Alpha [83]
Classic Jazz [75]	CRDT	Full	2022 [84]	Mainte-nance [84]
Ditto [85]	CRDT	Full	2018	Stable
Automerge [17]	CRDT	Full	2017 [86]	Stable [87]
LiveStore [88]	Event-sourced	Full	2024 [89]	Beta [89]
RxDB [90]	Toolkit	Configurable	2018 [91]	Stable [90]
Turso Sync [92]	SQLite	In beta	2024 [93]	Beta [94]
TinyBase [95]	Toolkit	Full	2022 [96]	Stable [96]
CouchDB [67] + PouchDB [97]	Document	Full	2005 / 2013 [98]	Stable [67]

[2]Year is the project-reported start year of each engine in its current form.

Some engines, such as Zero [73] and ElectricSQL [74], plug into existing Postgres databases and keep the server as the source of truth. Others, such as Classic Jazz [75] and Ditto [85], replace the backend entirely with decentralized CRDT-based data models. Still others, such as Convex [26], provide a fully managed backend where all mutations run server-side. Jazz v2 takes a related bundled, server-authoritative approach with snapshot-based merging on a trusted server and centrally enforced permissions [84].

Even within the server-authoritative category, engines differ in basic ways. ElectricSQL handles only the read path (syncing data to clients via "Shapes"), leaving writes to the application's existing API [99]. Zero handles both reads and writes through query-driven partial sync [73]. PowerSync provides first-class offline support with persistent SQLite on the client [29], while Zero treats the client as a cache rather than a full replica, noting that "sync engines are examples of caches" [100].

2.5. Complexity and abstraction theory

This section introduces three theoretical concepts that provide the analytical lens for evaluating sync engines in Chapters 5 and 6.

2.5.1. Essential versus accidental complexity

Brooks Jr. [101] distinguishes essential complexity, which is inherent to the problem, from accidental complexity, which is caused by imperfect tools. Past breakthroughs in software engineering such as high-level languages and time-sharing reduced accidental complexity [101]. Their gains diminish as accidental barriers fall. As Brooks Jr. argues, what remains is the essential difficulty: deciding what software should do and verifying that it does so correctly [101].

This distinction helps explain what sync engines can and cannot do. Using this distinction, sync engines remove accidental complexity: network plumbing, retry logic, and cache invalidation. The essential complexity remains: data consistency, conflict semantics, and business invariants [8]. Brooks Jr. [101] argues there is no "silver bullet" that provides an order-of-magnitude improvement. If sync engines only remove accidental complexity, they cannot commoditize the essential difficulty of maintaining complex application invariants across replicas [8, 101]. Brooks Jr. [101] also advocates "buy versus build" for applications. This thesis tests whether the same logic extends to middleware that constrains application architecture.

2.5.2. Leaky abstractions

Spolsky [102] formulates the Law of Leaky Abstractions: "all non-trivial abstractions, to some degree, are leaky." Abstractions hide implementation details, but the underlying

reality surfaces when things go wrong [102]. TCP abstracts unreliable packet delivery, but network latency still leaks through [102]. SQL abstracts data retrieval, but the query planner leaks through as performance variance [102].

Spolsky [102] specifically identified the remote-as-local abstraction as historically problematic, citing protocols such as the Network File System (NFS) and Server Message Block (SMB). This is the promise of sync engines: write to a local database, and the engine handles the network [21]. As Spolsky notes, dealing with leaks competently requires learning how the abstractions work and what they hide [102]. Adopting a sync engine does not eliminate the need for distributed systems expertise.

Leaky abstractions provide the vocabulary to describe *where* sync engines fail. Leaks manifest as performance degradation, unexpected conflict behavior, or inability to express domain constraints. For example, a sync engine that hides conflict resolution may use default merge semantics [5]. Shapiro et al. [8] show that for certain data structures, no single default is correct. When the default choice is wrong for the domain, the abstraction leaks.

2.5.3. Information hiding and module boundaries

Parnas [103] argues that modules should be defined by the design decisions they hide, not by processing steps. A well-designed module encapsulates a decision likely to change, exposing only a stable interface. The design process should begin by identifying difficult or changeable decisions, then structure each module to hide one such decision from the others [103].

In Parnas's terms, a sync engine is a module that hides the "how data moves between client and server" decision. This works well when synchronization is a contained concern. When a sync engine requires the UI and the backend to share knowledge of specific CRDT structures or conflict semantics, sync logic leaks across module boundaries and can undermine information hiding [31, 103].

If synchronization is a cross-cutting concern, it resists modularization and therefore resists generalization [103]. This argument predicts where generic sync engines fail. Custom sync implementations may be chosen precisely because they allow better hiding of sync complexity within a specific module, rather than spreading engine-specific constraints across the entire stack [103].

2.6. Related work

Academic research on local-first software and synchronization is growing but remains limited in scope. Most work focuses on CRDT correctness and programming models,

while practitioner content discusses specific engine implementations. No academic framework exists for navigating the synchronization spectrum or deciding when to adopt a sync engine versus building custom synchronization.

Niinikoski [104] investigated the suitability of local-first architecture for browser applications using a mixed-method approach combining literature review and developer interviews. The thesis concludes that fully serverless local-first systems are rarely practical and that hybrid models are more feasible. This thesis extends that finding by investigating *where* the boundaries lie and what factors determine the appropriate position on the synchronization spectrum.

Zhang et al. [105] studied programmer experience when building collaborative web applications with CRDTs. They compared developer experience across Automerge, Yjs, and Collabs, finding that participants struggled with issues such as distinguishing local from collaborative data and correctly initializing replicas [105]. This thesis shifts focus from CRDT libraries to sync engines and from developer experience to architectural decision-making.

Haas et al. [106] proposed LoRe, a programming model for verifiably safe local-first software. LoRe uses static analysis to verify safety invariants and selectively applies coordination where required [106]. This addresses the correctness problem from a language-level perspective, whereas this thesis addresses it from an architectural and organizational perspective.

The broader replication literature provides foundational context. Saito and Shapiro [7] remains the most thorough survey of optimistic replication techniques and covers the design dimensions that underpin sync engine architectures. Preguiça [25] surveys the CRDT research that has developed since Shapiro et al.'s original formalization [8]. Margara et al. [107] provided a multi-dimensional classification of distributed data-intensive systems, but targeted server-side systems (Kafka, Cassandra, Spark) rather than client-sync engines. None of these works addresses the emerging category of sync engines or the adoption decisions that practitioners face when choosing between them.

Two recent works discuss the category directly but stop short of classification. Kleppmann and Riccomini [58] place sync engines within the replication design space in technical secondary literature, but stay at a conceptual level and do not classify current engines or adoption criteria [58, pp. 220–222]. Jayakar [108] proposes nine dimensions for comparing sync platforms, but covers a broader set of systems than this thesis, including file synchronization, collaborative applications, CRDT libraries, and game netcode.

The academic coverage of local-first software remains thin, concentrated around Kleppmann et al. [5] and related work from the same research group [49, 52, 53].

This thesis addresses the gap between academic foundations and practitioner reality by constructing a decision framework grounded in both.

2.7. Summary

This chapter established the theoretical foundations for analyzing sync engines. Distributed systems constraints, formalized through CAP and PACELC, shape all synchronization architectures by forcing trade-offs between consistency, availability, and latency [18, 19]. Applications sit on a synchronization spectrum from server-authoritative request-response to fully local-first architectures [5]. CRDTs enable local-first by guaranteeing convergence without coordination, but they have limitations: global invariants are difficult to express, and conflict resolution remains application-specific [7, 8].

Sync engines aim to package synchronization as reusable infrastructure, abstracting network complexity from application code [21]. Three theoretical lenses help evaluate where this abstraction succeeds and fails. Brooks's distinction between essential and accidental complexity suggests sync engines can remove network plumbing but not domain-specific consistency requirements [101]. Spolsky's Law of Leaky Abstractions predicts that sync engines will leak when default behaviors conflict with domain needs [102]. Parnas's information hiding criterion questions whether synchronization can be contained within a module boundary [103].

The related work reveals a gap: academic research focuses on CRDT correctness and programming models, while practitioners discuss specific engine implementations. Existing literature does not connect the synchronization spectrum to the build-versus-buy decision. Chapter 3 describes how this thesis addresses that gap through practitioner interviews, content analysis, and a case study.

3. Research Approach

This chapter details the research approach. The study draws on a literature review and three empirical sources: practitioner content analysis of 69 sources, semi-structured interviews with 13 practitioners, and a case study. Each empirical source contributes to different research questions, and the combination enables triangulation across independent perspectives.

3.1. Research questions and scope

The research questions introduced in Chapter 1 guide the study design. Each additional research question (ARQ) maps to a distinct findings chapter and draws on different combinations of data sources.

Main RQ: To what extent can client-side data synchronization be generalized across web applications?

Additional research questions:

1. **ARQ1 (Taxonomy):** What architectural patterns exist across the synchronization spectrum from server-authoritative to local-first? (addressed in Chapter 4)
2. **ARQ2 (Trade-offs/Limits):** What trade-offs and limits do these architectural patterns introduce? (addressed in Chapter 5)
3. **ARQ3 (Fitness):** How do application requirements determine sync engine fitness? (addressed in Chapter 6)

The inclusion criteria, detailed in Chapter 4, require an engine to be actively supported and to either self-describe as a sync engine or function as integrated client-server sync infrastructure for application data. Engines must also have sufficient practitioner documentation to classify across all eight dimensions.

3.2. Literature review method

The literature review draws on academic literature for theoretical foundations and on practitioner content for engineering rationale. The literature search drew on ACM Digital Library and IEEE Xplore, with Google Scholar, Semantic Scholar, and arXiv as supplementary databases. The Aalto University library proxy provided access to paywalled sources. The search proceeded in two complementary modes. The first was keyword-driven search across the databases, using compound queries that combined thesis topics rather than single broad terms. The second was citation-graph traversal

through Semantic Scholar, starting from foundational works [5, 7, 8] and following both forward and backward citations.

Keyword queries combined concepts tied to specific chapter topics. Examples include "CRDT" with "authorization", "partial replication" with "web", and "sync engine" with "architecture taxonomy". Publication date filters were broadened when initial searches returned few results and narrowed when they returned too many. Vendor and system names supplemented topical queries when looking for papers about specific engines. An agentic web search through Anthropic Claude Opus 4.5 occasionally located papers that keyword queries had missed, particularly when the relevant terminology was not yet settled. Citation-graph traversal surfaced foundational works in distributed systems theory as recurring references, not through direct keyword searches.

Topical relevance was the primary inclusion criterion. Academic papers had to address sync engines, local-first software, optimistic replication, conflict-free replicated data types, consistency models, partial replication, conflict resolution, or authorization in distributed systems. The search excluded tangential applications, such as CRDTs in virtual reality or 6G crisis-area communication, unless they supported a specific argument elsewhere in the thesis.

Practitioner sources complement the academic literature where the field is too recent for peer-reviewed coverage. They include sync engine documentation, engineering blog posts, and technical specifications cited throughout the thesis, alongside a structured corpus of 69 conference talks, podcast episodes, and technical presentations analyzed using the framework method described in Section 3.4. Together, the academic and practitioner sources establish the theoretical foundations presented in Chapter 2 and inform the taxonomy dimensions developed in Chapter 4.

Because the field is recent, the study uses sources with different evidentiary weight. Peer-reviewed literature supports theoretical claims about consistency, replication, CRDTs, and distributed systems limits. Technical books and research essays provide conceptual framing where peer-reviewed work is still sparse. Vendor documentation supports product capability claims, such as offline writes, Postgres integration, and replication mechanisms. Practitioner talks, podcasts, and blog posts provide design rationale and adoption experience. Discord and social media material is used only for narrow factual clarification or provenance. Central findings about generalization, adoption, and limits require triangulation across source types or direct interview evidence.

3.3. Analytical framework

This study uses framework analysis [109, 110], a qualitative method where analytical categories are defined in advance from the research questions and applied systematically

across data sources. Framework analysis is suited to applied research with specific questions, a limited time frame, and a priori issues, where the goal is structured cross-case comparison rather than purely emergent theory generation.

Following Gale et al. [110], the analytical framework was developed before data collection and refined during early analysis. The framework consists of three category sets corresponding to the research questions:

1. **Taxonomy dimensions (T1–T8):** Eight architectural dimensions for classifying sync engines, derived from the literature review and refined through practitioner content analysis: authority model (T1), conflict resolution (T2), read/write path (T3), sync unit (T4), integration depth (T5), partial replication (T6), client persistence (T7), and network topology (T8). The analysis later reframed client persistence as offline capability because practitioners discussed persistence through the functionality it enabled. Each source is rated on a three-level scale: Substantive (S), the source discusses the topic with enough content to serve as thesis evidence; Weak (W), a passing mention without elaboration; or None (N), not discussed. An initial four-level scale distinguished Strong from Moderate coverage, but that boundary could not be reliably assessed during AI-assisted source extraction, so the two levels were collapsed into Substantive.
2. **Trade-off categories (F1–F6):** Six trade-off dimensions that characterize fitness decisions: performance versus consistency (F1), simplicity versus flexibility (F2), developer experience versus overhead (F3), offline versus online optimization (F4), vendor lock-in concerns (F5), and schema flexibility (F6).
3. **Decision factors (D1–D6):** Six factors that influence the build-versus-buy decision: authorization complexity (D1), data volume constraints (D2), schema evolution requirements (D3), operational complexity (D4), ecosystem maturity (D5), and use case fit (D6).

The F1–F6 and D1–D6 codes structured the coding pass, but the analysis then reorganized them thematically. D1, D2, and D3 became trade-offs and limits alongside the F categories in Chapter 5. D5 and D6 appear directly in Chapter 6 as ecosystem maturity and use-case fit. D4 and additional decision factors from the interview material moved into the speed and developer experience, backend compatibility, offline, abstraction level, and build-versus-buy categories there.

The framework analysis is applied to two of the empirical sources: the practitioner content corpus and the interview sample, both detailed in the following sections; the case study reported in Chapter 7 draws on the resulting findings but is not framework-coded itself. Both sources are analyzed against the same four practitioner groups, which hold different perspectives on the sync decision:

- **Group A (Sync engine vendors):** Founders and engineers who designed general-purpose sync engines for other engineers to use in their applications, rather than application-specific sync code.
- **Group B (Engine users):** Engineers who adopted existing sync engines for production applications, representing the "buy" side of the build-versus-buy decision.
- **Group C (Custom builders):** Engineers who built custom synchronization rather than adopting an engine.
- **Group D (Researchers):** Academics and thought leaders working on sync foundations.

The shared group structure enables cross-source comparison across the two data sources, with coverage by group detailed in the speaker coverage and participants tables below.

To enable cross-source triangulation, a second coding layer compares each coded point against interview findings:

- **[CONFIRMS]:** The practitioner position validates an interview finding.
- **[EXTENDS]:** The practitioner position adds nuance or depth to an interview finding.
- **[NEW]:** The practitioner position surfaces an insight not captured in interviews.
- **[CONTRADICTS]:** The practitioner position conflicts with an interview finding.

This two-layer coding scheme enables systematic comparison across 69 practitioner content sources and 13 interviews while preserving the ability to identify emergent themes that fall outside the a priori categories.

3.4. Practitioner content corpus

This study draws on a corpus of 69 publicly available practitioner content sources: conference talks, podcast episodes, and technical presentations that discuss sync engine architecture. These sources constitute naturally occurring data: practitioners discuss architectural decisions, trade-offs, and limitations in semi-structured formats without researcher prompting. Many speakers designed and built the sync engines under study, so the corpus contains substantial technical detail and design rationale.

The practitioner content serves three purposes. First, it provides triangulation by capturing independently stated practitioner positions that can be compared against interview findings. Second, it extends coverage to practitioner groups where interview recruitment proved difficult. Third, it informed interview design by grounding questions in documented practitioner positions.

3.4.1. Source identification

Sources were identified through purposive sampling starting from three anchor points:

1. **Conference proceedings:** All recorded presentations from Local-First Conf 2024 and 2025 [111] and Sync Conf 2025 [112], the two dedicated conferences for sync and local-first software.
2. **Podcast series:** All episodes of the Local First Podcast [113] that cover sync architecture or engine design. Selected episodes from devtools.fm [114], Syntax [115], and How About Tomorrow?, in which guests discussed sync engines or local-first architecture.
3. **Snowball extension:** When a speaker appeared in one source, other appearances were traced across podcasts, blog posts, and conference presentations.

The final corpus contains 69 sources spanning 2023 to early 2026, a period when the sync engine ecosystem matured rapidly. Podcast venues other than the Local First Podcast were sampled episode-by-episode rather than enumerated, so some relevant episodes from those venues may fall outside the corpus even when the speaker appears via another source.

Sources were included if they (1) feature a practitioner or researcher discussing sync engine architecture, design decisions, or adoption experience, (2) provide sufficient technical depth to map against the taxonomy dimensions or decision factors, and (3) were publicly available as of February 2026. Sources that discuss local-first or sync only in passing were excluded.

3.4.2. Corpus composition

Table 2 shows the distribution of sources across venues. Conference talks account for roughly half the corpus, and podcast episodes supply the complementary long-form discussion format.

Table 2: Practitioner content corpus composition by venue.

Venue	**Sources**	**Period**
Local-First Conf 2024	6	Jun 2024
Local-First Conf 2025	15	Jun 2025
Sync Conf 2025	13	Nov–Dec 2025
Local First Podcast	19	Jan 2024 – Jun 2025
devtools.fm	6	Jul 2024 – Jan 2026
Syntax	6	May 2024 – Nov 2025
How About Tomorrow?	2	Jun 2023, Mar 2025
Other (company videos)	2	Oct 2024, Jan 2026
Total	**69**	

3.4.3. Speaker coverage

The corpus provides recurring-speaker coverage across vendors (Group A), custom builders (Group C), and researchers (Group D), as Table 3 shows. Engineers using sync engines (Group B) are absent from recurring conference and podcast appearances; interviews close this gap. Several speakers appear in multiple venues, enabling cross-source consistency checks on the same practitioner's positions.

Table 3: Recurring speakers in the practitioner content corpus by practitioner group. Group A: sync engine vendors, B: engineers using sync engines, C: custom sync builders, D: researchers and thought leaders. Source counts include appearances across all venues.

Group	Speaker	System	Sources
A	James Arthur	ElectricSQL	5
	Aaron Boodman	Zero	5
	Anselm Eickhoff	Jazz	4
	Adam Fish	Ditto	3
	Kyle Mathews	ElectricSQL	2
	James Cowling	Convex	1
	Conrad Hofmeyr	PowerSync	1
B	*(no recurring speakers in corpus; coverage via interviews)*		
C	Tuomas Artman	Linear	3
	Arushi Bandi	Figma	1
	Ben Holmes	Simple sync engine	1
	Angelique Nehmzow	Notion	1
	Zhanna Sharipova	Anytype	1
D	Peter van Hardenberg	Ink & Switch	3
	Martin Kleppmann	Automerge	2
	Carl Sverre	mycelial	1
	Adam Wiggins	Ink & Switch	1

3.4.4. Processing pipeline

Each source was processed through a two-stage pipeline: AI-assisted transcription and summarization, followed by manual validation.

1. **Extraction:** Each source was processed using Google Gemini 3.1 Pro. Video sources (67 of 69) were provided as YouTube links, which the model can access directly. Two podcast episodes without video were provided as text transcripts. The model was given a structured extraction template with six categories: speaker background, technical claims, trade-offs acknowledged, comparisons to alternatives, opinions and predictions, and notable quotes. The full extraction prompt appears in Appendix A. All claims from video sources were extracted with timestamps. No thesis-specific context (research questions, analytical framework, or interview findings) was provided to the model, ensuring that the extraction captured source content without researcher bias.
2. **Manual validation:** The AI-generated summaries were validated against the original sources by reviewing timestamped segments. Claims that could not be verified

from the source material were discarded. Ambiguous points were re-checked by rewatching the relevant segment.

AI assistance made the corpus feasible to process, but the model output was treated as an index into the sources, not as evidence on its own. Evidence-bearing claims used in the findings chapters were checked against timestamped source segments, transcripts, documentation, or cited literature before they were treated as support. Unverifiable or ambiguous points were discarded, kept as background context, or rechecked against the relevant segment. Synthesis claims were retained only when supported by more than one evidence stream or by a clearly stated theoretical argument. The appendices preserve the extraction prompt, interview topic guide, and per-source coverage ratings.

The study was coded by one researcher, so it does not report inter-coder reliability. To make the single-coder analysis auditable, the thesis uses a fixed coding template, explicit category definitions, source-level coverage tables, and triangulation across practitioner content, interviews, literature, and the case study, with quantified counts reported for the practitioner-interview comparison.

3.5. Semi-structured interviews

Semi-structured interviews complement the practitioner content corpus with first-hand accounts of design decisions, adoption experience, and observed limits. This section describes the participant groups and rationale, recruitment, the resulting participant set, and the interview protocol.

3.5.1. Participant groups and rationale

The analytical framework defines four practitioner groups, each contributing to different research questions. Interview recruitment targeted three of the four: Groups A, B, and C. Groups A and C address the build-versus-buy tension from opposite sides. Vendors observe customer adoption patterns at scale and speak to design trade-offs and observed limits, contributing across all three ARQs: taxonomy dimensions from engine architecture (ARQ1), trade-offs and limits from design decisions (ARQ2), and fitness observations from customer patterns (ARQ3). Custom builders articulate why general-purpose solutions were rejected, contributing primarily to ARQ2 (limits of generalization) and ARQ3 (decision factors). Group B users provide first-hand production adoption accounts, contributing primarily to ARQ2 and ARQ3: what motivated the decision, what the engineer experience is like, and where the engines fall short. Group D was de-emphasized in interview recruitment because the practitioner content corpus already covers researcher positions broadly through conference talks and podcast episodes; no Group D interviews were completed.

3.5.2. Recruitment

Recruitment combined four channels. Snowball referrals from interviewees yielded the most participants: early interviewees referred colleagues and contacts in the sync engine community, and one vendor interviewee sent introductions to four additional contacts in a single message. Community outreach via Discord was the second most effective channel. Research introductions were posted in sync engine Discord servers (Zero, ElectricSQL, Convex, PowerSync, Jazz) and community servers (localfirst.fm, local-first/sync-conf), and several Group B participants responded directly. The thesis advisor connected the author with one Group A participant through a warm introduction. Cold outreach via X, Bluesky, and LinkedIn attracted some interest but converted fewer interviews than direct community engagement, with one Group C participant responding to a LinkedIn contact.

Recruitment was ongoing throughout the interview period (February–March 2026). Early interviews informed later recruitment: after completing three Group A vendor interviews, recruitment shifted toward Group B users and Group C custom builders where coverage was thinner.

Four factors gave the interviews technical depth. The author's professional position as an engineer at a B2B SaaS company evaluating sync engine adoption gave credibility with practitioners. The student and research status invited openness, as participants were willing to share candidly in an academic context. Personal experience with sync engines from side projects and experimentation supplied shared technical vocabulary. The practitioner content analysis and literature review provided familiarity with each participant's public positions and technical context before each interview, which kept conversations focused.

3.5.3. Participants

Table 4: Interview participants by group. Participants marked with * are pseudonymized at their request; others consented to named attribution. IDs (A1–C3) are used to reference participants in later sections.

Group	ID	Participant	Domain	Engine
A	A1	Pekka Enberg, Turso	Database sync (libSQL/ SQLite)	Turso Sync
	A2	Anselm Eickhoff, Jazz	Collaborative app framework	Jazz
	A3	Kevin De Porre, ElectricSQL	Postgres read-path sync	ElectricSQL
B	B1	Valter Kraemer, Astra	Brand protection SaaS	Zero
	B2	Sindre Trosterud, Tilit	Multi-tenant bookkeeping	Zero
	B3	Developer, non-profit*	Non-profit fund tracking	Zero
	B4	Abdul Osman, Gessic	Digital marketing and mobile apps	Zero
	B5	Krystian Gebis, ShipperCRM	Freight brokerage CRM	Zero + Convex
	B6	Engineer, field documentation*	Field site photo documentation	PowerSync
	B7	Engineer, B2B SaaS*	Field operations and equipment audits	PowerSync
C	C1	Jan Johannes	Browser development (shared spaces, messaging)	CouchDB + custom
	C2	CTO, field engineering platform*	Enterprise planning with mobile field tools	Custom
	C3	Jaakko Sirén, CRACI	CI/CD security and CRA compliance	TanStack DB + gRPC + custom

Table 4 lists the 13 interview participants across three groups. Group D (researchers) remains uncovered by interviews but is addressed through the practitioner content corpus.

The Group A sample stopped at three interviews. Within that sample, the three vendors independently raised the same architectural themes around authorization, sync

scope, and read-path generalization. Chapter 5 examines these themes in detail. An interview with an Evolu maintainer, suggested by A2, could have surfaced additional local-first positions that this sample does not cover.

Group B has the most participants with seven interviews, covering the variety of sync engine users across domains, engines, and team sizes. Five used Zero, providing strong coverage of that engine's adoption experience. Two used PowerSync: one a two-engineer team building field photo documentation on mobile only, and one a larger team building field operations and equipment audits across mobile and web.

Group C has three participants with contrasting approaches: one built on the existing CouchDB sync protocol and sits at the boundary between Groups B and C, one built fully custom synchronization from scratch, and one assembled a custom stack from modern components including TanStack DB and gRPC streaming.

3.5.4. Interview protocol

All interviews followed a semi-structured format of approximately 45 minutes. Of the 13 interviews, 12 used remote video calls and 1 took place in person; 9 were conducted in English and 4 in Finnish. Finnish quotes presented in this thesis were translated by the author. Each interview was recorded with verbal consent, and a written privacy notice offered participants the choice between named attribution and pseudonymization. Named attribution was most meaningful for Group A, where vendor founders and engineers speak as public representatives of their engines and their identity is part of the evidence base. For Groups B and C, the analytical signal comes from the application domain and engine choice, so participants who preferred pseudonymization could be reported under a generic role label while domain and engine remained visible in the table without weakening the analysis.

Each interview followed the same arc, adapted to the participant's group:

1. Background and context: the participant's product, role, and relationship to sync.
2. Technical architecture: how sync works in their system, or how they designed it for vendor participants.
3. Pain points and trade-offs: where things work well and where they break.
4. Decision factors: what drove the choice to build, buy, or design a sync engine.
5. Reflections: what they would do differently and their perspective on the field.

Questions were open-ended with prepared probes tailored to each group, drawn from the topic guide in Appendix B. Later interviews included probes informed by earlier findings, for example testing whether patterns observed in vendor interviews matched user experience in practice. Interviews were transcribed using Google Gemini 3.1 Pro in both English and Finnish, and analyzed the same day.

3.6. Framework application and triangulation

This section describes the coding procedure applied to practitioner content and interviews, and reports the resulting triangulation counts and how contradictions were treated.

3.6.1. Coding procedure

The analysis applied the analytical framework to both data sources following the seven stages of framework analysis [110]: transcription, familiarization, coding, developing the analytical framework, applying the framework, charting, and interpretation.

For practitioner content, the analysis proceeded in three passes:

1. **Open summarization:** AI-assisted extraction generated an open summary of each source that retained emergent themes outside the a priori categories.
2. **Framework mapping:** extracted points were tagged against the T1–T8, F1–F6, and D1–D6 categories.
3. **Triangulation coding:** each coded point was compared against interview findings and tagged with one of the triangulation codes.

For interviews, each interview was analyzed the same day using a structured analysis template with three sections corresponding to the research questions:

1. **ARQ1 (taxonomy):** the participant's sync architecture mapped against eight dimensions: authority model, conflict resolution, read/write path, sync unit, integration depth, partial replication, client persistence, and network topology.
2. **ARQ2 (trade-offs and limits):** engineering trade-offs, where the engine or approach struggles, and limits of generalization.
3. **ARQ3 (fitness and decision):** where the engine works well, engineer experience, what drove the architectural choice, and what alternatives were considered.

Notable quotes were extracted during analysis and linked to the relevant framework categories. Cross-case triangulation tracked how each participant's experience maps to the same dimensions across all interviews, with a coding table maintained as analysis material. The per-source practitioner content coverage ratings are included in Appendix C.

3.6.2. Triangulation results

Triangulation coding yielded 235 coded points across the 69 practitioner content sources: 103 confirmations (44%), 70 extensions (30%), 55 novel insights (23%), and 7 contradictions (3%). The high confirmation rate and low contradiction rate are consistent with convergence between interview findings and independently stated prac-

titioner positions, though as a single-coder study the rates support rather than prove convergence.

Contradictions were treated as evidence about scope. When sources disagreed, the analysis asked whether the disagreement followed from application domain, architecture, or unresolved practitioner judgment.

This most often affected conflict resolution. Zero users and several custom builders described conflicts as rare or avoidable, while CRDT-focused practitioners and conflict-visible systems emphasized conflict complexity. The final claim was therefore narrowed: conflict resolution is not a universal adoption concern, but it becomes central when full offline capability combines with concurrent edits to the same records. This conditional framing is used in Chapter 4, Chapter 5, and Chapter 6. The findings chapters return to these contradictions where they affect the argument, rather than treating the low contradiction rate as full consensus.

The framework categories served as a starting point, not a constraint. Several findings emerged outside the a priori categories. Vendor interviews pointed to authorization as a universal barrier to offline sync. User interviews showed adoption motivated by API simplification rather than sync ideology. A custom-builder interview drew the distinction between engineer-desired and customer-demanded features. The analysis incorporated these emergent findings into Chapter 5 and Chapter 6.

