Aalto University
School of Science
Bachelor's Programme in Science and Technology

Local-First Software: Promises and Pitfalls

Bachelor's Thesis

May 9, 2025

Otto Otsamo

Aalto University
School of Science
Bachelor's Programme in Science and Technology

ABSTRACT OF
BACHELOR'S THESIS

Author:	Otto Otsamo
Title of thesis:	Local-First Software: Promises and Pitfalls
Date:	May 9, 2025
Pages:	27
Major: **Code:**	Computer Science SCI3027
Supervisor: **Instructor:**	Professor Lauri Savioja Doctoral Researcher Juho Vepsäläinen (Department of Computer Science Engineering)

Software applications with collaboration features are typically built to store and process data on remote servers, making the client application completely dependent on those servers. This architectural pattern is simple to implement but has drawbacks: it is not possible to use the application without an internet connection, it is not always clear who owns the user's data, communication with the server may cause delays in actions taken by the user, and the application stops working if the service shuts down, potentially resulting in data loss.

Local-first software refers to a software design pattern that prioritises storing and processing data locally on the user's device instead of remote servers. In this approach, servers are typically only utilised as a synchronisation tool to enable collaboration between users.

This thesis reviews the benefits and drawbacks of each approach based on existing research and describes the common technologies typically used to build local-first software. The study demonstrates that the local-first pattern can be beneficial in applications involving asynchronous collaboration, but is unable to replace certain types of cloud-based software, such as those handling financial transactions or large datasets.

Currently, the development of local-first applications remains relatively limited; however, there is clear interest in the topic within both academic and professional communities. To advance local-first development, it would be beneficial to research how existing synchronous data structures could be adapted to support asynchronous collaboration and how the memory overhead and other limitations of local-first data structures can be mitigated.

Keywords:	cloud software, local-first software, software architecture
Language:	English

Aalto-yliopisto	KANDIDAATINTYÖN
Perustieteiden korkeakoulu	TIIVISTELMÄ
Teknistieteellinen kandidaattiohjelma	

Tekijä:	Otto Otsamo
Työn nimi:	Paikallislähtöisten ohjelmistojen hyödyt ja haasteet
Päiväys:	9. toukokuuta 2025
Sivumäärä:	27
Pääaine:	Tietotekniikka
Koodi:	SCI3027
Vastuuopettaja:	Professori Lauri Savioja
Työn ohjaaja:	Tohtorikoulutettava Juho Vepsäläinen (Tietotekniikan laitos)

Yhteistyöominaisuuksia tarjoavien sovellusten toiminta perustuu tyypillisesti tietojen tallentamiseen ja käsittelyyn etäpalvelimilla, jolloin asiakassovellus on täysin riippuvainen näistä palvelimista. Tämä pilviohjelmistojen (engl. cloud software) arkkitehtuurimalli on suosittu ja helposti toteutettavissa, mutta siihen liittyy myös haittapuolia: sovelluksen käyttäminen ei ole mahdollista ilman internet-yhteyttä; käyttäjän luomien tietojen omistajuus voi olla epäselvää; verkkoviiveet voivat aiheuttaa viiveitä käyttöliittymässä; ja sovellus lakkaa toimimasta palveluntarjoajan lopettaessa sen ylläpidon, mikä saattaa johtaa tietojen menetykseen.

Paikallislähtöiset sovellukset (engl. local-first software) pyrkivät ratkaisemaan näitä ongelmia tallentamalla ja käsittelemällä käyttäjän tietoja vain käyttäjän päätelaitteella. Tämän lisäksi voidaan hyödyntää etäpalvelimia, mutta tyypillisesti vain käyttäjien väliseen tietojen synkronointiin yhteistyön mahdollistamiseksi.

Tässä kandidaatintyössä tarkastellaan kunkin arkkitehtuurimallin hyötyjä ja haittoja aiemman tutkimuksen pohjalta sekä kuvataan paikallislähtöisten sovellusten kehityksessä tyypillisesti käytettyjä teknologioita. Työ osoittaa, että paikallislähtöisyys soveltuu etenkin asynkronisen yhteistyön sovelluksiin, mutta ei pysty täysin korvaamaan tietynlaisia pilviohjelmistoja, kuten rahaliikennettä tai suuria tietoaineistoja käsitteleviä sovelluksia. Paikallislähtöisiä sovelluksia kehitetään toistaiseksi melko vähän, mutta aiheeseen kohdistuu selkeää kiinnostusta sekä akateemisessa että ammatillisessa yhteisössä. On todennäköistä, että tämäntyyppinen ohjelmistokehitys yleistyy lähitulevaisuudessa. Siirtymän edistämiseksi olisi hyödyllistä tutkia lisää sitä, miten olemassaolevia tietorakenteita voidaan mukauttaa asynkroniseen yhteistyöhön sopivaksi ja miten niiden muistikuormaa ja muita rajoitteita voidaan pienentää.

Avainsanat:	ohjelmistoarkkitehtuuri, paikallislähtöiset ohjelmistot, pilviohjelmistot
Kieli:	Englanti

Contents

1 Introduction **6**

1.1 Why research local-first software? 6

1.2 Research question 7

1.3 Structure of the thesis 7

2 Design Patterns for Collaborative Software **8**

2.1 Collaborative software 8

2.2 Single-user applications 8

2.3 Cloud applications 9

2.4 Local-first software 10

2.5 Local-first limitations and drawbacks 11

2.6 Summary 12

3 Local-First Technologies **14**

3.1 Conventional data synchronisation methods 14

3.2 Conflict-free replicated data types 14

3.3 Operation-based CRDTs 15

3.4 State-based CRDTs 16

3.5 Delta state-based CRDTs 17

3.6 Concurrency semantics 17

3.7 Tools for local-first development 18

3.8 Examples of local-first applications 19

3.9 Summary 20

4 Discussion **21**

4.1 Feasibility of the local-first approach 21

4.2 Adoption of local-first technologies 21

4.3 Future of local-first software 22

5 Conclusion **23**

5.1 Main findings 23

5.2 Open questions 23

1 Introduction

In recent years, work, education and even recreational activities have increasingly moved online, making virtual collaboration an integral part of our everyday life [1]. Typically, software applications that offer collaboration features are built to store and process data entirely on remote servers, following a centralised, cloud-based architectural pattern [2]. This cloud-based approach is popular due to its simplicity from the implementation and maintenance perspective, but it has some drawbacks that *local-first software*, a design pattern proposed in recent research, aims to resolve [3].

