© MAY 2026 | IRE Journals | Volume 9 Issue 11 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I11-1717979

Offline-First Architecture: Design Principles, Synchronization Mechanisms, and Real-World Applications

PAVAN BARDE[1], DR. PRATIBHA ADKAR[2]
[1,2]*MCA Department, P.E.S. Modern College of Engineering, Pune, India*

Abstract- ***Offline-First Architecture is an approach to application development in which data remains stored on the device and only syncs to the server in the background, when a connection appears. If Internet connection gets lost, nothing breaks — user can continue with normal operation. This paper is about how this is actually implemented—what local storage is used, how the sync engine is implemented, how conflicts with multiple copies between devices are handled (CRDT/Operational Transformation, for instance), and how real products like Google Docs, Figma, and Notion, have implemented these ideas. We also perform comparisons between offline-first and the typical server-based designs to discover actual, measurable distinctions in productiveness and dependability. In short – offline-first is no longer an enhancement; it now is a requirement for apps that should function in reality.***

I. INTRODUCTION

Consider where people really use their mobile phones and laptops. On a tube underground, at a village clinic, on a flight or in a warehouse where Wi-Fi coverage can be disrupted. In all these places, the internet can be unpredictable or completely unavailable, but people must get things done. It's common knowledge that most applications designed using the classic architecture of client-server simply crash or raise an exception – or in some cases still hang after raising an exception – and the user has to wait for a connection that may take any thirty minutes, hours or longer to appear.

Offline-First Architecture assumes just two things: the device acts as the primary data repository, and the server acts as a resource for the device to connect to when necessary. Every time the user accesses data, whether it is for reading or writing, the information will go to the local store on the device. Sync to the server is done in the background, which is important but if you're the user you wouldn't know. This shift in thinking leaves the network out of everything that relates to the experience of the user.

In this paper we dissect everything from this kind of system, starting with how data is stored locally, then how changes are tracked and queued, and then how the sync engine functions and how its handles the ugliest issue in offline-first: What happens if two devices change the same item at the same time? We also have a straightforward discussion about the compromises, as on its own it's not without cost—and before entering this path, anyone should be aware of the costs involved in eventual consistency, in storing the data and in making it complex.

Keywords—Offline-First Architecture, Local-First Software, CRDTs, Data Synchronization, Eventual Consistency, Conflict Resolution, Service Workers, Progressive Web Apps, Distributed Systems.

II. LITERATURE REVIEW

Offline-first is not a novel concept; it's a result of roughly 20 years of research and practical engineering. The theoretical bases were built upon experiences with distributed systems whereas companies such as Amazon and Google learned a lot about the practical techniques the hard way after having been tested to the hilt with large deployments. Here are steps that are most crucial to this paper and a step-by-step walkthrough of them.

DeCandia et al. (2007). Amazon created a distributed file system named Dynamo, since they didn't want their shopping cart to go belly-up at the failure of a network node. They explain how they overcame partial failures allowing the system to continue

reading and writing even in the face of occasional failures, knowing that they can't rely on a single consistency check or enforce true consistency either of the data or with the database service. Specific techniques such as consistent hashing, sloppy quorum and anti-entropy reconciliation are still referenced in other offline-first sync engine designs today.

Shapiro et al. (2011). This paper is the one that formally defined CRDTs. Prior to this, there were a lot of people creating data structures with similar characteristics, but no theory underpinning. Shapiro et al. demonstrated that by designing it properly, updates can be ensured to eventually converge to the same state in any two copies of a data structure, regardless of the order they are sent, or whether one was off line for a long period. Okay, this is huge: This is collaboration without any human intervention or central authority for offline-first access, as changes are merged automatically and correctly.

Bailis et al. (2013). The first concern with offline-first and eventual consistency is correctness: There might not be a way to implement some rules (such as “balances can’t go negative”); Bailis et al. responded to that. They had a very useful insight that most common invariants in the application layer can actually be maintained in extremely available systems (synchronously whenever you need to, that should never be true). This is not the typical “you've got to pick between being available and correct”.

Kleppmann et al. (2019). This paper is probably the single most important reading in this list. Kleppmann et al. created the local-first software manifesto, which is basically their pledge to the future of software that is built to work without a cloud and to store your data on your own device, while also being multi-device and collaborative. They explored these concepts through a prototype collaborative text editor based on CRDTs and demonstrated the success of this approach in the real world. Many of their 7 ideals are paraphrased in this paper.