The combined data feeds into the findings chapters differently depending on the research question. For ARQ1 (taxonomy), conference talks by engine designers provide detailed architectural descriptions that complement documentation and interviews. For ARQ2 (trade-offs and limits), podcast discussions reveal candid assessments of where engines struggle and break down. For ARQ3 (fitness), talks by custom sync builders articulate why general-purpose solutions were rejected, and the interview material with adopters supplies the in-production view that the practitioner content corpus covers less directly.

3.7. Case study method

The study includes a case study at Teamspective, a B2B SaaS company that builds employee feedback, engagement, and people analytics tools for leadership enablement. The author works as an engineer at Teamspective, and the second thesis advisor works at the company, which allowed close collaboration on the case study design. Prior to the thesis, the author had not used any sync engine under study in production and had no financial or organizational ties to any sync engine vendor. Side-project experimentation with sync engines provided practical familiarity that informed interview design and case study work. Throughout the case study, an implementation diary captured decisions,

surprises, and operational events session by session, alongside deployment logs, support tickets, and commit history. These records, detailed in the Evidence subsection below, ground claims in artifacts rather than recall. Section 8.3 addresses residual concerns about insider access and self-evaluation. The case study complements the breadth of the 13 interviews by capturing implementation-level constraints, such as managed-infrastructure restrictions and authorization edge cases, that documentation and interviews do not surface. The case study evaluates the adoption of Zero for an existing production application. The case study tests whether the trade-offs and decision factors identified through interviews and practitioner content materialize in practice.

The case study targets the workspace configuration domain, a part of the application with known performance and data staleness issues that exercises relational data with permissions, roles, and group membership. Chapter 7 documents the full engine selection rationale, target domain justification, and implementation details.

3.7.1. Implementation approach

The team phased the integration into two slices, sized to limit exposure if any single change failed. The first covered user profile settings, a low-risk surface with two tables that validated the integration end to end. The second extended to workspace-scoped data through the feature flags page, and both shipped to production during the thesis period. Two parallel investigations ran alongside: a stress test that ported a complex query to Zero Query Language (ZQL), and an audit of the application's API endpoints against Zero migration criteria.

3.7.2. Evidence

The case study captures evidence in four shapes. Implementation feasibility comes from hands-on integration of Zero into Teamspective's stack, including schema generation, queries, mutators, build and deployment plumbing, and production infrastructure stand-up. Boundary testing draws on a stress test that ports a complex query to ZQL, an audit of the application's API endpoints against Zero migration criteria, and the slice deployments. Operational friction appears during managed-Postgres deployment, including provider support tickets, schema migration handling, and a high-availability failover incident. Adoption signals come from peer engineers asking to use Zero in other feature work. The implementation diary records all four evidence types. Deployment logs, support tickets, and commit history provide additional artifacts for operational friction and implementation work.

3.7.3. Analysis

The case study evaluates findings against the taxonomy developed in Chapter 4, the trade-off framework from Chapter 5, and the decision factors identified in Chapter 6. The analysis examines where on the synchronization spectrum the Zero-based architecture lands, which taxonomy dimensions it exercises, which trade-offs materialized in practice, and where the sync engine abstraction leaks.

3.8. Summary

This chapter described the methodology for investigating how broadly client-side data synchronization generalizes across web applications. The study combines a literature review of distributed systems theory with three empirical sources: practitioner content analysis of 69 sources from conferences and podcasts, semi-structured interviews with 13 practitioners across three groups of vendors, adopters, and custom builders, and a case study at Teamspective. Framework analysis structures the cross-case comparison across eight taxonomy dimensions, six trade-off categories, and six decision factors.

The practitioner content corpus and interview sample are analyzed against the same practitioner-group schema, supporting triangulation between independently stated positions and direct interview evidence. The case study grounds the findings empirically by evaluating Zero adoption for an existing B2B SaaS application against the taxonomy, the trade-off framework, and the decision factors. The following chapters present the results: Chapter 4 develops the synchronization pattern taxonomy, Chapter 5 identifies trade-offs and limits, and Chapter 6 evaluates application fitness and decision factors. Chapter 7 then applies and stress-tests these findings against Zero adoption at Teamspective.

4. Synchronization Pattern Taxonomy

This chapter addresses ARQ1: What architectural patterns exist across the synchronization spectrum from server-authoritative to local-first? The taxonomy classifies sync engines along eight dimensions derived from distributed systems literature, practitioner content analysis, and semi-structured interviews.

4.1. Classification methodology

This section explains how this thesis classifies sync engines: first by surveying prior work on adjacent classifications, then by describing the empirical approach used here, and finally by listing the engines in scope.

4.1.1. Existing approaches to sync engine classification

Existing classification work in this area comes from two sources: practitioner content that attempts to map the sync engine category directly, and academic literature on adjacent topics like replication, consistency, and CRDTs that informs the framing without classifying sync engines as a category. Neither produces a systematic architectural taxonomy of sync engines as integrated products, and this thesis builds on both.

Practitioner sources are not peer-reviewed, but they contain most of the classification work in the reviewed material, since academic study has so far focused on adjacent topics rather than sync engines as a category. These sources lack consistency in how they define sync engine architectures. Marketing categories such as "local-first" and "real-time" mix different architectural concepts under the same terms [13]. Browne [13] observes that "local-first" alone conflates three separate goals: offline functionality and speed, real-time updates, and collaboration. The localfirst.fm community comparison lists over 40 attributes across categories including networking, data storage, synchronization strategy, and authentication [57], but this resource is self-reported by engine vendors rather than systematically derived from architectural analysis. Jayakar [108] proposes a practitioner taxonomy of sync platforms across nine dimensions grouped into data model, systems requirements, and programming model. Jayakar's map compares a broader set of sync systems than this thesis, including file synchronization, collaborative applications, CRDT libraries, and game netcode, while this thesis focuses on general-purpose sync engines for application development. Several of Jayakar's dimensions overlap with this taxonomy: offline support corresponds to offline capability, centralization to authority model, and flexibility to integration depth. Jayakar's workload dimensions, including data size, update rate, input latency, and concurrent clients, are

treated as application fitness factors in Chapters 5 and 6, not as engine taxonomy dimensions.

Prior published work classifies adjacent concepts but not sync engines as a category, including both peer-reviewed publications and research essays where the field is too recent for peer review. Saito and Shapiro [7] classify optimistic replication along authority structure, transfer payload, and conflict resolution dimensions. Weidner [12], in a research essay, classifies server architectures for real-time collaborative editing. Margara et al. [107] provide a multi-dimensional classification of distributed data-intensive systems, covering databases and stream processors but not client-sync engines. Viotti and Vukolić [9] present a hierarchy of consistency models that underpin sync engine guarantees. Almeida [51] survey CRDT design approaches, classifying the data structures that some sync engines build on. Pandey et al. [116] outline a vision for local-first distributed property graphs, identifying challenges in extending CRDT-based replication to graph-structured data. De Porre [117] addresses programming models for replicated data, formalizing how sequential code can be adapted for eventually consistent systems. None of these works classifies sync engines as integrated products along architectural dimensions.

4.1.2. Approach taken in this thesis

This thesis takes a narrower and more empirical approach. It classifies general-purpose sync engines for application development, not file synchronization systems, collaborative editing systems, CRDT libraries, or game networking systems. It also separates engine architecture from application fitness. Authority model, read/write path ownership, and partial replication describe what an engine is. Workload factors such as data volume, update rate, input latency, and concurrent clients describe where that engine fits.

This separation is the main difference from both practitioner comparison tables and classical replication taxonomies. Practitioner resources are useful for coverage, but they often rely on vendor self-reporting or mix product capabilities with use-case fit. Replication literature provides foundational dimensions [7], recapped in Chapter 2: who has authority, what gets transferred between replicas, and how systems resolve conflicts. Current sync engines package those mechanisms with product-level choices, including integration depth, write-path ownership, and partial replication strategy. Those product-level choices determine adoption and fitness, which is why they become first-class dimensions here.

This taxonomy draws on three sources. The distributed systems literature provides established dimensions for replication systems [7], complemented by research essays on collaborative-editing architectures [12]. The practitioner content analysis of 69 sources,

described in Chapter 3, provides coverage statistics for each dimension and surfaces practitioner framings absent from academic literature. Semi-structured interviews with 13 practitioners validate which dimensions matter in practice and reveal their relative importance. The dimensions are ordered by practitioner-perceived importance, with the first six receiving substantial interview coverage and the last two, sync unit and network topology, included for completeness.

A dimension is included if it is *architecturally differentiating*: it separates sync engines into categories with meaningfully different properties. Four candidate dimensions were excluded, reconceptualized, or absorbed into others on this basis. Client persistence mechanism, whether IndexedDB, SQLite, OPFS, or in-memory storage, is not retained as a separate taxonomy dimension here. The taxonomy classifies the offline capability that persistence enables rather than the storage mechanism behind it. The initial analytical framework described in Chapter 3, which defines the a priori coding categories applied across all data sources, coded client persistence as a dimension (T7), but analysis showed that practitioners discuss persistence in terms of what it enables rather than as a choice in itself. Interviews confirmed offline capability as the most important differentiating dimension, leading to this reconceptualization during analysis, consistent with the framework refinement stage of framework analysis [110].

Three other candidate dimensions were not retained as separate dimensions in the final taxonomy, though for different reasons. Authorization architecture was reclassified rather than absorbed: all three vendor interviewees (A1, A2, A3) identified it as a shared cross-engine challenge, but the architectural choices it surfaces are common pain points rather than differentiators, so it is examined as a cross-cutting trade-off in Chapter 5 and a decision factor in Chapter 6. Data model is absorbed into integration depth and sync unit. For example, a Postgres-pluggable row-sync engine, a CRDT document engine, and an event-log engine differ both in what stack they require and in what unit they replicate. Consistency is similarly absorbed into offline capability, authority model, read/write path architecture, and conflict resolution strategy, with the resulting invariant and coordination trade-offs examined in Chapter 5. Read/write path architecture (T3) is kept as a first-class dimension because the read/write split is a recurring pattern across the classified engines, even though reviewed academic taxonomies and community comparisons typically treat it as part of other dimensions. The read/write path subsection below examines its scope.

The dimensions form a partial hierarchy of constraints. Offline capability constrains authority model: full offline writes require either queued reconciliation under server authority or decentralized authority through mechanisms such as CRDTs [8] or revision-tree replication [67]. Authority model in turn constrains conflict resolution strategy: server-authoritative systems can use mechanisms such as server reconciliation, optimistic concurrency control, last-write-wins, or snapshot-DAG merging, while decen-

tralized systems converge through CRDTs [8] or revision-tree replication [67]. Despite these constraints, meaningful architectural choices exist at each level. The remaining dimensions, namely integration depth, partial replication, read/write path, sync unit, and network topology, vary independently of this hierarchy, as Figure 2 illustrates.

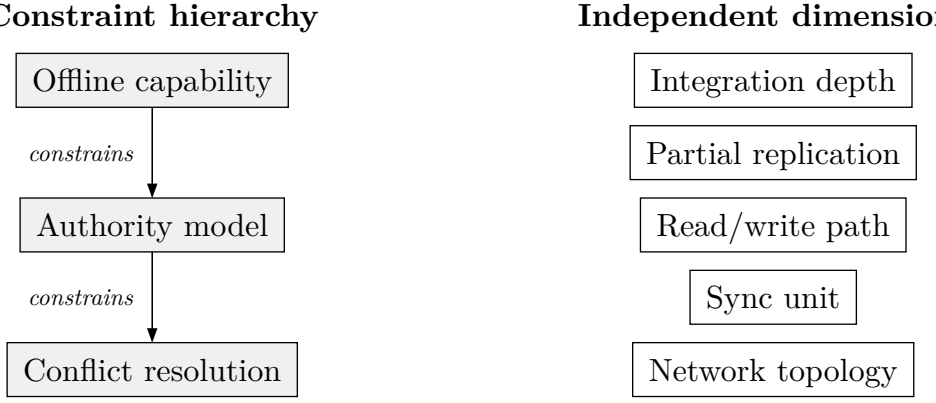

Figure 2: Taxonomy dimension structure. Offline capability constrains authority model, which constrains conflict resolution. The remaining five dimensions vary independently.

The classification applies the broad definition introduced in Chapter 2: a sync engine keeps client-side application state synchronized with one or more authoritative replicas. This covers systems with different persistence models, write locality, and backend ownership. Convex sits at the persistence boundary: its client state is reactive and in-memory, not a durable replica [26]. Automerge sits at the data-model boundary: it synchronizes CRDT documents rather than queryable relational state, making it the broadest data-model variant in the taxonomy [17, 55].

4.1.3. Scope and engines classified

This taxonomy is scoped to engines that are actively supported and either self-describe as sync engines or act as integrated sync infrastructure between client and server for application data. Engines must also have enough practitioner documentation to classify across all eight dimensions, which excludes several adjacent systems. Two are deprecated or unmaintained: Realm Sync, the cloud sync component of MongoDB Atlas Device Sync, reached end-of-life in September 2025 [118], and Triplit [119] has been inactive since early 2026. Firebase Realtime Database and Firestore [69] position themselves as backend-as-a-service platforms; they do not appear in the local-first practitioner corpus. Yjs [15] and Loro [16] are CRDT libraries; they ship no server runtime and do not position themselves as sync engines. Liveblocks [120] and Y-Sweet [121] provide infrastructure for collaborative editing, not general application sync. DXOS [122] and NextGraph [123] are frameworks where sync is one component among many. Basic [124] was considered but its conflict resolution and partial sync semantics are not yet documented in enough detail to classify.

Two boundary cases warrant inclusion despite their unusual profile. CouchDB and PouchDB remain actively supported and were the subject of a Local-First Conf 2025 talk by their co-creator [125]. Automerge [55] is included because van Hardenberg [17] describes it as a local-first sync engine, and automerge-repo provides networking and storage infrastructure that satisfies the definition proposed in Chapter 2, while operating on CRDT documents rather than queryable relational state.

The taxonomy classifies 14 sync engine instances across 13 projects: Automerge [17], Convex [26], CouchDB [67] with its JavaScript client implementation PouchDB [97], Ditto [85], ElectricSQL [74], InstantDB [81], Jazz in its Classic [75] and v2 [83] forms, LiveStore [88], PowerSync [59], RxDB [90], TinyBase [95], Turso Sync [92], and Zero [73]. Jazz appears twice because architectural changes between Classic and v2 cross multiple taxonomy dimensions, placing the two versions in different clusters.

The practitioner content corpus provides context on which dimensions practitioners actually discuss: substantive coverage runs between 40 and 67 of the 69 sources, or 58% to 97%. Because the corpus was selected for sync engine relevance, this coverage cannot independently validate the dimension selection, but it does show that each chosen dimension surfaces as an architectural concern in practitioner discourse. Partial replication (T6) sits at the lower bound with 40 of 69 sources. The gap is examined in that dimension's section below.

4.2. Taxonomy dimensions

The taxonomy identifies eight dimensions along which sync engines differ architecturally, ordered by practitioner-perceived importance. The first six dimensions received substantial interview coverage. The last two, sync unit and network topology, are included for completeness but showed dominant values with limited practitioner discussion.

4.2.1. Offline capability

Offline capability describes what functionality remains available when the network is unavailable. This dimension is the primary divider among sync engines: it constrains authority model, determines whether a conflict resolution strategy is needed, and narrows the field of viable engines for a given application.

Sync engines fall into three categories along this dimension. Engines with *no* offline capability require the network for all operations. Convex maintains only an in-memory cache with optimistic UI but no client-side persistence [26]. Engines with *read-only* offline capability cache data locally so reads work without the network, but writes require a server connection. Zero persists data to IndexedDB [126], but actively rejects

mutations while offline to prevent data loss, as the engine does not robustly support offline writes [50]. Zero recommends that application engineers implement UI that prevents user input while offline, acknowledging that even with mutation rejection, edge cases such as textarea input can lead to lost work.[3] ElectricSQL syncs data to clients via Shapes but provides no write path at all [99]. Engines with *full* offline capability support both reads and writes without the network, synchronizing changes on reconnection. PowerSync [59], Classic Jazz [75], Jazz v2 [83], InstantDB [127], Ditto [85], Automerge [17], RxDB [90], LiveStore [128], CouchDB [67], and TinyBase [95] all provide full offline support. Turso Sync provides offline writes through its embedded libSQL replica [94].

"The network is optional" is a central local-first ideal [5], yet the offline requirement is determined by the application domain rather than engineering preference. Interviews confirm a sharp divide. B6, who builds field documentation software using PowerSync, described offline as "more than essential," noting that field sites are often remote with unreliable connectivity. This requirement eliminated most engines from consideration. B6′s field workers cannot use an application that stops working when the network does. B6 also reported that this requirement leaves few alternatives, even when the chosen engine frustrates the team. B7, who builds a B2B field-operations platform using PowerSync, confirmed the same constraint from a different domain: field technicians "lose connectivity when they go under buildings" during equipment audits. Both PowerSync users independently confirmed that the engine was the only viable option for their offline requirements. C2, who built custom SQLite synchronization for a field engineering platform, confirmed the same constraint: "Offline writing is the most important feature [for any sync engine we'd adopt]." C1, who builds browser-based shared spaces using CouchDB, requires days-long offline capability for user data, relying on CouchDB's built-in replication protocol.

On the other side of the divide, four of the five Zero users (B1, B2, B4, B5) reported no need for offline capability. B3 is the exception: poor connectivity at some user locations made offline reads a factor in choosing Zero. These participants primarily value IndexedDB persistence for fast application startup rather than offline operation. B5 confirmed that Convex, which B5 uses for a separate internal tools project, has no client-side persistence at all. C3 similarly described the TanStack DB-based architecture as deliberately online-only, calling it "Sync Light" to distinguish from full offline-capable synchronization. A3 explained that ElectricSQL deliberately excluded offline writes because authorization made it untenable, choosing instead to serve the 80 to 90 percent of users who do not require offline.

[3]Zero offline mode documentation: https://zero.rocicorp.dev/docs/connection#offline. Aaron Boodman, Discord announcement in #rocicorp-asks-you, 2026.

The practitioner content analysis confirms this divide independently. Esterhazy [129] argues that "offline capabilities should not be seen as a binary choice" but rather as a slider, supporting the three-value classification used here. Boodman and Chase [21] explicitly frame Zero as trading offline writes for a superior online experience, while Fish [85] position Ditto's mesh networking as enabling "true offline" in environments with severe bandwidth constraints.

The offline requirement changes the question: it constrains which engines are viable, which authority models are possible, and how much architectural complexity the application must absorb. Practitioners who need full offline capability face fewer viable engines and higher architectural complexity, while those without the requirement can choose from a broader set of simpler engines. The platform maturity implications of offline capability, including the gap between native mobile and browser-based implementations, are examined in Chapter 5.

4.2.2. Integration depth

Integration depth describes how much of the application stack a sync engine replaces or constrains. This dimension determines whether adopting a sync engine is an incremental change to an existing system or a commitment that constrains the rest of the stack. Following the sync-engine versus syncing-datastore distinction introduced in Chapter 2, this taxonomy classifies backend replacement as integration depth rather than as an exclusion criterion.

This dimension has three values. *Bundled platform* engines replace the backend entirely, providing database, sync, and often authentication as a single service. Convex [26], Classic Jazz [75], Jazz v2 [83], and InstantDB [81] follow this model, offering simpler initial setup at the cost of constraining the data model and server-side logic [13]. A2 frames Jazz against the pluggable model: the bundled category packages the sync engine together with its own database rather than connecting a sync engine to an existing one. Turso Sync requires adopting Turso Cloud and libSQL [92], replacing the database layer rather than the full backend. *Pluggable* engines work alongside an existing database, adding sync as a layer without replacing existing infrastructure. Zero [50], ElectricSQL [74], and PowerSync [59] all connect to existing Postgres databases, enabling incremental adoption. PowerSync also supports MongoDB as a source database [59]. *Toolkit* engines provide building blocks that the engineer assembles into a complete system. RxDB [90] offers pluggable storage backends and optional replication plugins but leaves the sync backend to the engineer. Automerge follows the same pattern with pluggable storage and network adapters: the engineer provides client persistence (IndexedDB, filesystem, or a custom backend such as object storage) and operates the sync server [130].

Mathews et al. [88] contrast ElectricSQL's "modular" approach, where engineers bring their own database and client store, with the "all-in-one" packages offered by other sync engines. Practitioner content confirms that engineers frame sync engines primarily in terms of how deeply they integrate into the application stack. This distinction maps to a classical buy-versus-build trade-off. Bundled platforms absorb the accidental complexity [101] of integrating database, sync, and authentication by replacing those layers with a single managed service. Pluggable and toolkit engines instead accept more of that integration work in exchange for keeping the existing stack. Chapter 5 examines how this trade-off plays out in adoption decisions.

Among the interviewed adopters and custom builders, incremental adoption is a dominant requirement. B5, who migrated approximately 40% of a freight brokerage CRM to Zero over one year, described the engine's appeal in terms of integration shallowness: Zero embeds gradually as a cache layer without forcing a backend rewrite. The migration is deliberately partial. User-facing CRM data flows through Zero for instant reactivity, while the 51,000-record shipper reference database remains on tRPC queries because the volume exceeds what is practical to sync to clients. B4 similarly uses Zero for all user-facing state but routes long-running tasks such as large language model (LLM) inference through async mutators, keeping sync and non-sync workloads separate. C2 needed to adopt sync one entity type at a time, which only pluggable or toolkit engines support. B7 shows the depth of custom infrastructure that can surround a pluggable engine at enterprise scale, using PowerSync for read sync alongside a custom backend with transaction dispatching, failure recovery, and schema transformation tooling across hundreds of tables. The consequences of these integration choices for complex applications, including what sync engines cannot replace and the limits that force partial adoption, are examined in Chapters 5 and 6.

4.2.3. Authority model

Authority model describes which replica holds the canonical version of the data [7]. This dimension maps to the single-master versus multi-master distinction in optimistic replication [7] and to the consistency-availability trade-off in the PACELC framework discussed in Chapter 2 [19]. As established in the classification methodology, full offline writes constrain the authority model. Among the engines classified in this taxonomy, those supporting full offline writes are either server-authoritative with queued reconciliation, or decentralized through CRDTs [8] or revision-tree replication [67].

The majority of classified sync engines are *server-authoritative*. The server holds the source of truth and clients function as caches that receive updates. Zero [50], Convex [26], ElectricSQL [99], PowerSync [59], InstantDB [81], Jazz v2 [83], and Turso Sync [92] all follow this model. Server-authoritative designs are simpler because writes serialize

through one node. While the server is unreachable, no write can be authoritatively committed. Offline-capable variants accept writes locally and reconcile them once connectivity returns [7]. Four engines use *decentralized* authority, where any replica can accept writes and replicas converge through CRDTs [8] or revision-tree replication [67]. Classic Jazz [75], Ditto [85], and Automerge [17] use CRDTs. CouchDB uses revision trees with master-master replication, where any node can accept writes and propagate changes to peers [67]. LiveStore uses a *hybrid* approach, with a centralized eventlog as the source of truth while clients materialize state locally from the event stream [131]. RxDB and TinyBase leave the authority model *configurable*. RxDB exposes `pushHandler` and `pullHandler` functions through which the engineer chooses the replication backend and conflict handling strategy [90]. TinyBase is a client-side toolkit that lets the engineer compose persisters and synchronizers, with source-of-truth placement left to the application [95].

At the time of the interviews, all 10 adopter and custom-builder participants (B1 through B7 and C1 through C3) ran server-authoritative architectures in production. For Zero users B1 through B5, server authority is not an explicit choice but an implicit consequence of choosing a Postgres-backed sync engine. B6 and B7 similarly operate under PowerSync's server-authoritative model, with Postgres as the source of truth and client SQLite databases as read replicas. C2 and C3 both use their existing backends as the source of truth. C1 is the closest to a genuinely decentralized model: CouchDB's architecture lets the application decide authority, with the server as source of truth, the client as source of truth with the server as backup, or a cluster sharing state. C1′s deployment at the time of the interview was server-authoritative, with a planned three-tier hub-and-spoke topology that exercises the master-master capability further. The engine choice was driven by that capability rather than by present need. C1 describes the flexibility: "Every client can be source of truth."

All three vendor interviewees converge on server authority, though from different starting positions. A1 was pragmatic from the start, viewing sync primarily as latency optimization with the cloud as the canonical source of truth. A2 started with a decentralized peer-to-peer architecture in Classic Jazz using edge relay servers. Jazz v2 moves to server-enforced permissions similar to Postgres row-level security [83]. A2 explains the shift: "The big realization is that we don't have to be so extreme about it. We can be more pragmatic but still keep all of the nice things that local-first gives us." A3 made the most dramatic change, pivoting ElectricSQL from active-active replication to server-authoritative writes through Postgres. A3 frames this in CAP terms: "We gave up on the availability side to go to the consistency side." Both A2 and A3 cite authorization as the primary reason for moving toward server authority. Enforcing permissions in a decentralized system where clients can write while offline creates edge cases that are hard to resolve without application-specific policy, examined in detail in Chapter 5.

Boodman and Chase [50] position Zero's server as the source of truth with the client as a local cache. Cowling [26] describes Convex's model as serializable transactions executed on the server. Kleppmann et al. [5] propose that users should own their data, but as the vendor convergence above shows, the practical requirement for authorization pushes even ideologically local-first engines toward server-authoritative designs.

4.2.4. Partial replication

Partial replication describes how a sync engine controls which subset of data reaches each client. Boodman [132] argues that partial sync is what separates general-purpose sync engines from collaborative editing tools: syncing an entire database is straightforward, but syncing a dynamic, user-specific subset is the hard problem. Despite this importance, partial replication has the lowest practitioner content coverage of any dimension (40 of 69 sources), a gap that may reflect avoidance of the problem rather than its resolution.

The architectural divide is between *dynamic* approaches where the client's query determines what syncs and *static* approaches where sync rules are declared separately. Zero uses *query-driven* partial replication through incremental view maintenance (IVM): the client writes a standard query and the server maintains a materialized view of the result, streaming changes as the underlying data updates [73]. ElectricSQL uses *shapes*, where the client declares interest in a table with optional `WHERE` filters, and the server streams matching rows [99]. Shapes support cross-table filtering through subqueries, allowing clients to request related data across joins. Ditto [85] uses *subscription-based* partial replication where the client subscribes to specific queries. PowerSync uses *sync rules*, server-side filters defined per user or JSON Web Token (JWT) that determine which data each client receives [133]. Sync streams are positioned as the recommended approach for both new and existing projects [134, 135], with sync rules retained as a legacy form. Both PowerSync interviewees (B6 and B7) used sync rules during fieldwork. B7 was already migrating to sync streams at the time of their interview, treating the migration as in progress alongside their existing sync-rule deployment. Sync streams are server-defined but client-selected: clients subscribe at runtime to named streams declared in server-side YAML, rather than driving sync through their own queries [134]. Both forms therefore sit on the static side of the dynamic-static divide.

CouchDB supports partial replication within a single database through document selectors and design-document filter functions [136]. C1 combines multiple partial sync strategies in production, and separately uses many small independent databases to minimize conflicts and contain access-control scope: "Split up the entities that are synced into a lot of small parts that don't depend on each other." Automerge syncs entire documents including their full change history [17], with partial replication

operating at the document level: the application chooses which documents to load through automerge-repo, but within a document, all operations sync. Classic Jazz loads CoValues on demand as the application resolves references [75]. Jazz v2 replaces this with query-driven sync over a relational data model [83, 137]. Turso Sync operates at the database page level, fetching pages lazily as the client accesses data [138]. A1 describes a philosophy of many small databases rather than partial sync within one large database. RxDB leaves the replication protocol to the engineer and does not prescribe a partial sync strategy [90]. TinyBase synchronizes the entire `MergeableStore` as a unit through its synchronizer modules, with no built-in mechanism for filtering rows or tables [95].

Interviews confirm the dynamic-static divide as practically significant. Zero users B1 through B5 reported that query-driven partial replication works smoothly for their use cases, with client queries naturally defining what syncs. B2 specifically chose Zero because of its partial sync support, noting that solving the partial sync problem independently was not worth the effort. Both PowerSync users reported limitations of the sync-rules approach: B6 around expressiveness limits in modelling data requests, B7 around operational deployment costs at scale. B7 scopes partial replication at the site level while migrating to PowerSync's sync streams for finer-grained scoping. The practical consequences of static versus dynamic partial replication, including expressiveness limits, operational deployment costs, and the difficulty of building custom partial sync, are examined in Chapter 5.

4.2.5. Conflict resolution strategy

Conflict resolution strategy describes how concurrent modifications to the same data are resolved [7, 8]. As established in the classification methodology, this dimension is constrained by authority model: server-authoritative systems can use server reconciliation, optimistic concurrency control, last-write-wins, or snapshot-DAG merging, while decentralized systems converge through CRDTs [8] or revision-tree replication [67]. Saito and Shapiro [7] note that conflict resolution is typically domain-specific, and no single default works across all applications [8].