1.1 Why research local-first software?

Traditional cloud-based collaboration applications are completely dependent on remote servers being available [4]. These applications often employ a *software as a service* (SaaS) model, where both the application itself and the data it produces are hosted online. In practice, this means that it is not possible to use the application without an internet connection; ownership of the user's data can be unclear; communication with the server may cause delays in actions taken by the user; and the application stops working if the service shuts down, potentially resulting in permanent data loss [2]. In contrast, the local-first software design pattern prioritises storage and processing of data locally on the user's device. In this approach, servers are typically only utilised as a synchronisation tool to enable collaboration between users, which addresses many of the limitations associated with cloud software [2].

The local-first pattern was first proposed in 2019 in a research article by Kleppmann et al. [2]. Since the initial proposal, further research has been published, for example on consistent and safe data types for use in local-first applications [5] and programming models for handling the constraints involved with such data types [6]. Additionally, some general-purpose solutions for building local-first software are in development, such as the Automerge[1] library [7]. However, when compared to tools used to build cloud software, these solutions are still quite immature and lack important features [8]. Further research is needed to develop state synchronisation libraries that support a wide variety of data types, as well as meta-functionalities such as operation inversion (undo) and end-to-end encryption.

[1] `https://automerge.org`

1.2 Research question

This thesis aims to answer the following research question: **What are the potential benefits and drawbacks of using a local-first approach for developing collaborative applications?** These aspects are evaluated and compared with other common design patterns based on existing research on the topic.

1.3 Structure of the thesis

The remainder of this thesis is divided as follows: Section 2 of this thesis describes the evolution of these design patterns and their characteristics. Section 3 reviews the core technologies utilised to implement local-first software. Section 4 summarises and discusses the identified benefits and drawbacks of local-first software, and finally, Section 5 concludes the thesis and suggests focus areas for future research.

2 Design Patterns for Collaborative Software

This section describes the evolution of software architecture design for applications that offer collaboration features. It reviews the benefits of each approach over earlier technologies as well as aspects that were compromised.

2.1 Collaborative software

Collaborative software, also known as groupware [9], refers to applications designed to enable multiple users to work together on shared tasks, projects or documents. These applications can be categorised along two axes: time (synchronous or asynchronous collaboration) and location (co-located or remote collaboration) [10]. The main focus of this thesis is on applications that offer asynchronous collaboration, remote collaboration, or a combination of both.

2.2 Single-user applications

Early on, single-user applications typically stored and processed data on the user's own device [2]. With this kind of software, the user is always in control of the data, including backups, sharing, archival and deletion [2]. If necessary, files can be shared with other users using physical portable media or over the network, but the software itself has no dependency on any remote servers [2]. Data ownership is also clear: the provider of the software has no access to or responsibility over the data created by the user, and the user is free to choose other compatible software for viewing and modifying it [2].

As the internet became commonly accessible to the public in the 1990s, solutions started to emerge for enabling multi-user collaboration within existing single-user applications. These systems provided multi-user access to the application without requiring changes to the source code of the application or any special consideration from the users, a concept known as *collaboration transparency* [9]. Some of the first systems implementing this concept, like XTV and SharedX, were built for the X Window System [9]. These systems merged inputs from users, replicated them on a central instance of the application, and broadcast the graphical interface to each user [9].

Collaboration transparency systems offered an improved collaboration workflow compared to manual file sharing, but since only one user could control the shared application instance at a time and users could not view different portions of the shared data, concurrent work was not possible [9]. Additionally, since these systems were unaware of the application-specific semantics and broadcast the graphical user interface to all users, they were considerably demanding on network bandwidth [9]. Modern alternatives like

Virtual Network Computing (VNC) and Microsoft Remote Desktop (RDP) are commonly used today, but mainly for single-user remote access instead of multi-user collaboration.

2.3 Cloud applications

When a service stores its primary copy of data on a remote server, it is referred to as a "cloud application" [2]. If the client has a copy of the data, it is only persisted as a cache for performance purposes, and any changes must be sent to the server or they are eventually overwritten [2]. These kinds of applications are completely dependent on the servers, and the creator of the software is actually providing a service instead of a standalone application—hence, this type of software is commonly referred to as *Software as a Service* [2]. Figure 1 illustrates flow of data in a typical cloud application.

One of the main benefits of cloud applications is ease of collaboration: when data is stored centrally in one place and client applications treat it as the source of truth, multiple users can view and modify it in real time from any of their devices [2]. Since both data storage and processing are handled by a central server, cloud applications can handle very large datasets and heavy workloads while requiring minimal processing power or data storage capacity from the client devices [11].

Cloud applications typically require all data modifications to be confirmed with a server, which means there is a delay between the user taking action and the result being visible [2]. Some applications address this drawback by applying each change to local cached data and presenting the result to the user before receiving the confirmed result from the server—a model known as *optimistic user interface* [12].

