Bachelorarbeit

im Studiengang Medieninformatik

Implementation of a synchronization layer for external APIs in local-first applications

Implementation eines Synchronisations-Layer für externe APIs in Local-First Anwendungen

---

vorgelegt von Max Knerrich

an der Hochschule der Medien am 20. Juni 2025

zur Erlangung des akademischen Grades eines Bachelor of Science

Erstprüfer: Prof. Dr. Tobias Jordine

Zweitprüfer: André König, B.Sc.

Hiermit versichere ich, Max Knerrich, ehrenwörtlich, dass ich die vorliegende Bachelorarbeit mit dem Titel: „Implementation of a synchronization layer for external APIs in local-first applications“ selbstständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Ebenso sind alle Stellen, die mit Hilfe eines KI-basierten Schreibwerkzeugs erstellt oder überarbeitet wurden, kenntlich gemacht. Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden. Das KI-Werkzeug DeepL Write[1] wurde ausschließlich zum Lektorat und als Formulierungshilfe verwendet. Anschließend wurde sichergestellt, dass der Inhalt mit den intendierten Aussagen noch übereinstimmt. Außerdem wurde das KI-Werkzeug GitHub Copilot[2] für Optimierung und Generierung einzelner Programmcodes der Beispielanwendungen und der Messinstrumente verwendet. Bei der Erstellung der vorgelegten Studienleistung habe ich durchgehend eigenständig und beim Einsatz IT-/KI-gestützter Schreibwerkzeuge steuernd gearbeitet

Ich habe die Bedeutung der ehrenwörtlichen Versicherung und die prüfungsrechtlichen Folgen (§ 24 Abs. 2 Bachelor-SPO, § 23 Abs. 2 Master-SPO (Vollzeit)) einer unrichtigen oder unvollständigen ehrenwörtlichen Versicherung zur Kenntnis genommen.

Stuttgart, 20.06.2025 Max Knerrich

[1] https://deepl.com/write
[2] https://github.com/features/copilot

Abstract

The paradigm of local-first software is becoming increasingly popular, as it offers a compelling alternative to the already established cloud-first architecture, by providing fast interaction, resilient applications, and offline functionality. However, a substantial challenge remains in integrating local-first applications with external systems, as they do not offer the conflict resolution and stateful updates local-first applications require. This thesis bridges the gap, investigating how to enable local-first functionality while synchronizing with external data sources.

The primary contribution of this work is the design, implementation, and evaluation of a generic client-side synchronization layer for REST APIs. This thesis proposes an architecture based on a local write-log and a deterministic two-pass reconciliation algorithm to manage state changes between the local application and remote API. This architecture is designed to handle offline modifications, detect and resolve conflicts and optimize network communication. The purpose is to provide a reusable solution that unlocks the benefits of local-first development for applications consuming data from external REST APIs.

To validate this approach, a prototype issue-tracking application was developed that uses the proposed synchronization layer to interact with the GitHub REST API. The results demonstrate that the synchronization layer dramatically improves application responsiveness and offline capabilities. It reduces subsequent load times by up to 97%, ensures all local writes provide immediate visual feedback and notably reduces the number of API requests. This work confirms that a dedicated client-side synchronization layer is a viable and effective design pattern for creating modern, high-performance, local-first applications.

Zusammenfassung

Local-First Software wird immer populärer, da sie eine überzeugende Alternative zur bereits etablierten Cloud-First-Architektur darstellt, indem sie schnelle Interaktion, resiliente Anwendungen und Offline-Funktionalität bietet. Eine zentrale Herausforderung bleibt jedoch die Integration von Local-First-Anwendungen mit externen Systemen, da diese nicht die Konfliktlösungsalgorithmen und stateful Updates bieten, die Local-First-Anwendungen benötigen. Diese Arbeit schließt diese Lücke, indem sie untersucht, wie Local-First-Funktionalität ermöglicht werden kann, während gleichzeitig mit externen Datenquellen synchronisiert wird.

Der Hauptbeitrag dieser Arbeit ist das Design, die Implementierung und die Evaluation eines generischen clientseitigen Synchronisations-Layers für REST APIs. Die Arbeit schlägt eine Architektur vor, die auf einem lokalen Write-Log und einem deterministischen Two-Pass Reconciliation-Algorithmus basiert, um Änderungen zwischen der lokalen Application und der remote API zu managen. Diese Architektur ist darauf ausgelegt, offline Bearbeitung zu unterstützen, Konflikte zu erkennen und zu lösen sowie die Kommunikation über das Netzwerk zu optimieren. Ziel ist es, eine wiederverwendbare Lösung zu erstellen, die die Vorteile von Local-First Development für Applikationen erschließt, die Daten von externen REST APIs konsumieren.

Zur Validierung dieses Ansatzes wurde ein Prototyp einer Issue-Tracking-Anwendung entwickelt, die den vorgeschlagenen Synchronisations-Layer nutzt, um mit der GitHub REST API zu interagieren. Die Ergebnisse zeigen, dass der Synchronisations-Layer die Reaktionsfähigkeit der Anwendung und die Offline-Fähigkeiten dramatisch verbessert. Ladezeiten werden um bis zu 97 % reduziert, alle lokalen Updates liefern sofortiges visuelles Feedback und die Anzahl der API-Anfragen wird deutlich verringert. Diese Arbeit bestätigt, dass ein dedizierter clientseitiger Synchronisations-Layer ein praktikables und effektives Design Pattern für die Entwicklung moderner, performanter Local-First-Anwendungen ist.

Contents

List of Figures

List of Listings

List of Tables

List of Abbreviations

ACID	Atomicity, Consistency, Isolation, Durability
API	Application Programming Interface
CAP	Consistency, Availability, Partition Tolerance
CDC	Change Data Capture
CDN	Content Delivery Network
CRDT	Conflict-free Replicated Data Type
CRUD	Create, Read, Update, Delete
LWW	Last Write Wins
NAT	Network Address Translation
P2P	Peer-to-Peer
REST	Representational State Transfer
RTT	Round-Trip-Time
SPA	Single-Page Application
SSR	Server-Side Rendering
TTL	Time to Live
TTVF	Time to Visual Feedback

1 Introduction

Applications on the web and mobile devices increasingly rely on server-side business logic and use **APIs** to interact with external systems. However, this cloud-first model often results in sluggish user interfaces, fragile offline behaviour, and excessive network traffic. In contrast, the local-first paradigm advocates that applications should store and manipulate data locally by default. It can deliver fast interaction, seamless offline operation, and reduced dependency on network availability [1]. However, most public **APIs** expose only stateless, snapshot-based endpoints without built-in conflict resolution or push notifications, making direct local-first integration difficult.

Existing client-side libraries either require a bespoke server component or do not support the integration of external data at all. However, these client-side libraries cannot be used if the application wants to consume a generic third-party **API**. A principled architecture is needed to bridge the gap between a local-first application and an external **API**, preserving local-first ideals without modifying the remote service.

1.1 Motivation

Enabling robust local-first behaviour over arbitrary **APIs** promises substantial benefits: instantaneous user feedback, uninterrupted operation during network outages, and minimal bandwidth usage. These properties are increasingly important in domains such as project management, issue tracking, and collaborative editing, where users expect low-latency interactions, even on mobile or unreliable networks. A general-purpose synchronization layer that operates entirely in the client and interposes between the local store and any stateless **API** would unlock these benefits without requiring changes to existing cloud services.

1.2 Objectives

This thesis investigates the following research questions:

1. What architectural patterns best enable the synchronization of state between a local-first application and an external **API**, while upholding core local-first principles such as offline functionality and performance?
2. What mechanisms are required to reliably capture local data modifications made during offline periods and manage the data transformation needed to resolve schema differences between the local application and the remote **API**?
3. How can data conflicts be effectively detected and resolved when arising from concurrent modifications to the same data item on both the local application and the external API, particularly when the API lacks native conflict resolution support?
4. What are the fundamental trade-offs between application responsiveness, data freshness, and resource efficiency when designing a synchronization layer for a local-first application?

To address them, this work presents a modular synchronization layer and evaluates it in a prototype application against the GitHub Representational State Transfer (**REST**) **API**.

1.3 Structure

Chapter 2 - Related work surveys related work to this thesis.

Chapter 3 - Background presents theoretical background about caching, latency, consistency models, reactive systems like event sourcing. Then conflict resolution and local-first software are presented and explained.

Chapter 4 - Method explains the method chosen to answer the research questions. It states the requirements for the synchronization layer and scope of this research.

Chapter 5 - Design presents an architecture to solve the research topic.

Chapter 6 - Implementation explains how the architecture is implemented inside the prototype.

Chapter 7 - Evaluation analyses the implementation on how it solves the given requirements.
Chapter 8 - Summary provides a summary and conclusion of this thesis.

2 Related Work

The field of local-first software is still in early research, so a gap remains in the research regarding the integration of external **APIs** in local-first applications. To date, no known works address the challenges of leveraging external **APIs** while keeping the core principles of local-first software, like offline functionality, longevity, and low latency. However, two works share related goals or address specific aspects relevant to this research, while not focusing specifically on local-first software.

A central challenge in integrating external **APIs** is managing the state of dynamic data fetched from a third-party source. Therefore, research into advanced caching mechanisms for dynamic content is highly relevant for this research, as it explores techniques to maintain data consistency between dynamic data. *M. Sosnowski, R. von Seck, F. Wiedner, and G. Carle* [2] present a CRDT-based web caching approach to allow the caching of basic CRUD-operations by synchronizing different proxy caches with the help of CRDTs. This allows the proxies to accept writes from clients and synchronize with each other, synchronizing with the origin server at a later time, reducing the load of the origin server. Compared to a Time to Live (**TTL**) approach, CRDT web caching provides lower performance due to the synchronization overhead between the replicas, but reduces inconsistency windows, making it a viable option for dynamic data.

With Speed Kit, authors *W. Wingerath* et al. [3] propose an approach to dynamic caching by implementing a local on-device proxy, which splits requests into a highly cacheable anonymous version and the personalized content and merges the responses together to display the page. This makes caching applicable to personalized websites, which are typically considered unable to cache, while still protecting sensitive information, as all personalized data is handled on-device, and only anonymous data is remotely cached to improve performance.

While this existing academic research explores potential techniques for managing dynamic data, it is only research in the context of cloud-based applications, as the topic of local-first is still an emerging topic. While the area of integrating external **APIs** with local-first software lacks formal academic investigation, its practical relevance is demonstrated by several developer-driven projects. For instance, Overtone[3] is a local-first music client that uses external **APIs**, like Spotify, Tidal, or Soundcloud to provide the user with their music and playlists

[3] https://overtone.pro

on the respective service. Similarly, TinyHub[4] functions as a local-first GitHub client, which currently allows the user to read issues, pull requests and other statistics for their own or starred repositories.

The broader topic of synchronization in local-first software is addressed by multiple industry frameworks. ElectricSQL[5] manages partial synchronization of Postgres database subsets, while Zero[6] enables local synchronized queries for backend databases. However, these solutions require control over the backend infrastructure, as they need access to the system's database [4], [5] and are therefore not feasible to synchronize remote data.

[4]https://tinyhub.org
[5]https://electric-sql.com
[6]https://zero.rocicorp.dev

3 Background

In preparation for the research and results, this chapter provides background necessary to understand the main part of the thesis. This section starts by covering caching, along with latency. Furthermore, an introduction to consistency models, reactive systems, and conflict resolution, which form the foundation of local-first software. At last, local-first software is introduced.

3.1 Caching

A cache is a temporary storage area, where frequently accessed data is stored for rapid access [6]. Web caching is well established and is mostly implemented by client-side caches (e.g. browser caches) and shared caches, like Content Delivery Networks (**CDNs**), which provide cache results to multiple users, while being as close as possible to the user [7].