The classified engines use nine distinct strategies. *Server reconciliation* has the client submit mutations and the server execute them authoritatively, avoiding concurrent writes entirely. Zero follows this model [73]. *Optimistic concurrency control* (OCC) executes serializable transactions on the server with automatic retry on conflict. Convex uses this approach [26]. *Last-write-wins* (LWW) resolves conflicts by selecting the write with the highest timestamp and discarding earlier writes. LWW discards all but the latest write by design. The timestamp source determines which write that is, and how predictably. Logical clocks paired with a tie-breaker [23] avoid wall-clock skew altogether by not consulting wall time, while hybrid logical clocks [22] combine wall-clock time with

logical tie-breaking to tolerate bounded skew. Naive wall-clock timestamps, by contrast, can reorder causally related writes whenever client clocks diverge. Section 2.1.1 gives the underlying happens-before and HLC discussion. PowerSync [139], InstantDB [140], and Turso Sync [141] use LWW. TinyBase implements LWW through a CRDT construction with hybrid logical clocks. This makes the clock semantics explicit where other implementations leave them implicit [95]. *CRDTs* achieve deterministic convergence without coordination. Classic Jazz uses LWW per field on CoMaps and position-based CRDTs on CoLists [75]. Ditto uses delta-state CRDTs [85]. Automerge records every change with actor and timestamp, achieving deterministic convergence through operation-based CRDTs [17]. *Snapshot DAG* stores immutable application states linked by parent references, allowing concurrent branches to reconcile through deterministic merge [83]. Jazz v2 uses this model in place of Classic Jazz's per-field CRDTs. *Server total order* ensures all clients converge on the same event history by sequencing events through a centralized sync backend rather than resolving conflicts per-row. LiveStore implements this through a Git-style rebase model where clients pull remote events and rebase local unpublished events on top before pushing [142]; custom merge resolution is planned but not yet realized [131]. *Custom hooks* let the application define resolution logic. RxDB was originally built on PouchDB and inherited its conflict-handling approach. RxDB exposes this as a `conflictHandler` callback [90]. *Revision trees* preserve all conflicting versions for application-level resolution. CouchDB originated this approach [67], with PouchDB providing a JavaScript implementation that syncs with CouchDB-compatible servers [97]. *Application-owned* resolution applies when the engine delegates the write path entirely. ElectricSQL provides no write path, so conflict resolution is the application's responsibility [99].

In the interviewed sample, conflicts were rare among server-authoritative deployments and concentrated in the ones supporting full offline writes. All five Zero users (B1 through B5) reported zero conflicts in production: server reconciliation combined with single-user-per-record data models avoids the problem entirely. The two interviewees reporting actual conflicts, B6 and B7, both use PowerSync with full offline capability, where multi-device or multi-user editing of the same records creates conflicts that online-only Zero users avoid by serializing writes through the server. Whether conflict resolution matters in practice therefore depends on two factors outside this dimension: whether the application supports offline writes, and whether multiple users or devices modify the same records. Chapter 5 revisits these factors as a cost-benefit trade-off between resolving conflicts and designing them out. Chapter 6 then examines whether practitioners can in practice design them out at all.

4.2.6. Read/write path architecture

Read/write path architecture describes whether a sync engine handles the read path that delivers data from server to clients, the write path that accepts mutations from clients, or both. The read/write split was a recurring topic in practitioner interviews and a consistent pattern across the classified engines, which is why this taxonomy keeps it as a first-class dimension. Classical replication taxonomies [7] do not classify this dimension, and community comparison resources such as localfirst.fm [57] do not treat read/write path ownership as the unified architectural axis used here.

The classified engines differ in who owns write-path semantics. This dimension does not ask where code physically runs. It asks whether the engine defines how writes enter synchronization. In an *engine-owned* write path, writes use the engine's API, data model, protocol, or transaction model. The engine defines synchronization semantics, such as mutation replay, rollback, CRDT convergence, revision trees, or event ordering. Application code may still enforce authorization, validation, and domain logic inside that model. Zero [50] follows this pattern: mutators run optimistically on the client, re-execute through the server mutate endpoint, and roll back on failure. Convex [26] runs serializable transactions on the server through its mutation API. Classic Jazz [75], Ditto [85], Automerge [17], and LiveStore [88] converge writes through their respective CRDT or event-based mechanisms. InstantDB [81], Turso Sync [141], CouchDB [67], and TinyBase [95] also provide engine-owned write models. In an *engine-transported* write path, the engine queues, retries, or delivers writes, but the application backend owns execution semantics. PowerSync [29] follows this model: writes apply immediately to the local SQLite database and enter an upload queue, but the engineer defines an `uploadData()` function that calls any backend API [30, 143]. RxDB [90] similarly transports mutations through a configurable replication plugin. *Read-only* engines provide no write mechanism. ElectricSQL is the only classified engine that follows this model, syncing data via Shapes while delegating writes entirely to existing endpoints [99].

This dimension is independent from offline capability and authority model. ElectricSQL has read-only sync and read-only offline capability. Zero has an engine-owned write path but only read-only offline capability. PowerSync has an engine-transported write path and full offline capability. The read/write path describes the scope of the sync protocol, while offline capability describes what functions remain available without the network.

A3 explains ElectricSQL's read-only approach as a response to practitioner demand: "Most people just want the read part, so the reactivity part." A3 describes the difficulty of generalizing the write path as the primary driver for ElectricSQL's pivot from active-active replication to read-only sync, noting that business logic, authorization, and conflict resolution on the write side vary too much across applications. Custom

builders who arrived at split-path architectures without using sync engines confirm this observation independently. C3 built the CI/CD security platform with reads flowing through Connect RPC streaming into TanStack DB for reactive client state, while writes go through standard Connect RPC mutations to the server. C2 reports a similar pattern, where most mobile data is read-only and offline sync covers only a small subset of the application's write surface. B6 preferred PowerSync's engine-transported model because it leaves the write path in the application's existing HTTP layer, with the backend handling business logic before and after the upload queue. B7 follows the same architecture at larger scale, with PowerSync's upload queue sending client transactions to a custom backend that dispatches business use cases. The resilience patterns that practitioners build around the write path, including B7′s transaction replay system for server-side failure recovery, are examined in Chapter 5.

Mathews et al. [88] identify the read/write split as a differentiator in the sync engine landscape, contrasting ElectricSQL's read-only approach with engines that manage or transport writes. The trade-offs of generalizing the write path versus leaving it to applications, including implications for authorization and incremental adoption, are examined in Chapter 5.

4.2.7. Sync unit

Sync unit describes the kind of state or change the engine tracks and transfers between replicas. It captures both synchronization granularity and the data model implied by that granularity: rows, documents, events, operations, CRDT deltas, or database pages. This differs from partial replication. Partial replication asks which subset of data reaches a client. Sync unit asks what kind of unit exists inside that subset. Saito and Shapiro [7] identify the related state transfer versus operation transfer distinction as a core dimension of optimistic replication. State transfer sends current data values, which is simpler to implement but can be bandwidth-heavy [7]. Operation transfer sends individual modifications that are replayed to reconstruct state, which is more efficient but requires causal ordering [7, 23].

State transfer dominates among the classified engines. Zero streams rows through IVM [73]. ElectricSQL sends rows matching Shape subscriptions [99]. PowerSync replicates rows to client SQLite replicas by reading the Postgres write-ahead log (WAL) on the server [144]. Convex sends query results through reactive subscriptions [26]. Ditto uses delta-state CRDTs, transferring state differences rather than full snapshots [85]. InstantDB [127], RxDB [90], and CouchDB [67] also use state transfer at various granularities, from triples to documents.

Three engines use operation transfer. Classic Jazz syncs per-session append-only log entries, with CRDT types derived as views over the operation history [75]. Automerge

records individual changes such as keystrokes and field updates and synchronizes these operations between replicas, reconstructing state through deterministic replay [17]. A2 described this model: "The core of a CoValue is basically just a per-session append-only log. The CRDT types, like a CoMap and a CoList, are then a view over that history." LiveStore syncs events from a centralized eventlog, with clients materializing state locally from the event stream [131]. Turso Sync uses a hybrid approach, fetching database pages lazily on the read path while replicating logical statements on the write path [138, 141].

Interviews confirm state transfer dominance. All participants using off-the-shelf engines use state-based sync. C1 uses CouchDB's document-level state replication. C3 uses Connect RPC streaming of state updates through TanStack DB. No participant discussed sync unit as an explicit decision point. Participants appear to treat it as an implementation detail rather than an architectural choice.

4.2.8. Network topology

Network topology describes how replicas communicate with each other [7]. *Client-server* topology dominates the classified engines. Zero [50], ElectricSQL [99], PowerSync [59], Convex [26], Jazz v2 [83], InstantDB [81], LiveStore [89], RxDB [90], and Turso Sync [141] all use it. Classic Jazz's CRDT data model is peer-to-peer capable, but its production infrastructure uses centralized sync-and-storage servers [75, 145]. Automerge uses peer-to-peer with optional relay servers [17]. Ditto uses *mesh* networking across Bluetooth LE, Wi-Fi Direct, LAN, and WebSocket transports [85], and CouchDB [67] and TinyBase [146] support flexible topologies.

At the time of the interviews, all 10 adopter and custom-builder participants ran client-server in production. Only C1 cited topology as a decision factor, choosing CouchDB for its master-master support and a planned three-tier hub-and-spoke deployment. The finding is limited by the interview sample of web application practitioners, since mesh and peer-to-peer topologies serve edge and IoT use cases [85] that were not represented here. Practitioner sources mention topology mainly when discussing local-first peer-to-peer goals [5], not production architecture choices.

4.3. Classifying sync engines

Table 5 and Table 6 present the full classification of 14 sync engine instances across seven taxonomy dimensions. Table 5 covers the four dimensions that most strongly differentiate engines: offline capability, integration depth, authority model, and partial replication strategy. Table 6 covers three implementation dimensions: conflict resolution strategy, read/write path architecture, and sync unit. Network topology is excluded

from the tables because most engines use client-server, with Classic Jazz and Automerge on peer-to-peer with relay, Ditto on mesh, and CouchDB and TinyBase supporting flexible topologies. Engines are ordered by architectural cluster, as discussed below.

Table 5: Architectural classification of sync engines. Row shading indicates cluster: database-pluggable, bundled platform, CRDT-decentralized, singular.

Engine	Offline	Integration	Authority	Partial replication
Zero	Read-only	Pluggable	Server	Query IVM
ElectricSQL	Read-only	Pluggable	Server	Shapes
PowerSync	Full	Pluggable	Server	Sync Streams / Sync Rules[4]
Convex	None	Bundled	Server	Subscriptions
InstantDB	Full	Bundled	Server	Subscriptions
Jazz v2	Full	Bundled	Server	Query-driven
Classic Jazz	Full	Bundled	Decentralized	On-demand
Ditto	Full	Bundled	Decentralized	Subscriptions
Automerge	Full	Toolkit	Decentralized	Document-level
LiveStore	Full	Bundled	Hybrid	Full
RxDB	Full	Toolkit	Configurable	Configurable
Turso Sync	Full	Bundled	Server	Page-level
CouchDB	Full	Bundled	Decentralized	Filtered
TinyBase	Full	Toolkit	Configurable	Whole store

[4]PowerSync positions Sync Streams as the recommended approach for both new and existing projects [134, 135]. Sync Rules remain supported as a legacy form. Both PowerSync interviewees used Sync Rules during fieldwork, while B7 was migrating to Sync Streams.

Table 6: Implementation classification of sync engines. Row shading matches Table 5.

Engine	**Conflict resolution**	**R/W path**	**Sync unit**[5]
Zero	Server reconciliation	Engine-owned[6]	State / Operations
ElectricSQL	Application-defined	Read-only	State
PowerSync	Last-write-wins	Engine-transported	State / Operations
Convex	Optimistic concurrency	Engine-owned	State
InstantDB	Last-write-wins	Engine-owned	State
Jazz v2	Snapshot DAG	Engine-owned	Snapshots
Classic Jazz	CRDTs	Engine-owned	Operations
Ditto	Delta-state CRDTs	Engine-owned	Delta state
Automerge	CRDTs	Engine-owned	Operations
LiveStore	Server total order	Engine-owned	Events
RxDB	Custom handler	Engine-transported	State
Turso Sync	Last-write-wins	Engine-owned	Pages / Statements
CouchDB	Revision trees	Engine-owned	State
TinyBase	CRDTs (LWW)	Engine-owned	State

4.4. Sync engine clusters

The classification shows that engines cluster into four architectural groups with shared dimensional choices rather than sitting on a single spectrum, as Table 7 summarizes. The "sync engine" label alone does not capture this choice space: Browne needed to compare one engine from each cluster, Zero, Classic Jazz, and TinyBase [13], before eventually adopting Convex [147].

[5]Where two values appear separated by a slash, they refer to the read-path unit and the write-path unit respectively. Most engines transfer the same kind of unit on both paths and use a single value.

[6]Zero mutators execute on the application's push endpoint, but the engine defines the mutation protocol, optimistic application, server reconciliation, and rollback semantics. The application still owns authorization and business validation inside mutators.

Table 7: The fourteen sync engines cluster into four architectural groups rather than sitting on a continuous spectrum. Cluster colors match the row shading in Table 5 and Table 6.

Database-pluggable	**Bundled platforms**
Zero, ElectricSQL, PowerSync	Convex, InstantDB, Jazz v2
Integrate with existing Postgres backends; server-authoritative.	Replace the backend; server-authoritative; greenfield-oriented.
CRDT-decentralized	**Singular**
Classic Jazz, Ditto, Automerge	LiveStore, RxDB, Turso Sync, CouchDB, TinyBase
Decentralized authority via CRDTs; full local-first operation.	Each occupies a unique architectural position.

The *Database-pluggable* cluster (Zero, ElectricSQL, PowerSync) is the most relevant for engineers with existing backends. All three share server authority, pluggable integration, and state-based sync, but diverge on three dimensions. Zero and ElectricSQL support only read-only offline capability, while PowerSync provides full offline writes. ElectricSQL is the only engine in the taxonomy that provides no write path, delegating all mutations to the application's existing APIs. ElectricSQL positions TanStack DB as the recommended client library for local optimistic writes layered on top of its read-path sync [63]; TanStack DB sits outside ElectricSQL and does not change the read-path-only classification. The three engines also differ in partial replication: Zero uses query-driven incremental view maintenance, ElectricSQL uses shape filters, and PowerSync uses server-declared sync rules and the newer sync streams.

The *bundled platform* cluster (Convex, InstantDB, Jazz v2) targets greenfield projects where initial simplicity outweighs backend constraints. All three share server authority and bundled integration but differ in offline capability and conflict resolution. Convex offers no offline support. InstantDB and Jazz v2 both provide full offline: InstantDB uses a Pending Queue with IndexedDB persistence [127] and last-write-wins resolution [140], while Jazz v2 uses snapshot-DAG-based merges that preserve history rather than collapsing to last-write-wins or optimistic concurrency. Eickhoff [83] groups Jazz v2 with Convex and InstantDB as its closest peers, sharing the same feature set: relational data, permissions, sync, and blobs. Eickhoff acknowledges that maturity is the main advantage alternatives currently hold over Jazz v2's public-alpha state.

The *CRDT-decentralized* cluster (Classic Jazz, Ditto, Automerge) commits most fully to local-first operation. All three combine decentralized authority, CRDT-based conflict resolution, and full offline capability, three properties central to the local-first

ideal. The cost is constraining the data model to CRDT-compatible structures. They differ in sync unit, where Classic Jazz and Automerge transfer operations while Ditto transfers delta-state, and in network topology, where Classic Jazz and Automerge use relay servers while Ditto uses mesh networking. Classic Jazz now sits in maintenance mode, with Eickhoff [84] describing Jazz v2 as the successor. The sync unit also encodes the underlying data model: Automerge operates on CRDT documents, distinguishing it from the relational or graph structures used by the other engines.

The fourth group, *singular* engines (LiveStore, RxDB, Turso Sync, CouchDB, TinyBase), covers engines that do not cluster with the others. LiveStore uses a centralized eventlog as the source of truth with event-based sync [131] and full offline capability [128]. RxDB provides configurable components that the engineer assembles into a complete sync system [90]. Turso Sync integrates synchronization at the database engine level, using page-level reads and logical writes with offline capability still in beta [94, 138]. CouchDB combines multi-master decentralized authority, revision-tree conflict resolution, and document-level filtered replication. PouchDB is the JavaScript client implementation that adds full offline support in browsers and mobile apps [67, 97]. TinyBase is a configurable client-side library that supports peer-to-peer synchronization, server-mediated sync, and third-party integrations [146], with a `MergeableStore` CRDT for native conflict resolution [95].

Several patterns emerge across the clusters. Offline capability discriminates more than any other dimension. It constrains both the viable engine set and the authority model. Server authority dominates regardless of integration model: 8 of 14 instances use server-authoritative or hybrid approaches, with Classic Jazz, Ditto, Automerge, and CouchDB providing decentralized alternatives and RxDB and TinyBase leaving the choice to the engineer. Read/write path and sync unit show dominant values, with 13 of 14 instances providing some form of write path (managed or transported) and 9 using state transfer.

4.5. Summary

This chapter presented a taxonomy of eight dimensions for classifying sync engine architectures, derived from distributed systems literature, practitioner content analysis of 69 sources, and semi-structured interviews with 13 practitioners. The dimensions are ordered by practitioner-perceived importance: offline capability, integration depth, authority model, partial replication, conflict resolution, read/write path architecture, sync unit, and network topology. The first six dimensions structure the architectural choices practitioners actively reason about, while sync unit and network topology show

dominant values that practitioners treat more as platform defaults than as active design choices.

The dimensions form a partial hierarchy of constraints: offline capability constrains the viable authority models, which in turn constrain the available conflict resolution strategies, while the remaining five dimensions vary independently. Read/write path architecture is treated as a first-class dimension here. Reviewed academic classifications do not treat it as a separate dimension. Classifying 14 sync engine instances across these dimensions shows that engines fall into four clusters with shared architectural choices, detailed in Table 7: database-pluggable, bundled platform, CRDT-decentralized, and singular.

The classification supports the definition proposed in Chapter 2: a sync engine keeps client-side application state synchronized with one or more authoritative replicas. Chapter 2 motivates the "authoritative replicas" component from the replication literature's single-master versus multi-master distinction. The taxonomy supports this empirically: 8 of 14 instances are server-authoritative or hybrid, and Classic Jazz, one of the four decentralized engines, has moved to server-enforced permissions in Jazz v2 [83], which now sits in the bundled platform cluster.

The taxonomy is descriptive: it classifies what architectural patterns exist across the synchronization spectrum. How these patterns differ in trade-offs and limits is examined in Chapter 5. What determines sync engine fitness for different applications is examined in Chapter 6.

5. Trade-offs and Limits

This chapter addresses ARQ2: What trade-offs and limits do sync engine architectural patterns introduce? The taxonomy in Chapter 4 classifies sync engines along eight dimensions. The trade-offs below cover the engines classified there. Out-of-scope systems appear as references where relevant, without dimensional analysis. The chapter first identifies the trade-offs that follow from taxonomy dimensions and from cross-cutting adoption concerns. It then isolates the trade-offs that remain regardless of engine configuration, treating those as structural limits derived from the same trade-offs rather than as independent challenges. Of the seven trade-offs identified below, five map directly to taxonomy dimensions, while authorization and hidden integration costs cut across multiple dimensions.

5.1. Technical trade-offs

Trade-offs from the taxonomy dimensions become visible during implementation and operation. Seven trade-off areas emerged consistently across interviews and practitioner content. Table 8 lists them with engine positions and participant evidence.

Table 8: Seven technical trade-offs that emerged across interviews and practitioner content.

Trade-off	**Tension**	**Example evidence**
Consistency vs responsiveness	Local writes feel immediate but weaken consistency; server writes preserve consistency but add latency.	Zero optimistic mutator execution vs Convex server-only mutation execution; B1–B5; B2 selective fallback
Read/write path generalizability	Read path commoditizes across applications; write path resists generalization.	ElectricSQL pivot to read-only; A3, B6, B7, C2, C3
Integration depth	Pluggable engines enable incremental adoption; bundled engines require greenfield projects.	Pluggable vs bundled clusters; B5, B6, B7, C2
Authorization	Permission enforcement adds state-space complexity that all three vendors found to be structural.	Vendor convergence on server-enforced; A1, A2, A3
Partial replication	Static rules are rigid and operationally costly; dynamic queries hit scale limits.	PowerSync sync rules vs Zero query-driven; B5, B6, B7
Conflict resolution	Conflicts are rare in production; LWW dominates; richer resolution requires application-specific logic.	Zero, PowerSync (LWW); Automerge, Ditto (CRDTs); B1–B5 report none
Hidden integration costs	Schema evolution across client versions, client-generated IDs, browser maturity gaps, UI renderer mismatch, deployment requirements.	All engines; A2, B6, B7

5.1.1. Consistency versus responsiveness

The PACELC framework from Chapter 2 formalizes the trade-off between consistency and latency during normal operation [19]. Sync engines make this trade-off concrete. Engines that write locally first provide instant responsiveness but weaken consistency guarantees. Engines that serialize through the server guarantee consistency but add latency to every write.

Cowling [26] argues that centralized consistency is easier for engineers to reason about than eventual consistency. Convex executes all mutations as serializable ACID transactions on the server, providing a single consistent timeline that eliminates the ambiguity of concurrent writes [26]. This design trades responsiveness for correctness: every write requires a server roundtrip, but engineers reason about a single ordering rather than concurrent versions.

Zero takes the opposite position, applying mutations to a local cache before the server confirms them [21]. Interview participants B1 through B5, all Zero users, reported that this local-first write model delivers perceived instant responsiveness. B5 described the difference: with Convex, latency is "still fast, but in milliseconds," while with Zero, "you're trimming frames." B4 similarly chose Zero because the local write model eliminated loading states entirely from user-facing interactions.

Local writes have a cost. Kleppmann [148] identifies global invariants as a mathematical hard limit for systems that allow local writes without coordination. A uniqueness constraint across all users requires consensus, which conflicts with the availability that offline and local-first writes provide [148]. Köhler et al. [149] propose ConLoc, a system to automatically enforce invariants in local-first applications, but such mechanisms remain research prototypes not yet integrated into production sync engines. Practitioners navigate this trade-off by matching the consistency requirement to the domain. B2 treats consistency as a per-workflow choice: server-authoritative tRPC for accounting aggregations where consistency matters, and Zero for user-facing navigation where responsiveness matters. Abadi [19] frames this latency-consistency trade-off as a spectrum rather than binary.

5.1.2. Read-path and write-path generalizability

Table 9 sharpens the read/write asymmetry from Chapter 4 by crossing it against offline capability: read paths generalize regardless of network state, while write paths generalize online but resist offline. This asymmetry shapes both engine design and adoption patterns.

Table 9: Read and write paths crossed against network state. Three of the four cells generalize across applications; the offline write path resists because authorization, conflict resolution, and business logic vary by domain.

	Always-online	**Offline-capable**
Read path	**Generalizes** Server pushes deltas. Subscriptions, incremental view maintenance, and shape filters reuse across applications.	**Generalizes** Local replica serves reads from disk. Prefetch and sync reuse across applications.
Write path	**Generalizes** Engine owns mutation replay and reconciliation when given a canonical schema.	**Resists** Authorization, conflict resolution, and business logic vary across applications and resist standardization.

Arthur [99] articulates this asymmetry most directly. ElectricSQL's architectural pivot from full active-active replication to read-only sync reflects a deliberate conclusion that the write path varies too much across applications to commoditize [99]. A3 explained the reasoning: "Should you really do all the complexity and all the technical difficulties, and all the weird cases that come with it, if it only happens 10% of the time or 5% of the time? Probably no." A3 reported that most users only want the read-path reactivity rather than full bidirectional sync.

Interview evidence independently confirms this asymmetry. As detailed in Chapter 4, PowerSync transports writes but does not execute them, leaving the write logic to the application's backend. B6 valued this separation: "I like our approach the most so far, because it's just a normal HTTP layer and I can do everything before and after ingestion." B7 confirmed the same model at larger scale: PowerSync's upload queue feeds a custom backend that dispatches business logic.

C3 arrived at the same read/write split independently, without using a sync engine. The TanStack DB plus Connect RPC streaming architecture, detailed in Chapter 4, separates reactive read state from server-authoritative writes. C3 described this as "Sync Light": real-time read reactivity without the write-side complexity of full sync engines. C2 reported the same split: most mobile data is read-only, with offline sync limited to a small write surface.

C3 identified the main cost of the component-assembly approach: every CRUD route must manually emit an invalidation event to keep client subscriptions consistent. "When writing those CRUD routes, you have to always remember to send the invalidation

event." WAL-based engines such as Zero [73], ElectricSQL [150], and PowerSync [144] remove this overhead by replicating database changes directly from the WAL.

Vendor A3, adopters B6 and B7, and custom builders C2 and C3 all converge on the same split. The read path delivers data from a centralized source to clients, a pattern that maps well to HTTP caching and subscription-based streaming [99]. The write path involves business logic, authorization, and conflict resolution, which vary across applications and resist standardization. This asymmetry is a structural property of web application synchronization, not a limitation of particular engines.

5.1.3. Integration depth and incremental adoption

The integration depth dimension in Chapter 4 maps to the classical buy-versus-build trade-off in software engineering [101]. Bundled engines buy convenience but constrain future flexibility. Pluggable engines preserve flexibility but require more integration work [151]. For existing applications, this trade-off is asymmetric: pluggable engines enable incremental adoption, while bundled engines typically require greenfield projects or substantial rewrites.

Every interview group emphasized incremental adoption. B5, who migrated approximately 40% of a freight brokerage CRM to Zero over one year, described the engine's appeal as a "very fancy cache server that I can slowly start to incrementally embed into my app." C2 echoed this priority: only pluggable or toolkit engines support adopting sync one entity type at a time.

B6′s experience illustrates how integration depth constrains complex applications. B6′s application requires image resizing, PDF generation, computer vision processing, and background jobs, operations that cannot be expressed as sync operations. B6 concluded that backend replacement is "too optimistic. Maybe for simple consumer apps that's nice, but for our use case, it just gets too complex too quickly." B6 described a preferred architecture where sync is just another endpoint integrated into existing backend infrastructure rather than a separate service that duplicates authorization logic.

B7 confirmed this assessment from a larger enterprise context. B7′s deployment spans hundreds of tables, with PowerSync handling only the read sync; transaction handling, failure recovery, and schema management run on custom systems outside the engine. The sync engine is a component within a larger custom architecture, not a replacement for one.

5.1.4. Authorization converges on server enforcement

Authorization does not differentiate engines in Chapter 4 but affects all approaches. Three sync engine vendors independently converged on server-enforced authorization from different starting positions.

A3 described the problem that ended ElectricSQL's original decentralized approach: "Imagine someone does a mutation while being offline and they think they are still authorized, but concurrently, you revoked their rights." This question has no clean answer in a decentralized system where clients can write locally without server coordination. ElectricSQL's pivot to read-only server-authoritative sync removed the problem at the architectural level [99].

A2 reported that permissions are "where some of our bigger adopters are struggling the most," which drove Jazz's move toward server-enforced authorization. Eickhoff [84] explains the operational rationale: adopters preferred the experience of local-first data over the cryptography itself. Public-key complexity and unusual permission APIs proved too burdensome to scale. A2 described Jazz v2 as introducing a first-class "pending state" for local writes that exist before global permission acceptance. This design preserves the local-first writing experience while giving the trusted server final authority over what is accepted [83, 137]. A2 framed this as a pragmatic compromise that retains local-first ergonomics rather than a retreat from local-first principles.

Interview participants report the same difficulty, and authorization appears as a barrier to sync engine adoption in 26 of 69 practitioner content sources. B6 identified that authorization logic becomes split between the API layer and the sync rule layer, making some access patterns "a bit harder" to express. B7 implements dual authorization: sync rules for read-path access control and backend validation for write-path permissions. Custom builders C2 and C3 sidestep the problem entirely by using standard application-level authorization patterns outside the sync layer. Section 5.2.1 examines whether the vendor convergence reflects a navigable trade-off or a structural limit.

5.1.5. Partial replication: flexibility and operational cost

Partial replication, determining which subset of data reaches each client, is the least discussed dimension in practitioner content but one of the most operationally painful in interview evidence. Boodman [132] frames partial sync as a problem that current engines have not solved: syncing a whole database is straightforward, but syncing a dynamic, user-specific subset efficiently remains difficult. Terry [152] identifies five open questions for partial replication, including how devices specify items of interest, what happens when interests change, and what constraints apply to synchronization topologies with partial replicas. These questions remain unresolved in current sync engines.

The practitioner content analysis confirms this gap: only 40 of 69 sources discuss partial replication, the lowest coverage of any taxonomy dimension.

The taxonomy in Chapter 4 identifies an architectural divide between dynamic approaches, where the client's query determines what syncs, and static approaches, where sync rules are declared separately. Interview evidence reveals distinct failure modes for each approach.

Static sync rules create two independent pain points. B6 identified sync rules as the single biggest frustration with PowerSync: "Definitely the sync rules. Not only understanding them, but in some way it was hard to model actually how we wanted to request data." The rules are rigid, making it difficult to express granular access patterns. B6′s application scopes sync to the active project but wanted finer partitioning by entity type or time range that the sync rules could not express. B6 explicitly preferred ElectricSQL's dynamic approach: "I always liked the way Electric does that way more… more flexible, you just request shape data and you work with it." B7 reported a different pain: operational cost at enterprise scale. Redeploying sync rules triggers full database reprocessing, which takes hours for B7′s hundreds-of-gigabyte read replica and forces night-time deployment windows. B7 has worked around join limitations through denormalization maintained by database triggers. B6 cites flexibility, B7 cites operational cost; these are two independent indictments of the static sync rule model.