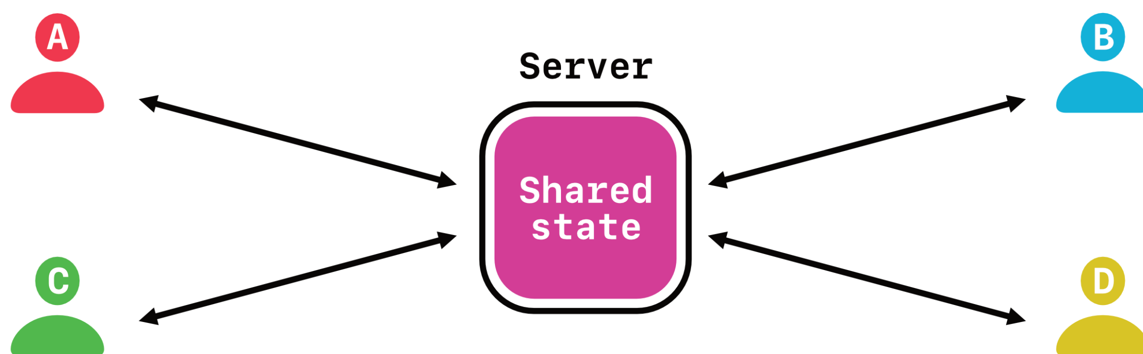

Figure 1: Illustration of data flow in a typical cloud application: users modify shared state stored on a central server, and cannot operate without a constant connection.

2.4 Local-first software

The local-first design pattern aims to combine the benefits of local and cloud applications by keeping the primary data store in the client application while still providing collaboration functionality [2]. This process involves implementing data synchronisation and conflict resolution systems due to each client managing their own state. In some cases relying on manual conflict resolution or a naive, deterministic algorithm like last-write-wins may be acceptable, but recent research proposes special data types that aim to preserve user intent and guarantee *eventual consistency* [5], [13]. These technologies are described in more detail in Section 3.

Figure 2 illustrates data flow in a typical local-first application. By storing the user's data locally, local-first applications can achieve improved privacy and security over cloud software [2]. First, data ownership is unambiguous, as the developer of the application is not involved in storing or processing the data. In the event of a data breach, the damage is limited to a single user's data since there is no need for centralised data persistence. Additionally, because the clients also handle processing locally, it is feasible to implement end-to-end encryption—ensuring that even when data is temporarily stored on a server for synchronisation purposes, it remains unreadable to any third parties [2].

Local-first software also includes potential for a more responsive user interface [2]. Since the primary copy of data in local-first applications is available locally and thus immediately accessible, any changes made by the user can be applied and be made visible with no delay, while the synchronisation to other users happens in the background [2]. The ability to apply changes locally also means the application can be used offline and changes can be synchronised later once a network connection—or some other form of communication—is available again. Furthermore, local access to data means there is no need for low-latency server communication, resulting in lower maintenance costs and improved service scalability [5].

The local-first approach has implications for developer experience as well. Since the local database can be used as the source of truth, state management in the client becomes simpler as there is no need to consider the state where something is pending a server response [3]. Local data also brings network resilience with its offline availability, reducing the need for multi-region backend services, content delivery networks or client-side caching solutions [4]. Asynchronous access to replicas and the absence of a centralised server-side bottleneck inherently improve service scalability as well [5].

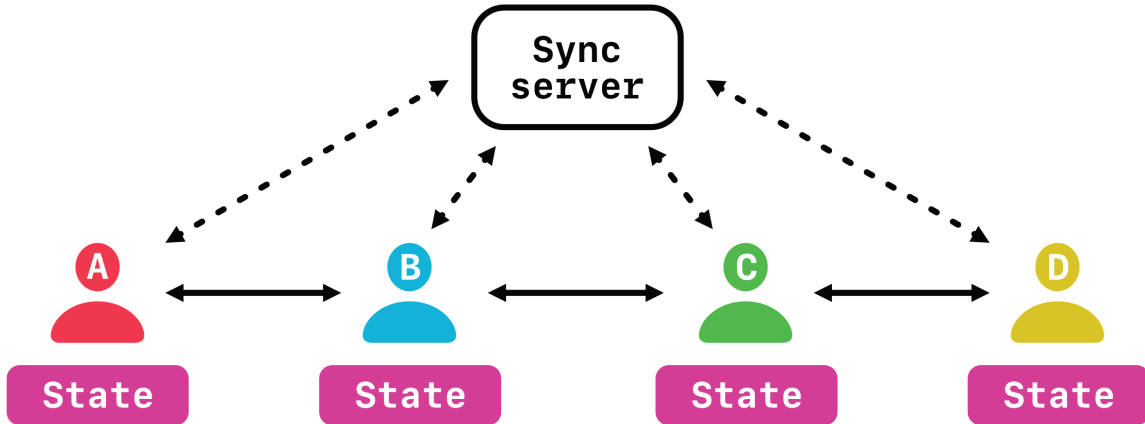

Figure 2: Illustration of data flow in a local-first application: each user has a copy of the data and synchronisation can be performed in a peer-to-peer network, or optionally via a synchronisation server.

2.5 Local-first limitations and drawbacks

Despite addressing many of the problems associated with cloud software, the local-first approach comes with its own challenges that make it unsuitable for some types of applications. For example, banking software and other systems with real-time constraints may require a single source of truth and immediate consistency of data [2]. Some software can involve operating on datasets that are too large to be stored on client devices and thus require a centralised server for storage and processing of data [3]. In practice, to keep storage requirements and initial load times reasonable, the data must be partitionable per user or otherwise constrained in size as the service scales [3].

Implementing a truly decentralised conflict resolution system can also result in performance and memory overhead, and require major engineering efforts compared to systems with a single central authority [14]. This is why some companies choose to adopt technologies inspired by the local-first proposal for conflict resolution, while keeping the traditional cloud software pattern as the main data synchronisation solution for easier implementation [14]. Since local-first software is a relatively recent paradigm, there are few established development frameworks available, meaning implementing these synchronisation systems often means building significant portions of the infrastructure from scratch. This can be a considerable engineering effort, especially for early-stage companies [15].