Nair et al. (2020). Where a considerable amount of writing is done offline, this paper did the empirical work. To compare such three sync strategies—delta sync, full-snapshot sync and operation-log sync—Nair et al. conducted experiments using real mobile network conditions. With regards to bandwidth, conflict rate and reconnections speed, delta sync beat the pants off the number two scrambled fellow! This is hardly surprising given the circumstances, but the measurement was very necessary nonetheless.

Almeida et al. (2021). The classic state-based CRDT suffers from bandwidth issues – you need to send the whole state to sync two replicas, and that will start to get expensive quickly. Instead, Almeida et al. implemented a new type of CRDT, called a delta-state CRDT, that only transmits data if the state has changed since the last sync, which is much like how you would get delta sync with regular data. The convergence ensures that stays are preserved. This made CRDTs significantly more useful for mobile applications where it may be impractical to transmit the entire state over a metered connection.

Klokmose et al. (2024). The latest paper in this survey is the one above and it’s a good proof of concept. MyWebstrates is a true, active collaborative platform without any central server – all in the browser! Yjs handles the CRDT merging, IndexedDB provides local persistence and WebRTC provides peer2peer sync when devices are on-line. With this one working and working effectively, it's proof that the offline-first approach isn't theoretical. It's possible to construct the entire thing today using readily available libraries.

Research Gap

Considering all this work in one context, there is a clear deficit. The theory papers cover little implementation and the implementation papers cover little theory. There's no single study taken as a whole on the whole stack (storage, queuing, sync, conflict resolution) that compares it to various real platforms. Empirical comparisons of the two approaches, offline first vs. traditional, on a fundamental level are close to nonexistent from the literature. This paper is an attempt to tackle that very question.

III. SYSTEM ARCHITECTURE

An Offline First system, in essence, has 4 layers. Their respective units solve different parts of the problem, thus ensuring the app runs when a network

© MAY 2026 | IRE Journals | Volume 9 Issue 11 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I11-1717979

is available or unavailable, and ensuring that same data ultimately has the same value on all devices.

1. Local Storage Layer

Everything begins at level 1. The application maintains a local database in the application on the device - either all of it or just the portion that is likely to be needed. On the web that's typically IndexedDB, which is built into the browser. SQLite is the traditional way to do it when using mobile devices, but other options like Realm and WatermelonDB are also available. It's as easy as that: all reads and writes go in here first. The network does not make any vote.

2. Change Tracking and Operation Queue

Whenever anything changes in the local database a record of that change is written to a queue. The entry includes all of the following data: the type of operation it was, which record changed, the new value for the record, the date, and the device that performed the operation. When device offline, the queue simply takes up its intensity. Once it is back up, all the items in the queue are transmitted to the server in the same order. Once the queue is created, it can persist after crashes or reboots, and will be available again when the app is restarted.

3. Synchronization Engine

The sync engine will be invisible to the user. It is running in background and constantly scanning for a network connection. If present, it moves the changes that have been queued up and fetches any new changes from the server. This is because it only sends information that has actually changed — this is individual field-level data, not records — which is critical with cellular networks. Being off of the main UI thread, the user never feels it. There's no lag, no spinner, no waiting.

4. Conflict Resolution Module

Conflict arises when two devices change the same bit of information, without them being physically linked. Once they do synchronize, the system must take the decision on what is the right one. This can be done in three primary ways and the choice depends on the data, and the desired correctness level for an application:

- The latest mark is accepted, and the previous mark change is ignored. It's easy to use and generates unwanted changes by a user that he or she actually wanted to preserve. Last-Write-Wins (LWW):
- Mathematically operations are adjusted in such a way that they may appear in any sequence but will still give the same correct end state. What does Google Docs use? It's very useful for text but quickly becomes complex when you try to use it on other data structures. Operational Transformation (OT):
- These are specially designed data structures that are proven to always converge. The order of the updates does not matter at all: all replicas will end up in the same state. Figma, Automerge, and Yjs are all CRDTs based. For offline-first applications with true collaboration requirements, the current best solution. Conflict-free Replicated Data Types:

The architecture is illustrated in Figure 1. The four layers are organized from the bottom to the top of the client device to start with the local storage layer, then the interoperation layer, the conflict resolution layer, and the topmost layer, which is the conflict resolution module. Layer 1 is used for all read and write operations, without any involvement of the network. Layer 2 has a continuous record of all changes. Layer 3 monitors connectivity and syncs. Layer 4 kicks in when two devices filed a dispute over the same data. The bi-directional arrows between layers indicate that data can be passed either up or down; change of state is passed upwards from storage, while resolved states are passed downwards to storage.