Dynamic approaches have their own limits. B5 reported that complex queries with many joins and large initial datasets overwhelm Zero's view syncer: "If I did a lot of joins, and the amount of data that needed to get replicated initially to the client was a lot, then basically no part of the view syncer would work." A2 reported that Classic Jazz's on-demand loading creates cascading latency because encrypted references must be resolved one at a time, producing N+1-style request waterfalls. Jazz v2 addresses this by moving to a relational data model with query-driven sync, where the trusted server can see row contents and fulfill bulk queries [83, 137].

Partial replication is where the gap between sync engine abstractions and production requirements is the widest. Neither static nor dynamic approaches have converged on a solution that scales without substantial engineering workarounds.

5.1.6. Conflict resolution in practice

Conflict resolution receives academic attention [7, 8] but has limited practical relevance for most adopters. Across all 13 interviews, conflicts are rare in production.

All five Zero users B1 through B5 report zero conflicts in production. Server reconciliation, combined with data models where records have clear single-user ownership, avoids the problem entirely. B4 confirmed: "It's pretty rare we have so many people working on the same thing that it gets a problem." C2 avoids conflicts by design through

an append-only data model where field measurements are added to the database and users cannot edit each other's measurements. A1 relayed a former Realm engineer's observation that practitioners care about knowing the conflict model rather than having sophisticated resolution: "Many developers don't seem to really want conflict resolution. They just want to know what the model is explicitly so they can build around it." A2 reached the same conclusion from the vendor side: "Our surprising learning with Jazz and our adopters has been that, 90% of the time, you actually don't need [ACID transactionality]. Eventual consistency is completely fine." Jazz v2 acts on this: it replaces CRDTs with a snapshot DAG that preserves history without per-client CRDT replay, while supporting globally consistent ACID transactions for the 10% case where stronger guarantees matter [84, 137]. A3 took this avoidance to the architectural extreme: ElectricSQL removed the write path entirely, leaving any conflict handling to the application's existing backend rather than the sync engine [99].

Real-time conflicts can occur in collaborative editing applications, but presence signals such as cursors and selections help users coordinate around simultaneous edits on the same data [153]. In this study, the interview cases that required manual or domain-specific conflict handling involved offline multi-user or multi-device modification of the same records. B6 reports the same user editing rich text notes on both desktop and mobile, overwriting changes through last-write-wins, and is considering Yjs [15] for collaborative rich text editing. B7 reports a different conflict pattern: a field worker making updates offline while a colleague deletes the same entity via the web platform. B7 mitigates this through soft deletes that enable entity recovery after conflict.

C1 articulated a structural distinction between two incompatible approaches to conflict handling. "CRDT-core" engines handle conflicts automatically through mathematical convergence guarantees, while "conflict-API-core" engines like CouchDB [67] expose conflicts to the application for domain-specific resolution. C1 argued that for applications with serious data integrity requirements, automatic conflict resolution is insufficient: "For stuff that has serious data and where it has serious consequences if you don't handle conflicts right, all of these sync engines are not usable." A3 pushed back on the assumption that conflicts never happen: "I wouldn't say that conflicts almost never happen, because they do happen." A3 argued that the question is whether the cost of handling them justifies the engineering complexity.

A3 also observed that only two CRDT designs are widely applicable in practice without per-application modification: last-write-wins registers and multi-value registers. A3 added that all other CRDT primitives require application-specific extension to be usable. Last-write-wins dominates because it requires no application changes, but it silently discards concurrent edits [8]. Engines that promise automatic conflict resolution through CRDTs provide a narrower guarantee than the marketing suggests.

Linear's architecture follows this use-case split in production. Artman [154] describes Linear's sync as fully custom with last-write-wins for atomic state changes such as priority or status, on the reasoning that "there's no merging you can do" between two discrete values. Linear only introduced CRDTs through Yjs years later when the team began building rich-text and collaborative document features where concurrent textual editing is the dominant case [154].

Most applications can design around conflicts through single-user record ownership, append-only models, or accepting last-write-wins. Applications that require richer conflict handling, such as collaborative text editing or concurrent field data collection, need domain-specific CRDT extensions or application-level resolution logic. No general-purpose sync engine provides this out of the box.

5.1.7. Hidden integration costs

Beyond the trade-offs discussed above, interviews identified five adoption costs: schema evolution across client versions, client-generated identifiers, platform maturity gaps, architectural mismatches between sync engines and UI frameworks, and deployment requirements that constrain runtime models.

Schema evolution becomes architecturally hard because many client versions in the field must continue to interoperate with a server schema that has moved forward. Mobile app store release cycles create version fragmentation that backend changes cannot eliminate, and B7 reported that schema validation changes on the backend can break older clients still holding old-version transactions in their upload queue. A2 described schema evolution as "maybe even harder than permissions." Adopter and vendor responses fall into two patterns. Both PowerSync users independently adopted an additive-only rule: only add columns and tables, never remove or rename, deprecating old paths only after all clients have updated. Jazz v2 takes a different path with Fluid Migrations, a bidirectional functional mapping between schema versions that lets newer and older schemas interoperate without locking the data model into additive change [137, 155].

Synchronizing schema definitions across the client SQLite store, the sync engine, and the server backend compounds the version-drift problem. B7 built custom tooling to generate client schemas for PowerSync [59] and Drizzle [156] from a Prisma [157] source of truth on Postgres. The tooling handles type mismatches such as JSON-to-text conversion and custom-to-standard index translation across hundreds of tables. This tooling was not planned from the start but became necessary when schema drift across multiple feature branches caused issues. Edwards et al. [158] systematize schema evolution as a general barrier to feedback loops and collaboration across interactive

programming systems, extending the problem beyond traditional database migration to contexts including live front-end programming and collaborative documents.

Client-generated identifiers are a concrete cost that sync engine documentation rarely highlights. B7 reported that moving from server-generated to client-generated surrogate keys was a notable team adjustment: "The WTF moment is usually we don't generate client-side IDs." B7 explained that client-generated IDs are necessary for idempotent writes in offline-capable sync: the backend uses the client-provided ID as an idempotency key for duplicate transaction handling from network retries.[7]

Platform maturity varies sharply between native mobile and browser environments. B7 runs PowerSync on both mobile with native SQLite and web with WebAssembly (WASM) SQLite over OPFS, and reported that browser offline introduces platform-level risks that native mobile does not. B6, who uses PowerSync only on mobile, never reported storage issues. The browser platform gap is examined as a fundamental limit in Section 5.2.

UI renderer mismatch creates a performance bottleneck between sync engine granularity and standard UI frameworks. Agrawal [159] identifies that sync engines deliver granular data deltas, but standard UI renderers such as React inefficiently re-render the entire component tree in response. Adopting a sync engine may therefore force changes in UI architecture [159].

Deployment requirements surface as a friction that pluggable sync engines add over component-assembly approaches. C3 observed that running Zero requires a long-running stateful service alongside the application backend, raising operational concerns about server-side memory and runtime separation. ElectricSQL and PowerSync share this constraint: each requires a sync service with container and file-system requirements that do not fit serverless or stateless deployment models, while component-assembly approaches reuse the existing backend's runtime.

The Teamspective case in Chapter 7 shows the constraint in practice: the zero-cache service initially fit the existing container-based deployment as a single disk-backed service. Production load and full-resync events later drove a migration to Zero's multi-node deployment, which separates upstream ingestion from client serving so clients remain reachable during upstream recovery. The split required additional persistent storage and ran into a host-platform constraint that pins disk-backed services to a single instance.

[7]Client-generated IDs typically use random UUIDs, whose 128-bit address space, illustrated by everyuuid.com, makes collisions statistically negligible at application scale.

5.2. Limits of generalization

The trade-offs above are challenges that engineers can navigate through design choices. Some of these challenges, however, remain constraints regardless of engine configuration. This section examines five such limits. Table 10 outlines them with structural causes and example evidence.

Table 10: Five limits that remain regardless of engine configuration.

Limit	**Why structural**	**Example evidence**
Authorization	The engine must track changing permissions across user replicas; offline writes made under stale permissions require reconciliation on reconnect.	A1, A2, A3 (vendor convergence); ElectricSQL pivot; Jazz v2 server-enforced
Global invariants and data volumes	Uniqueness across users requires consensus, which conflicts with local availability; sync replicates rows, not computed results.	B2 (tRPC fallback for aggregations); B5 (51,000-record limit)
Write-path resistance	Business logic, authorization, and conflict resolution vary across applications and resist standardization.	A3 (ElectricSQL pivot); B6, B7 (custom backends)
Browser platform immaturity	WASM SQLite, OPFS, and IndexedDB introduce platform risks absent on native mobile.	B7 (web vs mobile); browser quota limits
Partial sync and networking	Neither static rules nor dynamic queries have converged on a scalable solution.	Replicache partial-sync gap [132]; B5, B6, B7

5.2.1. Authorization as a hard limit

O'Brien [160] captures the structural mechanism: adding authorization to a working public-sync layer triggers a sharp complexity jump. The state the engine must maintain grows with the number of users and their permission sets, and both data and permissions change independently over time. O'Brien frames this as the harder production problem than data merging itself [160]. A permission change must propagate through the sync layer and remove data from all active client replicas, and the offline-mutation scenario A3 described compounds the problem with state created against permissions that no longer hold.

ElectricSQL responded to this pressure by removing offline writes entirely [99]. Jazz responded by moving to server-enforced permissions in v2 because the public-key model and permission APIs did not scale for adopters [83, 84]. Custom builders C2 and C3 took a different route, placing authorization outside the sync layer rather than inside it. Academic work has proposed CRDT-based access control mechanisms [161, 162], but these remain research prototypes not yet adopted by production sync engines. Separating authorization from the sync mechanism is the consistent vendor and builder response.

5.2.2. Global invariants and data volume boundaries

Kleppmann [148] identifies a theoretical boundary for local-first systems: global constraints such as unique values across all users require consensus, which conflicts with the availability that offline capability provides. This limit is mathematical, set by consensus requirements. No sync engine can guarantee both offline writes and unique constraints without the possibility of violations that must be resolved after reconnection.

Data volume creates a practical boundary that extends beyond this theoretical limit. B5 reported that a 51,000-record shipper database was too large to sync to clients, requiring that data to remain on traditional server queries. Artman [163] documents that Linear's architecture degraded at approximately 50,000 objects, requiring a custom "lazy" sync architecture that asynchronously fetches data only when accessed. B2 falls back to server-authoritative endpoints for financial aggregations because sync engines replicate rows, not computed results. B4 identified LLM token streaming as a concrete boundary: token-by-token responses generate database writes at a frequency that overwhelms the sync-through-database model, forcing B4 to route these writes through async mutators outside of Zero.

Sync engines are optimized for moderate-volume, user-generated content with interactive access patterns. Reference datasets, aggregation-heavy analytics, and high-frequency write streams fall outside this domain.

5.2.3. Write-path resistance to generalization

ElectricSQL's architectural pivot from full active-active replication to read-only sync is the strongest evidence that the write path resists generalization. A3 chose to serve "the use case that 80 or 90 percent of people want" rather than solving the full bidirectional sync problem [99]. A3 reported that the engineering cost of full bidirectional sync did not justify the share of users who needed it.

Domain-specific applications confirm this limit. Nehmzow [164] documents how Notion could not use standard CRDTs because their block-based data model requires operations like splitting and merging blocks, demanding custom implementations

beyond what generic engines support. Bandi [33] describes Figma maintaining two separate sync engines because canvas data and product metadata require fundamentally different sync strategies.

B7 revealed a gap between the sync protocol and production requirements: server-side failure recovery. B7 built a transaction replay system that persists every client transaction regardless of success or failure, allowing replay after fixing backend bugs or validation issues. B7 described the commitment: "We are basically promising zero lost data to our field workers." Sync engines handle client-to-server transport but not what happens when the backend fails to process a transaction. B7′s experience indicates that enterprise applications may require resilience infrastructure around the write path that sync engines do not provide.

5.2.4. Browser platform immaturity

B7′s experience with WASM SQLite and OPFS on the web shows that browser-based offline capability hits platform-level limits that native mobile does not. Kistner [165] documents that Chrome's incognito mode imposes a roughly 100-megabyte database size limit with unexpected errors when reached, and that Safari's incognito mode does not support OPFS at all. B7 reported encountering errors that "even the PowerSync team" cannot resolve, describing the experience as "somewhat building on bleeding edge."

This limit constrains the practical viability of browser-based local-first architectures independently of the sync engine. B6, who uses PowerSync only on mobile with native SQLite, never reported storage reliability issues. Mobile offline via native SQLite is mature and production-ready. Browser offline via WASM SQLite and OPFS still hits unresolved platform-level challenges. No sync engine can abstract away bugs in the browser's storage implementation.

5.2.5. Partial sync and networking as unsolved layers

Fish [85] argues that CRDTs solve data conflict resolution but production sync requires solving additional infrastructure. This includes mesh networking, bandwidth efficiency through property-level deltas, and partial replication at the network layer. Boodman [132] identifies partial sync as the central unsolved problem that forces teams to build custom backends, despite sync engines being marketed as solutions to the synchronization problem.

The interview evidence confirms that no partial replication approach has converged on a solution that scales without substantial engineering workarounds. Static sync rules are rigid and operationally expensive. Dynamic query-driven approaches break down under complex joins and large datasets. On-demand loading creates cascading latency.

Partial replication is harder than full-database replication [132] and may resist the generalization that sync engines attempt.

5.3. Summary

This chapter examined the trade-offs and limits that sync engine architectural patterns introduce, addressing ARQ2.

Seven technical trade-offs emerged from practitioner evidence. Engineers mix consistency levels within a single application rather than committing to one model uniformly. Vendors, adopters, and custom builders independently confirm that reads generalize well while writes resist standardization. Incremental adoption emerged as a universal requirement, favoring pluggable engines over bundled platforms. Authorization drove three vendors independently toward server authority. Partial replication has the lowest practitioner content coverage of any taxonomy dimension and is the most operationally painful in interview evidence. Conflict resolution receives extensive academic attention but rarely surfaces in practice. Hidden costs including schema evolution, client-generated IDs, and platform maturity gaps emerge only after adoption.

Five limits go beyond navigable trade-offs and act as constraints. Authorization combined with sync forces the engine to propagate permission changes across replicas and to reconcile offline-mutated state against revoked permissions on reconnect. Global invariants create a mathematical boundary, while data volume and high-frequency writes create practical boundaries. The write path resists generalization because business logic and conflict resolution are application-specific. Browser platform immaturity constrains web-based local-first architectures. Partial sync remains an open problem across current engines [132].

Chapter 6 examines which application types fit within these boundaries, and what factors drive the adoption decision in practice.

6. Application Fitness and Decision Factors

This chapter addresses ARQ3: How do application requirements determine sync engine fitness? Chapter 5 examined what sync engine architectures cost and where they break down. This chapter examines what they are good for: which application types fit, at what abstraction level, and what drives the adoption decision in practice. The fitness assessment covers the same engines classified in Chapter 4.

6.1. Application type fitness

Not all sync engines serve all application types equally well. Use case fit appears in 65 of the 69 practitioner sources. Interview evidence and practitioner content point to five application categories with distinct sync engine fitness. Table 11 compares them across offline requirement, key constraint, and example evidence.

Table 11: Sync engine fitness by application type, based on interview evidence and practitioner content analysis.

Application type	**Fitness**	**Offline**	**Key constraint**	**Example evidence**
Offline field apps	Strong	Required	Few viable engines (PowerSync, CouchDB)	B6, B7; C2 (custom)
B2B productivity tools	Strong	Not needed	Moderate data volumes	B1–B5 (Zero)
Collaborative editing	Partial	Varies	Requires CRDT or custom conflict resolution	Notion [164], Figma [33]; C1
Transactional	Poor	N/A	Needs global invariants incompatible with local writes	–
Data-intensive / analytics	Poor	N/A	Sync replicates rows, not computed results	–
"Sync Light" (read-only reactivity)	Alternative	Not needed	Component assembly, not a sync engine	C3 (TanStack DB)

6.1.1. Offline field applications

Offline field applications are the strongest fit for full-capability sync engines. B6 (field documentation), B7 (field-operations audits), and C2 (field engineering planning) all build applications where users operate in environments with unreliable or no connectivity. For these applications, offline capability is non-negotiable, and sync engines enable offline data collection with eventual server reconciliation. The viable engine set is severely constrained: both PowerSync users independently confirm that PowerSync is the only engine meeting their combined requirements of full offline capability and Postgres compatibility. Outside Postgres-tied workflows, CouchDB combined with PouchDB has been deployed in offline field-app contexts including humanitarian and infrastructure work [125]; no interview participant uses this stack. B6 summarized: "People won't use it without offline functionality." These applications accept higher architectural complexity, including the schema-evolution costs, sync rules, and client-generated IDs documented in Chapter 5, because the alternative is no product at all.

6.1.2. Real-time B2B productivity tools

Real-time B2B productivity tools are the broadest sweet spot for sync engines. B1 (brand protection), B2 (accounting), B3 (nonprofit bookkeeping), B4 (marketing tools), and B5 (freight CRM) all build applications where perceived responsiveness and developer velocity are the primary adoption drivers rather than offline capability. These applications use sync engines to eliminate loading states, simplify state management, and provide instant navigation. Artman [166] reports that Linear's sync architecture renders nearly every page locally, allowing roughly 1,000 concurrent users to run on minimal infrastructure. The fitness depends on moderate data volumes: B5′s migration is deliberately partial because a 51,000-record reference database exceeds what is practical to sync.

6.1.3. Collaborative editing applications

Collaborative editing applications such as documents, whiteboards, and design tools require conflict resolution beyond what last-write-wins sync engines offer. Within the taxonomy, CRDT-based engines are the closest fit but often require data-model-specific extensions. Outside the taxonomy, domain-specific implementations and specialized real-time collaboration libraries, such as Liveblocks [120] and PartyKit [65], address collaborative editing through different infrastructure. Notion's block-based data model required custom conflict-resolution logic for splitting and merging blocks, beyond what standard CRDTs provide [164]. For example, Figma maintains two separate sync engines because different data types require different sync strategies [33]. C1 observes that collaborative applications where multiple users edit the same data simultaneously

can generate "hundreds of conflicts per session." These applications typically require CRDT-decentralized engines such as Automerge and Ditto, or domain-specific custom implementations, and even then the conflict resolution must be tailored to the data model.

6.1.4. Transactional applications

Transactional applications requiring strict consistency, such as e-commerce checkout, financial transactions, or booking systems, are poorly served by sync engines that prioritize local responsiveness. These applications need global invariants, which Chapter 5 identifies as a mathematical hard limit for local-first writes [148]. Two customers cannot book the same hotel room, and eventual consistency does not resolve this race condition. Cowling [26] acknowledges this, positioning Convex's server-authoritative transactions as the appropriate model for applications requiring ACID guarantees at the cost of local responsiveness. Schickling [167] frames this as domain-driven: banking applications need strict consistency, while note-taking applications need immediate responsiveness.

6.1.5. Data-intensive and analytics applications

Data-intensive and analytics applications requiring aggregations, complex joins, or large-dataset queries hit the data volume boundaries documented in Chapter 5. The classified sync engines replicate individual rows or documents as established in Chapter 4, not computed results. B2 falls back to tRPC for accounting dashboard aggregations because computing them client-side from synced rows would be inefficient and potentially incorrect. B5 reported that complex queries with many joins overwhelmed Zero's view syncer. Linear hit a comparable scaling boundary at around 50,000 objects, leading to a custom "lazy" sync architecture [163]. These applications can use sync engines for the interactive user-generated content while keeping aggregations and reference data on traditional server-authoritative query endpoints.

6.1.6. Real-time read reactivity without offline

C3′s "Sync Light" approach combines TanStack DB with Connect RPC streaming for read-path reactivity without offline or conflict resolution. This sits between traditional REST and full sync engines. C3 described this as a deliberate choice: the application is explicitly online-only and avoids the hard problems that offline sync introduces. For many B2B web applications, the full sync engine abstraction provides more capability than the application requires.

6.2. Decision factors from practitioner evidence

Use case fit was examined in Section 6.1. There is no universally correct choice. Beyond use case fit, six further decision factors emerged from interviews and practitioner content. Table 12 enumerates them with example evidence.

Table 12: Six additional decision factors that emerged alongside use case fit.

Decision factor	**Description**	**Example evidence**
Speed and developer experience	Engineers adopt sync engines as state-management replacements, not for synchronization.	B1–B5 (Zero); Linear's local rendering [166]
Existing backend compatibility	Postgres compatibility acts as a binary filter; bundled platforms are only viable for greenfield projects.	B1–B5 require Postgres; A3 (ElectricSQL positioning); B5 (40% migration)
Offline requirement	Acts as a binary filter; few engines satisfy when offline writes are required.	B6, B7 (PowerSync as only fit); B1–B5 (not needed); C3 (online-only)
Ecosystem maturity	No widely accepted production standard yet; team size mediates impact through custom workarounds.	B3, B7 (pioneering cost); A2, A3 (vendor view)
Abstraction level	Higher tiers speed prototyping but constrain data model and deployment; lower tiers add control at the cost of maintenance.	Convex, Jazz v2 (high); Zero, ElectricSQL, PowerSync (medium); Yjs, custom (low); [13]
Build versus buy	Sync engines' primary competitor is engineers building custom rather than another engine.	C1, C2, C3 (would rebuild); A2 (Jazz's main competitor); B1 (cautious)

6.2.1. Speed and developer experience as adoption drivers

Interview participants frame sync engines as state management replacements and API simplifiers rather than as synchronization tools. Speed and developer experience drive adoption in this category, not synchronization itself.

B4 reported: "I was willing to try anything to get the speed down to instant." B4′s application served a 3-to-5-second initial load via Next.js and Supabase; after migrating

to Zero, the experience became "instant" after initial hydration. B5 valued Zero primarily for the local data access pattern rather than the synchronization capability, treating it as a cache layer over the existing application stack. B3 was "not even necessarily specifically drawn to sync" but chose Zero because "so much goes into building that API, keeping it online, and then all the wiring… a lot of those concerns are kind of just handled by using the sync engine." B2 reported that "half of all bugs come from frontend and backend being out of sync," a category "completely eliminated by sync." B2 further noted that this elimination extends beyond surface bugs to the additional glue code required for client-server state synchronization and the second-order effects of optimistic updates.

Practitioner content confirms that developer experience drives adoption more than user experience benefits. Multiple sources report that sync engines eliminate loading states entirely [168, 169]. Artman [166] reports that Linear's European data center runs almost idle with around 1,000 concurrent users, because nearly every read executes against the local replica. Mathews [169] and Artman [6] both report that sync engines remove entire categories of code: data fetching, cache invalidation, and optimistic update logic. Schickling [168] emphasizes that user interactions should feel immediate, without loading spinners between action and response.

The adoption question is "do I need faster perceived performance and simpler state management?" rather than "do I need synchronization?"

6.2.2. Existing backend compatibility and migration path

Existing infrastructure constrains the sync engine decision more than technical architecture. The interview evidence below points to a practical heuristic. For existing Postgres applications, Zero matches B1 through B5′s adopter requirements and PowerSync matches B6 and B7′s. ElectricSQL is positioned for this case by vendor A3. Greenfield projects fit Convex, Jazz, or InstantDB from the bundled-platform cluster in Section 4.4.

All five Zero users, B1 through B5, require Postgres compatibility. For these participants, the existing database is non-negotiable, and any engine requiring a different data store is immediately eliminated. B4 specifically eliminated bundled platforms because of infrastructure lock-in. A3 explained ElectricSQL's positioning as a library rather than a service: "We don't want to say to our customers that they have to rewrite their entire stack on top of our product." ElectricSQL deliberately integrates shallowly into existing Postgres stacks, requiring no backend changes beyond granting read access [99].

Migration path extends beyond database compatibility. C2 ranks migration path as the single most important requirement: "The number one requirement is not having to do a massive migration." C2 envisions adopting sync one entity type at a time, keeping

the rest of the application on existing HTTP APIs. B5 demonstrates this incremental approach in practice, with 40% of the application on Zero and 60% on tRPC after approximately one year of migration. Bundled platforms face adoption resistance from engineers with existing backends.

A1 from Turso offered a contrasting perspective: many sync use cases are a "boring latency optimization" rather than an architectural shift. If latency matters and the application reads more than it writes, a sync layer on an existing database provides value with minimal disruption.

6.2.3. The offline requirement as binary filter

The offline requirement acts as a binary filter that immediately narrows the viable engine set and changes the vendor relationship. As established in Chapter 4, offline capability is the most discriminating taxonomy dimension.

Both PowerSync users (B6, B7) independently confirm that PowerSync is the only viable option for offline capability combined with Postgres compatibility. B7 put it directly: "Only PowerSync fits our criteria." B7 reports that at the time of evaluation in 2024, ElectricSQL was rebuilding and lacked mobile SDKs, and Replicache was closed source. This lack of alternatives creates a vendor relationship driven by necessity rather than satisfaction. B6 reports: "I would really try hard not to use PowerSync again… but I think I would have to again because there's nothing that replaces what we needed today." B7, by contrast, would "definitely" choose PowerSync again. The difference may reflect team size and tooling capacity. B7′s larger engineering team can absorb rough edges through custom workarounds; B6′s two engineers cannot.

All five Zero users, B1 through B5, reported no need for offline capability. Zero's IndexedDB-backed client store persists data across reloads but is not designed for durable offline use [50]. The default query TTL of five minutes [170] bounds how long inactive queries remain cached while the application is running, not how long persisted state remains usable offline. C3 designed "Sync Light" as an online-only architecture and deliberately avoided the offline-write complexity that full sync engines must address.

C2 challenged the assumption that real-time collaboration is a universal requirement: "No customer has ever asked for [real-time collaboration]. It's an engineer's disappointment, not a real customer need." For C2′s field engineering platform, offline writes matter but collaborative editing does not. As Browne [13] observes, the "local-first" umbrella conflates three separable requirements: offline capability, real-time responsiveness, and multi-user collaboration. Different requirements lead to different engine choices, and engineers benefit from evaluating each independently.

6.2.4. Ecosystem maturity and risk tolerance

Ecosystem maturity appears in 45 of 69 practitioner content sources. The recurring concern is that the sync engine field is too young for production reliance.

B7 reflected on the PowerSync adoption: "We probably underestimated the amount of job it takes to have a very faithful sync engine." B7′s experience illustrates the pioneering cost: custom tooling for schema generation, workarounds for WASM/OPFS browser limitations, and operational processes adapted around sync rule deployment constraints. B3, by contrast, chose Zero partly because of its relative maturity compared to Jazz, observing that Jazz's data storage model remained "a little fuzzy." Zero declared its 1.0 release stable in March 2026 [76], while Jazz v2 entered public alpha in April 2026 [83].

Team size and engineering sophistication mediate the impact of ecosystem immaturity. B7′s larger engineering team built custom infrastructure: schema generation tooling, a hexagonal architecture that abstracts the database, and a transaction replay system. This infrastructure absorbs the rough edges of an immature engine. B6′s team of two engineers lacks the capacity for such workarounds and experiences the immaturity more acutely.

Wienert [171] observes that recent local-first sync attempts including ElectricSQL and InstantDB have not been "pulled off yet fully" as production-ready defaults. Kleppmann [172] similarly argues that it is "too early to standardize." Jayakar [173] argues from Dropbox's experience that the set of edge cases a sync system must handle is "astronomically large," and that all possibilities will eventually occur. The same dynamic applies to sync engines more broadly: building a correct implementation pushes the decision toward buying generalized solutions unless the team has significant engineering resources. Teams adopting sync engines today must accept that APIs, migration paths, and best practices will change.

6.2.5. Abstraction level

Sync engines range from low-level CRDT libraries to fully integrated platforms. Each level balances what engineers can do against what the engine constrains. Spolsky [102] warns that all non-trivial abstractions are leaky, and sync engines are no exception. Practitioner content documents three abstraction tiers with distinct failure modes [151].

High-abstraction platforms such as Convex [26], InstantDB [81], and Jazz v2 [83] handle database, authentication, and sync as a single integrated service. These platforms offer a faster path to a working prototype than approaches that require separately integrating database, sync, and authentication layers. Mathews [169] describes sync engines as enabling prototyping speed where the "prototype is the production system." Bundled

platforms limit the data model, server-side logic, and deployment options in exchange for the integration. Wiggins [174] observes that breaking down app silos requires unfettered data access, and that software vendors are generally incentivized to keep data locked up. The same logic extends to sync: commoditizing sync through proprietary engines risks creating a new lock-in pattern, undermining the data ownership that local-first aims to preserve. A2 framed the trade-off explicitly, describing Jazz as "very explicitly not a sync engine that talks to an existing database, but a sync engine plus a database."

Medium-abstraction sync layers such as Zero, ElectricSQL, and PowerSync integrate with existing Postgres backends, adding sync as a layer without replacing existing infrastructure. Prokopov [175] framed this as a stack collapse, where the application reads and writes against a local replica rather than treating the database as a remote service. This tier enables incremental adoption, as B5′s partial migration demonstrates, but introduces complexity at the integration boundary. Bandi [33] warns against overusing the abstraction: at Figma, applying LiveGraph to all data types caused scaling problems. B5 experienced the boundary in one production migration: Zero's view syncer worked for simpler queries but failed when joins grew complex. Engineers must learn where the abstraction applies and where to fall back to traditional patterns.