2.6 Summary

Collaboration in the software world has evolved from simple screen sharing and file sharing programs to modern real-time applications, where a large number of users can work on shared documents simultaneously. While collaborative software is most commonly built as cloud applications, recent research [2],[16],[5],[6] proposes a local-first pattern as an alternative. Benefits of the local-first approach over cloud-based applications include improved privacy and data control, reduced latency in the user interface, and the ability to collaborate asynchronously without a constant internet connection. However, drawbacks are introduced as well, such as increased implementation complexity and real-world constraints that are difficult or impossible to model in a local-first manner. Table 1 presents a comparison of typical characteristics of cloud-based software and local-first software.

Aspect	Cloud-based software	Local-first software
Offline functionality	Typically completely unavailable without an internet connection [2].	Fully functional offline, data is synchronised later when a connection is available [2].
Responsiveness	May involve latency due to server round-trips [3].	Nearly instantaneous interface updates due to local data processing [3].
Data ownership	Data is stored on remote servers; ownership may be ambiguous [2].	Data is stored and processed locally on the user's device, thus access is completely controlled by the user [2].
Privacy and security	Data is typically accessible by the service provider. Central storage is an attractive target for data breaches [2].	Local data processing makes end-to-end encryption implementation straightforward [2].
Failure tolerance	Service downtime makes the application unavailable [3].	Resilient to server downtime, software may be usable even after permanent shutdown of the service [3].
Scalability	Constant synchronisation requires expensive low-latency infrastructure, for example multi-zone deployments [4].	Asynchronous synchronisation is not affected by slow servers, reducing infrastructure cost and complexity [4].
Development complexity	Supported by established frameworks, tooling and conventions [17].	Higher development effort upfront and fewer widely adopted tools available [14].
Suitability	Best for applications requiring strong consistency guarantees, central coordination or operation on large datasets [3].	Best for applications prioritising offline use, decentralised collaboration and privacy [2].

Table 1: Comparison of local-first and cloud-based software characteristics

3 Local-First Technologies

This section reviews common technologies employed to build local-first software while highlighting their benefits as well as the limitations and constraints introduced by adopting them.

3.1 Conventional data synchronisation methods

Many widely used text editors, such as Google Docs, rely on *operational transformation* (OT) for collaborative editing [16]. OT algorithms are based on operations created from user actions, serialised by a central server, and synchronised back to clients to ensure that they all agree on the final state [18]. Since the clients are constantly synchronised with the shared state, the operations can be discarded immediately once they have been applied on the server [18].

In contrast with these centralised tools, some are based on a completely decentralised approach. For example, the version control tool Git operates on independent repositories that can be set up based on project requirements. Typically, contributors work on their personal repositories and asynchronously push revisions to one or more remote repositories [19]. These revisions consist of series of *commits* which, while internally stored as snapshots, can for convenience be represented as *diffs* that describe changes to a file since the previous commit [20]. Two revisions can then be combined by performing a *three-way merge*, where the changes in the diverging states are compared against a common ancestor. This enables completely asynchronous and decentralised workflows [19]. However, overlapping modifications result in conflicts that must be manually resolved by the user during the merging process [20].

3.2 Conflict-free replicated data types

In local-first software, one of the main tools utilised in merging the collaborators' changes is a specialised class of data types known as *conflict-free replicated data types* (CRDTs), originating from distributed systems research in 2011 [5], [21]. These data types enable automatic conflict resolution and *eventual consistency*. Using these data types, each client produces independent replicas that can be later merged and can provably converge to a correct common state [21].

CRDTs achieve conflict resolution and consistency by constraining all operations to have three properties: associativity, commutativity and idempotence [21], [22]. Operations are associative and commutative if they can be applied in any order without having an effect on the result [22]. For example, when two users each modify a text file, it is not known

whose change will be applied first, thus, the resulting text must be the same regardless of order. In addition, idempotence guarantees an operation can be applied any number of times without changing the result—this means there is no need to keep a log of which operations have been applied [21].

It is important to note that while CRDTs offer eventual consistency, they do not enforce constraints, and thus most systems cannot be completely modelled with CRDTs [5]. Some invariants require synchronisation by nature: for example, if an account balance must always be non-negative, it is impossible to ensure this is not violated by concurrent withdrawals in separate replicas [5]. A consistency requirement like this can be satisfied by applying changes sequentially, known as *strong consistency*, and CRDTs are often used in combination with this approach for operations that are compatible with eventual consistency [5].

3.3 Operation-based CRDTs

Operation-based CRDTs are a class of CRDTs that track the state of the application as incremental operations. Implementing an operation-based system is relatively simple, and the size of messages synchronised between participants stays small due to their incremental nature [23]. However, each operation must be applied individually on each replica, potentially causing processing overhead. Furthermore, correctly handling duplicate messages requires careful consideration in order to guarantee idempotence of the operations [23].

Compared to operational transform systems or others that rely on a central server, maintaining the state of an application using operation-based CRDTs typically requires storing the history of operations for longer. For example, in the case of a text editing application, a span of text removed by one user must not be deleted from the data structure but simply marked as removed instead [16]. This process creates objects known as *tombstones*, which allow other concurrent changes to the deleted piece of data to be successfully merged later [16]. The storage and processing overhead caused by these tombstone objects can be mitigated by including *causality metadata* in the operations, which is used to track which clients have already incorporated the change into their local replica [24]. Once all clients have "seen" a tombstone, it is guaranteed that no further changes can be made to the data referenced by the tombstone, and therefore it can be permanently deleted [18], [24].

Figure 3 demonstrates the fundamental operation of a simple operation-based counter CRDT. In the diagram, circled numbers represent actions taken by the user, and the numbers along the timelines represent each participant's local state after each operation has been applied. Here, the participants eventually reach an agreement on the correct

state despite the asynchronous operations initially causing the states to diverge. However, there is no protection against duplicate operations since the messages do not include information about the sender's state. A duplicated message would therefore result in permanently divergent state between the participants.