While caching is heavily used for static content, caching dynamic or user-generated content is more challenging. Most systems are based on **TTL** and can be unsuitable for dynamic data, as they can lead to long inconsistency windows and high staleness if the content changes before the **TTL** expires [2].

J. Zhong, R. D. Yates, and E. Soljanin [8] propose two metrics to measure the freshness of the cached content:

- **Age of Synchronization (AoS)** measures the time elapsed since the local cache and the remote source became desynchronized.
- **Age of Information (AoI)** measures the total time elapsed since the data held in the cache was originally generated at the source.

While these are meaningful metrics to evaluate a caching implementation, there are real-world scenarios where content does not lose its value simply because time has passed. *B. Abolhassani, J. Tadrous, A. Eryilmaz, and S. Yüksel* propose a different metric in [9] called **Age of Version (AoV)**, which counts the integer difference between the versions in the database and cache. Figure 1 shows how **AoV** compares against the other two metrics.

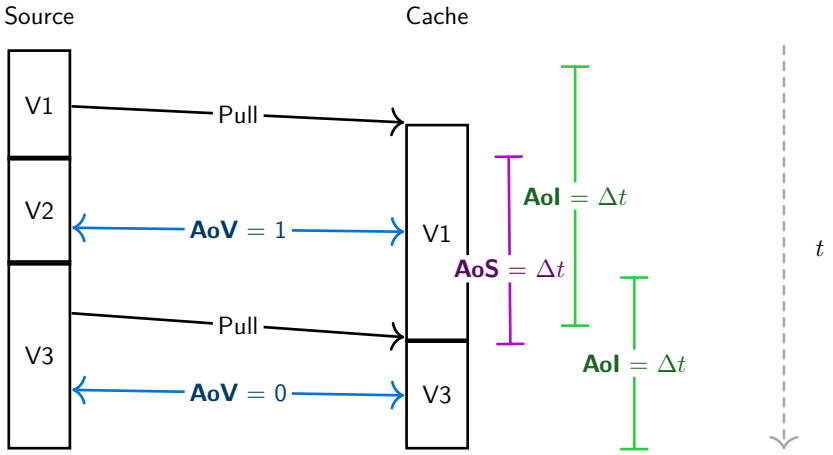

Figure 1: Illustration of freshness metrics in a pull-based system

Minimizing these metrics to maintain data integrity requires a strategy to update or invalidate stale copies. In client-server architectures, this is often done through active invalidation, which however is quite complex [2]. To solve this, *B. Abolhassani, J. Tadrous, A. Eryilmaz, and S. Yüksel* propose two approaches, push-based caching, and pull-based caching. While push-based systems are only valid for traditional client-server approaches, where you have control over the server, the client orchestrates pull-based caching. In it, the local cache in the client decides when to request the latest version from the backend, based on its knowledge of user requests, but without necessarily knowing the age of the content [9].

This distinction between push and pull strategies, illustrated in Figure 2, is central to the research problem of this thesis. In the pull-based model, the client periodically requests updates from a passive server. In the push-based model, an active server sends updates to listening clients.

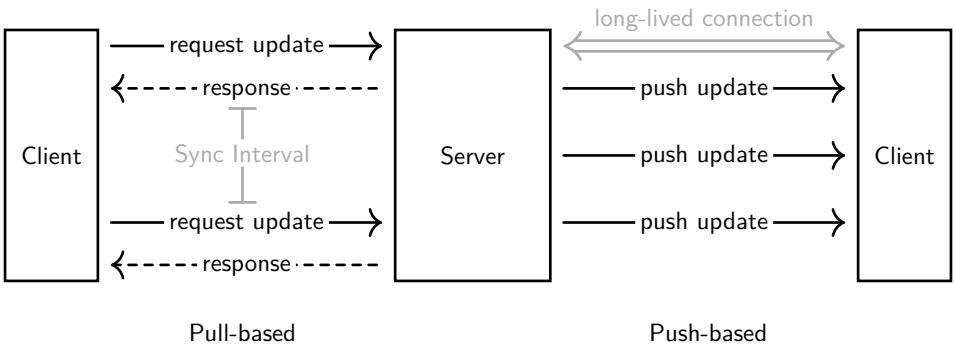

Figure 2: Difference between a pull and push-based synchronization

While push-based synchronization frameworks like Zero[7] or Jazz[8] exist, they depend on a developer-controlled, sync-aware server. However, synchronizing with an external third-party **API** forces a pull-based model, as the remote server does not push updates. This research addresses the resulting gap in tooling by designing a generic, pull-based synchronization layer for this common but underserved problem.

3.2 Latency

In human-computer interaction, latency is defined as the delay between input action and output response [10]. As such, latency can be caused by various sources. For example, in a cloud-based web-application a user action might look like the following:

1. A user inputs text into a form.
2. The user submits the form and sends it to the server.
3. The server processes the update (e.g. saving it in the database).
4. The server responds to confirm the update.
5. The client displays the updated dataset.

All steps in this process can take time and add up to the total latency experienced by the user. While the client can be responsible for some of the latency, for example through render blocking client-side JavaScript code, most of the latency in traditional systems is caused by the request and response round-trip to the server. The client might hide this latency by using a pattern called Optimistic UI [11], where it already shows the updated dataset, even though the request is still in progress. But until the request is complete, there is still the possibility that the request might fail [1]. Thus, it still exposes the latency when an error occurs.

N. Tolia, D. G. Andersen, and M. Satyanarayanan [12] studied the effect of latency on user productivity with the following results: latencies below 150 ms do not impact user productivity. Between 150 ms and 1000 ms the latencies are noticeable, and the users becomes increasingly aware of the response time. For latencies above 1000 ms, users become unhappy and frustrated, leading to a drop in productivity [12].

A similar conclusion was reached by *J. Nielsen*, stating that 100 ms is the limit where users still feel that they are directly manipulating the UI and it is reacting instantaneously. A delay or latency of one second is the limit for the user's flow of thought to stay uninterrupted, but

[7] https://zero.rocicorp.dev
[8] https://jazz.tools

notice the delay. Above 10 seconds is the limit for keeping the user's attention without giving feedback indicating an expected time of completion [13].

3.3 Consistency Models

Data consistency refers to how and when observers see updates made to data in the system [14]. According to *M. van Steen and A. S. Tanenbaum*, a consistency model is a way to tell when local write operations are propagated to other processes in the system, and the effects for local read operations [7]. Consistency in this context is not the same consistency introduced in Atomicity, Consistency, Isolation, Durability (**ACID**), as the consistency in **ACID** relates to the guarantee that when a transaction finishes, the database is in a consistent state [14]. In the Consistency, Availability, Partition Tolerance (**CAP**) Theorem, consistency is defined by how a shared and replicated data appears as single, up-to-date copy [7].

M. van Steen and A. S. Tanenbaum describe multiple consistency models. For the context of this research strong and weak consistency are introduced. In a **strong consistency** system, all replicas will always return the most recent version of the data. When updates occur on one replica, all other replicas will reflect this change immediately.

If a system allows inconsistencies but convergence to a consistent state after no updates have taken place in a long time, the system is called **eventual consistent**, a version of weak consistency. Furthermore, if the system also ensures that even if there are conflicting updates across different replicas, it will still convergence to one state, it is **strong eventual consistent** [7].

3.4 Reactive Systems

The architectural patterns described in this section are often employed to build systems that adhere to the principles of the reactive paradigm. This paradigm proposed in the reactive manifesto by *J. Bonér, D. Farley, R. Kuhn, and M. Thompson* [15] defines four core traits for modern systems:

- **Responsive**: The system must respond in a timely manner
- **Resilient**: The system must stay responsive even in the face of failure
- **Elastic**: The system must stay responsive under varying workloads
- **Message-driven**: The system must rely on asynchronous message-passing to establish a clear boundary between components

While proposed for distributed systems, these traits do share similarities to the ideals and requirements of local-first software, which will be explained in Section 3.6. Two patterns for building such message-driven systems are Event Sourcing and Change Data Capture (**CDC**). Both event sourcing and **CDC** are patterns used to create a stream of events representing changes in an application's state. However, they differ fundamentally in their approach, the nature of the events they produce, and their impact on the application's design.

3.4.1 Event Sourcing

In event sourcing, the application's state is not stored in its current form. Instead, all changes to the state are stored as a log of domain events. These are high-level, business-relevant facts, such as "IssueCreated" [16]. This log is append-only and events are immutable. The application state is then derived by deterministically materializing the events into a state, which can then be used to render the application [17].

This enables the application to completely rebuild the application state by re-running the events from the log inside an empty application and allows the application to view the exact state the application was at any point in time [16].

The usage of an event sourced system also offers replay functionality, as the log can be reversed and then played back in a different order if events have been received in the wrong sequence, making it a powerful architecture for asynchronous systems [16].

However, it is also an intrusive pattern, as the entire application must be designed around the concept of producing these events.

3.4.2 Change Data Capture

In contrast, **CDC** is a non-intrusive pattern for capturing data changes. The application interacts with a traditional database, while the **CDC** process captures change events from a database transaction log and forwards them to other consumers. [18].

CDC allows the application state to be externalized and used to update external stores, while the application only needs to interact with the database [18].

3.5 Conflict Resolution

One of the seven ideals for local-first software, which will be introduced in full in Section 3.6, is seamless collaboration. This requires that several people can contribute to the same data store, document, or file. However, this may result in a conflict, which needs to be resolved either automatically by a conflict resolution strategy or manually by the user [1].

There are different strategies for resolving conflicts in a convergent way:

1. Give each **write** a unique ID. The write with the highest ID is kept, while other writes are discarded. If the ID is a timestamp, this approach is known as Last-Write-Wins (LWW). However, this will result in data loss, as writes are discarded.
2. Give each **replica** a unique ID. Writes from a replica with a higher ID have precedence overwrites from replicas with lower IDs. This also is prone to data loss.
3. Merge the values together. This is the only option that does not suffer from data loss, but might not always be possible, for example opposing Boolean values [17].

Another strategy is to record all the writes into a data structure that preserves all the information and resolve the conflict at a later time, for example by prompting the user [17].

3.5.1 Conflict-free Replicated Data Types

A Conflict-free Replicated Data Type (**CRDT**) is a data structure which provides the following properties:

1. Each replica can be modified without coordinating with other replicas.
2. Replicas that received the same set of updates reach the same state [19].

Resulting, **CRDTs** support concurrent modification while guaranteeing a convergence. This is done by attaching additional metadata to the data structure and providing an algorithm that handles conflicts deterministically [20].

3.5.2 In Event-Based Systems

While event sourcing (Section 3.4.1) and **CDC** (Section 3.4.2) are data modelling techniques and do not inherently include conflict resolution, they however provide a basis to write custom conflict resolution logic. Event sourcing can additionally provide additional context to the algorithm, as each event expresses the intent of the user.

A solution for conflict resolution in a system using some sort of event log, either from **CDC** or event sourcing, would be to determine a global total order of events. This can be done by the use of logical clocks or by applying a source-of-truth strategy, where one replica takes precedence over the other. If a total order is established, the Last Write Wins (**LWW**) approach can be applied [17]. As the event log is append-only and immutable, this would not result in data loss, as already resolved conflicts can be undone and resolved differently, by playing back the event log and committing a new event. [17].

3.6 Local-First Software

Local-first software is a concept for collaborative software, first introduced in 2019 by *M. Kleppmann, A. Wiggins, P. Van Hardenberg, and M. McGranaghan*, in which the advantages of cloud software, such as collaboration, always up-to-date applications and worldwide access, are implemented in local software to regain ownership and control of the data for the user [1].