Low-abstraction toolkits and libraries such as Yjs [15], Loro [16], RxDB [90], and custom implementations give engineers full control over the sync stack. Linear illustrates this approach: the team spent approximately six months building core application infrastructure including the sync engine before shipping the product [6]. C2 and C3 both built custom synchronization because no off-the-shelf partial sync mechanism fit their requirements. This approach demands more expertise and engineering investment but avoids the constraints that higher abstractions impose.

Browne's case shows how the abstraction-level trade-off can shift over a product's lifetime [13, 147, 176]. Browne rejected Zero because of schema duplication, Classic Jazz because of signed-out user handling, and TinyBase because of WebSocket management overhead, before building a custom solution with Dexie.js and PlanetScale [13]. The custom solution provided full control but created maintenance problems: only Browne had deep enough understanding of the sync code to safely modify it, blocking the rest of the team from shipping features [147]. Browne ultimately migrated to Convex, a high-abstraction platform, because the maintenance burden of the custom solution outweighed the platform's constraints [147]. On Convex, the team later hit a platform outage triggered by a client reconnection storm [176]. Even high-abstraction platforms can still expose engineers to reconnection storms, retry strategies, and platform throughput limits [176]. Complexity did not disappear but shifted from managing sync logic to managing connection lifecycle. This journey illustrates Spolsky [102]'s law of

leaky abstractions. Sync engines abstract how data moves but cannot abstract network connections, database write throughput, or memory consumption.

6.2.6. Build versus buy as the primary competitor

A2 identified the primary competitor to Jazz not as another sync engine but as the engineer's decision to build custom: "The competitor is 'we think we can build it ourselves' first." The decision is not primarily which sync engine to pick but whether to adopt a sync engine at all.

All three custom builders would rebuild the same way. C2 stated: "It would be hard to see a managed service working for our use case. It would need to be some component of our existing TypeScript backend." C3 independently arrived at a component-assembly approach, combining TanStack DB for reactive client state with gRPC streaming for server updates, using general-purpose components rather than sync-specific building blocks. C1 uses CouchDB's replication protocol [67] and would not replace it with a newer sync engine because the conflict visibility that CouchDB provides is essential for the application's data integrity requirements.

The same Linear case shows the upside of the build path: the upfront investment enabled faster feature development later [6]. Browne's case shows the opposite path, from custom sync to a high-abstraction platform after maintenance cost became the larger constraint [147]. Building custom gives control, but maintenance cost scales with team size and product complexity.

B1, despite being a satisfied Zero user, observed: "I know how to build a bad sync engine. I don't know how to build a good one." Jayakar [173]'s correctness argument applies here: most teams underestimate the complexity of handling all race conditions and edge cases. Practitioner content reflects the same caution: 17 sources substantively discuss build-versus-buy.

The build-versus-buy decision hinges on three factors. First, whether the application's requirements fit within the constraints of available engines. If they do, buying avoids reinventing verified infrastructure. Second, whether the team can maintain custom synchronization over time without the maintenance burden outweighing platform constraints. Third, whether the application requires capabilities that no available engine provides, such as C2′s domain-specific partial replication or C1′s conflict-visible revision trees.

6.3. Summary

This chapter evaluated sync engine fitness for different application types and identified the factors that drive the adoption decision, addressing ARQ3.

Application fitness varies by domain. Full-capability sync engines best serve offline field applications, despite the added complexity. Real-time B2B productivity tools sit in the broadest sweet spot. Collaborative editing requires CRDT-based or custom implementations beyond what general-purpose engines provide. Transactional and data-intensive applications are poorly served by the sync model. The emerging "Sync Light" category, real-time read reactivity without offline or conflict resolution, could fit many B2B web applications that do not require offline capability.

Six additional decision factors emerged alongside use case fit. Speed and developer experience drive most adoption decisions. Existing backend compatibility, particularly Postgres, constrains the viable engine set. The offline requirement acts as a binary filter that narrows viable engines and changes the vendor relationship. Ecosystem maturity mediates adoption success. Abstraction level trades ease of adoption against architectural control: connection lifecycle, database throughput, and memory consumption still require direct attention even with high-abstraction platforms. The build-versus-buy decision is the primary decision point rather than engine comparison.

Chapter 7 reports the Teamspective case study, which applies these decision factors and the trade-off framework to a production adoption. Chapter 8 then synthesizes the findings from Chapter 4, Chapter 5, this chapter, and the case study to address the main research question: to what extent can client-side data synchronization be generalized across web applications?

7. Implementation Case Study: Adopting Zero at Teamspective

This chapter reports a case study of Zero adoption at Teamspective, a production B2B SaaS platform. The case tests the trade-offs from Chapter 5 and the decision factors from Chapter 6 in practice. Adoption was straightforward at the API layer, costly at the deployment boundary, and bounded by the authorization model. Both the user-scoped slice and the workspace-scoped slice shipped to production. No other parts of the application moved to Zero during the study period. The case also adds evidence on developer experience as an adoption driver and identifies one open question about the scope of sync-engine query layers. Chapter 3 details the case study method.

7.1. Context and motivation

Teamspective is a B2B SaaS platform for leadership enablement that provides tools for employee engagement, people analytics, and feedback. The current architecture uses type-safe REST APIs with SWR for immutable in-memory caching, placing it at "application-level caching" on the synchronization spectrum defined in Chapter 2.

Workspace configuration data, including users, members, groups, and attributes, is loaded frequently across the application and is performance-sensitive. Two pain points motivated the case study. First, changes made by other users remain invisible until the affected user refreshes the page. SWR invalidates the cache only on the current user's mutations, so cross-user updates such as permission changes or feature flag toggles do not propagate until the next network refetch. Users have reported this staleness problem through customer support, and managers receiving new access often only see the change after a manual reload. Second, workspace data loading contributes to initial page load times because the server must be queried on every cold start.

SWR's session caching already handled route changes within the single-page application (SPA) quickly, so cold-start hydration was the latency Zero could improve. The main user-facing benefit is data freshness: administrative changes such as role grants or feature flag toggles should propagate to affected users without requiring a page reload. Task and action-planning views are plausible later targets because additions and edits by one user could appear on other users' screens without a reload.

7.2. Decision process

Teamspective adopted a sync engine in consultation with thesis advisor Ian Tuomi, partly motivated by early interview findings that Postgres-compatible sync engines fit B2B SaaS applications with existing backends. Postgres compatibility was the primary filter, consistent with the decision factor identified in Chapter 6: all five Zero users and both PowerSync users in the interview sample required Postgres compatibility. This narrowed the viable set to four options: Zero, ElectricSQL with TanStack DB, PowerSync, and a custom implementation.

A custom implementation was not pursued because building reliable synchronization infrastructure requires significant engineering investment, as the build-versus-buy evidence in this chapter confirms, and the team's priority is delivering product value rather than infrastructure work. Three factors ruled PowerSync out. B7 reported limited web platform maturity compared to its mobile experience. Its full database resync on schema changes is incompatible with Teamspective's frequent schema migrations. Its sync rules model, documented in Chapter 5, requires more upfront configuration than Zero's query-based approach. The choice between ElectricSQL and Zero came down to API design, documentation quality, community momentum, and Zero's track record from Replicache.

Teamspective belongs to the real-time B2B productivity tool category identified in Chapter 6: a Postgres-backed B2B application where perceived responsiveness and developer velocity drive adoption rather than offline capability. The case tests whether its permission model hits the authorization complexity limit identified in Chapter 5, where permission enforcement adds state-space complexity that all three sync engine vendors found to be structural.

7.3. Scope

Workspace configuration is the case-study domain. The data covers users, members, groups, attributes, and feature flags, with permissions and roles that change as administrators reassign access. The domain fits the case study because it loads on every authenticated page, has known staleness in production, and has the permissions and relational structure that make synchronization hard.

The implementation worked in two slices, following the tracer-bullet pattern [177]. A tracer bullet builds a thin end-to-end path through every layer of a new system and uses that path to learn about the layers before committing to a larger build. Each slice was narrow in data scope and code paths, even where the workspace-scoped slice's feature flags reach every user. This sizing kept the revert path back to the REST

architecture cheap if the evaluation did not justify continuing. In this case the user-scoped slice both shipped a low-risk user-facing feature and acted as the learning vehicle for the deployment infrastructure that any later sync-engine surface in the application would depend on. The user-scoped slice built the full integration and shipped the user profile settings page on top of it. The integration covered schema generation, queries, mutators, React integration, server endpoints, CI, and production infrastructure. The settings page has two tables and owner-scoped access, which keeps the slice low-risk to break without product impact. The workspace-scoped slice reused the infrastructure and targeted feature flags. Feature flags require a workspace-scoped query path that filters by an explicit `workspaceId` argument and checks the caller's membership in that workspace. The Implementation section reports each slice in turn.

Two feasibility investigations ran alongside the slices. A complex-query stress test ported a 15-table query with viewer-dependent visibility to Zero Query Language (ZQL) and tested the limits of what the query language can express. An API endpoint audit classified the application's 437 endpoints by Zero migration feasibility and estimated the migratable share.

7.4. Zero's schema and authorization

Zero is a database-pluggable sync engine for relational web applications [50]. Two of its design choices shape much of the case study below: the separation of schema from query, and the placement of authorization in application code.

Zero requires its own schema definition in application code, defining a single global row shape per table [178]. Which rows actually sync is determined per-query: applications define named queries and the engine syncs the matched rows to each subscribed client [170].

Authorization runs in application code on both paths [179]. Read authorization sits in `.where()` clauses on centrally defined queries against a server-resolved authorization context. Write authorization sits in server-side mutator functions that can run arbitrary validation, transformation, or rejection logic before the transaction. Together, the row-shape ceiling and the query-layer placement of authorization drive many of the case study's design decisions, including schema-level column filtering and the workspace-scoped slice's authorization context.

7.5. Implementation

The implementation work spans the user-scoped slice with integration infrastructure, two parallel feasibility investigations, the production infrastructure deployment, and the workspace-scoped slice.

7.5.1. Infrastructure and the user-scoped slice

To keep the Postgres database as the single source of truth, I built a code generator that introspects the Postgres catalog and emits Zero's schema. The generator accepts a table allowlist with optional column-level filtering. Column-level filtering became necessary when the initial table-level allowlist would have placed sensitive columns, such as persistent login codes, in the global synced row shape. The risk was caught during development before any cross-user exposure, but the architectural lesson held. Any query against the table would have replicated those columns to whichever clients subscribed, and columns omitted at the schema level cannot be selected back into queries. In Zero, column selection is global, not per-query or per-viewer.

A missing `.where()` clause on a centrally defined query would constitute a data leak, similar to a missing authorization check in any server API. Centralizing queries and mutators in shared definitions reduces this risk relative to per-endpoint authorization scattered across handlers.

The first deployment target was the user profile settings page, because it uses only two tables and is not critical to the product. This was enough to validate the supporting infrastructure at low risk. The core Zero integration, including schema generation, queries, mutators, the React provider, and server endpoints, took approximately one day. Integration with the existing build and deployment tooling took another day or two: pnpm resolution conflicted with transitive dependencies, Docker build paths changed, and routing conflicted with the existing API prefix. None of these problems were Zero-specific; they reflect integration boundary friction in existing build and deployment tooling.

The initial design used a feature flag to switch between Zero and REST inside a component, but the gradual-rollout benefit was outweighed by the cost of maintaining two parallel query paths. Conditional routing would have spread throughout the application, and the data-access code would have nearly doubled. The design would also have limited the production traffic visible to Zero, reducing what could be learned about its connection and query patterns at full scale. The implementation instead deploys Zero per component, with each component using either Zero or REST. The workspace-scoped slice is one exception: because Zero syncs a subset of the data already returned

by the existing `getTeam` REST endpoint, the application can fall back to REST data for feature flags when Zero has not yet connected.

7.5.2. Feasibility exploration: complex query stress test

The settings page deployment proved Zero works for a simple case with two tables and owner-based access. To explore the limits, I ported one of the most complex queries in the codebase to Zero. The query combines structured evaluations, growth feedback, and praise feedback with viewer-dependent visibility, anonymity rules, and aggregates across 15 tables, including feedback, templates, and group memberships.

The stress test identified eight capabilities that ZQL cannot express. Six are query-expressiveness gaps: set operations across heterogeneous query branches, filters over relationships defined in joined tables, aggregates, JSONB path access, derived columns, and cross-column comparisons. A more expressive ZQL could plausibly close these, though doing so likely involves engine-side work in query planning and incremental view maintenance. Aggregates, for instance, have remained an open item on Zero's tracker.[8]

The remaining two, viewer-dependent privacy at the sync boundary and conditional visibility filtering, are different in kind. These collide with the row-shape ceiling rather than a missing query primitive, requiring per-row or per-viewer column projections that the single global row shape cannot express [180]. Per-viewer-derived field values would require either a query layer that supports per-row column selection, or per-viewer-materialized rows in the database. PostgreSQL itself can express the same per-viewer projection logic in raw SQL; the limit is in ZQL's row-shaped query model, not in what the underlying data can express. Each gap forces either client-side post-processing over synced rows for aggregation or transformation cases, or keeping the query on the REST path when the gap touches sensitive data that cannot be safely synced in the first place.

In the stress-test domain, viewer-dependent visibility, rather than the count of missing query primitives, set the boundary on what Zero could migrate. Data the user owns, such as profile and settings, maps cleanly to Zero's `.where('userId', ctx.userId)` pattern, where `ctx` is the server-resolved authorization context introduced earlier. Data with viewer-dependent visibility rules cannot use the same owner-where pattern: raw source rows cannot be safely replicated, since any field on a synced row is readable to that user [180]. For example, the author name on a feedback row may be hidden from the receiver but shown to the receiver's manager. In the REST architecture, the server resolves these rules at request time and returns only the values each user can see. Zero pushes authorization filtering into the sync query layer, so the REST runtime resolution does not port directly. Visibility rules have to be expressed as sync query

[8] Zero issue tracker for aggregates: https://bugs.rocicorp.dev/p/zero/issue/3040. Accessed 2026-05-23.

predicates before the row leaves the server, or the data has to be materialized into per-viewer projections.

7.5.3. API endpoint audit

To assess broader migration feasibility, I audited all 437 API endpoints against Zero migration criteria. Approximately 55% of endpoints could be migrated, counting reads with owner-based or workspace-scoped authorization, writes through server-side mutators, client-side aggregation over already-synced data, and multi-query decomposition for cross-table filters. Approximately 23% could not be migrated: heavy analytics aggregations, external API proxies, authentication flows, file I/O, and AI token streaming. The remaining endpoints sat on the boundary between the two and would have required case-by-case rework to classify. The raw percentage matters less than which endpoints, when migrated, would improve user experience. The highest-value targets for perceived performance and data freshness were reads on every page load that currently display loading spinners, and frequently-changing cross-user data such as task and action-planning views where a manager and a direct report often work together. For writes, Zero would standardize the existing hand-rolled optimistic-update patterns and reconcile concurrent edits at the engine level.

7.5.4. Production infrastructure

Deploying Zero on Teamspective's managed Postgres platform, Render, surfaced constraints that did not appear in local development. Render does not grant the privileges required to configure logical replication, create schema-wide publications, or install event triggers. Each privileged operation required a support ticket. The cumulative wall-clock time across these support cycles, including back-and-forth on what we were trying to enable, totaled roughly two weeks. The stress test and endpoint audit ran in parallel during this period. The initial publication replicated the entire `public` schema, which exceeded practical limits for the SQLite replica disk. We moved write-only audit tables to a new `archive` schema, which reduced the replication scope by approximately 16 gigabytes and brought the production public schema to just under 10 gigabytes. Zero's documentation describes custom Postgres publications as a workaround for permission-constrained providers [181]. Narrowing the publication to the tables covered by the Zero schema would let teams shrink the replication scope directly and reduce resync time on a self-managed Postgres. On Render, publication scope changes require superuser access through support, so the deployment stayed on a `public`-schema publication. Moving Zero-relevant tables into a separate schema would have required SQL refactoring across queries that were not naturally aligned to a sync boundary.

Without event trigger support, every schema change that touches replicated tables triggers a full replica resync. Resync becomes a recurring operational cost rather than a one-time setup cost. The privileges required to configure Zero on a managed Postgres platform vary by provider, and Zero's documentation includes provider-specific setup sections for several common platforms [181].

A measured schema change on the staging environment, where the replica is 1.66 GB, took 1 minute 48 seconds from detection to a fully resynchronized replica, with total client-perceived downtime of approximately 2 minutes. During resynchronization, the `zero-cache` process exits intentionally and restarts automatically, the Zero client retries the connection every five seconds, and locally cached data remains readable in the browser, even in this predominantly online deployment. Even this 2-minute downtime would have been unacceptable for routine schema migrations. The projected production cost on the 9.7 GB replica was approximately 12 minutes by linear extrapolation, an estimate later confirmed at 14 minutes by the high-availability (HA) failover documented below. Most schema changes should be unnoticeable to users, but in this configuration each one would have produced a multi-minute service interruption.

The recurring resync cost prompted us to build a custom workaround. The initial implementation wrapped each migration with hooks that emitted schema-snapshot WAL messages via `pg_logical_emit_message()` before and after the migration's DDL statements. A patched `zero-cache` processed these messages as incremental schema changes without requiring event trigger installation. Three small patches handled WAL message ordering edge cases that arose because DDL and data changes running in the same migration transaction reached the WAL before the post-migration schema snapshot. This workaround was discussed in the project's Discord[9]. The upstream maintainers later shipped a built-in `zero_0.update_schemas()` Postgres function in version 1.5, which performs the same diff-and-emit work internally and exposes it as a SQL call [182, 183].

Adopting the upstream primitive removed the need to copy schema-introspection queries into the migration runner, but the migration boundary alone did not close the original ordering gap. The application-side response was a SQL rewriter that parses each migration before it is applied and emits `SELECT zero_0.update_schemas();` between every top-level statement. Statement-level snapshots eliminate the ordering gap, and two of the three original patches were dropped. One four-line patch is the only remaining Zero modification in the deployment.

A subsequent vendor-side high-availability failover gave the first production-scale measurement of full resync: the provider's primary-side restart terminated the WAL

[9]Discord thread: https://discord.com/channels/830183651022471199/1494256522283192400. Accessed 2026-04-28.

connection, the replication slot did not survive, and `zero-cache` wiped the local replica and ran a full initial sync. Client-perceived downtime was approximately 14 minutes on the 9.7 GB production replica, scaling roughly linearly with the staging measurement. The full-replica-wipe behavior on slot loss is a property of Zero's design: a consumer dependent on a logical replication slot cannot resume from the slot's last position once it is gone, so recovery scales with the upstream's row count. PostgreSQL 17 added failover-safe replication slots [184, 185], but the resilience improvement depends on both Zero and the host platform configuring their respective halves [186]. Render had not exposed the server-side configuration at the time of the case study.

A second slot-loss event days later wiped the replica again, this time without an HA failover on the upstream side. Subsequent mornings surfaced a distinct failure mode in which memory pressure on the single-node `zero-cache` produced repeated out-of-memory restarts under surge load. The single-node deployment runs upstream ingestion and client serving in the same process, so resync competes for memory with the active WebSocket pool.

Both failure modes motivated migrating to Zero's multi-node deployment shape, in which a replication-manager service owns upstream ingestion and view-syncer services serve WebSocket clients from separate local replicas restored from S3-compatible object storage [187]. The split decouples upstream recovery from client-serving capacity: view-syncers continue serving while the replication-manager runs a resync, so a slot-loss event need not terminate active client traffic.[10] The deployment requires S3-compatible object storage for Litestream's continuous WAL backup [188], which Render does not provide directly. We use Cloudflare R2 [189] with client-side Age encryption [190] applied before upload, so the replica in R2 is encrypted with a key the storage provider does not hold.

The slot-loss incidents also prompted disabling Postgres HA on the upstream cluster. For a sync engine consuming Postgres through a logical replication slot, an HA failover terminates the connection, and on providers without the PostgreSQL 17 failover-safe slot mechanism the slot does not survive, triggering a full resync that scales with upstream row count [184]. Two other factors reduced confidence in HA as a net positive: the provider experienced its own downtime within HA mode, and the application's broader HA configuration had not in practice prevented application-side downtime. The team accepted some risk of unscheduled primary downtime in exchange for eliminating failover-driven slot loss.

Multi-node deployment ran into another Render platform limit. Running multiple view-syncer instances would have enabled rolling deploys onto warm peers and spread the deploy-time reconnect load across them [187]. Render does not allow scaling any

[10] Aaron Boodman of Rocicorp confirmed this graceful-degrade behavior in Discord correspondence on 2026-05-12.

service with an attached persistent disk to multiple instances [191]. The view-syncer retains a SQLite replica on disk, so this constraint pins the service to a single instance. The deployment therefore runs a single view-syncer on a larger plan rather than multiple smaller instances behind a load balancer. Zero's open-source client retries on a uniform five-second cadence with no client-side jitter.[11] Disconnected clients therefore reconnect in bursts onto a cold-starting view-syncer at each deploy. During the case-study period, the resulting reconnect storm was absorbed by the larger plan's memory headroom without further out-of-memory events.

Hosting `zero-cache` on a separate subdomain matches Zero's recommended deployment topology [179]. This required migrating the application's session cookie from host-only to parent-domain scoping so that `zero-cache` could read it, in a system that had relied on host-only cookies for years. The migration was implemented as a small middleware in the existing session-refresh path, so the rollout did not require coordinated changes across the rest of the application.

7.5.5. The workspace-scoped slice and authorization context

The workspace-scoped slice required propagating workspace identity into Zero's query path so that workspace-scoped queries could check the caller's workspace membership. The team first tried designs that placed workspace identity in the Zero connection's authorization context rather than in the query arguments, to avoid repeated `workspaceId` arguments and membership-join logic on every query. Two such designs were rejected before deployment. The first kept workspace identity in the authorization context, populated from a server-validated cookie. Zero identifies its per-query state by query name and arguments.[12] With `workspaceId` absent from the arguments, two tabs running the same query for different workspaces would look identical to Zero, and state that should stay separate could be reused across them. The second design carried a server-precomputed list of accessible workspaces in the authorization context. This would have required resolving the list on the server before the Zero client could connect, delaying every page load and adding client-side complexity around connection initialization and request waterfalls. Workspace-membership changes would also have propagated only when the list refreshed on a timer, rather than as the underlying data changed.

The team eventually settled on the simpler pattern of keeping workspace identity in the query rather than the connection. Zero connection context carries only `userId`,

[11] Verified against the open-source Zero client at commit `e707eaf`: `RUN_LOOP_INTERVAL_MS = 5_000` in `packages/zero-client/src/client/zero.ts`, and the sleep helper in `packages/shared/src/sleep.ts` uses plain `setTimeout` without randomization.

[12] The query hash combines name and arguments via `hashOfNameAndArgs` at `packages/zero-protocol/src/query-hash.ts`.

and `workspaceId` is passed as an explicit query argument. A reusable query wrapper composes a `whereExists` check against the user's direct membership or against an active superadmin grant. The relevant membership and admin-grant tables were added to the client schema, and membership changes propagate via Zero's native IVM on the next WAL event, without an auxiliary refresh interval. The slice shipped to production. Feature flag toggles propagated live to connected clients, and Sentry recorded no Zero-related errors in the three business days following the deployment, a window too short to claim stabilization.

7.6. Analysis and framework application

The Zero-based architecture sits in a hybrid position on the synchronization spectrum: the user profile settings page and workspace feature flags operate as a sync engine, while the rest of the application remains at application-level caching through SWR. This section revisits the trade-offs from Chapter 5 and the decision factors from Chapter 6 in light of the case-study findings, and connects them to the theoretical foundations in Chapter 2.

7.6.1. Confirmed trade-offs

Four trade-offs from Chapter 5 materialized in the case study: hidden integration costs, client-generated identifiers, schema-level visibility, and authorization complexity.

Hidden integration costs materialized at the build toolchain level rather than in Zero's own API. This confirms the interview finding that such costs emerge only after adoption, and the case study extends the finding from code-level to infrastructure-level friction, as discussed in the next subsection.

Client-generated identifiers are required for Zero inserts because optimistic creates need an ID that matches the server's eventual state [192]. Auto-incrementing integers cannot provide this: a client-guessed ID diverges from the server-assigned ID, and any optimistic relationships created against the guessed ID point at the wrong row [192]. The implementation kept row creation on the REST API rather than redesigning the primary key columns from auto-incrementing integers to client-generated UUIDs. This compromise confirms the interviewee reports of client-generated IDs as a friction point in sync engine adoption and is the reason the initial deployment is limited to update-only mutations.

Schema-level visibility required column-level filtering because the initial table-level allowlist would have placed sensitive columns, such as persistent login codes, in the global synced row shape. With Zero, the shipped schema determines the client data

surface and there is no per-endpoint field selection. The concern was known from interviews but was still surprising when it materialized during development.

Authorization complexity surfaced in the form predicted by Section 5.2.1: owner-based data syncs cleanly through `.where()` filters on a user identity column, and viewer-dependent data requires server-side resolution or materialized permission tables. The stress test identified that boundary: ZQL cannot express the required per-viewer authorization rules. This particular boundary, ZQL's row-shape limit, is a query-format choice in Zero's current implementation rather than a fundamental synchronization constraint. Raw PostgreSQL can compute the per-viewer projections, and a more expressive query layer or materialized per-viewer rows would handle it. The broader authorization-complexity limit identified in Chapter 5 remains intact, since it concerns how sync engines as a category surface permission state rather than which row shapes Zero can express. The materialized-table option moves complexity from the query layer into the data layer: every change to a role, team, or permission template must fan out to the materialized rows it affects, typically through database triggers or application-side fan-out. Teamspective has precedent for materialized permission tables from survey participations, but the feedback domain's viewer-dependent rules would require new fan-out infrastructure to keep the materialized state in sync as roles, teams, and templates change.

7.6.2. Managed infrastructure friction extends the hidden-cost finding

Teamspective's deployment surfaced a second layer of hidden cost, namely managed-infrastructure friction, that the interview findings, focused mainly on code-level integration, did not capture. The Production infrastructure section above documents the Render-specific privilege constraints, the WAL workaround, and the recurring full-resync cost in detail. B7 reported a structurally similar pattern with PowerSync, where redeploying sync rules triggers full database reprocessing that takes hours and forces night-time deployment windows. The same operational pattern, recurring full reprocessing as a deployment tax, appears in Zero on a managed platform but at a different layer. In PowerSync the cost is inherent to the engine, since changing sync rules invalidates the replica by definition. In Zero the cost depends on the host platform, since elevated-privilege Postgres features may not be exposed by every managed provider. On managed Postgres providers that expose more of these privileges than Render does, teams would have more control over publication scope, event triggers, and replication slot configuration than the case study could exercise. Teams evaluating a sync engine on their managed Postgres platform should budget for this friction explicitly because sync-engine documentation does not always surface the full cost of provider-specific workarounds.

The case study also identifies a managed-infrastructure trade-off that distinguishes sync engines from stateless web services running against the same managed Postgres. A stateless service tolerates HA failovers because each request reconnects cheaply. A sync engine consumes Postgres through a long-lived logical replication slot, and an HA failover terminates that connection. On providers that have not configured the failover-safe slot mechanism added in PostgreSQL 17, the slot may not survive the failover, as the production-infrastructure section documents. Sync engine fitness on a managed Postgres platform therefore depends not only on the platform's feature compatibility but also on the operational profile of its HA mode and the sync engine's recovery semantics under slot loss.

A second framework refinement concerns the sync engine's own deployment shape. Zero's single-node deployment runs upstream ingestion and client serving in the same process, which is operationally simple but couples recovery cost to client-serving capacity under load. Zero's multi-node deployment splits ingestion from serving, which is designed to absorb heavy events such as full resync without terminating client traffic, at the cost of additional services, S3-compatible storage, and the operational work of running multiple processes. Sync engine adopters should anticipate the single-node-to-multi-node transition as production load grows, rather than treating either shape as a permanent choice.

Rocicorp has launched Cloud Zero, a managed service for deploying Zero on the vendor's infrastructure [193]. This would absorb the Zero-side operational complexity of running multi-node services and Litestream-driven object storage, but would not change the upstream Postgres provider's privilege model, slot survival behavior, or HA mode. Zero integrates with Postgres through logical replication, schema introspection, and replication slot management, and the abstraction leaks at each of these points when the host Postgres restricts the relevant features.

7.6.3. Setup difficulty confirms the adoption pattern

The core Zero integration was straightforward, which confirms the adoption pattern from interviews in which speed and developer experience drive the decision. B3′s observation about Zero offering the shortest path to a live application matches the case study experience at the code level. The case study qualifies this finding: "easy" applies to the sync engine's own abstractions, while the surrounding ecosystem, including monorepo build tooling, Docker, and managed Postgres, adds friction that is easy to underestimate. Within the team, two of the three peer engineers asked to use Zero in their own feature work ahead of any organizational push, and one had previously used Zero on a personal side project. This mirrors the developer-experience pull reported in interviews.

7.6.4. Connection to theoretical foundations

The case study findings connect to three of the foundational principles discussed in Chapter 2. Brooks Jr. [101]'s distinction between essential and accidental complexity frames the managed infrastructure friction. Zero reduces the essential complexity of synchronization by providing a general-purpose read-path reactivity abstraction and a server-reconciled write path. Replacing an application-level cache with a sync engine introduces new accidental complexity at integration boundaries: the build toolchain, the deployment infrastructure, and the managed database platform. The multi-node deployment migration is a concrete example: decoupling upstream ingestion from client serving is accidental complexity created by the production load profile, not essential complexity of synchronization itself. These frictions do not reflect inherent limits of synchronization; they reflect that sync engines have not yet been commoditized across the full deployment stack.