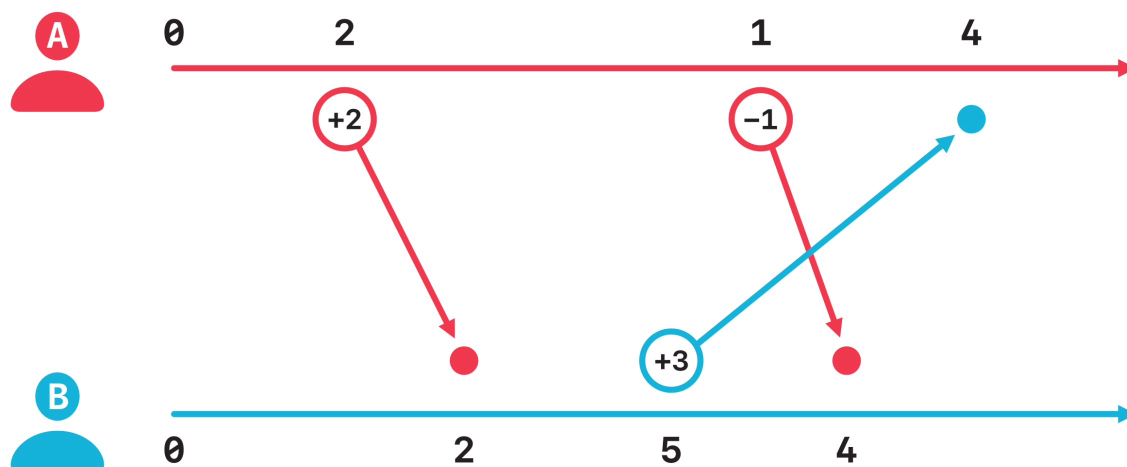

Figure 3: Illustration of an operation-based counter CRDT

3.4 State-based CRDTs

State-based CRDTs are a complementary alternative to operation-based CRDTs. In a state-based system, a message represents the whole state of the application, and synchronisation of the messages is performed by *merging* the different states [25]. This approach means operations can be batched and the resulting state can be sent and applied to other replicas with a single message instead of individual operations, and message duplication is not an issue [23]. However, sending the entire application state introduces communication overhead, as the size of the state grows as more changes are made or more participants are added [23]. In practice, state-based CRDTs are only useful for data types that have a small state size.

Figure 4 demonstrates the operation of a state-based alternative for the counter CRDT presented earlier. Here, the state maintained by each participant includes the latest known values from all participants, and the final result is evaluated by calculating the sum of these values. When a user performs an action, a message with the whole state is sent, and the receiver updates their local state accordingly. This provides a straightforward solution for ensuring duplicate messages have no effect.

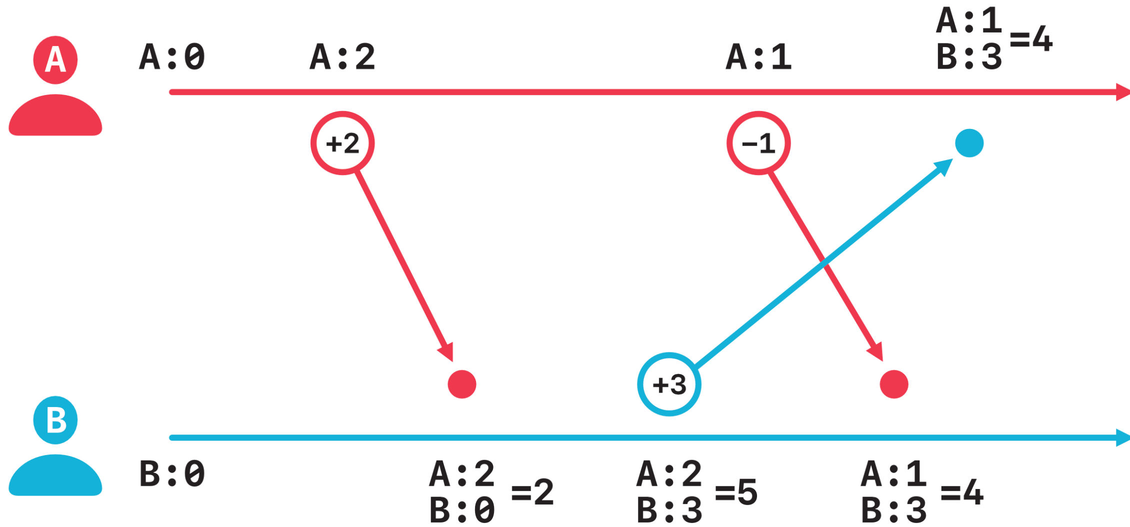

Figure 4: Illustration of a state-based counter CRDT

3.5 Delta state-based CRDTs

Delta state-based CRDTs are a variant of the state-based design, aiming to remove the requirement of sending the complete state while still ensuring idempotence of the system for compatibility with unreliable communication methods [23]. This is achieved by calculating incremental states or *deltas* that are synchronised by *joining* them into existing state on replicas [23]. A delta state-based system combines the benefits of both operation-based and state-based CRDTs, but introduces the drawback of requiring causal ordering of messages [23].

3.6 Concurrency semantics

CRDTs have been proposed for many common data types, such as registers, counters, sets, and lists. However, for most of these data types, multiple solutions exist for supporting the core CRDT properties: concurrent data modification and convergence to a common state [26].

In the case of some simple data types, the supported operations are naturally commutative, thus no special consideration is required for convergence, and implementation is straightforward [25]. For example, counter data types represent a single shared number and support increment and decrement operations. Since the order of summation does not affect the result, the concurrency semantics of counter CRDTs are unambiguous [26].

More commonly, CRDTs include non-commutative operations and must define arbitration rules for concurrent modifications [26]. These rules must preserve the sequential semantics of the original data type while providing a deterministic result for concurrent

execution [25]. For example, a set data type supports addition and removal of elements, and concurrently performing both operations for the same element is not well-defined without additional semantics. This can be resolved by declaring that additions always win over removals, and doing both concurrently results in the element belonging to the set. Alternatively, removals can win over additions, or timestamps from loosely synchronised clocks can be attached to the operations for last-writer-wins semantics [26].