Local-first software is defined by seven ideals [1]:

- **No loading**: the user should never have to wait for a request.
- **Multiple devices**: the user should have a way to seamless switch the device he is using without manually copying data on to another device.
- **Offline**: the application should keep working, even on a slow or no internet connection.
- **Collaboration**: the application should enable the user to collaborate with other users on the same document, in real-time or asynchronously.
- **Longevity**: the data created by the users should be accessible forever, even if the application service is not available any more.
- **Secure and private**: the user's data should only be accessible by the user and endangered by a security breach of a central database.
- **User control**: the user should have full access and control over their data.

In short, local-first software aims to provide the user with collaboration, privacy, and ownership of data. Furthermore, the software should feel fast and allow the user to work seamlessly on multiple devices.

However, in practice not every software that is local-first implements all seven ideals in full. Either because the use case does not comply with all seven ideals, as for example not every app needs collaboration, or for technical limitations, like to large datasets to save on the device.

Therefore, local-first software should be seen more as a spectrum, trying to implement all seven ideals, if feasible. In practice, especially ideals five (longevity) and seven (user control) are not high priorities for most businesses [21].

3.6.1 Local-First as a Distributed System

A networked computer system in which processes and resources are sufficiently spread across multiple computers is called a distributed system [7].

Local-first software is not inherently a distributed system, as the definition of local-first is quite broad. However, local-first software and distributed systems share similar challenges and requirements, but depending on the specific application, these are sometimes for different reasons. Most local-first applications need a distributed data store, to satisfy the **multi-device** ideal, yet not for availability reasons, but because the user expects each device to have the same data.

The difference can be seen when evaluating the eight fallacies of distributed systems for local-first software. These eight fallacies are [22]:

1. *The network is reliable.*
2. *Latency is zero.*
3. *Bandwidth is infinite.*
4. *The network is secure.*
5. *Topology does not change.*
6. *There is one administrator.*
7. *Transport cost is zero.*
8. *The network is homogeneous.*

Local-first software does not suffer from the same eight fallacies. From the perspective of the user, fallacies 1, 2, 5 and 8 are solved, since the access to the data is always local and no network is needed [21]. As the user only interacts with local data, their workflow is not confined by the reliability, latency, topology and homogeneously of the network. This also reduces the complexity required for developers to introduce new features in the application, as the need to consider how the data is transferred over the network is eliminated, thereby reducing the overall complexity of the application development process [23].

However, this is not the case for the whole application, as the complexity is just moved to the synchronization layer, which needs to synchronize these changes between devices and collaborators and is still exposed to these fallacies.

This is also stated by the local-first fallacies, proposed by *B. Zelenka*:

1. *Everybody is online* [21]:
   Not every user is always online and available for synchronization.
2. *Local-first does not suffer Tesler's Law* [21]:
   Tesler's Law states that for any system there is a certain amount of complexity that cannot be reduced [24]. In the local-first context the complexity is moved into the synchronization layer, which might be handled by a framework.
3. All programs are weakly consistent:
   Sometimes some problems in an application might not be weakly consistent [21]. For example, transferring funds in a banking application should always be handled through strong consistency, as the user needs to know their exact amount of funds before making a transfer [1].

3.6.2 Local-First as a Decentralized System

A decentralized system distinguishes itself from a distributed system, as processes and resources are **necessarily** spread across multiple computers [7]. While the motivation for a distributed system is the need to improve the performance of a single computer, a distributed system has in itself the need to spread across multiple computers. The system's continued operation relies on the collective contributions of its end-users [7].

Not every local-first software is a decentralized system, but local-first can be a fitting architecture to realize a local-first system. Since every client node can handle the business logic themselves, only data synchronizing between the nodes is needed, which can happen through a Peer-to-Peer (**P2P**)-System, removing the need for a central server.

However, this introduces additional complexity and challenges, like discovering new nodes [25] and **NAT** traversal [1]. Another actively research challenge in the **P2P** local-first software space is authentication, authorization, and identity, since without a central server, establishing trust between nodes can become complex [26].

3.6.3 Challenges

While local-first software addresses many problems associated with cloud software, it is not suitable for all types of applications. For example, software with real-time constraints like banking software might require strong consistency, making local-first software unfeasible [27]. However, even in strongly consistent applications, there might be some use-cases that can be improved by a local-first approach. In the banking software example, while making transfers and looking at balances should be strongly consist and therefore only done online, looking at past transactions can be solved by a local-first approach to make it available offline, as long as it is clearly communicated to the user that the information might not include all transaction.

Conflict Resolution

While conflict resolution is solved on a technical level in most local-first software, e.g. by using **CRDTs**, there is no guarantee that a merge of two local edits result in a semantically correct and meaningful result. Given an example where person A and person B both edit a replica while being offline. If person A changes the title from "Dogs" to "Birds" and person B changes the conclusion of the article, a **CRDT** will merge both edits correctly without a conflict (as shown in Figure 3). However, the title and the content of the article are semantically not correct, as the title does not describe the content correctly.

This challenge however is only a challenge in highly collaborative applications where multiple users edit the same documents simultaneously. For other types of applications, this challenge can be mostly irrelevant, as for example single-user applications the scenarios where a user commits drastic changes to the document on two different devices while being offline are quite slim.

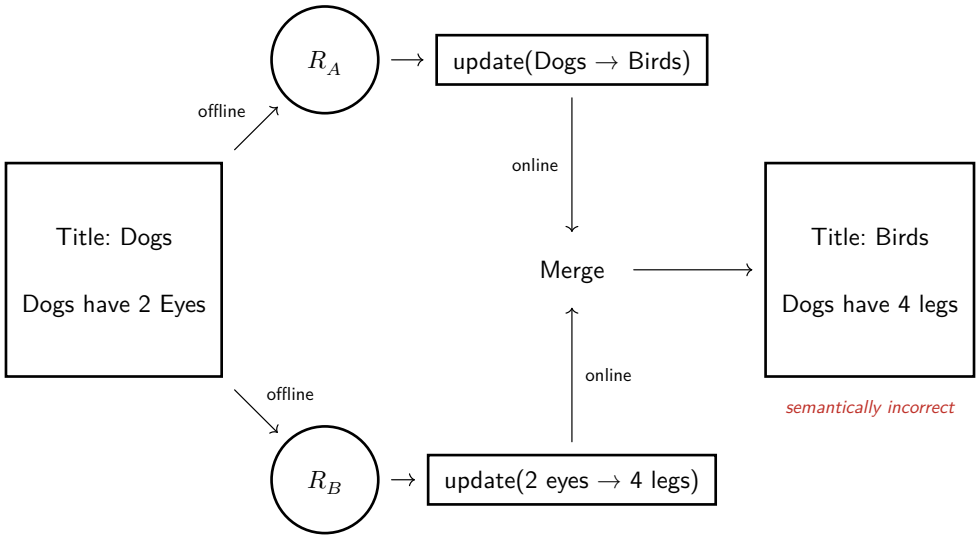

Figure 3: Example of technical correct merge that is semantically incorrect

Initial Load and Data Size

As local-first software uses a local copy of the data to operate, the first time it is used the whole dataset needs to be downloaded. Depending on the size of the dataset, this can take some time, where the user might be unable to use the software [27]. Additionally, some datasets are too large to even fit on the client device, making local-first challenging.

Security and Access Control

In cloud-based applications, security and access control is done on the server. In local-first software, the user needs a partition of the data on their local device. This means that the user should only be able to synchronize down the data he is allowed to have access to. Another challenge is how to revoke access to data that has already been synchronized to the device, if for example on user turns out to be a malicious actor [27].

4 Method

To answer the research questions, requirements for the synchronization layer were defined. These requirements are based on the past theory and research presented in Chapter 3. Using these requirements, an architecture was designed and implemented, using a real-world use case as the bases for the prototype. This prototype serves as the basis for the evaluation, and was compared against a second artifact, based on a client-server architecture.

4.1 Requirements

The requirements are grouped into local-first requirements, based on the ideals proposed in the original essay introducing local-first by *M. Kleppmann, A. Wiggins, P. Van Hardenberg, and M. McGranaghan* [1], and additional efficiency requirements for the synchronization layer.

R1 Using the app should feel instantaneous. To achieve that, the implementation should respond to input within 100 ms to feel instantaneous to the user [13].

This requirement is based on the first ideal of local-first software in Section 3.6, which states that using the software should feel fast.

R2 When offline, the application should still be functional. Reading data should work the same and most writes or changes should still work if no verification is needed by the external system.

This is the requirement to satisfy the third ideal, which states that apps should work while using a high latency network or being completely offline.

R3 The local features of the app should still work indefinitely if the remote **API** is disabled, with the last received data from the remote system, thus satisfy ideal five, the long now.

The other four local-first ideals, while still important for other local-first software research, were outside the scope of this project due to time-constraints, as stated in Section 4.3.

While these requirements ensure that the synchronization layer is sufficiently valid for a local-first application, additional requirements are needed to increase the efficiency of the synchronization layer.

R4 The user should be able to read their own writes immediately. This requirement supports **R2**, as offline writes require the user to read their own writes without having to synchronize first.

R5 The synchronization cost should be as low as possible. As most external **APIs** impose rate limiting [28] or pay per usage, sending requests to the **API** will inevitably generate some cost. Additionally, as JavaScript is a single-threaded language, running the synchronization too often will lower the compute time available to the application.

R6 The storage overhead required by the synchronization layer should be as low as possible. This includes the storage space needed for the event log, as well as the amount of additional synchronization metadata needed for each item. As web browsers limit the maximum amount of storage per domain (Example: Firefox storage quota is 10% of the disk size on which the profile is stored) [29], this is necessary to provide as much storage as possible to the application to store its data.

R7 The synchronization should minimise resource usage to still allow **R1** to be fulfilled during active synchronization operations. Because of the single-threaded nature of JavaScript, it is necessary to make sure that an active synchronization is not render-blocking.

A summary of these requirements can be found in Table 1.

R1	Respond to user input in under 100 ms
R2	Support offline reads and writes
R3	Work if remote **API** is disabled
R4	Read own writes
R5	Low synchronization cost
R6	Minimise storage requirements
R7	Active synchronization should not degrade application performance

Table 1: Requirements Summary

4.2 Prototype Use Case

To evaluate the requirements against a realistic use case, a prototype of an application has been developed. The chosen use case is a project management application, in which projects can be created and issues tracked. To provide external data for the synchronization layer, the GitHub **REST API** [30] has been chosen for its popularity and in-depth documentation.

This is inspired by popular project tracking applications like Jira[9], Asana[10] or Linear[11], which all feature synchronization between their respective platforms and GitHub.

The prototype allows the user to create projects and connect them with their respective GitHub repository. For each project, the user can see and edit issue details like status, title, or description. Issues and repositories are kept in synchronization with their external GitHub counterpart.

However, the prototype also provides local-only features to show how local-only data can be stored alongside synchronized data, without the risk of data loss. The user can create local-only projects, which do not synchronize to GitHub. Additionally, issues can have additional states apart from the two states provided by GitHub. Issues can additionally also have different priorities. However, it should be noted that these local features have been incorporated solely for the purpose of contextual enhancement, as they are not relevant for the research component of this thesis.

4.3 Beyond the Scope

The primary objective of this thesis is to develop an architecture that allows local applications to synchronize their data with an external data source via an **API**. To maintain a clear focus on this objective, the scope of the investigation must be defined. The following areas are deliberately excluded from the scope of this work.

User Experience and Interface Design

While a prototype was developed to validate the synchronization layer, a comprehensive user experience design process was not undertaken.

[9] https://www.atlassian.com/software/jira
[10] https://asana.com
[11] https://linear.app/

Multi-Device or Multi-User collaboration

The complexities of multi-device and multi-user collaboration represent a substantial research area in their own right. This thesis establishes a foundation for a single-user local-first functionality, leaving the vast topic of collaboration for future work.