The same friction can be read through Spolsky [102]'s law of leaky abstractions. Zero and Electric abstract synchronization behind query and mutator APIs, but both are tightly coupled to Postgres and the abstraction leaks at the database layer: engineers must still provision logical replication, publications, and event triggers for sync to work at all. On a managed platform that restricts these features, the leak widens because each workaround is provider-specific and not fully captured in the sync engine's own documentation.

A second leak appears at the query layer. ZQL's row-shape cannot express per-viewer column projections even though the database can compute them [178, 180]. Integrators must either restructure the data into per-viewer rows or keep the query on REST.

The authorization findings connect to the end-to-end argument [31]. Zero pushes authorization to application-defined query and mutator code. Read authorization sits in query definitions as `.where()` clauses on user identity, and write authorization sits in server mutator functions. No intermediary system enforces permissions automatically. This follows the end-to-end principle and explains why authorization complexity remains the dominant limit: authorization is application-specific by the end-to-end argument, and sync engines honor this by not trying to generalize it. Per-viewer field projections follow the same pattern: ZQL leaves them to application-specific code or materialized data structures rather than absorbing them into the row-shape model.

7.6.5. Fitness framework reconciliation

The Teamspective case fits the real-time B2B productivity tool category in Table 11. The settings page deployment confirms this classification for owner-based data, which maps cleanly to Zero's `.where('userId', ctx.userId)` pattern. The feedback domain

stress test illustrates the authorization complexity boundary of that fitness. The workspace-scoped slice extends the case to workspaceId-filtered queries, where an explicit `workspaceId` argument composes with a `whereExists` authorization gate at the query layer. The shipped slice confirms that the user-scoped and workspace-scoped patterns coexist within the same query language.

The managed Postgres findings suggest that the fitness framework should be qualified by infrastructure context. A B2B productivity application on self-hosted Postgres has higher fitness for Zero than the same application on a managed provider that restricts logical replication and event triggers. The infrastructure context is a decision factor that this thesis framed primarily as "existing backend compatibility" at the database level; the case study refines it to existing backend *deployment* compatibility, including the platform's privilege restrictions.

7.7. Implications for the framework

The strongest implication is that ZQL's row-shape limit, not authorization complexity in general, is what set the migration boundary in this case. This sharpens one of the framework's authorization findings into a query-format choice that a more expressive query layer could change. The case study also extends the hidden-cost finding to deployment infrastructure, validates sliced rollout as an adoption pattern, and adds within-team evidence for developer experience as an adoption driver. One structural question stays open about the scope of sync-engine query layers.

ZQL cannot express the required per-viewer rules in queries, but the rules can move to materialized data, where the cost shifts to keeping the materialized state in sync. The framework can recommend materialized permission tables as a path forward, but the cost does not disappear: every change to roles, teams, and templates triggers updates across the materialized state. For this case, the framework treats ZQL's row-shape limit as a query-format choice in the current sync engine rather than a fundamental synchronization constraint, while the broader authorization-limit question stays open.

Fitness assessment must cover deployment-infrastructure compatibility, not only data-domain fit. Privileged Postgres features and the recurring resync cost are decision factors that the data-model framing alone does not surface.

The sliced rollout served a second purpose beyond feature-level risk management. The tracer-bullet shape established a thin end-to-end path through every layer of the new deployment, which the team used to learn the operational profile of Zero on a managed Postgres platform before extending the deployment to the workspace-scoped surface [177]. Two slot-loss events, a memory-pressure cascade under load, the multi-node migration, and the HA decision all played out on the user-scoped slice's two-table footprint,

with impact bounded to the user profile settings page rather than the application's core workflows. The case suggests a deployment-strategy addition to the framework: size the first slice for operational learning rather than feature delivery, and defer broader rollout until the initial slice has stabilized in production. The Teamspective workspace-scoped slice shipped after the slot-loss incidents, the multi-node migration, and the HA decision had been addressed, though the host platform's single-instance constraint on disk-backed services remained unresolved at that point. Within-team evidence for developer experience as an adoption driver also appeared, with peer engineers asking to use Zero in their own feature work ahead of any organizational push.

One question the case study raises but cannot resolve concerns the scope of ZQL and other sync-engine query layers, where three response options stay open. First, the query language can expand so that viewer-dependent rules and per-row column projections become expressible: sync engines would carry more application logic but generalize across more domains. Second, query layers can stay row-shaped and accept that complex authorization moves to the data layer through materialized tables: the engine stays simple but each application carries fan-out infrastructure. Third, sync engines can accept that some application-specific business logic does not fit at all, and applications can design hybrid architectures where the sync engine covers the cases it serves well and REST covers the rest. The Teamspective case tilts toward the third response in practice, since the slice scope explicitly excludes endpoints where authorization is too complex to materialize. Materializing some permissions into dedicated tables is already on the team's roadmap, motivated as much by REST-side query performance and code clarity as by sync suitability, which would shift more of the surface toward the second response over time. A single case cannot generalize.

8. Discussion

This chapter synthesizes findings across the three research questions and discusses implications for practitioners and researchers. The main research question asks to what extent client-side data synchronization can be generalized across web applications. The three additional research questions provide the evidence: ARQ1 maps the architectural patterns, ARQ2 identifies their trade-offs and limits, and ARQ3 evaluates fitness across application types.

Sync engines generalize partially. Read paths and online interaction patterns generalize across application types. Offline writes remain bounded by authorization, conflict resolution, partial replication, and domain logic.

8.1. Synthesis of findings

Four claims frame the answer given in Chapter 9. Sync engines cluster into discrete architectural groups rather than spreading across a spectrum. The online/offline boundary determines what can be generalized. Latent synchronization problems appear in many production applications as unrelated bugs. Incremental adoption on existing infrastructure makes generalization practical.

8.1.1. Sync engines do not form a spectrum

Chapter 2 defines a synchronization spectrum from server-authoritative request-response to fully local-first architectures. Sync engines, however, cluster into discrete groups rather than spreading evenly across this spectrum. The taxonomy in Chapter 4 classifies 14 sync engine instances along eight dimensions and reveals four architectural clusters rather than a single spectrum. Database-pluggable engines such as Zero, ElectricSQL, and PowerSync integrate with existing Postgres backends. Bundled platforms such as Convex and InstantDB replace the backend entirely. CRDT-decentralized engines such as Jazz, Ditto, and Automerge provide genuine local-first operation. Singular engines such as LiveStore, RxDB, and Turso Sync occupy unique positions outside these three groups.

Choosing a sync engine is a question of which architectural cluster fits the application's constraints, not of "how much sync." The most discriminating dimension is offline capability: it constrains which authority models are viable, which in turn constrains conflict resolution and authorization strategies. An application's offline requirement determines which cluster is relevant.

8.1.2. The online/offline boundary determines generalization

For always-online applications, synchronization generalizes well on the read path. The read path delivers data from server to clients through subscriptions, incremental view maintenance, or shape-based filtering. Vendors, adopters, and custom builders all converge on the read path as the part of sync that can be commoditized.

The write path can also generalize for online applications when the engine owns write-path semantics. Zero applies mutations optimistically on the client, re-executes them on the server, and rolls back on failure [126]. This model avoids conflicts because writes serialize through the server. All five Zero users reported no production conflicts. PowerSync illustrates the boundary: it transports writes to the application's backend but does not execute them. Business logic and conflict resolution remain with the engineer. Engineers can still mix consistency levels inside one application. They can fall back to server-authoritative endpoints where strict consistency matters.

For offline-capable applications, generalization becomes conditional. Offline writes create state that must be reconciled when connectivity returns, and the reconciliation forces application-specific decisions about authorization and conflict resolution. The broader offline-capable architecture must also handle partial replication and schema evolution across replicas. Chapter 5 examines each in detail. Vendor architectures handle these demands differently. ElectricSQL excludes offline writes [99]. Jazz keeps them under server validation in v2 [83]. PowerSync delegates write execution to the application backend by design. Together these architectures point to the same boundary: the write path for offline applications varies too much across domains to commoditize.

Mapping engines across the read/write and online/offline combinations confirms the same split. On always-online reads, subscriptions, incremental view maintenance, and shape filters generalize the pattern, with Zero, ElectricSQL, and Convex each using one of these. On offline-capable reads, a local replica serves reads regardless of network state, as in PowerSync, Jazz, and Ditto. On always-online writes, generalization works when the engine owns mutation semantics: Zero [50] and Convex [26] take this path with different consistency models. On offline-capable writes, no engine fully generalizes, and PowerSync's engine-transported model, Jazz v2′s pending-state model with server-enforced permissions, and decentralized CRDT engines all leave different parts of the boundary to the application.

8.1.3. Synchronization problems appear as unrelated bugs

Many applications have synchronization problems they do not recognize. Applications built with traditional request-response or client-side caching encounter data staleness, stale permissions, and inconsistent state across concurrent users. These issues

are often not identified as sync problems until they manifest in production. B2 reported that roughly half of the bugs in their team's application originate in mismatches between client-side and server-side state. B6 reports that even single-user multi-device usage causes data loss through last-write-wins overwriting.

The Teamspective case study illustrates the pattern directly. The application uses `useSWRImmutable` for nearly all queries with no global cache invalidation. A user who edits a document and returns to the document list can see the stale title because the save did not invalidate the list query. Concurrent edits fall to last-write-wins or leave users on divergent views of the same record. Admin changes, including role grants, management permission changes, and feature flag toggles, do not reach affected sessions without a manual refresh. The team treated these as unrelated refresh bugs rather than as symptoms of a shared synchronization problem. Teamspective users opened support tickets when newly granted permissions did not appear. Refreshing was not their first instinct, and from the user's perspective it should not need to be.

A sync engine maintains consistency between server and client state as a baseline, reducing this class of bug within the surface the engine covers. As B2 noted in Chapter 6, the savings reach further: applications can shrink the hand-written cache and state-sync code they maintain, along with the additional handling each optimistic update brings for rollback and reconciliation. Query-oriented sync engines such as Zero derive the client view from queries against the local replica, so both client and server can avoid carrying parallel state representations for the synced surface. The adoption driver across interview participants is speed and developer experience rather than synchronization as a goal in itself.

8.1.4. Incremental adoption makes generalization practical

The theoretical fitness of a sync engine matters only if adoption is practical. Incremental adoption on existing infrastructure is the condition that makes generalization viable. Every interview group emphasized this requirement. B5 migrated 40% of a freight CRM to Zero over one year while keeping the rest on traditional APIs. C2 named migration path as the deciding factor in evaluating any sync engine. B6 and B7, despite different satisfaction levels with PowerSync, both operate the sync engine as a component within a larger backend, not as a replacement.

Postgres compatibility is a prerequisite for most practitioners. All five Zero users require it. Both PowerSync users require it. Bundled platforms that replace the database face adoption resistance from engineers with existing production systems. Greenfield products may see less resistance, as B5′s adoption of Convex for a separate internal product alongside Zero on the main application illustrates.

The Teamspective case illustrates both conditions. The team did not shortlist database-bundled platforms such as Convex and InstantDB, because replacing Postgres would have required renegotiating data processing agreements and redoing transfer impact assessments, a vendor-change cost that would have needed roughly a 10x benefit over database-pluggable alternatives to justify. Zero adoption is strictly additive: Zero runs alongside the existing REST API with each component using one or the other, and the first end-to-end slice took roughly two weeks of elapsed calendar time, most of which was spent waiting on managed-Postgres support tickets rather than on application development.

8.2. Implications for practice

This section presents five implications for engineers evaluating sync adoption: how to sequence the adopt-assemble-build decision, when offline requirements dominate, what latent problems sync engines remove, where to keep concerns outside the sync layer, and how local data enables client-side computation.

8.2.1. Adopt before assembling, assemble before building

The interview evidence points to an ordering for synchronization strategy. Sync engines have already solved data delivery, cache consistency, and optimistic updates. Adopting one is simpler than assembling equivalent functionality from libraries. B1 makes the same point from the adopter's perspective: practitioners can build inadequate sync engines, but rarely good ones. Jayakar [173] argues that race conditions and edge cases make building a correct sync engine extremely hard. Browne's experience shows that custom sync maintenance cost can grow to exceed the constraint cost of adopting an engine [147].

If no sync engine fits, the next option is assembling general-purpose components. C3′s "Sync Light" architecture combines TanStack DB for reactive client state with Connect RPC streaming for server updates, achieving real-time read reactivity without adopting a sync-specific engine. This approach requires more integration work but avoids the constraints that sync engines impose.

Building fully custom synchronization is appropriate when the application requires capabilities that no available engine provides. C2 needed domain-specific partial replication and C1 needed conflict-visible revision trees, neither of which any sync engine offered at decision time. These gaps may narrow as engines mature.

8.2.2. Evaluate the offline requirement first

The offline requirement narrows the viable engine set and changes the trade-off profile. Applications that require offline writes face a constrained market, with PowerSync as the only Postgres-compatible option with full offline capability, and must accept higher architectural complexity including dual schemas, sync rules, and client-generated identifiers.

Applications without an offline requirement have broader options and simpler trade-offs. For this group, speed and developer experience drive the decision more than synchronization capability.

8.2.3. Sync engines solve problems before they appear

Applications built with client-side caching can carry latent synchronization problems. Data staleness, stale permissions, and inconsistent state across concurrent users appear when each component fetches data independently and treats responses as snapshots. Two mechanisms recur in the interview and case-study evidence: cache invalidation is easy to forget on each mutation, and when an invalidation runs it is scoped to the local session, so changes by other users do not surface at all. These issues are often recognized only after users report them in production. The Teamspective case study confirmed the pattern: permission, feature flag, and post-save staleness appeared as individual refresh bugs rather than as symptoms of a shared synchronization problem.

A sync engine maintains consistency between server and client as a baseline. Engineers working in single-user development environments rarely anticipate these bugs, which a sync-maintained baseline removes.

8.2.4. Do not force everything through sync

Sync engines are not a replacement for all server communication. Engineers should sync interactive, user-facing data while keeping other concerns on traditional infrastructure. B5 keeps a 51,000-record reference database on traditional queries because the volume exceeds what is practical to sync. B4 routes LLM token streaming through async mutators outside of Zero because token-by-token database writes overwhelm the sync model. File uploads, background jobs, image processing, and heavy aggregations remain outside the sync engine's domain, as B6′s experience illustrates.

8.2.5. Client-side data enables client-side computation

When a sync engine replicates data to the client, the data becomes available for local computation beyond what the sync engine itself provides. Client-side search, filtering, and aggregation on the user's data subset happen without server roundtrips. Artman

[166] reports that Linear runs nearly all read operations locally, while its European deployment supports roughly 1,000 concurrent users on two CPU cores at about $80 per month. For the kind of interactive workloads that local-first architectures target, modern end-user devices, including mobile devices, are capable of handling computation that would otherwise run on the application's server instances [194, 195]. Moving computation to the client reduces server load and improves responsiveness.

This resource shift may also have a sustainability dimension. Siffre et al. [194] report that local-first architectures can substantially reduce network traffic and server uptime by relocating computation to existing client resources, with a Google Docs experiment showing roughly 160 times more network traffic in online editing than in offline editing of the same content. The benefit is conditional rather than universal: synchronization and merge work shifts to clients rather than disappears, and the net energy effect depends on application characteristics [194].

B7 demonstrated an extension of this pattern where an LLM communicates directly with the local SQLite database, running arbitrary queries on the user's authorized data subset. B7 notes: "Most of the time when products want to enable this, they have to make a custom language or a custom layer to validate that what the client is doing isn't breaching authorization." Local data sidesteps this authorization problem because the database contains only what the user is allowed to see. As AI-driven features become common in enterprise applications, this may become a practical advantage of the sync engine architecture.

8.3. Validity and limitations

The following subsections cover internal, external, and construct validity, researcher positionality, and the strength of each major claim. Reliability is addressed in Chapter 3.

8.3.1. Internal validity

The study relies on semi-structured interviews, practitioner content analysis, and a case study. Each method introduces potential biases.

Interview findings depend on participant recall and willingness to share negative experiences. Vendor participants in Group A may emphasize their engine's strengths and downplay limitations. Including three separate groups, namely vendors, adopters, and custom builders who rejected sync engines, mitigates this bias. Where vendor claims conflict with user experience, the user evidence takes precedence.

The practitioner content analysis uses AI-assisted extraction of 69 sources. The pass 1 extractions compress varying depths of discussion into similar-length summaries, which

limits the granularity of the coding. The three-level rating scale (Substantive/Weak/None) was adopted specifically because the finer Strong/Moderate distinction could not be reliably assessed from AI-extracted summaries. Manual validation against original sources reduces extraction errors, but subtle misinterpretations may remain.

The interview coding followed the framework analysis method, with all interviews analyzed on the same day using a consistent template. Later interviews included probes informed by earlier findings, which improves depth but may introduce confirmation bias toward emerging themes.

8.3.2. External validity

The findings reflect a specific moment in a fast-moving field. Zero reached stable 1.0 in March 2026 [76] during the study period, and Jazz v2 entered public alpha in April 2026 [83]; the specific engine classifications may date as engines add capabilities or change direction. The interview sample is weighted toward Zero users, who account for five of seven adopters; no primary adopter of Convex, Jazz, InstantDB, or Ditto was interviewed, though B5 uses Convex on a separate internal product. The practitioner content corpus partially compensates for this gap, with 69 sources covering a broader set of engines and practitioners.

The case study examines one company, Teamspective, using one engine, Zero, on one managed Postgres provider, Render, in one application domain, B2B SaaS workspace configuration. The implementation took place within a small engineering team with competing product priorities, which limited how much ground the case study could cover within the thesis period. Generalizing the case-study findings to other domains, engines, company sizes, or managed Postgres providers would require additional validation. The B2B productivity-tool fit observed across the five Zero adopters does not extend to B2C contexts without further evidence.

The case study covered a narrow surface of the application. Both slices, user-scoped and workspace-scoped, shipped to production. The implementation exercised only update mutations against existing rows. Insert paths with auto-incrementing identifiers were not exercised because the pain point that motivated the case study, stale data on the read path, did not require new row creation. Only a small part of the application runs on Zero, so findings about its behavior under additional workspace tables, more complex authorization queries, or higher mutation volume would require further slices.

8.3.3. Construct validity

The taxonomy dimensions are derived from three independent sources, namely distributed systems literature, practitioner content analysis, and semi-structured interviews. This triangulation reduces the risk that the taxonomy reflects only one

perspective. One dimension, read/write path architecture, emerged from interviews and has no precedent in prior academic work, which limits its theoretical grounding but strengthens its practical relevance.

The trade-off and decision factor categories F1 through F6 and D1 through D6 were defined before data collection as part of the analytical framework and refined during analysis. Pre-defined categories may miss emergent themes. The triangulation coding layer captured 55 novel insights not anticipated by the initial framework, confirming that the coding was not limited to a priori categories.

8.3.4. Researcher positionality

The author works as an engineer at Teamspective, the case study company. This provides insider access to the codebase, infrastructure, and user feedback. Familiarity with the existing system may bias the evaluation of how sync adoption compares to the status quo.

The choice of Zero as the case study engine was informed by the same interview findings that the thesis presents. Early interviews indicated Zero as a strong fit for B2B SaaS with existing Postgres backends, and the case study then evaluates whether this is true. This creates a potential confirmation loop, where the research that informed the engine choice is also used to evaluate it. The study mitigates this risk in three ways. The interview sample includes custom builders in Group C who rejected sync engines and can articulate why generalized solutions fail. The vendor interviews cover three different engines, namely Turso, Jazz, and ElectricSQL, not just Zero. Prior to the thesis, the author had not used any sync engine under study in production and had no financial or organizational ties to any sync engine vendor.

8.3.5. Claim strength

Some conclusions rest on triangulated evidence across all three interview groups, the practitioner content corpus, and the case study; others depend primarily on the case study or specific subsamples. Table 13 maps each major conclusion to its supporting evidence and what would extend the claim further.

Table 13: Major conclusions mapped to supporting evidence and what would extend the claim further.

Conclusion	**Supported by**	**Generalizing further would require**
Four architectural clusters; not a single spectrum	14-engine classification across eight dimensions; vendor self-positioning aligns with cluster boundaries	Reclassifying engines emerging after the study period; cluster stability under more instances
Online/offline boundary determines generalization	Three vendor pivots converge on server authority for offline writes; the read/write boundary generalizes in three of four cells across the 14-engine taxonomy (Table 9); offline-write resistance recurs across vendor, adopter, and custom-builder interviews	Engines resolving offline-write authorization without server enforcement; CRDT-based access control at scale
Authorization as a structural limit	Vendor convergence across A1, A2, A3; O'Brien's theoretical argument [160]; case study stress-test confirms predicted edge cases	Empirical decentralized-authorization deployments at scale
Hidden costs extend to deployment infrastructure	Case study managed-Postgres friction; B7 PowerSync schema-resync evidence; C3 deployment-cost observation	Multi-provider case studies; provider-by-provider audit
Developer experience drives adoption	All five Zero adopters cite developer experience as central; practitioner content from Schickling [168], Mathews [169], and Artman [6] confirms; case study peer engineers volunteered to use Zero ahead of organizational push	Adoption studies in larger organizations and outside the Postgres-Zero ecosystem
B2B productivity-tool fit	Zero adopters B1–B5 build B2B productivity-tool applications matching the profile	B2C application case studies; productivity tools outside SaaS

The cluster taxonomy and online/offline boundary are the most strongly supported claims; engine classification, vendor pivots, and triangulated interview evidence converge on them. For deployment-infrastructure hidden costs, the case study is the primary evidence; multi-provider studies would extend the claim. B2C fit is untested, and the productivity-tool finding holds only for the five Zero adopters interviewed.

8.4. Future work

This study classified 14 sync engine instances and interviewed 13 practitioners. A wider survey covering more engines and more production applications would test whether the taxonomy dimensions and fitness findings hold at greater scale. The taxonomy captured engines at a point in time, and Zero's 1.0 release during the study period illustrated how quickly the field changed. Longitudinal studies could track how engine architectures evolve and whether the limits identified here are resolved by future engines.

The case study evaluated a single engine at a single company. Comparative studies across multiple engines, companies, and application domains would strengthen the generalization claims. Quantitative measurement of migration costs, developer productivity, and data freshness improvements across a larger sample would complement the qualitative findings.

This thesis identified authorization, partial replication, and offline write-path generalization as inherent limits. Future work could explore whether new architectural approaches resolve these limits. CRDT-based access control mechanisms are already an active area of research [161, 162]. New partial replication strategies, such as the questions Terry [152] posed nearly two decades ago, remain open. Engines that bridge the online/offline boundary without requiring the application to handle conflict resolution would expand the domain where generalization is viable.

The "Sync Light" pattern identified by C3, combining TanStack DB with streaming for read-path reactivity without full sync, also warrants further investigation. This middle ground between traditional caching and full sync engines may serve a large segment of web applications that do not require offline capability or conflict resolution.

Local data replication also enables client-side computation, including AI-driven queries on the user's data subset. ElectricSQL introduced Electric Agents in April 2026, framing multi-agent coordination explicitly as a sync problem and treating agent sessions as syncable streams [196]. Sync architectures applied to non-synchronization purposes deserve dedicated study.

9. Conclusion

Sync engines aim to make synchronization a reusable infrastructure layer rather than custom application code. This thesis investigated to what extent that promise holds. The study examined which application types sync engines serve well, where generalization encounters limits, and what determines fitness in practice. The study combined a literature review, practitioner content analysis of 69 sources, semi-structured interviews with 13 practitioners, and a case study.

9.1. Contributions

This thesis contributes a decision framework for sync engine adoption in a field where terminology and adoption criteria remain unsettled. The framework identifies a boundary: where client-side data synchronization can be generalized as reusable infrastructure for web applications, and where the abstraction breaks down. That boundary follows the interaction between an application's online/offline requirement and its read/write path. The taxonomy, trade-off and limits analysis, application-fitness model, and Teamspective case study let engineers locate their application relative to that boundary before committing to an architecture.

First, the taxonomy classifies 14 sync engine instances into four architectural clusters: database-pluggable, bundled platform, CRDT-decentralized, and singular. The taxonomy is derived from distributed systems literature, practitioner content, and interview evidence rather than marketing categories. Offline capability is the most discriminating dimension, because it constrains authority model, conflict resolution, and write-path architecture.

Second, the trade-off analysis identifies seven technical trade-offs and five limits of sync engine generalization. The read path generalizes well, while the write path resists generalization for offline-capable applications. Authorization, global invariants, and partial sync are limits rather than navigable trade-offs.

Third, the fitness analysis explains where sync engines fit. Real-time B2B productivity tools are the broadest sweet spot for sync engines. Speed and developer experience, rather than synchronization itself, drive most adoption decisions. Incremental adoption on existing Postgres infrastructure makes generalization practical.

Fourth, the Teamspective case study applies the framework to Zero adoption in a B2B SaaS application on managed Postgres. The case confirms the fit for owner-based data, sharpens the authorization limit for viewer-dependent rules, shows that managed-infra-

structure friction adds cost, and adds organic-pull evidence to the developer-experience adoption driver.

9.2. Four architectural clusters

ARQ1 asked what architectural patterns exist across the synchronization spectrum from server-authoritative to local-first. Sync engines cluster into four architectural groups rather than sitting on a single spectrum: database-pluggable engines such as Zero, bundled platforms such as Convex, CRDT-decentralized engines such as Jazz, and singular engines such as LiveStore. Server authority dominates the field: 8 of 14 instances use server-authoritative or hybrid approaches. One dimension, read/write path architecture, emerged from practitioner interviews and distinguishes between engine-owned, engine-transported, and read-only write paths, a distinction absent from prior taxonomies.

9.3. Trade-offs and limits

ARQ2 asked what trade-offs and limits these architectural patterns introduce. The answer combines practitioner interviews, practitioner content analysis, and case study evidence to distinguish navigable trade-offs from fundamental limits.

Seven trade-offs emerge from the practitioner evidence. Engineers selectively apply different consistency levels to different parts of their applications. Reads generalize well while writes resist standardization. Incremental adoption is a universal requirement. Authorization drove three vendors toward server authority. Partial replication is the least discussed but most operationally painful dimension. Conflict resolution is academically prominent but practically rare. Hidden costs including schema evolution, client-generated identifiers, and platform maturity gaps emerge after adoption.

Five limits emerge from convergent evidence across interviews, practitioner content, and the case study, rather than from isolated implementation problems. Authorization complexity explodes the state space when combined with sync. Global invariants create a mathematical boundary between offline availability and consistency. Application-specific business logic and conflict resolution prevent the write path from generalizing. Browser platform immaturity constrains web-based local-first architectures independently of the sync engine. Partial sync remains unsolved despite being a defining feature of sync engines.

9.4. Application fitness

ARQ3 asked how application requirements determine sync engine fitness. Offline field applications benefit most from full-capability sync engines despite higher complexity. Real-time B2B productivity tools are the broadest sweet spot, with adoption driven by speed and developer experience. Collaborative editing requires CRDT-based or custom implementations. The sync model serves transactional and data-intensive applications poorly.

The offline requirement acts as a binary filter that narrows the viable engine set. Existing backend compatibility, particularly Postgres, constrains the decision further. The primary decision is whether to adopt a sync engine at all, not which engine to pick.

The Teamspective case study confirms these findings in practice. Teamspective dismissed database-bundled platforms before the shortlist on compliance and data protection grounds, which confirms the existing-backend constraint, and the managed Postgres setup extends that constraint to include the privileges granted by the provider.

9.5. The extent of generalization

The main research question asked to what extent client-side data synchronization can be generalized across web applications. The answer is conditional on the online/offline boundary. Sync engines generalize as web application infrastructure for always-online applications, but break down where offline writes meet application-specific authorization, conflict resolution, and business logic.

Client-side data synchronization can be generalized for always-online web applications. The read path generalizes across application types through subscriptions, incremental view maintenance, and shape-based filtering. The write path generalizes when the engine owns write-path semantics, as Zero demonstrates through mutation replay, server reconciliation, and rollback.

Generalization breaks down for offline-capable applications. Offline writes require conflict resolution, create authorization edge cases, and demand coordinated schema management across server and client replicas. These constraints are structural; engineering effort cannot remove them. The trade-off analysis shows that three sync engine vendors converged independently on this conclusion from different starting positions.

9.6. Closing remarks

Sync engines generalize partially, not universally. They work best when the application stays within the online boundary and can express interactive data through engine-owned or read-only sync paths. They break down for offline writes, where authorization, conflict resolution, partial replication, and domain logic become application-specific. The contribution of this thesis is therefore a boundary. It shows where generalized synchronization is a useful infrastructure abstraction and where custom application logic remains necessary.

For engineers evaluating sync engines, the offline requirement is the most consequential filter. Applications without offline-write requirements have a low-effort path through database-pluggable engines that integrate with existing Postgres backends. Adding offline writes brings structural constraints that no current engine fully resolves; those constraints should shape the architectural decision rather than be left to the implementation.

Whether the boundary moves over time is itself the open question. CRDT-based access control could ease the authorization limit. New partial replication strategies could reduce operational friction. Architectures that handle offline writes without exposing conflicts to the application would expand where generalization is viable. The "Sync Light" middle ground, read-path reactivity with server-authoritative writes, suggests another path: narrower abstractions may serve applications that do not need offline capability.