3.7 Tools for local-first development

To help with the non-trivial task of implementing a robust conflict resolution system, some general-purpose development tools have been created. For example, the Automerge library provides a JSON data structure implemented as a CRDT to enable automatic conflict resolution and data synchronisation, and includes additional features such as end-to-end encryption support [27]. A popular alternative is Yjs, which also utilises CRDTs, but supports nested maps, lists and text instead of standard JSON [28],[29]. Solutions that operate on relational data are also in development, though not yet widely adopted [30],[31].

Some libraries provide partial local-first functionality. For example, the Replicache synchronisation engine is a client-side datastore that enables optimistically applying changes locally, then synchronising them to the server asynchronously [32]. This achieves some of the benefits of local-first software, most importantly a responsive user interface, while still treating the remote server as the source of truth [32]. Table 2 compares the core aspects of these libraries.

	Conflict resolution	**Synchronisation model**	**Supported data types**
Automerge	CRDT [27]	Client-server or peer-to-peer [27]	JSON [27]
Yjs	CRDT [28]	Client-server or peer-to-peer [29]	Nested maps, lists and text [29]
Replicache	Server as authority, programmatic conflict resolution via *mutators* [32]	Client-server [32]	JSON [32]

Table 2: Comparison of libraries commonly utilised in local-first development

3.8 Examples of local-first applications

Despite local-first being only in its early stages, the first applications utilising this approach are already emerging. For example, the note-taking application AnyType[2] is built according to local-first principles, utilising CRDTs to offer a decentralised collaboration experience, including local data processing and conflict resolution, peer-to-peer synchronisation and end-to-end encryption [33]. Likewise, the personal budgeting application Actual[3] uses CRDTs to achieve fast loading times and high reliability synchronising across devices regardless of network conditions [34],[35].

Some applications employ architectures partially inspired by the local-first pattern, integrating it with traditional approaches to achieve a desirable balance. For example, the user interface design software Figma has adopted some local-first technologies in their multi-user editing functionality. While it does not implement pure CRDTs, it utilizes a modified version with relaxed consistency requirements, supplemented by a central authoritative server [14]. Similarly, the project management application Linear employs a proprietary synchronization engine that incorporates several local-first principles, such as local data processing, offline support, and a responsive user interface. Additionally, adopting a local-first approach has enabled Linear to reduce infrastructure requirements and lower operational costs [15]. Table 3 compares the conflict resolution mechanisms of some applications that offer local-first features.

	Purpose	**Data synchronisation**	**Local-first features**
AnyType	Note-taking and planning	CRDT-based conflict resolution (Any-Sync) [33]	Offline functionality, responsive user interface, end-to-end encryption [33]
Actual Budget	Personal budget management	CRDT-based conflict resolution [35]	Offline functionality, responsive user interface, end-to-end encryption [35]
Linear	Project management	Proprietary synchronisation engine [15]	Partial offline functionality, responsive user interface, real-time collaboration [15]
Figma	User interface design	CRDT-inspired conflict resolution with server as central authority [14]	Responsive user interface, real-time collaboration [14]

Table 3: Comparison of some local-first applications

[2]`https://anytype.io`
[3]`https://actualbudget.org`

3.9 Summary

Traditional synchronisation methods, such as operational transform (OT), rely heavily on centralised servers and immediate state synchronisation. This simplifies conflict resolution but limits offline capabilities. In contrast, decentralised systems like Git support asynchronous workflows but require manual conflict handling when merging changes.

Conflict-free replicated data types (CRDTs) enable automatic conflict resolution and eventual consistency for certain types of operations. However, some systems have strong consistency requirements that cannot be modeled with CRDTs. Operation-based CRDTs offer efficient communication with their incremental updates, but incur storage and processing overhead. State-based CRDTs simplify message handling and eliminate duplication issues, but suffer from increasing communication overhead as the state size grows. Delta state-based CRDTs combine features of operation-based and state-based CRDTs. Table 4 compares the features of each type of CRDT.

	Operation-based CRDT	**State-based CRDT**	**Delta state-based CRDT**
Message content	User actions (operations) [25]	Complete state [25]	Partial state diffs (deltas) [25]
Benefits	Small message size (low bandwidth)	Simple merge logic; tolerates duplicate or lost messages [25]	Small message size, tolerates duplicate or lost messages [25]
Drawbacks	Requires exactly-once message delivery [25]	Large message size (high bandwidth)	Complex merge logic [25]

Table 4: Comparison of different types of conflict-free replicated data types

4 Discussion

Local-first is a rapidly evolving paradigm that addresses many issues characteristic of cloud software. However, particularly at this early stage of development, it introduces a different set of challenges that need to be carefully considered by those seeking to adopt a local-first approach in their upcoming products.

4.1 Feasibility of the local-first approach

Although the local-first approach imposes significant constraints on the development of collaborative applications—such as the need for careful asynchronous conflict resolution—it also offers benefits that can justify the associated development effort. The improved user interface responsiveness and offline functionality are valuable features, even for a user without a technical background. Users concerned with privacy and security are likely to prefer software that allows local data storage and end-to-end encryption—features commonly associated with local-first designs. Even if user preferences are ignored, local-first architectures can reduce infrastructure complexity and costs, offering advantages over cloud software even when applied only partially. Nonetheless, it should be noted that traditional cloud-based approaches remain more practical in some types of applications, such as those involving financial transactions, real-time consistency constraints, or the handling of very large datasets or complex computations.

4.2 Adoption of local-first technologies

The number of companies adopting local-first technologies is still relatively low compared to those opting for traditional cloud-based architectures. This can be at least partially attributed to the relatively recent emergence of the local-first paradigm. Although some development frameworks and tools are already available for building local-first applications, they are not yet very mature or widely adopted, and their use can appear risky for companies selecting an architectural foundation for their new product.