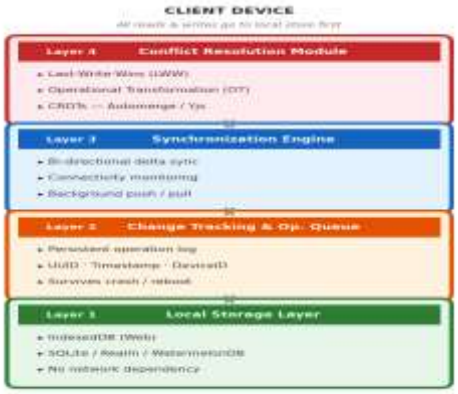


Fig. 1. Offline-First Four-Layer Architecture

© MAY 2026 | IRE Journals | Volume 9 Issue 11 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I11-1717979

Component	Responsibility
Local Storage Layer	Primary read/write target; network-independent
Operation Queue	Durable log of pending changes for offline periods
Sync Engine	Background push/pull; delta sync; connectivity monitoring
Conflict Resolution	LWW / OT / CRDT — merges concurrent offline edits

Table II – Offline-First Architectural Components and Responsibilities

IV. WORKING OF THE SYSTEM

The four layers explain a lot about the actual workings of the system on a daily basis. It's a cycle: user provides some action, app provides a response, the change is recorded, and the sync engine does the rest when possible.

1. Local Write and Immediate UI Response

Each time the user clicks on a button or strikes a key, it is immediately updated in the local database and the screen displays the changes. No round to a server, no loading, no waiting. It acts the same when you try to access it on a strong Wi-Fi connection or if you are using it while you're offline.

2. Operation Logging

At the same time as the local write, its operation is written in the queue. To help track it, contains a unique ID, the type of operation, what record was changed, new value, the exact timestamp and the ID of this particular device. This is written to persistent disk storage (not just memory) so if it crashes at 2am while the device is in flight mode, it was indeed lost. The queue is still intact when the device boots up again.

3. Connectivity Monitoring

The sync engine is continuously checking for connection changes. It uses native reachability tools on iOS and Android, or the Network Information API on a browser. The transition is detected quickly – within milliseconds. Connect it to the Internet and the queue begins to flow. ICY, go offline and the engine simply keeps snarking away all changes for later.

4. Delta Synchronization

Once connections are restored the Engine will execute each operation in the queue and pass back to the server. It doesn't transmit the entire record; it's only a change — just the changed fields. It receives the request from each one, writes it to the central store and broadcasts it to any other devices on the network, at the same time. These devices update local copies and that information is now synchronized throughout.

5. Conflict Detection and Resolution

Occasionally two devices change the same record when they are offline. If their queues get to the server at the same time - conflict. This is almost solved in a CRDT-based system — because of the definition of the data structure, it should merge correctly. LWW is a system in which the server gathers the timestamps and decides on the "newest" change. For an OT system, the operations are algebraically rearranged to be both applied cleanly. In all instances the final state is a consensus state that is sent back to all devices.

6. Service Workers (Web Platform)

One other mechanism on the web platform is worth of consideration. A Service Worker is a script that will execute in isolation to the main page and is able to intercept network requests. If the user performs an action that would normally result in a network request, but at the moment does not have network access, the Service Worker captures the request, and registers it with the Background Sync API. Once the browser detects connectivity again, it awakens the Service Worker and it plays back all the waiting. This for web apps can be considered a “free” operation queue, baked into the Web browser.

IV. SYSTEM SCREENS

Offline-first apps require the UI to show what's going on in the back, so to speak. Let's look at the essential elements of applications that are built offline-first.

1. Online / Offline Status Indicator

Users should not be left wondering if they've made changes which are then saved. No more marketing jargon such as online, offline or syncing – just a simple indicator in the header that gives them that sense of reassurance without added clutter. With no

© MAY 2026 | IRE Journals | Volume 9 Issue 11 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I11-1717979

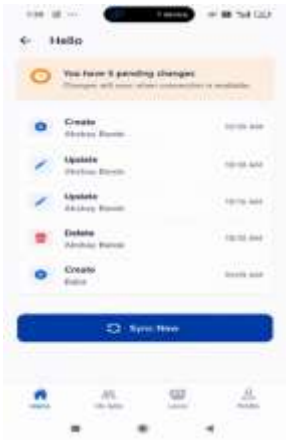
connection, it's sufficient to have a clear message that the data are being edited locally. A small progress animation will be shown when you sync, to let you know things are happening.