For now, the evidence places the boundary here. Sync engines are reusable infrastructure within it and a starting point for application-specific code beyond it. The structural limits identified here are open research problems, not engineering oversights.

Bibliography

[1] J. Nielsen, *Usability Engineering*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994. ISBN 978-0-08-052029-2.

[2] S. Egger, T. Hossfeld, R. Schatz, and M. Fiedler, "Waiting Times in Quality of Experience for Web Based Services," in *2012 Fourth International Workshop on Quality of Multimedia Experience*, July 2012, pp. 86–96. doi: 10.1109/QoMEX.2012.6263888.

[3] Vercel, "SWR," Feb. 2026, https://swr.vercel.app/, [Accessed Feb. 09, 2026].

[4] TanStack, "Optimistic Updates | TanStack Query React Docs," Nov. 2025, https://tanstack.com/query/latest/docs/framework/react/guides/optimistic-updates, [Accessed May 25, 2026].

[5] M. Kleppmann, A. Wiggins, P. van Hardenberg, and M. McGranaghan, "Local-First Software: You Own Your Data, in Spite of the Cloud," in *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Oct. 2019, pp. 154–178. doi: 10.1145/3359591.3359737.

[6] T. Artman, "Linear, Sync Engines, Rethought Startup MVP," Oct. 2024, https://www.youtube.com/watch?v=XTMyOtvBJ_g, [Accessed Jan. 31, 2026].

[7] Y. Saito and M. Shapiro, "Optimistic Replication," *ACM Comput. Surv.*, vol. 37, no. 1, pp. 42–81, Mar. 2005, doi: 10.1145/1057977.1057980.

[8] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-Free Replicated Data Types," in *Stabilization, Safety, and Security of Distributed Systems*, 2011, pp. 386–400. doi: 10.1007/978-3-642-24550-3_29.

[9] P. Viotti and M. Vukolić, "Consistency in Non-Transactional Distributed Storage Systems," *ACM Comput. Surv.*, vol. 49, no. 1, pp. 19:1–19:34, June 2016, doi: 10.1145/2926965.

[10] C. Sun and C. Ellis, "Operational Transformation in Real-Time Group Editors: Issues, Algorithms, and Achievements," in *Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work*, Nov. 1998, pp. 59–68. doi: 10.1145/289444.289469.

[11] A. Imine, M. Rusinowitch, G. Oster, and P. Molli, "Formal Design and Verification of Operational Transformation Algorithms for Copies Convergence," *Theoretical Computer Science*, vol. 351, no. 2, pp. 167–183, Feb. 2006, doi: 10.1016/j.tcs.2005.09.066.

[12] M. Weidner, “Architectures for Central Server Collaboration,” June 2024, https://mattweidner.com/2024/06/04/server-architectures.html, [Accessed Feb. 15, 2026].

[13] T. Browne, “My Chaotic Journey to Find the Right Database,” Feb. 2025, https://www.youtube.com/watch?v=3gVBjTMS8FE, [Accessed Feb. 05, 2026].

[14] TanStack, “QueryCache | TanStack Query Docs,” Feb. 2026, https://tanstack.com/query/latest/docs/reference/QueryCache, [Accessed Feb. 09, 2026].

[15] P. Nicolaescu, K. Jahns, M. Derntl, and R. Klamma, “Yjs: A Framework for Near Real-Time P2P Shared Editing on Arbitrary Data Types,” in *Proceedings of the 15th International Conference on Engineering the Web in the Big Data Era - Volume 9114*, June 2015, pp. 675–678. doi: 10.1007/978-3-319-19890-3_55.

[16] Loro Labs, “Loro – Reimagine State Management with CRDTs – Loro,” Feb. 2026, https://loro.dev/, [Accessed Feb. 16, 2026].

[17] P. van Hardenberg, “Ink and Switch, Automerge,” Jan. 2026, https://www.youtube.com/watch?v=9f2owS3nDHs, [Accessed Jan. 31, 2026].

[18] S. Gilbert and N. Lynch, “Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services,” *SIGACT News*, vol. 33, no. 2, pp. 51–59, June 2002, doi: 10.1145/564585.564601.

[19] D. Abadi, “Consistency Tradeoffs in Modern Distributed Database System Design: CAP Is Only Part of the Story,” *Computer*, vol. 45, no. 2, pp. 37–42, Feb. 2012, doi: 10.1109/MC.2012.33.

[20] R. T. Fielding, “Architectural Styles and the Design of Network-Based Software Architectures,” Publication, Irvine, 2000.

[21] A. Boodman and A. Chase, “What Is Sync?,” July 2025, https://zero.rocicorp.dev/docs/sync, [Accessed Feb. 04, 2026].

[22] S. S. Kulkarni, M. Demirbas, D. Madappa, B. Avva, and M. Leone, “Logical Physical Clocks,” in *Principles of Distributed Systems*, 2014, pp. 17–32. doi: 10.1007/978-3-319-14472-6_2.

[23] L. Lamport, “Time, Clocks, and the Ordering of Events in a Distributed System,” *Commun. ACM*, vol. 21, no. 7, pp. 558–565, July 1978, doi: 10.1145/359545.359563.

[24] W. Vogels, “Eventually Consistent,” *Queue*, vol. 6, no. 6, pp. 14–19, Oct. 2008, doi: 10.1145/1466443.1466448.

[25] N. Preguiça, “Conflict-Free Replicated Data Types: An Overview,” June 2018, http://arxiv.org/abs/1806.10254, [Accessed Feb. 09, 2026]. doi: 10.48550/arXiv.1806.10254.

[26] J. Cowling, “How to Design a Sync-First Database,” Dec. 2025, https://www.youtube.com/watch?v=ngN3PoXDFJI, [Accessed Jan. 31, 2026].

[27] E. A. Brewer, “Towards Robust Distributed Systems,” 2000, https://sites.cs.ucsb.edu/~rich/class/cs293b-cloud/papers/Brewer_podc_keynote_2000.pdf, [Accessed Feb. 09, 2026]. doi: 10.1145/343477.343502.

[28] J. Browne, “Brewer's CAP Theorem,” Jan. 2009, https://www.julianbrowne.com/article/brewers-cap-theorem/, [Accessed Feb. 09, 2026].

[29] C. Hofmeyr, “SQLITE Persistence on the Web,” Dec. 2025, https://www.youtube.com/watch?v=7yNG1aj7-Aw, [Accessed Jan. 31, 2026].

[30] PowerSync, “Writing Data,” https://docs.powersync.com/client-sdks/writing-data, [Accessed Apr. 04, 2026].

[31] J. H. Saltzer, D. P. Reed, and D. D. Clark, “End-to-End Arguments in System Design,” *ACM Trans. Comput. Syst.*, vol. 2, no. 4, pp. 277–288, Nov. 1984, doi: 10.1145/357401.357402.

[32] E. Wallace, “How Figma's Multiplayer Technology Works | Figma Blog,” Oct. 2019, https://www.figma.com/blog/how-figmas-multiplayer-technology-works/, [Accessed Feb. 15, 2026].

[33] A. Bandi, “Arushi Bandi (Figma) - A Tale of Two Sync Engines,” Nov. 2025, https://www.youtube.com/watch?v=2FB-Yrf8eLo, [Accessed Jan. 31, 2026].

[34] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser, “Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System,” in *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, Dec. 1995, pp. 172–182. doi: 10.1145/224056.224070.

[35] Postman, “2025 State of the API Report,” 2025, https://www.postman.com/state-of-api/2025/, [Accessed May 24, 2026].

[36] R. T. Fielding, M. Nottingham, and J. Reschke, “Hypertext Transfer Protocol (HTTP/1.1): Caching,” Request for Comments RFC7234, June 2014. doi: 10.17487/RFC7234.

[37] M. Nottingham, “HTTP Cache-Control Extensions for Stale Content,” Request for Comments RFC5861, May 2010. doi: 10.17487/RFC5861.

[38] Vercel, “Automatic Revalidation,” Dec. 2025, https://swr.vercel.app/docs/revalidation, [Accessed May 25, 2026].

[39] TanStack, “Important Defaults | TanStack Query React Docs,” Apr. 2026, https://tanstack.com/query/latest/docs/framework/react/guides/important-defaults, [Accessed May 25, 2026].

[40] TanStack, "Query Invalidation | TanStack Query React Docs," Dec. 2025, https://tanstack.com/query/latest/docs/framework/react/guides/query-invalidation, [Accessed May 25, 2026].

[41] "Cache," https://swr.vercel.app/docs/advanced/cache, [Accessed Feb. 09, 2026].

[42] "persistQueryClient | TanStack Query React Docs," https://tanstack.com/query/latest/docs/framework/react/plugins/persistQueryClient, [Accessed Feb. 09, 2026].

[43] "Window: localStorage Property - Web APIs | MDN," Nov. 2025, https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage, [Accessed Feb. 09, 2026].

[44] "IndexedDB API - Web APIs | MDN," Apr. 2025, https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API, [Accessed Feb. 09, 2026].

[45] "Origin Private File System - Web APIs | MDN," July 2025, https://developer.mozilla.org/en-US/docs/Web/API/File_System_API/Origin_private_file_system, [Accessed Feb. 09, 2026].

[46] D. Meyer, "Discover RxDB Storage Benchmarks | RxDB - JavaScript Database," Jan. 2026, https://rxdb.info/rx-storage-performance.html, [Accessed Feb. 09, 2026].

[47] "Service Worker API - Web APIs | MDN," Nov. 2025, https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API, [Accessed Feb. 09, 2026].

[48] A. Feyerke, "Designing Offline-First Web Apps," Dec. 2013, https://alistapart.com/article/offline-first/, [Accessed May 02, 2026].

[49] P. van Hardenberg and M. Kleppmann, "PushPin: Towards Production-Quality Peer-to-Peer Collaboration," in *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*, Apr. 2020, pp. 1–10. doi: 10.1145/3380787.3393683.

[50] A. Boodman and A. Chase, "When To Use Zero," July 2025, https://zero.rocicorp.dev/docs/when-to-use, [Accessed Feb. 04, 2026].

[51] P. S. Almeida, "Approaches to Conflict-free Replicated Data Types," *ACM Comput. Surv.*, vol. 57, no. 2, pp. 51:1–51:36, Nov. 2024, doi: 10.1145/3695249.

[52] M. Kleppmann and A. R. Beresford, "A Conflict-Free Replicated JSON Datatype," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 10, pp. 2733–2746, Oct. 2017, doi: 10.1109/TPDS.2017.2697382.

[53] G. Litt, S. Lim, M. Kleppmann, and P. van Hardenberg, "Peritext: A CRDT for Collaborative Rich Text Editing," *Proc. ACM Hum.-Comput. Interact.*, vol. 6, no. CSCW2, pp. 531:1–531:36, Nov. 2022, doi: 10.1145/3555644.

[54] P. Nicolaescu, K. Jahns, M. Derntl, and R. Klamma, "Near Real-Time Peer-to-Peer Shared Editing on Extensible Data Types," in *Proceedings of the 2016 ACM International Conference on Supporting Group Work*, Nov. 2016, pp. 39–49. doi: 10.1145/2957276.2957310.

[55] Ink & Switch, "Automerge," Feb. 2026, https://automerge.org/, [Accessed Feb. 09, 2026].

[56] D. Wang, A. Mah, and S. Lassen, "Google Wave Operational Transformation," technical report, July 2010.

[57] J. Schickling and J. Martin, "Local-First Landscape," Jan. 2026, https://www.localfirst.fm/landscape, [Accessed May 17, 2026].

[58] M. Kleppmann and C. Riccomini, *Designing Data-Intensive Applications*, 2nd Edition. California, US: O'Reilly Media, Inc., 2026. ISBN 978-1-0981-1905-8.

[59] PowerSync, "PowerSync: Backend DB - SQLite Sync Engine | For Postgres, MongoDB, MySQL," Jan. 2026, https://www.powersync.com/, [Accessed Jan. 25, 2026].

[60] "Offline-First Apps Made Simple: Supabase + PowerSync," https://www.powersync.com/blog/offline-first-apps-made-simple-supabase-powersync, [Accessed Jan. 21, 2026].

[61] "Introduction to Apollo Client," https://www.apollographql.com/docs/react, [Accessed May 02, 2026].

[62] T. Linsley, "TanStack DB," May 2025, https://www.youtube.com/watch?v=hy9pNJMFfyM, [Accessed Feb. 22, 2026].

[63] J. Arthur, "Introducing TanStack DB," June 2025, https://www.youtube.com/watch?v=ia9FpY_Sw_4, [Accessed Feb. 22, 2026].

[64] "Subscriptions | PostGraphile," Nov. 2025, https://postgraphile.org/postgraphile/5/subscriptions, [Accessed May 02, 2026].

[65] "PartyKit - Turn Everything into a Realtime Multiplayer App," https://www.partykit.io/, [Accessed May 02, 2026].

[66] L. Kawell, S. Beckhardt, T. Halvorsen, R. Ozzie, and I. Greif, "Replicated Document Management in a Group Communication System," in *Proceedings of the 1988 ACM Conference on Computer-supported Cooperative Work*, Jan. 1988, p. 395. doi: 10.1145/62266.1024798.

[67] Apache Software Foundation, "Apache CouchDB," Feb. 2026, https://couchdb.apache.org/, [Accessed Feb. 10, 2026].

[68] D. Katz, “Blogging Couch,” Apr. 2005, https://web.archive.org/web/20050414075925/http://damienkatz.net/2005/04/blogging_couch.html, [Accessed May 25, 2026].

[69] “Firebase Realtime Database,” https://firebase.google.com/docs/database, [Accessed Feb. 10, 2026].

[70] C. Metz, “'Firebase' Does for Apps What Dropbox Did for Docs,” Apr. 2012, https://www.wired.com/2012/04/firebase/, [Accessed Feb. 10, 2026].

[71] “Replicache: Framework for Local-First Web Apps.,” https://replicache.dev/, [Accessed Feb. 10, 2026].

[72] A. Boodman, “Release v0.3.0 $c\cdot$ Rocicorp/Replicache”, Sept. 2020, https://github.com/rocicorp/replicache/releases/tag/v0.3.0, [Accessed May 25, 2026].

[73] A. Boodman, “General-Purpose Sync with IVM,” June 2025, https://www.youtube.com/watch?v=39CizIAHpw0, [Accessed Jan. 31, 2026].

[74] J. Arthur, “ElectricSQL, A Local First Sync Engine,” July 2024, https://www.youtube.com/watch?v=EY9zs3z6jCE, [Accessed Jan. 31, 2026].

[75] A. Eickhoff, “Oops, My Sync Engine Has Become a Database,” Dec. 2025, https://www.youtube.com/watch?v=wils2KFCgEU, [Accessed Jan. 31, 2026].

[76] A. Boodman, “Zero 1.0,” Mar. 2026, https://zero.rocicorp.dev/docs/release-notes/1.0, [Accessed Apr. 30, 2026].

[77] J. Arthur, “Electric 1.0 Released | Electric,” Mar. 2025, https://electric.ax/blog/2025/03/17/electricsql-1.0-released, [Accessed May 22, 2026].

[78] K. Botha, “PowerSync Update: November 2023,” Dec. 2023, https://powersync.com/blog/powersync-update-november-2023, [Accessed May 22, 2026].

[79] A. Cole, “Announcing Convex 0.1.5,” July 2022, https://news.convex.dev/announcing-convex-0-1-5/, [Accessed May 25, 2026].

[80] T. Ballinger, “Announcing Convex 1.0,” July 2023, https://news.convex.dev/announcing-convex-1-0/, [Accessed May 22, 2026].

[81] InstantDB, “InstantDB: A Modern Firebase,” Feb. 2026, https://www.instantdb.com/, [Accessed Feb. 16, 2026].

[82] “About - InstantDB,” https://www.instantdb.com/about, [Accessed May 22, 2026].

[83] A. Eickhoff, “What Is Jazz?,” Apr. 2026, https://jazz.tools/blog/what-is-jazz, [Accessed Apr. 30, 2026].

[84] A. Eickhoff, “What We Learned from Classic Jazz,” Apr. 2026, https://jazz.tools/blog/what-we-learned-from-classic-jazz, [Accessed Apr. 30, 2026].

[85] A. Fish, “CRDTS Solved Conflicts, Not Sync,” Dec. 2025, https://www.youtube.com/watch?v=UmLSpcausF0, [Accessed Jan. 31, 2026].

[86] O. Henry and P. van Hardenberg, “Towards Production: Getting in Shape,” Dec. 2020, http://inkandswitch.github.io/automerge-rs/post/towards-production/, [Accessed May 22, 2026].

[87] “Introducing Automerge 2.0,” Jan. 2023, https://automerge.org/blog/automerge-2/, [Accessed May 22, 2026].

[88] K. Mathews, J. Cowling, J. Schickling, and A. Boodman, “Sync Panel Discussion - Sync (Sync),” Dec. 2025, https://www.youtube.com/watch?v=vnhlEC0aG3I, [Accessed Jan. 31, 2026].

[89] J. Schickling, “LiveStore: SQLite-based Data Layer for Local-First Apps,” Nov. 2024, https://expo.dev/blog/local-first-application-development-with-livestore, [Accessed May 22, 2026].

[90] D. Meyer, “RxDB - JavaScript Database,” Dec. 2025, https://rxdb.info/, [Accessed Feb. 09, 2026].

[91] D. Meyer, “RxDB README,” May 2026, https://github.com/pubkey/rxdb/blob/eb7e3c285c07ba132efc3c648fabcefb1562bf2b/README.md, [Accessed May 25, 2026].

[92] N. Sivukhin, “Introducing Databases Anywhere with Turso Sync,” Oct. 2025, https://turso.tech/blog/introducing-databases-anywhere-with-turso-sync, [Accessed Feb. 09, 2026].

[93] G. Costa, “Introducing Offline Writes for Turso,” Oct. 2024, https://turso.tech/blog/introducing-offline-writes-for-turso, [Accessed May 22, 2026].

[94] “Offline Sync Public Beta,” Mar. 2025, https://turso.tech/blog/turso-offline-sync-public-beta, [Accessed Feb. 09, 2026].

[95] J. Pearce, “TinyBase,” July 2024, https://www.youtube.com/watch?v=1DyaHuTyJkE, [Accessed Jan. 31, 2026].

[96] “TinyBase,” https://tinybase.org/, [Accessed Feb. 16, 2026].

[97] Apache Software Foundation, “PouchDB, the JavaScript Database That Syncs!,” Feb. 2026, https://pouchdb.com/, [Accessed Feb. 10, 2026].

[98] “Releases $c\cdot$ Apache/Pouchdb”, https://github.com/apache/pouchdb/releases, [Accessed Feb. 10, 2026].

[99] J. Arthur, “ElectricSQL, Read-Path Syncing, PGLite,” Dec. 2024, https://www.youtube.com/watch?v=gNVpNU4gWos, [Accessed Jan. 31, 2026].

[100] A. Boodman, “Your Web App but Good:,” July 2024, https://www.youtube.com/watch?v=rqOUgqsWvbw, [Accessed Jan. 31, 2026].

[101] F. P. Brooks Jr., “No Silver Bullet Essence and Accidents of Software Engineering,” *Computer*, vol. 20, no. 4, pp. 10–19, Apr. 1987, doi: 10.1109/MC.1987.1663532.

[102] J. Spolsky, “The Law of Leaky Abstractions,” Nov. 2002, https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/, [Accessed Jan. 12, 2026].

[103] D. L. Parnas, “On the Criteria to Be Used in Decomposing Systems into Modules,” *Commun. ACM*, vol. 15, no. 12, pp. 1053–1058, Dec. 1972, doi: 10.1145/361598.361623.

[104] J. Niinikoski, “Local-First Architecture: Investigating the Suitability for Modern Browser Applications,” Master's thesis, Aalto University, Espoo, 2025.

[105] Y. Zhang, M. Weidner, and H. Miller, “Programmer Experience When Using CRDTs to Build Collaborative Webapps: Initial Insights,” in *PLATEAU 13th Annual Workshop at the Intersection of PL and HCI*, Mar. 2023. doi: 10.1184/R1/22277341.v1.

[106] J. Haas, R. Mogk, E. Yanakieva, A. Bieniusa, and M. Mezini, “LoRe: A Programming Model for Verifiably Safe Local-first Software,” *ACM Trans. Program. Lang. Syst.*, vol. 46, no. 1, pp. 2:1–2:26, Jan. 2024, doi: 10.1145/3633769.

[107] A. Margara, G. Cugola, N. Felicioni, and S. Cilloni, “A Model and Survey of Distributed Data-Intensive Systems,” *ACM Comput. Surv.*, vol. 56, no. 1, pp. 16:1–16:69, Aug. 2023, doi: 10.1145/3604801.

[108] S. Jayakar, “A Map of Sync,” Oct. 2024, https://stack.convex.dev/a-map-of-sync, [Accessed May 02, 2026].

[109] J. Ritchie and L. Spencer, “Qualitative Data Analysis for Applied Policy Research,” *Analyzing Qualitative Data*. Routledge, London, pp. 173–194, 1994. doi: 10.4324/9780203413081.

[110] N. K. Gale, G. Heath, E. Cameron, S. Rashid, and S. Redwood, “Using the Framework Method for the Analysis of Qualitative Data in Multi-Disciplinary Health Research,” *BMC Medical Research Methodology*, vol. 13, p. 117, Sept. 2013, doi: 10.1186/1471-2288-13-117.

[111] “Local-First Conf 2025,” https://www.localfirstconf.com/, [Accessed Jan. 25, 2026].

[112] “Sync Conf | Nov 12, 2025 in San Francisco.,” https://syncconf.dev/, [Accessed Jan. 25, 2026].

[113] J. Schickling, "Local-First Podcast," https://www.localfirst.fm/, [Accessed May 25, 2026].

[114] "Devtools.Fm," https://devtools.fm/, [Accessed Jan. 31, 2026].

[115] "Syntax - Web Development Podcast," Jan. 2026, https://syntax.fm/, [Accessed Jan. 31, 2026].

[116] A. Pandey, S. Dumbrava, M. Shapiro, C. Ferreira, M. Pereira, and N. Preguica, "Towards Local-First Distributed Property Graphs," in *Proceedings of the 12th Workshop on Principles and Practice of Consistency for Distributed Data*, Apr. 2025, pp. 22–29. doi: 10.1145/3721473.3722139.

[117] K. De Porre, "When Sequential Code Meets Replicated Data," Doctoral dissertation, Brussel, 2022.

[118] "Device Sync Deprecation - Atlas App Services - MongoDB Docs," https://www.mongodb.com/docs/atlas/app-services/sync/device-sync-deprecation/, [Accessed May 25, 2026].

[119] "Triplit | The Fullstack Database," https://triplit.dev/, [Accessed Feb. 16, 2026].

[120] "Liveblocks | Realtime Infrastructure for Multiplayer Apps and Agents," https://liveblocks.io/, [Accessed May 17, 2026].

[121] "Jamsocket/y-Sweet," May 2026, https://github.com/jamsocket/y-sweet, [Accessed May 17, 2026], Jamsocket.

[122] "DXOS Home," https://dxos.org/, [Accessed May 17, 2026].

[123] "NextGraph - Decentralized, Encrypted and Local-First Platform and Framework, towards the Better Internet We All Deserve!," https://nextgraph.org/, [Accessed May 17, 2026].

[124] "Basic | The Personal Computing Platform," https://basic.tech/, [Accessed May 17, 2026].

[125] J. Lehnardt, "15 Years of Local First: A Best-of Report from the Field," June 2025, https://www.youtube.com/watch?v=_x0kmL-AZ00, [Accessed Jan. 31, 2026].

[126] "Mutators," https://zero.rocicorp.dev/docs/mutators, [Accessed May 25, 2026].

[127] J. Averbukh, S. Parunashvili, D. Woelfel, and D. Harris, "A Backend for AI-coded Apps," Apr. 2026, https://www.instantdb.com/essays/architecture, [Accessed May 02, 2026].

[128] J. Schickling and N. Burk, "Offline Support," Dec. 2025, https://docs.livestore.dev/patterns/offline/, [Accessed May 25, 2026].

[129] P. Esterhazy, "Two Myths about Building Offline-Capable SaaS Apps," July 2024, https://www.youtube.com/watch?v=oKCFuYhD7c8, [Accessed Feb. 22, 2026].

[130] "Repositories," https://automerge.org/docs/reference/repositories/, [Accessed Mar. 08, 2026].

[131] J. Schickling and N. Burk, "Syncing," Jan. 2026, https://docs.livestore.dev/reference/syncing/, [Accessed May 25, 2026].

[132] A. Boodman, "Replicache and Zero, Building Sync Engines for the Web," Feb. 2025, https://www.youtube.com/watch?v=wubqEnUisP8, [Accessed Jan. 31, 2026].

[133] "Sync Rules (Legacy)," https://docs.powersync.com/sync/rules/overview, [Accessed May 24, 2026].

[134] "Sync Streams," https://docs.powersync.com/sync/streams/overview, [Accessed May 17, 2026].

[135] "Migrating from Sync Rules," https://docs.powersync.com/sync/streams/migration, [Accessed May 17, 2026].

[136] Apache Software Foundation, "2.1. Introduction to Replication — Apache CouchDB® 3.5 Documentation," 2025, https://docs.couchdb.org/en/stable/replication/intro.html, [Accessed May 02, 2026].

[137] A. Eickhoff, "The Four Fresh Ideas behind Jazz," Apr. 2026, https://jazz.tools/blog/four-fresh-ideas-behind-jazz, [Accessed May 02, 2026].

[138] "Partial Sync," https://docs.turso.tech/sync/partial, [Accessed Feb. 09, 2026].

[139] "Handling Update Conflicts," https://docs.powersync.com/handling-writes/handling-update-conflicts, [Accessed May 24, 2026].

[140] J. Averbukh, "InstantDB Conflict Resolution Comment," Apr. 2026, https://news.ycombinator.com/item?id=47707632#47720031, [Accessed May 25, 2026].

[141] "Usage," https://docs.turso.tech/sync/usage, [Accessed Feb. 09, 2026].

[142] J. Schickling and N. Burk, "How LiveStore Works," Dec. 2025, https://docs.livestore.dev/evaluation/how-livestore-works/, [Accessed May 25, 2026].

[143] PowerSync, "App Backend Setup," https://docs.powersync.com/configuration/app-backend/setup, [Accessed Apr. 04, 2026].

[144] C. Hofmeyr, "Postgres Logical Replication Challenges & Solutions," May 2024, https://powersync.com/blog/postgres-logical-replication-challenges-solutions, [Accessed May 25, 2026].

[145] "Sync and Storage: Jazz Cloud or Self-Hosted | Jazz," https://classic.jazz.tools/docs/react/core-concepts/sync-and-storage, [Accessed May 25, 2026].

[146] "Architectural Options | TinyBase," https://tinybase.org/guides/the-basics/architectural-options/, [Accessed May 02, 2026].

[147] T. Browne, "Why I Moved Away from SQL," July 2025, https://www.youtube.com/watch?v=gZ4Tdwz1L7k, [Accessed Feb. 09, 2026].

[148] M. Kleppmann, "CRDTs, Automerge, Generic Syncing Servers & Bluesky," Feb. 2024, https://www.youtube.com/watch?v=BNUXsCRQj3Q, [Accessed Jan. 31, 2026].

[149] M. Köhler, G. Zakhour, P. Weisenburger, and G. Salvaneschi, "Consistent Local-First Software: Enforcing Safety and Invariants for Local-First Applications," *IEEE Transactions on Software Engineering*, vol. 51, no. 1, pp. 53–65, Jan. 2025, doi: 10.1109/TSE.2024.3477723.

[150] "Postgres Sync | Electric," https://electric.ax/sync/postgres-sync, [Accessed May 25, 2026].

[151] W. Bos and S. Tolinski, "Sync Engines and Local Data," July 2025, https://syntax.fm/924, [Accessed Jan. 31, 2026].

[152] D. B. Terry, "Partial Replication," *Replicated Data Management for Mobile Computing*. Springer International Publishing, Cham, pp. 49–55, 2008, https://doi.org/10.1007/978-3-031-02477-1_5, [Accessed Mar. 30, 2026]. doi: 10.1007/978-3-031-02477-1_5.

[153] C. Gutwin and S. Greenberg, "A Descriptive Framework of Workspace Awareness for Real-Time Groupware," *Computer Supported Cooperative Work (CSCW)*, vol. 11, no. 3, pp. 411–446, Sept. 2002, doi: 10.1023/A:1021271517844.

[154] T. Artman, "Linear," July 2023, https://www.youtube.com/watch?v=Vk15EYX6C8g, [Accessed May 22, 2026].

[155] G. Litt, P. van Hardenberg, and O. Henry, "Project Cambria: Translate Your Data with Lenses," Oct. 2020, https://www.inkandswitch.com/cambria/, [Accessed Mar. 29, 2026].

[156] "Drizzle ORM - Next Gen TypeScript ORM," https://orm.drizzle.team/, [Accessed May 24, 2026].

[157] "Prisma ORM | Type-Safe ORM for Node.js and TypeScript," https://www.prisma.io/orm, [Accessed May 24, 2026].

[158] J. Edwards, T. Petricek, T. van der Storm, and G. Litt, "Schema Evolution in Interactive Programming Systems," *The Art, Science, and Engineering of Programming*, vol. 9, no. 1, p. 2, Oct. 2024, doi: 10.22152/programming-journal.org/2025/9/2.