Despite CRDT-based conflict resolution systems enabling impressive features like offline-compatible collaboration, the complexity and additional constraints involved in implementing such systems are likely to drive the adoption of hybrid architectures instead of pure local-first software. Combining CRDT-like conflict resolution algorithms with the traditional single-source-of-truth approach has proven viable in the industry [14] and provides an optimal balance between local-first benefits and cloud software convenience.

Although adoption of local-first technologies has been relatively slow, there is clear evidence of growing interest in this emerging paradigm within both academic and

professional communities. This is reflected in community initiatives such as *Local-First Conf*—a dedicated conference first held in May 2024 [36]—and a podcast on the same theme, *localfirst.fm*, which launched in January 2024 [37].

4.3 Future of local-first software

As demand for highly scalable and performant collaboration software grows along with user awareness of the privacy and security aspects, local-first designs are likely to become more common. Some companies that have opted for the local-first approach, such as Linear [15], are also sharing their experiences and actively promoting the model, which is likely to improve confidence in the technology among potential adopters. As larger companies begin to adopt emerging paradigms such as local-first, they often contribute to the development of tooling and frameworks, further accelerating adoption across the industry.

5 Conclusion

The aim of this thesis was to answer the following research question: **What are the potential benefits and drawbacks of using a local-first approach for developing collaborative applications?** To address this, I reviewed the high-level evolution of collaborative software architectures and compared their key characteristics with those of local-first software. I also analysed the advantages and limitations of some of the technologies commonly utilised in local-first development.

5.1 Main findings

The findings highlight that local-first software enhances user experience through increased user interface responsiveness, offline functionality, and improved security and privacy. It is also beneficial for service scalability and can reduce infrastructure complexity. However, certain challenges are also introduced, such as asynchronous conflict resolution complexity and limited availability of widely adopted development tools. Nevertheless, the technology is developing rapidly and likely to become more popular in the near future.

5.2 Open questions

Some open questions remain for future research:

- Can standardised, general-purpose frameworks for local-first software development be designed to accelerate the adoption of this architectural approach?
- Is it possible to streamline the development of hybrid systems that integrate operations adhering to the eventual consistency guarantees of conflict-free replicated data types (CRDTs) with operations requiring strong consistency?
- Can existing database solutions be utilised in large-scale local-first software, or is developing a purpose-built database necessary?

References

[1] Charlie Lin, Wei-Chieh Wayne Yu, and Jenny Wang. Cloud collaboration: Cloud-based instruction for business writing class. 4(6):p9. ISSN 1925-0754, 1925-0746. doi: 10.5430/wje.v4n6p9. URL `http://www.sciedu.ca/journal/index.php/wje/article/view/5884`.

[2] Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark 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*, Onward! 2019, pages 154–178. Association for Computing Machinery. ISBN 978-1-4503-6995-4. doi: 10.1145/3359591.3359737. URL `https://doi.org/10.1145/3359591.3359737`.

[3] Why local-first software is the future and its limitations | RxDB - JavaScript database, . URL `https://rxdb.info/articles/local-first-future.html`. Accessed 2025-03-07.

[4] van der Merwe Phillip. Local-first key concepts: Developer benefits of local-first. URL `https://www.powersync.com/blog/local-first-key-concepts-developer-benefits-of-local-first`. Accessed 2025-04-03.

[5] Mirko Köhler, George Zakhour, Pascal Weisenburger, and Guido Salvaneschi. Consistent local-first software: Enforcing safety and invariants for local-first applications. 51(1):53–65. ISSN 1939-3520. doi: 10.1109/TSE.2024.3477723. URL `https://ieeexplore.ieee.org/document/10713276`. Conference Name: IEEE Transactions on Software Engineering.

[6] Julian Haas, Ragnar Mogk, Elena Yanakieva, Annette Bieniusa, and Mira Mezini. LoRe: A programming model for verifiably safe local-first software. 46(1):2:1–2:26. ISSN 0164-0925. doi: 10.1145/3633769. URL `https://dl.acm.org/doi/10.1145/3633769`.

[7] Automerge. A JSON-like data structure (a CRDT) that can be modified concurrently by different users, and merged again automatically. URL `https://github.com/automerge/automerge`. Accessed 2025-04-28.

[8] Johannes Schickling. #4 – Martin Kleppmann: CRDTs, Automerge, generic syncing servers & Bluesky – localfirst.fm. URL `https://localfirst.fm/4`. Accessed 2025-04-28.

[9] James Begole, Mary Beth Rosson, and Clifford A. Shaffer. Flexible collaboration transparency: supporting worker independence in replicated application-sharing

systems. 6(2):95–132. ISSN 1073-0516. doi: 10.1145/319091.319096. URL https://dl.acm.org/doi/10.1145/319091.319096.

[10] Georgia Bafoutsou and Gregoris Mentzas. Review and functional classification of collaborative systems. 22(4):281–305. ISSN 02684012. doi: 10.1016/S0268-4012(02)00013-0. URL https://linkinghub.elsevier.com/retrieve/pii/S0268401202000130.

[11] Pham Phuoc Hung, Tuan-Anh Bui, Mauricio Alejandro Gómez Morales, Mui Van Nguyen, and Eui-Nam Huh. Optimal collaboration of thin–thick clients and resource allocation in cloud computing. 18(3):563–572. ISSN 1617-4909, 1617-4917. doi: 10.1007/s00779-013-0673-z. URL http://link.springer.com/10.1007/s00779-013-0673-z.

[12] Denys Mishunov. True lies of optimistic user interfaces. URL https://www.smashingmagazine.com/2016/11/true-lies-of-optimistic-user-interfaces/. Accessed 2025-02-14.

[13] Werner Vogels. Eventually consistent. 52(1):40–44. ISSN 0001-0782. doi: 10.1145/1435417.1435432. URL https://dl.acm.org/doi/10.1145/1435417.1435432.