Security and privacy

Security, privacy, and encryption is an ongoing research area in the local-first space. A security or privacy analysis, which would involve threat modelling, penetration testing and other security examinations is a specialized effort that is beyond the defined boundaries of this research.

API Versions

Additionally, handling **API** version changes and optimisations for multiple different **API** types beyond the selected example (**REST API**) will remain outside the scope of this research.

Detecting Remote Updates between full Datasets

Detecting updated data from two different full snapshots is also not investigated by this research, as the chosen example **API** (GitHub) can already return only updated items after a given timestamp and the focus of this research was on how to enable offline writes to be synchronized.

5 Design

Based on the requirements presented in the preceding chapter, this chapter presents the foundational architecture designed to address the central research problem of enabling local-first capabilities for an application that must synchronize with an external third-party **API**. This architecture was developed through a structured design process, beginning with an evaluation of potential solutions before the final architecture was specified.

Accordingly, this chapter is organized into three main sections. First, it introduces and analyses two competing approaches and justifies the chosen approach. Then, the most substantial part of the chapter offers a detailed exposition of the chosen architecture, its core components, and its operational logic. Finally, the chapter concludes with a presentation of the prototype that was developed to demonstrate the design.

5.1 Suggested Solutions

Two distinct architectural solutions were theorized based on established principles of local-first software and the related work presented in Chapter 2. The following sections will describe these solutions and outline their respective mechanisms, providing a basis for comparison.

5.1.1 Proxy Approach

The first solution is based on a proxy approach, like Speed Kit [3], but adapted for a local-first architecture. In this proxy approach the application would send a request to the external **API** like it would in a client-server architecture. A on-device proxy would intercept the request and save the response from the external **API** in a local data store on the device, as shown in Figure 4.

With this approach the data is stored in two separate locations. The local only data is stored in the database chosen by the application and the proxy stores a snapshot of the remote data to enable offline reads. The local snapshot is saved with a **TTL**. If the application requests data while being online, the proxy will check if the data in the local snapshot is still inside the **TTL**. If so, the local snapshot will be returned, if not it will be fetched from the remote **API** and the response then used to update the local snapshot. This approach, however, increases latency compared to a direct local read, as it may require a network roundtrip to the external

server. While offline, the local snapshot will be returned no matter the **TTL**, as there is no remote available to update the snapshot.

Online writes are directly sent to the remote **API**, and the response is used to update the local snapshot of the data. However, offline writes can be difficult, as the proxy must cache the writes while also changing the local snapshot of the remote data to allow the application to read its own writes. The application also has no method to force synchronization, as it only sends requests and is under the impression that it is interacting directly with the remote **API**.

Additionally, the storage space needed for the local snapshots can grow quite large, as the whole **API** response must be saved locally, even though the application might only require a small subset of the response.

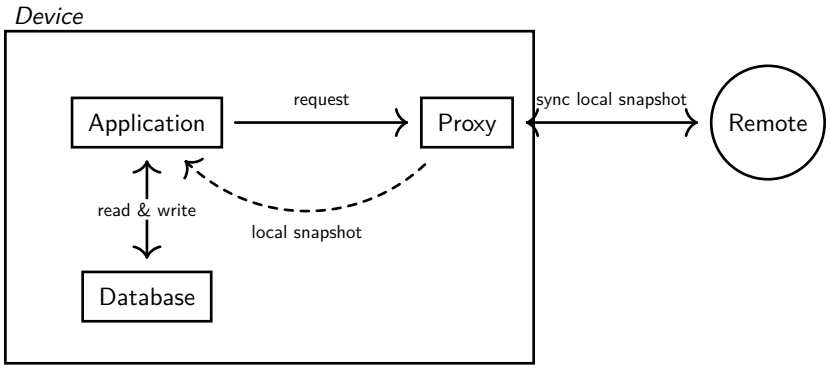

Figure 4: Proxy Approach

5.1.2 Synchronization Layer Approach

In the synchronization layer approach, the application interacts with the local data store like it would in a local-first approach. The synchronization layer is then responsible to synchronize the local database directly with the remote **API**, as shown in Figure 5. The application developer therefore does not interact with the **API** directly, as the synchronization layer completely handles updates from and to the remote system.

This allows the application to read data directly from the data store, as the synchronized data is present alongside the local only data. Creating or updating data is also done directly inside the data store, as writes are first performed in the local store and then sent in bulk to the remote **API** at a specified synchronization interval.

A live query allows the application to display synchronized results immediately, as it subscribes to the data store and re-renders the UI if changes appear. The application does not know about the existence of the synchronization layer, since interacting with synchronized data or local-only data is done in the same way.

Additionally, the synchronization layer only stores the relevant subset of the **API** response, that is required by the application.

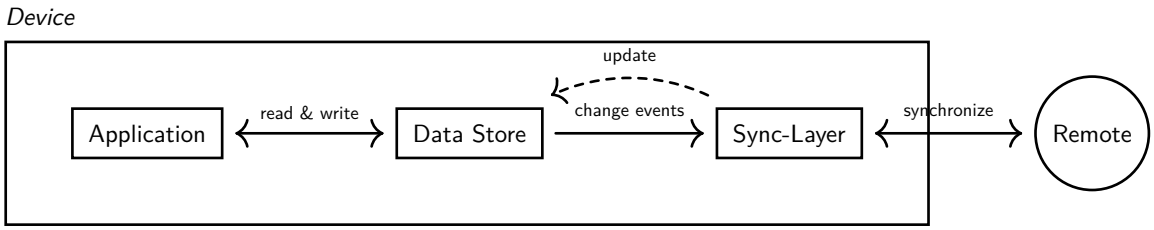

Figure 5: Synchronization Layer Approach

5.1.3 Approach Chosen

For the purposes of this research, the synchronization layer approach was chosen. While the proxy approach may appear more simplistic, a deeper analysis reveals substantial additional limitations. The synchronization layer approach was selected due to its advantages in the following key areas.

Performance

In the proxy approach, remote data needs to be merged with the local data for every request before it can be displayed. This will result in additional computation needed for each request, resulting in higher latency as this work will be done on the main thread, thereby possibly blocking other computation and the rendering of the UI.

Control and Observability

Furthermore, the proxy approach provides no mechanism for the application to explicitly control or observe the synchronization state, which can lead to suboptimal user experience as the user is provided no context if or why their information might be stale.

Data Modelling Complexity

The proxy approach also has limited ability to provide partial synchronization. The following example, drawn from the selected use case (Section 4.2), illustrates this point: The user creates

two projects, but only one of them should be synchronized with GitHub. In the synchronization layer approach, the local-only project is marked as such, meaning it will be ignored by the synchronization layer. Both projects use the same database schema and tables to render the user interface. However, in the proxy approach, only the synchronized project is stored within the proxy, with additional data (priority and additional issue states) stored in the application database. All of the local project's data has to be stored in the application database, as the proxy has no knowledge of the project, creating additional complexity for the application developer, who needs to accommodate this in their database schema and application logic by making database fields optional or storing the data in different tables.

Storage Efficiency

Lastly, since one of the requirements in Section 4.1 is to minimise the storage overhead needed for synchronization, the proxy layer's need to store full snapshots of the API response will not meet this requirement.

The following Table 2 summarizes why the synchronization layer was ultimately chosen by comparing the approaches on how the meet the defined requirements.

Requirement	Proxy	Synchronization Layer
R1: Respond to user input in under 100 ms	◐[12]	●
R2: Support offline reads and writes	◐, (complex offline writes)	●
R3: Work if remote **API** is disabled	◐, (see **R2**)	●
R4: Read own writes	◐, (see **R2**)	●
R5: Low synchronization cost	●	●
R6: Minimise storage requirements	○, (saves unnecessary data from response)	●

[12] if the TTL of the local data is expired, the request is sent to the remote, which can result in higher response time

Requirement	Proxy	Synchronization Layer
R7: Active synchronization should not degrade application performance	●	●

● Fully meets requirement; ◐ Partially meets or complex to implement;
○ Does not meet requirement; ● Depending on implementation

Table 2: Comparative Evaluation of Architectures

5.2 Architecture

The designed architecture in this research is based on two components. Necessary synchronization metadata is stored in a local data store, which is included into the application database. The synchronization object is responsible for scheduling and performing the necessary synchronization operations.

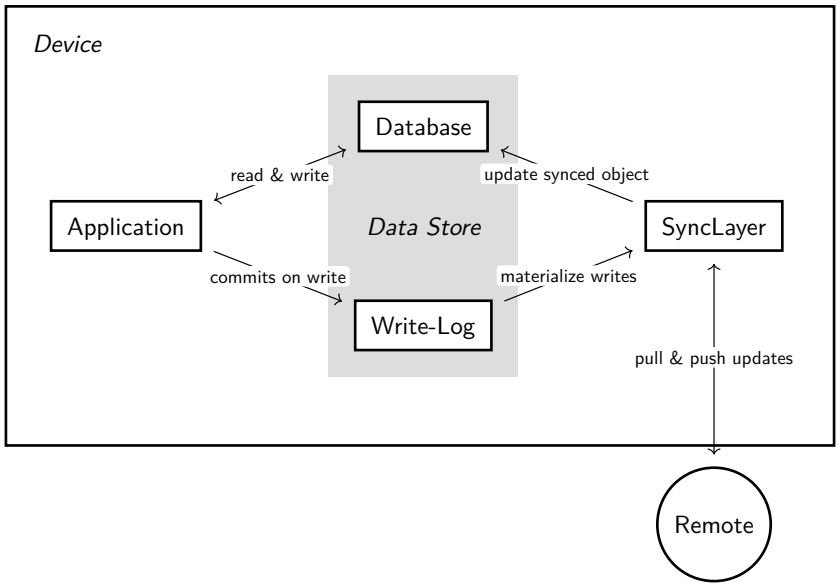

Figure 6: High Level Architecture of an Application using the Synchronization Layer

5.2.1 Data Store

The data store consists of a write-log table, which logs all local writes between synchronizations. It stores the updated fields with the old and new data for each write. Additionally, for

each item, metadata is added to track each object with the remote counterpart. Furthermore, the data store also stores the last synchronization timestamp for each table. The write-log is inspired by log-based **CDC** (Section 3.4.2), however it only captures events about items that need to be synchronized. If the data store provides a replication log, it can be used to generate the write-log. All events in the write-log save the change method, ID and table of the object, as well as the delta of the changes. Assuming the title of an issue was updated from "Old Title" to "New Title", the committed event would look like Listing 1.

```
{
  "number": 1,
  "object_id": 1,
  "object_table": "issues",
  "method": "update",
  "old_data": {
    "title": "Old Title"
  } ,
  "new_data": {
    "title": "New Title"
  }
}
```

Listing 1: JSON representation of a write-log event

The `number` of the event is used to order the event log and is incrementally increased as new events are committed. The fields `object_id` and `object_table` are needed to determine the specific database row that was updated. `method` provides context for the synchronization layer, as it needs to know which endpoint to send the request. `old_data` and `new_data` only log the delta of changes to keep the event log as small as possible.

5.2.2 Synchronization Object

The synchronization object is responsible for the whole synchronization logic. This section explains all the necessary task performed by the synchronization object to keep the local data store and remote system in synchronization on a higher level. The specific implementation is explained in Section 6.2.2.

Connection Status

If the application is offline, the synchronization object will pause synchronization until the connection is online again.

Scheduling

The synchronization object will schedule individual synchronization tasks for each synchronized table of the data store. For each table, an individual synchronization interval and one of two synchronization types (read-only or full) can be set. Additionally, a table can be excluded from automatic interval-based synchronization by marking it as manual synchronization only. In the case that the last synchronization is older than the specified interval, a new task is started asynchronously to synchronize the table.