[159] D. Agrawal, “Sync Engine's Best Friend: Fine-Grained Rendering,” June 2025, https://www.youtube.com/watch?v=YQT26cnCKqo, [Accessed Jan. 31, 2026].

[160] B. O'Brien, “Can Sync Be Network-optional?,” Dec. 2025, https://www.youtube.com/watch?v=sN99A7KWUJ0, [Accessed Jan. 31, 2026].

[161] P.-A. Rault, C.-L. Ignat, and O. Perrin, “Distributed Access Control for Collaborative Applications Using CRDTs,” in *Proceedings of the 9th Workshop on Principles and Practice of Consistency for Distributed Data*, Apr. 2022, pp. 33–38. doi: 10.1145/3517209.3524826.

[162] M. Sidibe, “Policy-CRDT: Conflict-Free Replicated Data Type with Remove-Wins Strategy for Convergent Access Control in Asynchronous Environments,” Dec. 2025, https://www.preprints.org/manuscript/202512.1467, [Accessed Mar. 30, 2026]. doi: 10.20944/preprints202512.1467.v1.

[163] T. Artman, “Building a Synchronous Experience with Asynchronous Data: Linear's Sync Engine,” June 2025, https://www.youtube.com/watch?v=bnOpm3a1fRE, [Accessed Jan. 31, 2026].

[164] A. Nehmzow, “Conflict Resolution x Notion Blocks,” Dec. 2025, https://www.youtube.com/watch?v=AKDcWRkbjYs, [Accessed Jan. 31, 2026].

[165] R. Kistner, “The Current State Of SQLite Persistence On The Web: May 2026 Update,” Nov. 2025, https://powersync.com/blog/sqlite-persistence-on-the-web, [Accessed May 25, 2026].

[166] T. Artman, “Unexpected Benefits of Going Local-First,” June 2024, https://www.youtube.com/watch?v=VLgmjzERT08, [Accessed Jan. 31, 2026].

[167] J. Schickling, “Prisma, Effect and the Rise of Local First Development,” Apr. 2024, https://www.youtube.com/watch?v=DvLq8GguZ9Q, [Accessed Feb. 22, 2026].

[168] J. Schickling, “Local First and TypeScript's Missing Library with Johannes Schickling,” May 2024, https://syntax.fm/767, [Accessed Jan. 31, 2026].

[169] K. Mathews, “Benefits of Using a Sync Engine, Personal Local-First Apps, ElectricSQL,” Mar. 2024, https://www.youtube.com/watch?v=RP7DB4w94do, [Accessed Jan. 31, 2026].

[170] A. Boodman and C. Adams, “Queries,” May 2026, https://zero.rocicorp.dev/docs/queries#query-caching, [Accessed May 21, 2026].

[171] N. Wienert, “Tamagui, One Stack, Zero, and Universal Apps,” Nov. 2024, https://www.youtube.com/watch?v=0R4nA046N30, [Accessed Feb. 22, 2026].

[172] M. Kleppmann, “The Past, Present, and Future of Local-First,” May 2024, https://www.youtube.com/watch?v=NMq0vncHJvU, [Accessed Jan. 31, 2026].

[173] S. Jayakar, “Dropbox, Convex,” Apr. 2025, https://www.youtube.com/watch?v=sUd8eBdwBHU, [Accessed Jan. 31, 2026].

[174] A. Wiggins, “Local-First Software: Pragmatism vs Idealism,” Dec. 2025, https://www.youtube.com/watch?v=VeaVlAvAhck, [Accessed Jan. 31, 2026].

[175] N. Prokopov, “Local First Is Not Going to Win, but That's Okay,” June 2025, https://www.youtube.com/watch?v=ZJW3wAo5nxI, [Accessed Jan. 31, 2026].

[176] T. Browne, “My Biggest Failure to Date,” June 2025, https://www.youtube.com/watch?v=yXhbzFNUeh8, [Accessed Feb. 05, 2026].

[177] D. Thomas and A. Hunt, *The Pragmatic Programmer*, 20th Anniversary Edition. Great Britain: Addison-Wesley, 2019. ISBN 978-0-13-595705-9.

[178] “Zero Schema,” https://zero.rocicorp.dev/docs/schema, [Accessed May 07, 2026].

[179] “Authentication,” https://zero.rocicorp.dev/docs/auth, [Accessed May 07, 2026].

[180] “ZQL,” https://zero.rocicorp.dev/docs/zql, [Accessed May 07, 2026].

[181] “Connecting to Postgres,” https://zero.rocicorp.dev/docs/connecting-to-postgres#render, [Accessed Apr. 20, 2026].

[182] D. Tong, “Feat(Zero-Cache): Schema Change Improvements by Darkgnotic $c\cdot$ Pull Request #5895 $c\cdot$ Rocicorp/Mono”, Apr. 2026, https://github.com/rocicorp/mono/pull/5895, [Accessed Apr. 29, 2026].

[183] A. Boodman, “Zero 1.5,” May 2026, https://zero.rocicorp.dev/docs/release-notes/1.5, [Accessed May 16, 2026].

[184] “PostgreSQL: Release Notes,” https://www.postgresql.org/docs/release/17.0/, [Accessed May 07, 2026].

[185] “29.3. Logical Replication Failover,” Feb. 2026, https://www.postgresql.org/docs/18/logical-replication-failover.html, [Accessed May 07, 2026].

[186] “Zero-Cache Config,” Apr. 2026, https://zero.rocicorp.dev/docs/zero-cache-config, [Accessed May 16, 2026].

[187] “Self-Hosting Zero,” Apr. 2026, https://zero.rocicorp.dev/docs/self-host, [Accessed May 16, 2026].

[188] “Litestream - Streaming SQLite Replication,” 2026, https://litestream.io/, [Accessed May 16, 2026].

[189] “Cloudflare R2,” Apr. 2026, https://developers.cloudflare.com/r2/, [Accessed May 16, 2026].

[190] F. Valsorda, “FiloSottile/Age,” May 2026, https://github.com/FiloSottile/age, [Accessed May 16, 2026].

[191] “Persistent Disks,” https://render.com/docs/disks, [Accessed May 20, 2026].

[192] "Supported Postgres Features," https://zero.rocicorp.dev/docs/postgres-support, [Accessed Apr. 20, 2026].

[193] "Deploy on Cloud Zero," May 2026, https://zero.rocicorp.dev/docs/cloud-zero, [Accessed May 16, 2026].

[194] L. Siffre, T. Ledoux, R. Pawlak, and J. Guery, "Local-First Software for Green IT," in *2025 11th International Conference on ICT for Sustainability (ICT4S)*, June 2025, pp. 58–68. doi: 10.1109/ICT4S68164.2025.00015.

[195] L. Siffre, T. Ledoux, R. Pawlak, and J. Guery, "Local Computing vs. Cloud Computing: An Empirical Study of Energy Consumption," in *Proceedings of the 18th IEEE/ACM International Conference on Utility and Cloud Computing*, Dec. 2026, pp. 1–10. doi: 10.1145/3773274.3774278.

[196] K. Mathews, "Introducing Electric Agents — the Agent Platform Built on Sync | Electric," Apr. 2026, https://electric.ax/blog/2026/04/29/introducing-electric-agents, [Accessed Apr. 30, 2026].

[197] J. McCloy, "The Unreasonable Advantage of Building Local-First," June 2024, https://www.youtube.com/watch?v=PttluvOS5ZI, [Accessed Jan. 31, 2026].

[198] J. Arthur, "Little Elephants Everywhere," July 2024, https://www.youtube.com/watch?v=ZlHWSpIYixk, [Accessed Feb. 22, 2026].

[199] P. van Hardenberg, "Local First: The Secret Master Plan," June 2025, https://www.youtube.com/watch?v=9s8OA08ggbM, [Accessed Jan. 31, 2026].

[200] R. Andersson, "Playbit Sync," June 2025, https://www.youtube.com/watch?v=KMcmMm8v5_E, [Accessed Feb. 22, 2026].

[201] P. Butler, "CRDTs as a Temporal Data Structure," June 2025, https://www.youtube.com/watch?v=b4fDzmNE50U, [Accessed Feb. 22, 2026].

[202] A. Eickhoff, "How Local First Is Accidentally Perfect for the AI Age," June 2025, https://www.youtube.com/watch?v=e3-yIWGNBLQ, [Accessed Feb. 22, 2026].

[203] A. Fish, "Why Local First Matters to Businesses," June 2025, https://www.youtube.com/watch?v=eXrm3A0kl7M, [Accessed Jan. 31, 2026].

[204] J. Martin, "The Last Mile of Local First," June 2025, https://www.youtube.com/watch?v=VCT4J4g2Ung, [Accessed Jan. 31, 2026].

[205] J. Schickling, "Sync Different: Event Sourcing in Local-First Apps," June 2025, https://www.youtube.com/watch?v=nyPl84BopKc, [Accessed Feb. 22, 2026].

[206] Z. Sharipova, "Local First & Social Software in the Era of LLMs," June 2025, https://www.youtube.com/watch?v=lGNZxnUe5z0, [Accessed Jan. 31, 2026].

[207] M. Weidner, “Collaborative Text Editing without CRDTs or OT,” June 2025, https://www.youtube.com/watch?v=5CFrpd0sG-g, [Accessed Feb. 22, 2026].

[208] C. Sverre, “Why Physical Replication Still Matters,” Dec. 2025, https://www.youtube.com/watch?v=QoKzDyH2MEA, [Accessed Jan. 31, 2026].

[209] F. McSherry, “Synchronizing Data Across Computation,” Dec. 2025, https://www.youtube.com/watch?v=X8VSxyKpgPA, [Accessed Feb. 22, 2026].

[210] I. Gozalishvili and C. Joel, “Your Data, Your Rules & the Way to Share Them,” Dec. 2025, https://www.youtube.com/watch?v=4n-2AXhbZPw, [Accessed Feb. 22, 2026].

[211] S. Wang, “Always Be Pair Programming,” Dec. 2025, https://www.youtube.com/watch?v=Tn3HgAKGZOU, [Accessed Feb. 22, 2026].

[212] P. van Hardenberg, “An Intro to Local-First,” Jan. 2024, https://www.youtube.com/watch?v=0bjAI57_cDk, [Accessed Jan. 31, 2026].

[213] A. Boodman, “From Google Gears to Replicache & Reflect.Net,” Jan. 2024, https://www.youtube.com/watch?v=cgTIsTWoNkM, [Accessed Jan. 31, 2026].

[214] R. Andersson, “Playbit, Software Quality, Data Models Tradeoffs,” Mar. 2024, https://www.youtube.com/watch?v=46ROIJ2NtKM, [Accessed Feb. 22, 2026].

[215] J. Long, “Actual Budget, Hybrid Logical Clocks & Absurd-SQL,” Apr. 2024, https://www.youtube.com/watch?v=rEz3shZJtvI, [Accessed Feb. 22, 2026].

[216] D. Raad, “Local-First SaaS,” May 2024, https://www.youtube.com/watch?v=aAdQBDAjjEk, [Accessed Feb. 22, 2026].

[217] M. Wonlaw, “Cr-Sqlite, Syncing Strategies and Incremental View Maintenance,” June 2024, https://www.youtube.com/watch?v=oGGAehNQuG0, [Accessed Feb. 22, 2026].

[218] M. Weidner, “Architectures for Central Server Collaboration,” Sept. 2024, https://www.youtube.com/watch?v=SvTq-vcruYE, [Accessed Feb. 22, 2026].

[219] A. Eickhoff, “Jazz,” Oct. 2024, https://www.youtube.com/watch?v=Ydt3zRGfkdM, [Accessed Jan. 31, 2026].

[220] S. Gentle, “Google Wave, Eg-Walker, Creativity, AI,” Feb. 2025, https://www.youtube.com/watch?v=ynz2C42nDRY, [Accessed Feb. 22, 2026].

[221] P. Butler, “Jamsocket,” Mar. 2025, https://www.youtube.com/watch?v=1o3MvimlmQ8, [Accessed Feb. 22, 2026].

[222] B. Holmes, “Astro, Simple Sync Engine & Warp,” May 2025, https://www.youtube.com/watch?v=nye4mh6OOl8, [Accessed Jan. 31, 2026].

[223] A. Fish, “Ditto, Realm,” June 2025, https://www.youtube.com/watch?v=PAY5A8wFHSk, [Accessed Feb. 22, 2026].

[224] A. Eickhoff, “Jazz Tools,” Dec. 2024, https://www.youtube.com/watch?v=E7UNbIf6oA4, [Accessed Feb. 22, 2026].

[225] G. Costa, “The SQLite Takeover with Turso's Glauber Costa,” Aug. 2024, https://syntax.fm/803, [Accessed Jan. 31, 2026].

[226] S. B. Schmidt, “Prisma ORM: Local First, Typed SQL Queries and Serverless with Søren Bramer Schmidt,” Oct. 2024, https://syntax.fm/839, [Accessed Feb. 22, 2026].

[227] L. Houssier, “Extreme Native Perf on the Web with Superhuman,” July 2025, https://syntax.fm/918, [Accessed Feb. 22, 2026].

[228] T. Linsley, “Fullstack TanStack! The Scoop with Tanner Linsley,” Nov. 2025, https://syntax.fm/954, [Accessed Feb. 22, 2026].

[229] A. Elmore, D. Raad, and C. Enns, “Replicache, Local First, and Sunburns,” June 2023, https://tomorrow.fm/26, [Accessed Feb. 06, 2026].

[230] A. Boodman, “Aaron Boodman on Building Zero, Local-First, and Living in Hawaii,” Mar. 2025, https://tomorrow.fm/126, [Accessed Feb. 06, 2026].

[231] Convex, “Sync Protocols and the Truth Behind Local-First,” Oct. 2024, https://www.youtube.com/watch?v=1vtp52Ytc_w, [Accessed Jan. 31, 2026].

[232] CultRepo, “Local-First Software: Taking Back Control of Our Data | a Mini-Doc,” Jan. 2026, https://www.youtube.com/watch?v=10d8HxS4y_g, [Accessed Jan. 31, 2026].

A. Practitioner Content Extraction Template

This appendix contains the prompt template used for AI-assisted extraction of practitioner content sources. The same prompt was used for all sources in the corpus to ensure consistent extraction. No thesis-specific context (research questions, interview findings, or analytical framework) was provided to the model, so that the extraction captures source content without researcher bias. The framework mapping was performed manually by the researcher in a separate analysis pass.

A.1. Extraction Prompt

The following prompt was used with Google Gemini 3.1 Pro. Each source was provided as a YouTube video link alongside this prompt.

Context: This video discusses software synchronization engines, local-first software, or related client-server data synchronization architectures. Focus extraction on technical architecture, engineering trade-offs, and practitioner experience.

Watch this video and extract the following information. Be comprehensive and cite timestamps for all claims.

1. Speaker background

Who is speaking, what is their role, and what product or system do they represent? If multiple speakers, identify each.

2. Technical claims

What specific technical architecture, design decisions, or implementation details are described? Include: data model, sync protocol, conflict resolution strategy, consistency model, client storage, server infrastructure, and any other architectural details discussed. Cite timestamps.

3. Trade-offs acknowledged

What limitations, challenges, or trade-offs does the speaker discuss? Include both technical trade-offs (performance, consistency, complexity) and practical trade-offs (adoption cost, migration difficulty, team expertise). Cite timestamps.

4. Comparisons

Does the speaker compare their approach to alternatives? What do they say about other engines, protocols, or architectural approaches? Note both explicit comparisons and implicit positioning. Cite timestamps.

5. Opinions and predictions

What subjective positions does the speaker take about the future of sync, local-first, or related topics? Distinguish clearly between factual claims and opinions. Cite timestamps.

6. Notable quotes

Verbatim quotes that are particularly clear, provocative, or well-articulated. Prioritize quotes where the speaker articulates a position concisely or reveals an insight from direct experience. Include timestamps.

A.2. Framework Mapping (Manual Pass)

After extraction, the researcher mapped each extracted point against the following a priori categories derived from the research questions. This mapping was performed manually without AI assistance.

- **RQ1 (Taxonomy dimensions):** authority model, conflict resolution, read/write path, sync unit, integration depth, partial replication, client persistence, network topology
- **RQ2 (Trade-offs and fitness):** where the approach works well, where it breaks, engineering cost, developer experience, application fitness indicators
- **RQ3 (Decision factors and limits):** what drove the architectural choice, adoption barriers, limits of generalization, build-vs-buy rationale
- **Emergent themes:** points that do not fit the above categories but represent recurring patterns across sources

B. Interview Guide

This appendix presents the topic guide used for the 13 semi-structured interviews. The guide defines topic areas and example probes rather than fixed questions, as conversations were adapted to each participant's background and expertise. All interviews followed the same arc but varied in emphasis depending on the participant's group and prior knowledge from the practitioner content analysis. Four interviews were conducted in Finnish, with topics translated by the interviewer during the session. Later interviews included additional probes informed by earlier findings.

B.1. Common Arc (All Groups)

1. **Background and context:** the participant's product, role, and relationship to synchronization.
2. **Technical architecture:** how sync works in their system, or how they designed it for vendor participants.
3. **Pain points and trade-offs:** where things work well and where they break.
4. **Decision factors:** what drove the choice to build, buy, or design a sync engine.
5. **Reflections:** what they would do differently and their perspective on the field.

B.2. Group-Specific Topic Areas

B.2.1. Group A: Sync Engine Vendors

- Customer adoption patterns: which application types succeed, which struggle
- Abstraction limits: what customers request that the engine cannot or will not provide
- Design trade-offs: what was scoped out and why
- Market perspective: why teams still build custom, what would change adoption

B.2.2. Group B: Engine Users

- Adoption decision: what motivated the choice, what alternatives were evaluated
- Reality versus expectations: what was harder than expected, what still requires custom work
- Limits encountered: where the engine could not do what was needed, workarounds used
- Retrospective: whether they would choose the same approach again

B.2.3. Group C: Custom Sync Builders

- Decision to build custom: what requirements led to rejecting off-the-shelf solutions
- Implementation experience: hardest problems, ongoing maintenance burden
- Trade-offs: what was gained by building custom, what was lost
- Market perspective: whether off-the-shelf engines could serve their needs today

C. Practitioner Content Coverage Summary

This appendix summarizes the dimension coverage across the 69 practitioner content sources analyzed in this study. Each source is rated using a three-level scale: Substantive (S), Weak (W), or None (N). A source receives a Substantive rating if it discusses the dimension with enough content to serve as thesis evidence, whether through technical detail, architectural description, comparison, or experience-based discussion. A Weak rating indicates a passing mention without elaboration. The ratings were assigned during the framework mapping pass described in Chapter 3. The dimensions use the analytical framework codes (T1–T8) from the initial coding framework, which differ from the presentation order in Chapter 4.

The analytical framework dimensions are: T1 Authority model, T2 Conflict resolution, T3 Read/write path, T4 Sync unit, T5 Integration depth, T6 Partial replication, T7 Offline capability (originally coded as client persistence, reconceptualized during analysis), T8 Network topology. Where a source appeared in multiple venues with different coverage, the stronger rating is used.

Table 14: Practitioner content source coverage across taxonomy dimensions. S = Substantive, W = Weak, N = None. Left: conference talks (Local-First Conf 2024, Local-First Conf 2025, and Sync Conf 2025). Right: podcasts and other talks (localfirst.fm, other podcasts, Syntax.fm, How About Tomorrow?, and other YouTube).

Src	T1	T2	T3	T4	T5	T6	T7	T8
[172]	S	S	S	W	S	N	W	S
[166]	S	S	S	S	S	S	S	S
[100]	S	W	S	S	S	S	S	S
[197]	W	N	W	N	S	N	N	S
[198]	W	N	S	S	S	W	S	S
[129]	W	N	S	S	S	S	N	W
[199]	S	S	S	S	S	W	W	S
[163]	S	W	S	S	S	S	S	S
[73]	S	W	S	S	S	S	S	S
[159]	N	N	W	S	S	S	N	W
[200]	S	S	S	S	S	W	W	S
[201]	S	S	W	S	S	W	S	W
[202]	S	W	S	S	S	S	W	W
[203]	S	S	S	S	S	W	S	S
[125]	S	S	S	S	S	S	S	S
[204]	W	S	S	W	S	N	W	W
[175]	S	S	S	S	S	W	S	S
[205]	S	S	S	S	S	W	S	S
[206]	S	S	S	S	S	W	S	S
[207]	S	S	S	S	S	W	S	S
[63]	S	S	S	S	S	S	W	S
[174]	W	S	W	W	S	N	W	W
[33]	S	S	S	S	S	S	S	S
[26]	S	S	S	S	S	S	S	S
[75]	S	S	S	S	S	S	S	S
[85]	S	S	S	S	S	S	S	S
[29]	W	W	S	S	S	S	S	W
[164]	S	S	S	S	S	W	W	S
[208]	S	S	S	S	S	S	S	S
[209]	S	W	S	S	S	S	W	S
[160]	S	S	W	S	S	S	W	S
[210]	S	S	S	S	S	S	S	S
[88]	S	S	S	S	S	S	S	S
[211]	W	S	W	N	N	N	N	N

Src	T1	T2	T3	T4	T5	T6	T7	T8
[212]	S	S	S	S	S	S	S	S
[213]	S	S	S	S	S	S	S	S
[148]	S	S	S	S	S	W	S	S
[169]	S	W	S	S	S	S	S	S
[214]	S	S	S	S	S	N	S	S
[215]	S	S	S	S	S	S	S	S
[216]	S	S	S	S	S	S	W	S
[217]	S	S	S	S	S	W	S	S
[95]	S	S	S	S	S	S	S	S
[218]	S	S	S	S	S	W	S	S
[6]	S	S	S	S	S	W	S	S
[219]	S	S	S	S	S	S	W	S
[99]	S	S	S	S	S	S	S	S
[220]	S	S	S	S	W	N	W	S
[221]	S	S	S	S	S	W	S	S
[173]	S	S	S	S	S	S	S	S
[222]	S	S	S	S	S	W	S	S
[62]	S	S	S	S	S	W	S	S
[223]	S	S	S	S	S	S	S	S
[74]	S	S	S	S	S	S	S	S
[167]	S	S	S	S	S	S	S	S
[224]	S	S	S	S	S	S	S	S
[171]	S	W	S	S	S	S	S	S
[132]	S	S	S	S	S	S	S	S
[17]	S	S	S	S	S	W	S	S
[168]	S	W	S	S	S	W	S	S
[225]	S	S	S	S	S	W	S	S
[226]	S	W	S	S	S	S	S	S
[227]	S	S	S	S	S	S	S	S
[151]	S	S	S	S	S	S	S	S
[228]	S	S	S	S	S	S	S	S
[229]	S	S	S	S	S	S	S	S
[230]	S	S	S	S	S	S	S	S
[231]	S	S	S	S	S	W	S	S
[232]	S	S	S	S	S	W	S	S

The following tables summarize the trade-off dimension (F1–F6) and decision factor dimension (D1–D6) coverage using the same rating scale. The trade-off dimensions are: F1 Performance vs consistency, F2 Simplicity vs flexibility, F3 DX vs runtime overhead, F4 Offline vs online-first, F5 Vendor lock-in vs integration, F6 Schema flexibility vs safety.

The decision factor dimensions are: D1 Authorization complexity, D2 Data volume limits, D3 Schema evolution, D4 Operational complexity, D5 Ecosystem maturity, D6 Use case fit.

Table 15: Practitioner content source coverage across trade-off dimensions. S = Substantive, W = Weak, N = None. F1 Performance vs consistency, F2 Simplicity vs flexibility, F3 DX vs overhead, F4 Offline vs online-first, F5 Vendor lock-in, F6 Schema flexibility.

Src	F1	F2	F3	F4	F5	F6
[172]	S	S	W	S	S	N
[166]	S	S	S	S	N	N
[100]	S	S	S	S	N	N
[197]	N	S	S	N	S	N
[198]	N	S	S	N	N	S
[129]	N	S	S	S	N	N
[199]	N	S	S	N	N	S
[163]	S	N	S	S	N	N
[73]	S	S	S	N	N	N
[159]	S	N	S	N	N	N
[200]	S	S	N	N	S	N
[201]	S	N	S	N	N	N
[202]	N	S	S	N	N	W
[203]	S	S	S	S	S	N
[125]	S	S	S	S	W	N
[204]	W	S	S	N	S	W
[175]	N	S	S	W	N	N
[205]	S	S	S	W	N	N
[206]	N	S	N	N	S	S
[207]	S	S	S	N	S	N
[63]	S	S	S	W	S	N
[174]	W	N	N	S	S	N
[33]	W	S	S	W	W	W
[26]	S	S	S	W	W	N
[75]	S	S	N	S	N	S
[85]	S	S	N	S	N	N
[29]	S	S	S	S	N	N
[164]	S	S	N	S	N	S
[208]	S	S	N	S	S	N
[209]	S	S	S	N	S	N
[160]	S	S	N	S	S	N
[210]	N	S	S	N	N	S
[88]	S	S	S	S	S	N
[211]	N	S	N	N	N	N

Src	F1	F2	F3	F4	F5	F6
[212]	S	S	W	S	W	S
[213]	S	S	S	S	W	W
[148]	S	S	S	W	S	S
[169]	W	S	S	W	S	W
[214]	S	S	S	W	W	S
[215]	S	S	S	S	W	W
[216]	S	S	S	S	W	W
[217]	S	S	S	W	W	S
[95]	W	S	S	S	S	S
[218]	S	S	W	S	S	S
[6]	S	S	S	S	S	N
[219]	W	S	S	W	S	W
[99]	S	S	S	W	S	N
[220]	S	S	W	S	N	S
[221]	S	S	S	S	W	N
[173]	S	S	W	S	N	S
[222]	W	S	S	S	N	N
[62]	N	S	S	W	N	S
[223]	S	S	S	S	S	N
[74]	S	S	S	S	S	W
[167]	N	S	S	S	N	N
[224]	N	S	S	S	S	S
[171]	S	S	S	S	S	N
[132]	S	S	S	S	S	S
[17]	S	W	W	S	W	S
[168]	S	S	S	S	W	W
[225]	S	S	S	S	S	W
[226]	S	S	S	S	S	S
[227]	S	S	S	S	W	W
[151]	S	S	S	S	S	W
[228]	S	S	S	W	S	S
[229]	S	S	S	S	S	W
[230]	S	S	S	S	S	W
[231]	S	S	S	S	S	W
[232]	S	S	S	S	S	W

Table 16: Practitioner content source coverage across decision factor dimensions. S = Substantive, W = Weak, N = None. D1 Authorization complexity, D2 Data volume limits, D3 Schema evolution, D4 Operational complexity, D5 Ecosystem maturity, D6 Use case fit.

Src	D1	D2	D3	D4	D5	D6
[172]	N	N	N	S	S	S
[166]	S	S	N	S	N	S
[100]	N	S	N	N	S	S
[197]	N	N	N	N	S	S
[198]	N	N	N	N	S	S
[129]	N	N	N	N	N	S
[199]	S	N	S	N	S	N
[163]	N	S	N	N	N	S
[73]	S	S	N	N	N	S
[159]	N	N	N	N	W	W
[200]	N	N	N	S	N	S
[201]	N	S	N	N	S	N
[202]	S	N	N	N	W	S
[203]	N	N	N	S	S	S
[125]	W	S	N	S	S	S
[204]	N	N	N	N	W	S
[175]	N	N	N	N	W	S
[205]	N	N	W	N	W	S
[206]	S	N	N	W	S	S
[207]	N	N	N	N	S	S
[63]	S	W	W	S	S	S
[174]	S	N	N	N	S	S
[33]	S	W	W	W	W	S
[26]	S	N	N	S	S	S
[75]	N	S	N	N	S	S
[85]	N	S	N	S	S	S
[29]	N	S	N	S	S	S
[164]	N	N	S	N	N	S
[208]	S	S	N	N	N	S
[209]	N	S	N	S	N	S
[160]	S	N	N	S	N	S
[210]	N	N	N	N	N	S
[88]	S	S	N	S	S	S
[211]	N	N	N	N	N	N

Src	D1	D2	D3	D4	D5	D6
[212]	S	S	S	S	S	S
[213]	W	S	S	W	S	S
[148]	W	S	W	S	S	S
[169]	N	W	W	S	S	S
[214]	N	N	W	N	S	S
[215]	N	S	W	S	S	S
[216]	S	S	S	S	W	S
[217]	W	W	S	W	S	S
[95]	S	S	S	W	S	S
[218]	S	S	W	W	S	S
[6]	N	S	N	S	S	S
[219]	S	W	W	W	S	S
[99]	N	S	W	S	S	S
[220]	N	S	N	N	S	S
[221]	N	S	S	S	N	S
[173]	S	S	S	S	S	S
[222]	N	N	N	W	N	S
[62]	N	N	N	N	S	S
[223]	N	S	N	S	S	S
[74]	W	S	S	S	S	S
[167]	N	N	N	N	S	S
[224]	S	S	S	S	S	S
[171]	N	S	N	S	S	S
[132]	S	S	S	S	S	S
[17]	S	S	W	W	W	S
[168]	S	W	W	S	S	S
[225]	W	S	W	S	W	S
[226]	S	S	S	S	W	S
[227]	W	S	W	S	S	S
[151]	W	W	S	S	S	S
[228]	W	W	W	W	S	S
[229]	W	S	W	S	S	S
[230]	S	S	W	S	S	S
[231]	S	S	W	S	W	S
[232]	S	W	W	S	S	S