[14] Evan Wallace. How figma's multiplayer technology works | figma blog. URL https://www.figma.com/blog/how-figmas-multiplayer-technology-works/. Accessed 2025-04-11.

[15] Tuomas Artman. Scaling the linear sync engine. Accessed 2025-04-12. URL https://www.youtube.com/watch?v=Wo2m3jaJixU.

[16] Geoffrey Litt, Sarah Lim, Martin Kleppmann, and Peter van Hardenberg. Peritext: A CRDT for collaborative rich text editing. 6:531:1–531:36, . doi: 10.1145/3555644. URL https://dl.acm.org/doi/10.1145/3555644.

[17] Stack Overflow. 2024 Stack Overflow developer survey. URL https://survey.stackoverflow.co/2024/technology#1-cloud-platforms. Accessed 2025-04-28.

[18] David A. Nichols, Pavel Curtis, Michael Dixon, and John Lamping. High-latency, low-bandwidth windowing in the jupiter collaboration system. In *Proceedings of the 8th annual ACM symposium on User interface and software technology*, UIST '95, pages 111–120. Association for Computing Machinery. ISBN 978-0-89791-709-4. doi: 10.1145/215585.215706. URL https://dl.acm.org/doi/10.1145/215585.215706.

[19] Diomidis Spinellis. Git. 29(3):100–101. ISSN 1937-4194. doi: 10.1109/MS.2012.61. URL https://ieeexplore.ieee.org/document/6188603/?arnumber=6188603. Conference Name: IEEE Software.

[20] Prem Kumar Ponuthorai and Jon Loeliger. *Version control with Git: powerful tools and techniques for collaborative software development*. O'Reilly Media, Inc, third edition edition. ISBN 978-1-4920-9119-6 978-1-4920-9116-5. URL https://learning.oreilly.com/library/view/version-control-with/9781492091189/.

[21] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *Stabilization, Safety, and Security of Distributed Systems*, volume 6976, pages 386–400. Springer Berlin Heidelberg. ISBN 978-3-642-24549-7 978-3-642-24550-3. doi: 10.1007/978-3-642-24550-3_29. URL http://link.springer.com/10.1007/978-3-642-24550-3_29. Series Title: Lecture Notes in Computer Science.

[22] Peter Bailis and Ali Ghodsi. Eventual consistency today: limitations, extensions, and beyond. 56(5):55–63. ISSN 0001-0782. doi: 10.1145/2447976.2447992. URL https://dl.acm.org/doi/10.1145/2447976.2447992.

[23] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Efficient state-based CRDTs by delta-mutation. In Ahmed Bouajjani and Hugues Fauconnier, editors, *Networked Systems*, volume 9466, pages 62–76. Springer International Publishing. ISBN 978-3-319-26849-1 978-3-319-26850-7. doi: 10.1007/978-3-319-26850-7_5. URL http://link.springer.com/10.1007/978-3-319-26850-7_5. Series Title: Lecture Notes in Computer Science.

[24] Carlos Baquero, Paulo Sergio Almeida, and Ali Shoker. Pure operation-based replicated data types. URL http://arxiv.org/abs/1710.04469.

[25] Paulo Sérgio Almeida. Approaches to conflict-free replicated data types. 57(2):51:1–51:36. ISSN 0360-0300. doi: 10.1145/3695249. URL https://dl.acm.org/doi/10.1145/3695249.

[26] Nuno Preguiça. Conflict-free replicated data types: An overview. URL http://arxiv.org/abs/1806.10254.

[27] Martin Kleppmann and Alastair R Beresford. Automerge: Real-time data sync between edge devices. URL https://mobiuk.org/abstract/S4-P5-Kleppmann-Automerge.pdf. Accessed 2025-05-06.

[28] Yicheng Zhang, Matthew Weidner, and Heather Miller. Programmer experience when using CRDTs to build collaborative webapps: Initial insights. doi: 10.1184/R1/22277341.v1. URL https://kilthub.cmu.edu/articles/conference_contribution/Programmer_Experience_When_Using_CRDTs_to_Build_Collaborative_Webapps_Initial_Insights/22277341/1. Publisher: Plateau Workshop.

[29] Petru Nicolaescu, Kevin Jahns, Michael Derntl, and Ralf Klamma. Yjs: A framework for near real-time p2p shared editing on arbitrary data types. In Philipp Cimiano, Flavius Frasincar, Geert-Jan Houben, and Daniel Schwabe, editors, *Engineering the Web in the Big Data Era*, pages 675–678. Springer International Publishing. ISBN 978-3-319-19890-3. doi: 10.1007/978-3-319-19890-3_55.

[30] Geoffrey Litt, Nicholas Schiefer, Johannes Schickling, and Daniel Jackson. Riffle: Reactive relational state for local-first applications. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, pages 1–16. ACM, . ISBN 979-8-4007-0132-0. doi: 10.1145/3586183.3606801. URL `https://dl.acm.org/doi/10.1145/3586183.3606801`.

[31] Rocicorp LLC. Zero docs. URL `https://zero.rocicorp.dev/docs/introduction`. Accessed 2025-05-06.

[32] Rocicorp LLC. How Replicache works | Replicache docs. URL `https://doc.replicache.dev/concepts/how-it-works`. Accessed 2025-05-06.

[33] Any Association. Protocol overview. URL `https://tech.anytype.io/any-sync/overview`. Accessed 2025-04-14.

[34] Actual Budget. A local-first personal finance app. URL `https://github.com/actualbudget/actual`. Accessed 2025-04-15.

[35] James Long. Actual: Using CRDTs in the wild. URL `https://jlongster.com/using-crdts-in-the-wild`. Accessed 2025-04-15.

[36] Local-first conf 2024, . URL `https://www.localfirstconf.com/local-first-conf-2024`. Accessed 2025-02-14.

[37] Johannes Schickling. localfirst.fm - a podcast about local-first software development. URL `https://www.localfirst.fm/`. Accessed 2025-04-14.