Transforming the data

Because the data store and remote **API** may employ vastly different data schemas, the synchronization object must transform the data from the remote schema to the local schema and vice-versa. However, the application developer is tasked with providing the synchronization object with a function to transform the data, as it has no knowledge about the context of the data. This function can consist of one-to-one mappings of differently named fields (e.g. a remote `body` field to a local `description` field) or more complex transformation. For example, in the application prototype developed in this research, an issue can be in one of 4 states (*Done*, *Open*, *In Progress*, *Backlog*), while GitHub issue can only be *open* or *closed*, therefore a transformation must be done between the formats.

Get and categorize Changes

All items that have at least one local or remote change, are categorized with one of the following categories described in Table 3.

Category	Example using Prototype	Action needed
New Remote	Issue was created on GitHub	Save locally
New Local	Issue was created using the local-first application	Push to remote
Update Remote	Updated the title of an issue on GitHub	Update local item

Category	Example using Prototype	Action needed
Update Local	Updated the title of an issue in the local-first application	Push changes to remote item
Deleted by Remote	Deleted an issue on GitHub	Delete local item
Deleted by Local	Deleted an issue in the local-first application	Delete remote item
Conflicts	Changed the title of an issue on GitHub and the description in the local-first application	Merge changes and update local and remote item

Table 3: Categories of changes

This is done with a deterministic two-pass algorithm, which classifies every modified item into one of the categories in Table 3. In the first pass, it iterates through the collection of items received from the remote **API**. For each item, it determines its category relative to the local data:

- **Deleted by Remote**: If the item is deleted by the remote.
- **New Remote**: If no corresponding local item can be found, the item is new to the system.
- **Update Remote**: If a local item exists but has no corresponding write-log entries, the item has only been changed on the remote system.
- **Conflict**: If a local item exists and the local item also has entries in the write-log, it signifies that the item has been modified by both systems since the last synchronization

The second pass iterates over the write-log entries for each item. If an item has already been classified by the first pass, it will be skipped as these have already been categorized as **Conflict**. For the remaining items, which represent purely local modifications, it performs the following categorization:

- **Deleted by local**: If the write-log for the item contains a delete event, it was deleted locally. If the write-log does also contain a create event it will not be categorized, as it has been created and deleted between synchronizations, so no update to the remote system is needed.
- **New Local**: If the write-log contains a create event for the item, it was created locally and has not been published to the remote.

- **Update Local**: If the write-log for the item only contains update event it has been updated locally without a corresponding remote change.

Generating field-exact change

The remote **API** only returns the items changed since the last synchronization. However, it always returns a snapshot of the full item including all fields, regardless of if the specific field was updated or not. To merge the remote and local changes however, a changeset is needed that only includes the actual changed fields since the last synchronization.

If there are no local changes, this can be done by comparing the remote item with the item in the database. However, if the item was also updated locally since the last synchronization, this is not feasible, as it could overwrite the local change.

A solution would be to additionally save the last received snapshot of the data, however this would result in increase of storage required. However, since the `old_data` field is stored into the write-log, the state of the item at the last synchronization can be materialized by undoing events in the log, as shown in Figure 7 ($n \rightarrow n-1$). The materialized item is then used to compare against the remote item and a field exact changeset is generated.

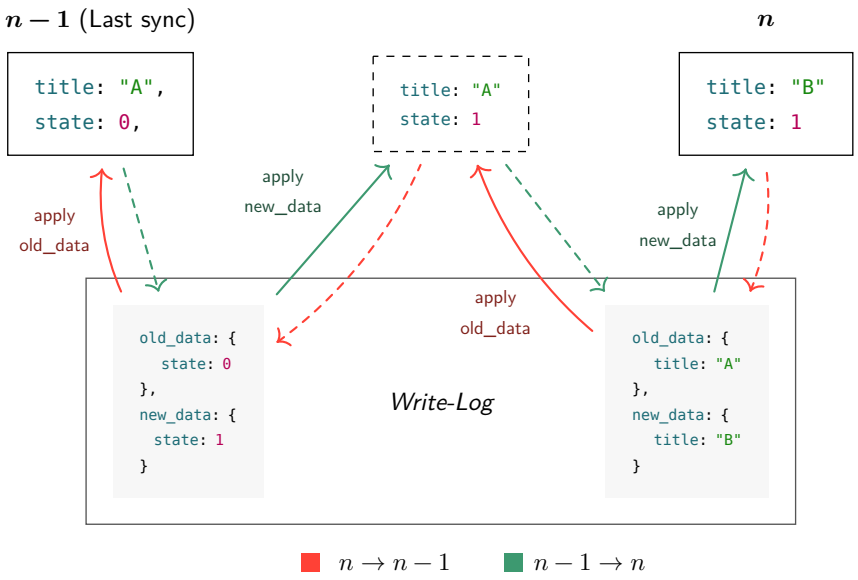

Figure 7: Materializing the write-log

Conflict Resolution

Advanced technologies like **CRDTs** offer automated merging, however, they need to be used by all replicas in the system, which is not available when synchronizing with a third-party **API**. For this research, the write-log was used as a basis for custom conflict resolution, like the explanation in Section 3.5.2. The changes from the remote were turned into write events, a total order was determined and a **LWW** was enforced. For the prototype, a source-of-truth policy was adopted for the total order, with GitHub determined as the source of truth, meaning changes from GitHub were inserted, and therefore also applied, **after** the local writes. As a result, if both GitHub and the local application change a field on an issue, the GitHub change will always be given precedence, no matter when they were made. This was chosen as the application is only used by a single user, while GitHub can be used by multiple users, to avoid overwriting updates on the remote by other users. However, this is purely a use-case decision, not a universal rule for the synchronization layer and could be implemented differently by other applications in the future just by defining a different total order. If the application and GitHub change different fields, both changes will be merged.

Pushing updates to the Remote

Pushing the updated data to the remote system is done using a queue. Items of the categories "New Local", "Updated Local", "Deleted Local" and "Conflicts" (see Table 3) are put on the queue to be sent to the **API**. The queues pushes the updates asynchronously, with multiple active requests at once, while still respecting the rate-limits imposed by the **API**.

Applying changes to the local data store

At the end of the synchronization task for a table, the local data store is updated with two sets of information: remote modifications determined in the first pass of the algorithm, as well as with the return data from the pushQueue for information that the remote system exclusively creates. This includes the `remote_id`, which serves to assign remote updates to their local item.

Clearing Write-Log

After successful synchronization of the table, the write-log can be reset to free storage on the device.

5.2.3 Example Synchronization of an Issue

To illustrate a synchronization with an example, Figure 8 shows how one issue is synchronized between local and remote. Writes are done directly on the database, which then internally logs them to the write-log. At the start of a synchronization, all updates from the remote are pulled down. To obtain the original state of the issue, all writes are undone, by materializing all `old_data` fields (see Listing 1 and Figure 7).

The original state of the issue is then compared against the issue provided by GitHub, to generate a delta change. This is needed as GitHub only provides a complete snapshot of the issue, which cannot be used to merge the local and remote edits together. As GitHub is the chosen source of truth, this delta is then added to the end of the write-log and applied on top of the local edits, resulting in the final issue object.

This issue is then pushed to the remote GitHub **API** as well as to the local data store to update the application. After successful synchronization, the write-log is cleared to save storage.

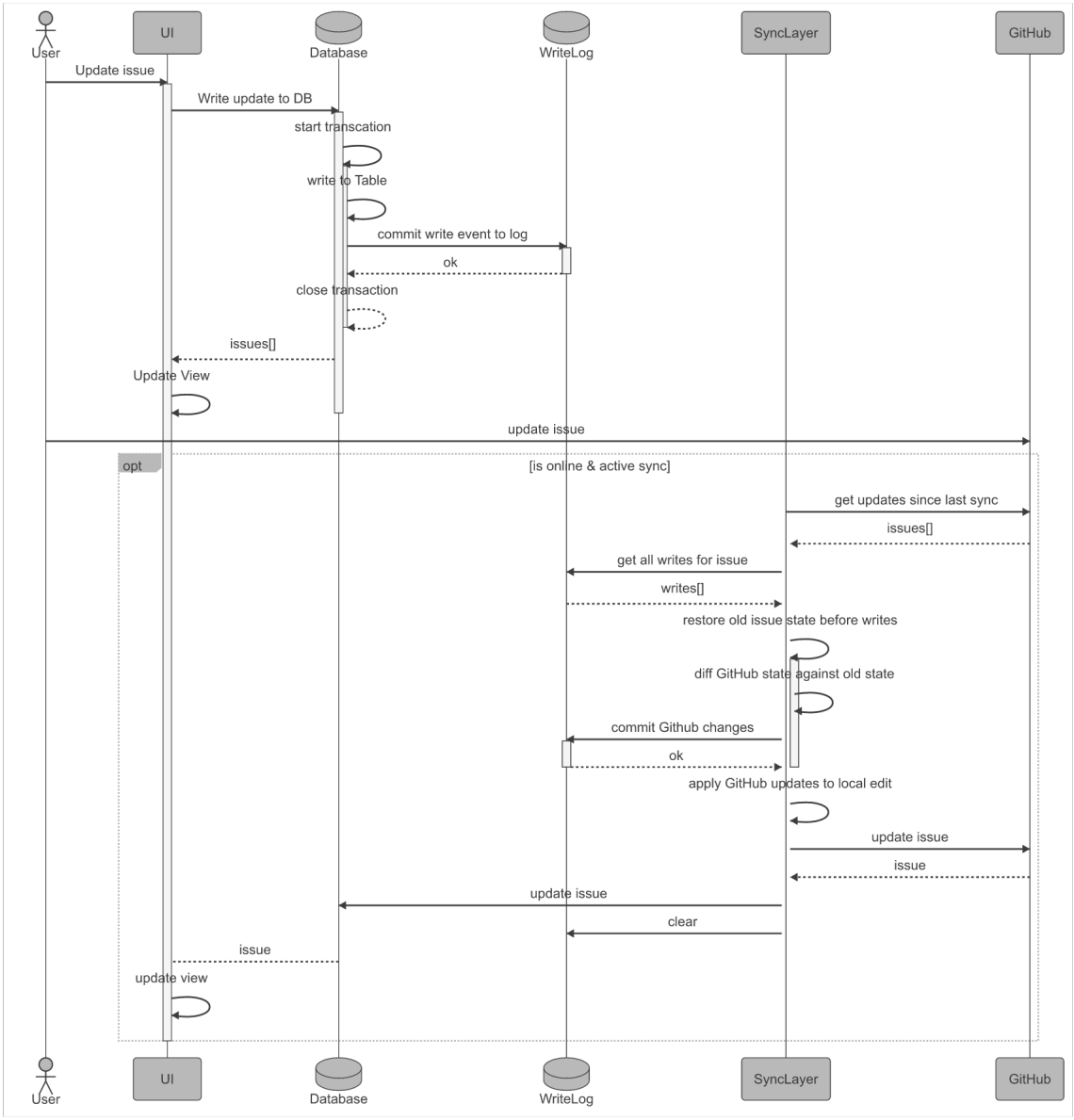

Figure 8: Example of an issue synchronizing between GitHub and local

6 Implementation

Having established the architectural principles in Section 5.2, the implementation details of the synchronization layer and application prototype will now be presented.

6.1 Technology Stack

The prototype application was developed as a Single-Page Application (**SPA**) to be used in modern browsers targeting Baseline 2025[13], a label for feature compatibility across web browsers. Technology choices for both the application and synchronization layer were influenced by bundle size and performance, due to limited resource availability in browsers for storage and memory, and to decrease initial load size. Dependencies were only added if it seemed necessary, using native JavaScript and browser standards wherever possible, resulting in only two runtime dependencies (Dexie.js and TinyTick). TypeScript was chosen as the language to provide type safety and increase the developer experience.