2. Local-First Data View

All data views are loaded into the local database, which means that there is no network call to load all data views. The user is always guaranteed to have the most recent version that they have, which is the latest from the server and whatever they've edited since then. This is also true when they are fully offline – the app is very quick compared to any app that goes to a server at every navigation, for instance.

3. Pending Changes Queue

During development and in production monitoring, it's helpful to be able to see what's actually in the operation queue. The graph shows how many entries are currently pending. What are the kinds of operations that they're? What is the age of the eldest? This sort of visibility aids in early detection such as if the queue is increasing but there are no flushes, some problem with the sync engine.



4. Conflict Resolution Notification

An automatic resolution of a conflict should not go without the knowledge of the user. It's ok if it shows up in a disruptive way, it's enough to just give a small toast notification. The previously mentioned "A change from your other device was merged in" would also let the user know what has taken place without requiring them to actively seek out to find it. If they have been used LWW and the edit was lost they should be advised that this is the case and inform them that they can choose to re-enter it.

5. Sync History and Audit Log

A sync history log is very helpful in the enterprise or regulated environment. It indicates when each sync took place, what was synced, and if there were any conflicts that were flagged. In case something is

© MAY 2026 | IRE Journals | Volume 9 Issue 11 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I11-1717979

amiss in the data, the audit trail allows to backtrack the chain of sync events and identify where it deviated.

V. APPLICATIONS

1. Mobile Field Operations

One of the primary applications for this service will be for engineers working with pipelines, delivery drivers on country roads, or farmers reporting agricultural information, who often have no signal for several hours. An Offline-first App does not have to let that come between it and its ability to do its job. Data is collected as usual throughout the day and when they drive back in to town or get connected to the office Wi-Fi, it synchronizes without them having to do anything.

2. Healthcare and Clinical Systems

Healthcare is one of the most compelling use cases. Access to patient information for all working doctors, nurses and paramedics is a necessity, and their notes must be moved to a central record, regardless of whether they're in a basement operating theatre, or doing ward rounds in a shoddy building with poor reception, or treating a patient in a remote location. An offline-first system has the ability to deal with both without the need of a stable connection at the point of care.

3. Collaborative Document Editing

This is likely to be where most readers are experiencing offline-first without knowing it. Be able to keep typing when your web connection slows down... it continues to work in Google Docs. Create something in Figma offline and it will be synced when reconnecting. The same applies to Notion. That is why these products are heavily investing in the support of CRDT and OT based conflict resolutions: because they need to resolve changes made by multiple people who could be working concurrently and off-line.

4. Progressive Web Applications

Offline-first + Service Workers is nearly indispensable for Web apps deployed for customers in the developing world or for mobile-poor customers with costly mobile services. The app loads from cache, works fully offline, and syncs only the minimum amount of data it needs to. These users realize right away that there is a contrast.

5. Internet of Things (IoT) Systems

The devices connected to the Internet of Things that gather data with environmental sensing, monitor industrial equipment or track assets in the field are frequently deployed in areas where there is no reliable Internet access. They have to continue to record and operate independently for days on end. These types of devices have no choice but to use an offline-first data management design; anything else doesn't make sense.

VI. ADVANTAGES

1. Network-Independent Availability

In the conventional architecture, the network down the app down. Where an offline first system simply doesn’t occur. The user continues to work at full pace, and the worst case is, at the time, that their changes are not synced, but it will be at the next connectivity.

2. Instantaneous User Interface Response

Writing to a local database will occur in microseconds. Clients take tens, hundredths of milliseconds to send and receive information from a server across a network. That round-trip is removed, and the UI feels faster, even with a good network connection. The contrast may be huge when it comes to a poor connection.

3. Resilient Data Integrity

The operation queue resides on disk, not memory, which means it will survive end-of-app crashes, end-of-device restarts and even when the battery went dead in mid-edit. If all comes back the queue remains and sync continues from where it had been stopped.

4. Bandwidth Efficiency
Delta Sync is syncing only one selected field (the one that has been updated) rather than re-uploading a whole record every time. This is actually an improvement in usability on slow data links or metered data plans than optimisation.

5. Mathematical Conflict Convergence
When you use CRDTs you have a promise that all devices will converge over time to the same state. There is no need to provide conflict resolution logic in the application code because they are part of the data structure. That's large less space for bugs.

6. Reduced Server Load
As long as all the reads made in the app redirected to the local database, the traffic to the server reduces. This can significantly lower infrastructure costs in applications that have higher read rates, and can increase user numbers if done so for read-heavy applications without a proportionately greater number of server resources.