6.1.1 Database

For the database, IndexedDB has been chosen as it is a browser standard implemented by all modern browsers [31]. For additional developer experience, Dexie.js[14], a minimalistic wrapper for IndexedDB, has been chosen. Its support for reactive observable queries enables the application to automatically re-render if remote data is added to the database [32], which IndexedDB does not support natively.

Dexie.js provides a simple API to create tables and interact with data. A simple Dexie.js database is shown in [33]. Operating on the Database is done through simple Create, Read, Update, Delete (**CRUD**) functions, an example provided by [33].

[13] https://web.dev/baseline/2025
[14] https://dexie.org

```
import Dexie, { type EntityTable } from 'dexie';

interface Issue {...}

const db = new Dexie('IssueDatabase') as Dexie & {
  issue: EntityTable<
    Issue,
    'id'
  >;
};

db.version(1).stores({
  issue: '++id'
});
```

Listing 2: Create a Dexie.js database

```
await db.issue.add({id: 1, title: "Update Dependencies", status: 0})
await db.issue.update(1, {status: 1});
await db.issue.delete(1);
```

Listing 3: CRUD functions

6.1.2 Framework

Svelte[15] and its meta-framework SvelteKit[16] have been chosen as the framework for its fast reactive signal-based state, which is achieved by the usage of a compiler [34]. The compiler additionally compiles only the needed library code alongside the application code, resulting in smaller bundle, with a sample Svelte's app compressed bundle size of 7.3 kBs compared to 49.6 kBs in React for the same application [35].

```
<script>
  let count = $state(0);
</script>

<button onclick={() => count++}>
```

[15] https://svelte.dev
[16] https://svelte.dev/docs/kit

```
  clicks: {count}
</button>
```

Listing 4: Simple incremental counter using Svelte

6.1.3 Task Orchestration

For orchestrating task inside the synchronization layer, the task orchestration library TinyTick[17] was chosen for its minimal bundle size of 2.8kB gzipped [36].

6.2 Synchronization Layer

The Implementation of the synchronization layer consists of three parts. The database adapter, which establishes the write-log, the general synchronization object, responsible for all task that can be generalized between multiple **APIs**, and an interchangeable **API** adapter, used to interact with the chosen **API**.

6.2.1 Database Adapter

As IndexedDB does not provide a replication log, a Dexie.js add-on was developed to create a write log and automatically commit events to it if synchronized stores are updated. This add-on allows the application developer to mark tables as synchronized tables by extending the default database creation (Listing 2), as shown in Listing 5.

```
import { X_Sync_Addon_Dexie } from '$lib/x-sync';
import { Dexie, type EntityTable } from 'dexie';

interface Issue {...}

const db = new Dexie('IssueDatabase', {
  addons: [X_Sync_Addon_Dexie], // Adding the Addon
}) as Dexie & {
  issues: EntityTable<Issue, 'id'>;
};

db.version(1).stores(
  {
```

[17] https://tinytick.org/

```
    issues: '++id, github_number, user',
    local_table: "++id"
  },
  { sync: ['issues'] }, // Mark table issues for synchronization
);
```

Listing 5: Create a Dexie.js database with the addon. `issues` will be synchronized

To create the write-log, the add-on overrides the `db.version(1).stores()` method to add the additional stores `_writeLog` and `_syncState` which is used to persist the synchronization state for tables. Additionally, the schema of each synchronized table is extended with the `remote_id` field, which is indexed to query items by their `remote_id`.

Furthermore, the add-on overrides the `create()`, `update()`, `delete()` methods to automatically commit events to the write-log if they are used to update a synchronized table. The implementation for overriding the `add()` method is shown in Listing 6. To focus on the key logic, boilerplate code has been omitted for clarity.

```
db.Table.prototype.add = Dexie.override(
  db.Table.prototype.add,
  (original) =>
    async function (this: Dexie.Table, ...args: unknown[]) {
      ...
      const isSyncedStore = db._syncedStores.includes(this.name);
      if (!isSyncedStore) {
        return original.apply(this, args);
      }
      const result = db.transaction(
        'rw',
        ['_writeLog', this.name],
        async () => {
          const original_res = await original.apply(this, args);
          await db._writeLog.add({
            object_id: result,
            table: this.name,
            method: 'create',
            old_data: null,
            new_data: args[0] as object,
          });
          return original_res;
```

```
            },
          );
          return result;
        },
    );
```

Listing 6: Overriding the `db.TABLE.add()` method for sync

When the function is used for a synchronized store, the call to the original implementation, as well as the creation of a new write-log event, is wrapped within a single database transaction. This use of a transaction is critical, as it guarantees that the creation of a new item and its corresponding write-log event are an atomic operation. If either part fails, the entire transaction is rolled back [37], preventing the database from ever entering an inconsistent state where a change has been made, but not logged, which is essential for the synchronization process and the recreation of the original state needed for generating the field-exact change, described in Section 5.2.2.

6.2.2 Generic Synchronization Object

The architectural design from Chapter 5 is realized in a reusable TypeScript class `SyncBase`. This abstract class encapsulates the entire synchronization logic. To target a specific third-party **API**, developers extend `SyncBase` and implement its abstract methods (`pullData`, `pushCreate`, etc.), thereby creating a specialized adapter. Figure 9 illustrates this relationship (Internal helper methods and attributes have been omitted for clarity). The abstract `SyncBase` defines the protocol for pulling and pushing data, while a concrete subclass (`GitHubSync`) provides the **API**-specific implementations. This pattern yields a modular interface that can be reused across multiple third-party endpoints without modifying the synchronization core.

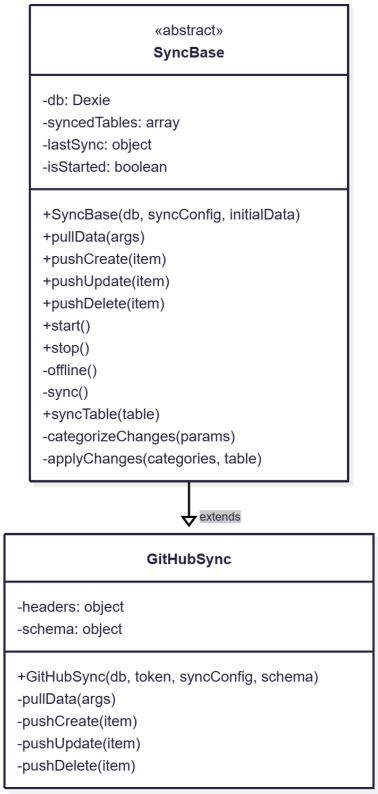

Figure 9: General Synchronization Object and API adapter.

Lifecycle and Scheduling

The `SyncBase` class handles the entire lifecycle of the synchronization process.

Synchronization scheduling is done by a dispatcher method `sync()`, which is scheduled to run every ten seconds using the TinyTick manager. The `sync()` method iterates through all synchronized table and schedules a `syncTable(table)` task if the configurable synchronization interval of the table (defaulting to 5 minutes) has elapsed.

Connection status is checked to prevent failed network requests due to flaky network conditions. It uses the `window.navigator.onLine` Boolean property provided by the browser as a primary check [38]. An enhancement, it also leverages the `window.navigator.connection.effectiveType` property of the Network Information API [39], which at the time of publishing are only browsers based on the Chromium Project[18], to pause synchronization on slow or unstable networks, improving the system's robustness. The imple-

[18] https://www.chromium.org

mentation currently only executes the synchronization if the `effectiveType` is `4g`, meaning a consistent Round-Trip-Time (**RTT**) below 270 ms and a minimum downlink of 700 Kbps [40]. This is done to avoid the failing of the synchronization requests due to flaky network conditions. The lifecycle automatically resumes when a stable connection is re-established.

Core Synchronization Workflow

The `syncTable(table)` is the core method orchestrating the synchronization process. When executed either by the `sync()` dispatcher or manually by the application, it performs a sequence of high-level operations:

1. **Data Fetching**: It fetches all the data from the remote **API** using the abstract `pullData()` function and retrieves local modifications for the table from the write-log.

2. **Change Categorization**: These two datasets are then processed using the two-pass algorithm described in Section 5.2.2, classifying every modified item into a specific category. For the items in the **Conflicts**, **Update Remote** and **Update Local** categories the final, resolved field-exact changeset inserted into the list. For items of the **New Remote** category, the application can also provide initial data for fields that do not exist on the remote system, for example the priority of the issue in the implemented prototype. For the **New Local** items, the list contains the local item from the database. As the prototype of the use case does not implement destructive (delete) actions by either system, the categorization of deletes is currently not implemented.

3. **Applying the Changes**: Finally, it passes the resulting categories object to the `applyChanges()` method to execute the necessary updates.

The `applyChanges()` method, shown in Listing 7, orchestrates the final phase. It uses the `PushQueue`, a simple asynchronous queue that accepts asynchronous functions (`pushCreate()` and `pushUpdate()`) and resolves a promise when all items resolve. It handles rate limiting by only pushing a certain amount of items concurrently with a total limit per minute, both of which can be adjusted, to stay under the secondary rate limits exposed by GitHub [28]. The **API** adapter can return necessary updates to the local system based on the response data of the **API**, for example adding the `remote_id` to **New Local** items after successful creation on the remote, which is added to the `localUpdatesFromRemote` array. This array is then applied to the local data alongside the changes from the categories **New Remote**, **Update Remote** and **Conflicts**, inside a single atomic transaction to ensure data consistency.

After successfully applying the changes, the write-log is cleared to recover storage on the device, as the entries are no longer needed.

```
// Simplified for clarity
async function applyChanges(categories, table) {
  const localUpdatesFromRemote = [];

  ...

  // 10 concurrent, 80 per minute to stay under GitHubs Rate Limit
  const pushQueue = new PushQueue(10, 80);

  for (const item of categories.newLocal) {
    pushQueue.add(async () => {
      return pushCreate({...}).then((result) => {
        localUpdatesFromRemote.push(result);
        return result;
      });
    });
  }

  ...

  // Mark queue as done and wait for all operations to complete
  await pushQueue.done();

  await this.db.transaction('rw', localTable, async () => {
    db[table].bulkAdd(newRemote);
    db[table].bulkUpdate(updatedRemote);
    if (localUpdatesFromRemote.length > 0) {
      db[table].bulkUpdate(localUpdatesFromRemote);
    }
    db[table].bulkUpdate(conflicts);
  });
}
```

Listing 7: Applying the changes

Supporting Algorithms for Change Resolution

The core workflow relies on several specialized helper algorithms to correctly resolve changes.

- **Materialize Changes**: To turn multiple write log updates into one field-exact change object, the write-log must be materialized as shown in Figure 7 ($n-1 \rightarrow n$). This is done by a simple reduce function, displayed in Listing 8.

```
const data = writeLog.reduce(
  (acc, event) => Object.assign(acc, event.new_data),
  {},
 );
```

Listing 8: Materializing the write-log

- **Restoring Initial Item**: For items that have both local and remote changes, the initial state of the item during the last synchronization must be restored. This can be done by walking back the write-log, shown in Figure 7 ($n \rightarrow n-1$). This is be done by reversing the write-log before materializing like Listing 8 and then applying it on top of the current state of the object.

- **Generate Field-Exact Change**: Generating the field-exact change between two items A and B is done by a simple key-value difference algorithm. It works by comparing each key-value pair of both items and appending the key-value pair of B if it does not match the key-value pair of A. The implementation of this algorithm is a simple for loop over all keys of item B, illustrated by Listing 9.

```
  let differences = {};

  for (const key in B) {
    if (Object.hasOwn(B, key)) {
      if (Object.hasOwn(A, key) && A[key] !== B[key]) {
        differences[key] = B[key];
      }
    }
  }
}
```

Listing 9: Difference algorithm used to generate field-exact change

- **Conflict Resolution**: To resolve the remote and local changes to a single object, three steps are followed:

1. Restore the initial Item
2. Generate field-exact change between the initial item and the remote item
3. Append the change to the write-log, as GitHub is determined to be the source-of-truth.
4. Materializing the new write-log to generate the resolved changes object

6.2.3 GitHub API Adapter

In the synchronization layer, **API** Adapters are responsible with implementing the **API** specific logic of the `SyncBase` class. Each **API** needs its own adapter implementation, which can be shared across multiple applications using the same remote **API**. For the GitHub **REST API**, the class `GitHubSync` extending the abstract `SyncBase` class was developed. It provides the specific logic required to pull and push data to GitHub's endpoints, handles authentication and data transformation.

The design of the adapter is centred around a strongly typed `schema` configuration object, provided by the application developer. This object defines how items of local tables map to remote entities (e.g. issues, repositories).

Schema Object

This `schema` object requires the application developer to define several key properties for each remote entity the application uses.

- **Table name**: Maps the remote entity (e.g. issues) to the actual table name in the local database.
- **Bidirectional data lens**: The application developer provides the adapter with bidirectional data lens to transform the remote data into the local format and vice-versa. It consists of:
  `toLocal(apiResponse)`: transforms a raw **API** JSON-result into the local schema used by the database. This is where logic such as mapping `remote.body` to `local.description` or converting GitHub's state ("open" or "closed") to the application's four-state status system occurs, as previously stated in Chapter 5, Subsection "Transforming the data."
  `toRemote(localItem)`: performing the reverse action, turning a local change object or database item into the format expected by the remote **API**.
- **API-specific logic:** The adapter can also require additional logic to be provided by the application developer. For instance, the GitHub issues are nested under repositories, requiring the names and owners of these repositories. In the `GitHubSync` adapter the application developer therefore must provide a `repos_to_fetch()` function, returning the names and owners of all repositories whose issues the application wants to synchronize.

Implementing Abstract Methods

In the adapter, the abstract methods defined inside the `SyncBase` object need to be implemented. This subsection describes how these are implemented inside of the specific `GitHubSync` adapter.

`pullData()` is responsible for fetching data from GitHub. The `SyncBase` class provides the arguments `tablename` and `since` (timestamp of the last synchronization). A switch statement based on the table name is used to call the correct fetching logic. In the example in Listing 10, it first calls the `repos_to_fetch()` function provided by the application on initialisation inside the `schema` object to get a list of all repositories on which issues should be fetched. It then initializes parallel requests to fetch the issues for each repository. It uses a `paginatedFetch()` helper function to automatically fetch all pages, as the GitHub API pages the results in batches of 30 per default [41]. For each item returned by the **API** it calls the `toLocal()` function to transform it into the local format.

`pushCreate()` and `pushUpdate()` operate in a similar way, by first turning the local format from the database, provided by `SyncBase`, into the format expected by GitHub using `toRemote()`. The application developer must additionally provide a `getRepo()` function, returning the corresponding repo name and user required for the `POST` or `PATCH` request to the GitHub **API**.

```
async function pullData({table, since}) {
 switch (table) {
   case schema.issues.tableName: {
     const repos = await this.schema.issues.repos_to_fetch();
     const issues = await Promise.all(
       repos.map(
       (repo) => paginatedFetch(`/${repo}/issues?since=${since}`,
           {...}
         )
       .then((res) => {
           return res.map((issue: RemoteIssue) =>
             this.schema.issues.toLocal(issue, repo),
           );
         }),
       ),
     )
     ...
     return issues;
   }
```

```
    }
  }
```

Listing 10: Fetching issues using `pullData()`

6.3 Prototype Application: GitHub Issue Tracker

The application allows the user to create projects. A project can be connected with a GitHub repository, marking it as a synchronized project. Figure 10 shows a newly created Project that was connected with the GitHub repository "thesis-test-static-10", which already contains ten issues. Inside a project, issues can be created with a title, description, status (one of "Open", "In Progress", "Done", "Backlog") and priority ("Low", "Medium", "High"). The user can create new projects, start and stop the synchronization as well trigger synchronization manually. Issues can be created and updated, which will be synchronized with GitHub.

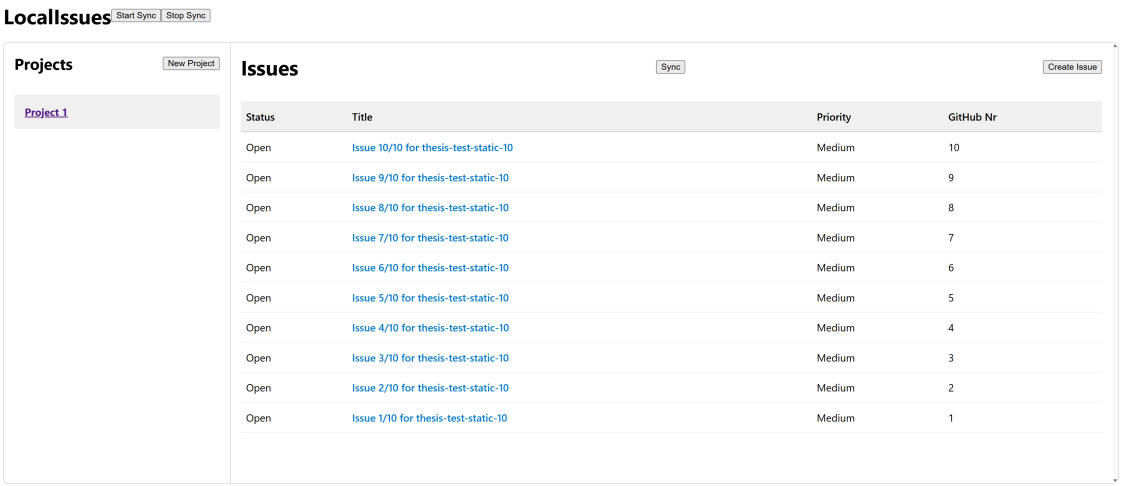

Figure 10: View of a project in the application after initial synchronization with GitHub

6.3.1 Database Setup

For persistent storage the application uses Dexie.js. The adapter provided by the synchronization layer is included in the creation of the `db` Dexie instance, creating 3 tables for `issues`, `projects`, and `repositories`. The tables `issues` and `repositories` are marked as synchronized for the synchronization layer, as Listing 11 is showing. In the current implementation of the synchronization layer, the synchronized tables must use a primary key named `id` with

the JavaScript type of `number`. The `id` of a project is used as a foreign key for the related tables `issues` and `repositories`, and therefore indexed, alongside other indexes needed by the application. The `db` instance is then exported to be used in other files and frontend components.

```
import type { Issue, Project, Repository } from './schema';

export const db = new Dexie('LocalIssueDB', {
  addons: [X_Sync_Addon_Dexie],
}) as Dexie & {
  issues: EntityTable<Issue, 'id'>;
  projects: EntityTable<Project, 'id'>;
  repositories: EntityTable<Repository, 'id'>;
};

db.version(1).stores(
  {
    issues: '++id, github_number, project_id',
    projects: '++id, name',
    repositories: '++id, project_id',
  },
  {
    sync: ['issues', 'repositories'],
  },
);
```

Listing 11: Application database setup

6.3.2 Synchronization Object Initialization

To use the synchronization layer, the prototype application must initialize the `GitHubSync` adapter. This process involves creating a new instance of the class and providing it with a concrete configuration object defined in Section 6.2.3.

The configuration object is substantial, defining everything from synchronization intervals to the complex data transformation logic. The complete, unabridged code for this initialization is provided in Appendix A (Listing 14) for reference. To illustrate the key concepts, an abbreviated version of the configuration is shown in Listing 12. It highlights the three main parts of the configuration: supplying the database instance and authentication token, configuration of the synchronization behaviour (`syncConfig`), and the data mapping `schema`.

```
export const sync = new GitHubSync<Schema, typeof db>({
  db,
  token: PUBLIC_GITHUB_TOKEN,
  syncConfig: {
    issues: { syncInterval: 60 }, // Sync every 60 seconds
    repositories: {
      mode: 'manual',     // Only sync when manually triggered
      path: 'r',          // Read-only access
    },
  },
  schema: {
    issues: {
      tableName: 'issues',
      initialData: { priority: 1 },
      repos_to_fetch: () =>
        db.repositories
          .filter((repo) => repo.project_id !== undefined)
          .toArray()
          .then(...),
      toLocal: (remote) => ({
        title: remote.title,
        description: remote.body,
        status: remote.state === 'open' ? 0 : 2,
        ...
      }),
      toRemote: (local) => {...},
    },
  ...
  },
});
```

Listing 12: Key points of the initialization

The configuration object allows flexibility for the synchronization layer due to the following exposed configurations by the generic `SyncBase` object and the more focused `GitHubSync` adapter:

- **Synchronization Behaviour (`syncConfig`):** The application developer has fine-grained control over each table. Here, `issues` are configured for frequent, automatic, bidirectional synchronization, while `repositories` are read-only and will only be synchronized when manually triggered by the user.

- **Data Schema (`schema`):** This enables the synchronization layer to use the same database schema as the application. The application developer can provide the name of the table that corresponds to the remote entity, as well as add initial data for **New Remote** items. Additionally, it provides the concrete data lens functions (`toLocal` and `toRemote`) required by the adapter. For instance, the prototype's `toLocal` function implements the specific logic to map GitHub's binary `open`/`closed` state to the application's more granular multi-state status field. Similarly, the `repos_to_fetch` function is implemented to query the local database for the list of repositories to synchronize, satisfying the API-specific requirement of the adapter.

By providing this single, comprehensive configuration object, the prototype application equips the synchronization layer to communicate effectively with the GitHub API, bridging the gap between the local-first database and the remote service.

The exported instance `sync` of the `GitHubSync` object can then be used inside a frontend component to start or stop the synchronization, or to manually sync a table, as shown in Listing 13.

```
<button onclick={() => sync.start()}>Start Sync</button>
<button onclick={() => sync.stop()}>Stop Sync</button>

<button onclick={() => sync.syncTable("issues")}>
  Manual Sync Issues
</button>
```

Listing 13: Svelte markup interacting with the synchronization layer

7 Evaluation

This chapter evaluates the implemented synchronization layer architecture against the requirements defined in Chapter 5 and provides evidence-based answers to the research questions introduced in Chapter 1. To provide empirical data, the prototype application described in Chapter 6 was benchmarked against a second client-server SvelteKit application, using Server-Side Rendering (**SSR**). At first, the setup for the benchmarks will be described in detail, followed by a results-and-analysis section structured by key requirements. At last, a discussion of the findings and a summary of the evaluation's limitations will be provided.

7.1 Evaluation Setup

Each benchmark was run five times to reduce random variation by averaging. Benchmarks were run using Playwright[19] version 1.52.0 with the Chrome browser, on a Windows 11 Professional (Build 26100.3775 (24H2)) desktop computer with a 3.4 GHz 16-core AMD Ryzen 9 5950X and 32GB of 3200 MHz DDR4 memory. The computer was connected to the internet by a fiber-to-the-curb connection with a maximum uplink of 100 Mbps and a maximum downlink of 20 Mbps, with a 15 ms ping to GitHub's servers.

Benchmarks were performed on both the prototype application and the second client-server application. The client-server application leverages SvelteKit's **SSR** using load functions for data fetching [42] and form actions to submit data from the client to the server [43]. All interaction with the **API** was done by the server and then submitted to the client. The server was deployed on the same machine, however the ChromeDevToolsProtocol was used to simulate a similar latency of 15 ms to the server [44]. The client-server baseline deliberately excludes client-side caching or optimistic UI patterns to establish a clear comparison point. While production applications might employ these techniques, they introduce additional complexity for the application developer, making the comparison less meaningful for evaluating the core synchronization contribution.