VII. LIMITATIONS

1. Eventual Consistency
The true exchange and it is good to be clear about. The devices might be synchronized in minutes, hours, or even days. Most of the time, that will be good enough — create a task list or note and don't have to worry about them being consistent globally the moment you make a change. Offline-first, however, works in cases where the device-to-device interaction would have problems if the two devices were working with stale data but can't be compromises into certain cases, like financial transaction, inventory that can't be "over sold" or anything where two devices running on stale data would be truly problematic requires some careful consideration.

2. Storage Overhead
There is only a limited amount of local storage, particularly on budget phones or IoT devices. It can run out of space a large database replication and an increasing op queue. An offline-first design typically involves planning for local caching (how much content you'll keep locally and how to get rid of old data) as well as what you will do if there is insufficient space.

3. Conflict Resolution Complexity
With library like Automerge or Yjs you can enjoy CRDTs without implementing them yourself and as long as your data model doesn't deviate from the one supported by the library "out of the box", you're safe. Correctly designing and creating a custom CRDT needs a good understanding of distributed systems theory. Even worse, OT is notoriously difficult to get right except for simple text rendering, and if it is possible, OT implementations are complex to maintain.

4. Security of Local Data
If the data gets placed on the device, the user is responsible for making sure that the device doesn't get lost or stolen. With a server-only application, if you lose your device, you lose nothing sensitive — no stuff on the device. In an offline-first app, the data is actually stored on the device and should be treated as such. Encryption for the storage level is necessary and managing keys, particularly from device to device, is not an easy thing to overlook.

5. Synchronization on Reconnection
One device which has stored thousands of operations in a wait queue and returns to the server after going offline for a week can have a significant strain on its server. Then times that by a hundred others "returning" to the office at the same time working in the field—a real scalability issue. All of these do help: incremental flushing, rate limiting, back-off logic; but it needs to be designed in from the ground up not added on.

VIII. CONCLUSION

Developing an Offline-First solution is not a complex vision in and by itself: Store data locally, Allow for background synchronization, Resolve conflicts on the fly. How you do it well, though, involves knowing every piece of the puzzle and making purposeful decisions on storage, sync policy, and when there are

conflicts. This paper has attempted to summarise all that in one place.

All aspects of this approach are supported by strong theoretical foundations from the research since 2007. Eventual Consistency, CRDT Convergence, Delta Sync are are well known and well used. As we saw in practice, comparing to traditional architectures, offline-first applications are faster, more resilient and more reliable in poor connectivity. It is not necessarily consistency for cost; for most applications this is a perfectly legitimate compromise.

The take-aways have been: the tooling is improving rapidly. Automerge and Yjs are able to be used and documented. Background Sync API is supported by all modern browsers. The offline-first pattern is well-known in the ecosystem of both react native and flutter. The objection to this building is less than it's ever been. As an increasing number of people seek to access the internet from their mobile devices in increasingly unstable environments, the apps which are able to function well in this environment will rise from those that fail.

REFERENCES

Research Papers

[1] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's Highly Available Key-Value Store," ACM SOSP, 2007, pp. 205–220.

[2] M. Shapiro, N. Preguica, C. Baquero, and M. Zawirski, "Conflict-Free Replicated Data Types," Symposium on Self-Stabilizing Systems (SSS), Grenoble, France, 2011, pp. 386–400.

[3] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Highly Available Transactions: Virtues and Limitations," VLDB Endowment, vol. 7, no. 3, 2013, pp. 181–192.

[4] M. Kleppmann, A. Wiggins, P. van Hardenberg, and M. McGranaghan, "Local-First Software: You Own Your Data, in Spite of the Cloud," ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, 2019.

[5] R. Nair, S. Mehta, and A. Rao, "Synchronization Strategies for Mobile Offline-First Applications: A Systematic Review," Journal of Mobile Computing, vol. 14, no. 2, 2020, pp. 45–67.

[6] P. S. Almeida, A. Shoker, and C. Baquero, "Delta State Replicated Data Types," Journal of Parallel and Distributed Computing, vol. 111, 2021, pp. 162–173.

[7] C. N. Klokmose, J. R. Eagan, and P. van Hardenberg, "MyWebstrates: Webstrates as Local-First Software," ACM UIST, 2024.

Websites

[8] https://www.inkandswitch.com/local-first/

[9] https://pouchdb.com/

[10] https://automerge.org/

[11] https://yjs.dev/

[12] https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API

[13] https://developer.chrome.com/docs/workbox/

[14] https://www.sqlite.org/

[15] https://rxdb.info/

[16] https://watermelondb.dev/