To measure the perceived performance of both approaches, this research introduces the **TTVF** as a key metric, defined as the time measured between the user action initiation and UI rendering of updated data.

In total, three test were performed, summarized in Table 4.

[19] https://playwright.dev

	Scenario Description	Metrics measured
T1	Initial & Subsequent Load	**TTVF**
T2	Bulk & Conflict Synchronization	**TTVF**, CPU & memory usage, **API** request count
T3	Sustained Write Load	**TTVF**, CPU & memory usage, **API** request count

Table 4: Summary of performed benchmarks

- Scenario **T1** measures the time required for the application to become interactive when loading a project for the first time and on subsequent visits. To assess performance under varying conditions, this test was performed against repositories containing 10, 50, and 250 issues.
- **T2** simulates high-volume concurrent updates. A small (5) and large (25) batch of issues were created and updated simultaneously by GitHub and the application over the course of five minutes, while measuring **TTVF**, the resource utilization of the synchronization cycle as well as the amount of request sent to the **API**. The test was performed with both a 30 and a 60-second synchronization interval.
- In **T3** an issue is updated every 15 seconds over the course of five minutes to simulate a user actively working inside the application. The test was run with the same two synchronization intervals of 30 seconds and 60 seconds. The resource utilization, network activity and performance consistency using the **TTVF** metric were monitored.

Unlike latency or resource-usage, there is no established, fully automated benchmark or metric for offline behaviour. The key consideration is the user's ability to read existing data, queue new writes, and recover gracefully when connectivity returns. As it was not possible to quantify offline in a meaningful way using a single metric, a qualitative assessment was performed instead, using a scorecard.

To test the storage efficiency of the synchronization layer, a new database was created with a single project, containing 50 issues and 50 local writes. The IndexedDB data was exported once before synchronization and once after synchronization. This was done by executing Listing 15, provided in the appendix. To compare against a baseline without the synchronization layer, Listing 16, provided in the appendix, was executed in the developer console of the browser to remove all data only used by the synchronization layer from the database. This includes

the added tables `_writeLog` and `_syncState`, as well as the `remote_id` field on each issue and repository.

7.2 Results

Having detailed the evaluation methodology in the preceding section, this section now presents and analyses the collected data. At first, the performance and responsiveness of both applications will be explored, then, the offline capabilities of both applications will be explained. Finally, the resource and storage efficiency of the synchronization layer will be evaluated.

Performance and Responsiveness

Performance and responsiveness are governed by two requirements: R1 (fast feedback) and R8 (non-blocking synchronization). To verify these, the **TTVF** metric was measured in Test T1 under two scenarios:

To verify the read performance of the application, **TTVF** was measured in **T1** under two scenarios:

- Initial Load: the first time a project is opened.
- Subsequent Load: re-opening the same project.

Figure 11 shows the mean **TTVF** for loading projects of varied sizes. The corresponding Table 9 can be found in the appendix.

While the initial load times for small repository sizes was similar over both applications, an increase in the size of the repository leads to linear increase in initial load time for the local-first application, compared to a lower increase for the client-server application. However, this was to be expected, as each fetched issue from the remote has to be transformed into the local format and inserted into the database, while the cloud-approach directly displays the data as is and most of its wait time is due to the round-trip to the server instead of the actual size of the data. Since initial load times occur infrequently (only if creating a new project with repository that has existing issues), these longer latencies are not as relevant as subsequent load times that happen every time the project is opened.

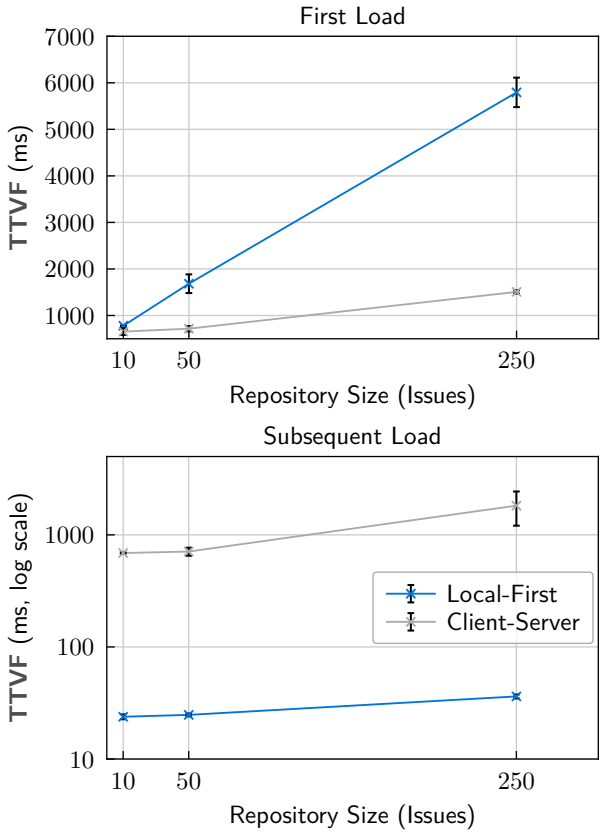

*Figure 11: Mean **TTVF** for loading a project*
Error bars show ± σ, Note that the y-axis for the 'Subsequent Load' plot is logarithmic.

For subsequent loads, where the local-first app exclusively reads from the IndexedDB, the mean **TTVF** falls well below the 100 ms threshold, with load times around 97-98% faster compared to the client-server application.

The read performance of the application was evaluated by measuring the **TTVF** multiple times across two write-intensive tests **T2** and **T3**. Figure 12 plots the mean update and create times across both tests, while Figure 13 shows how the create and update times are distributed in the local-first application.

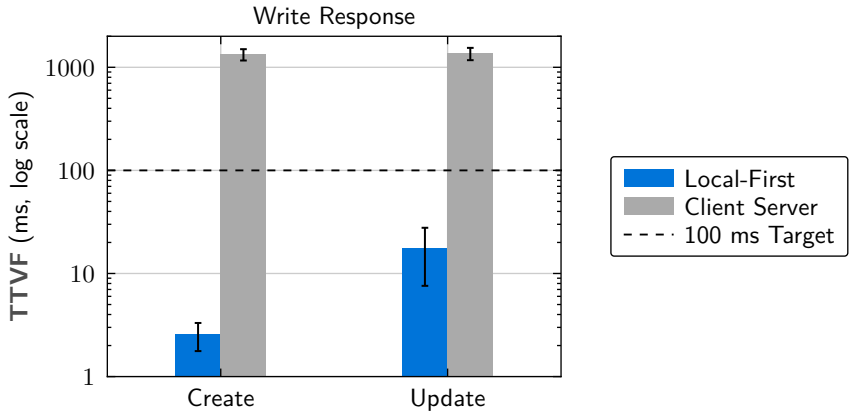

*Figure 12: Mean write performance across **T2** and **T3***
Error bars show ± σ, plotted with a logarithmic y-axis.

In both the local-first and cloud-first application, creating an issue is faster than updating an issue, however this is due to the fact that updating an issue is done on a separate page (the issue overview page), while creating an issue is done inside a popup on the same page as the data view, therefore the user has to wait for the navigation to finish before seeing the updated data view. In the local-first application, all writes are below the 100 ms threshold required by **R1**, with a mean creation time of 2.6 ms (± 0.8ms σ) and mean update time of 17.67ms (± 10.1ms σ). The client-server baseline does not meet this threshold with a mean creation time of 1331.6 ms (± 167.5 ms σ) and a mean update time of 1358.1ms (± 185.5 ms σ), also missing the 1000 ms threshold, therefore resulting in a higher user frustration [10].

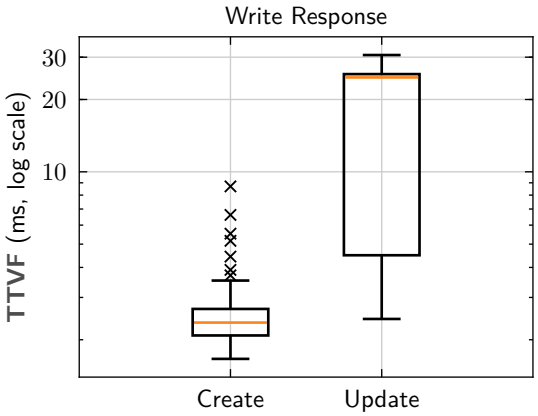

*Figure 13: Write performance of the local-first application across **T2** and **T3** plotted with a logarithmic y-axis.*

As the test **T2** and **T3** are performed with active synchronization, and every write is below the threshold of 100 ms, the synchronization layer does not block the rendering of the UI during active synchronization, therefore satisfying **R7**.

Offline Capability

As the prototype application lacks a service worker, its offline support is restricted to the period after the JavaScript bundle has been fetched via an active connection. The synchronization layer will only correctly serve data and create writes once all application assets have been loaded, it cannot bootstrap itself from entirely offline. This limitation falls outside the scope of the synchronization layer's responsibilities, which only concern data storage, conflict resolution and request batching. Instead, it is the application's responsibility to implement a service worker (or equivalent) for offline asset caching and startup [45]. Due to time constraints, a service worker was not added to the prototype. Adding one would enable true 'first-load offline' behaviour without affecting or complicating the synchronization layer itself.

Table 5 provides a qualitative feature scorecard that directly maps to the offline ideals formalized in requirements **R2** ("app works while offline"), **R4** ("read-own-writes"), and **R3** ("app survives API unavailability"), as unavailability can be considered a form of unlimited offline state. Both applications were tested if they pass the given offline use case. Each row of Table 5 corresponds to one essential offline capability:

- Offline reads: ability to query and display cached data with no network
- Offline create/update: acceptance of new or edited records while offline
- Read-your-own-writes: immediate visibility of local modifications during disconnection
- Queued writes and reconciliation: persistent logging of local changes and automatic synchronization on reconnect
- Seamless transition online: resumption of bidirectional synchronization without user intervention

Use Cases	Local-First	Client-Server
Read data when offline	●	○
Create/update items offline	●	○
Read offline created/updated items	●	○

Use Cases	Local-First	Client-Server
Queue writes and reconciles on reconnect	●	○
Transitioning from offline to online	●, Automatic synchronization	○, Manual refresh required

● Pass; ○ Fail

Table 5: Offline feature scorecard

Synchronization Efficiency

To evaluate requirement **R5** and **R7**, the total number of GitHub API requests (GET, POST, PATCH) issued by the local-first prototype (at 60 s and 30 s synchronization intervals) and the client-server baseline was measured across **T2** and **T3**. Additionally, CPU usage (in percent) and memory usage (in MB) was measured.

Figure 14 and Table 6 show that in the client-server application, GET request increase with the size of the repository and amount of updates, as the whole repository gets refetched. This also increases due to pagination, as responses over 30 items get split up into multiple pages, requiring multiple GET requests. As the local-first application only fetches the incremental updates since the last synchronization, its amount of GET request depends on the number of remote updates (U) and the **API**'s page size (P), following the formula

$$N_{\text{GET}} = \frac{\text{Total Test Duration}}{\text{Synchronization Interval}} * \lceil \frac{U}{P} \rceil$$

therefore 5 (for 60 second interval) and 10 (for 30 second interval) requests, as the tests ran for 5 minutes and the remote updates were never larger then the page size. For the creation of items (POST), the amount of **API** request is the same across both applications, the same for updates to different items (POST), exactly 1 PATCH or POST per write. For updates to the same item, the local edits between the interval get collapsed into one big update request, resulting in 5 request (60s interval) and 10 (30s interval), compared all 20 update request being send by the client-server approach.

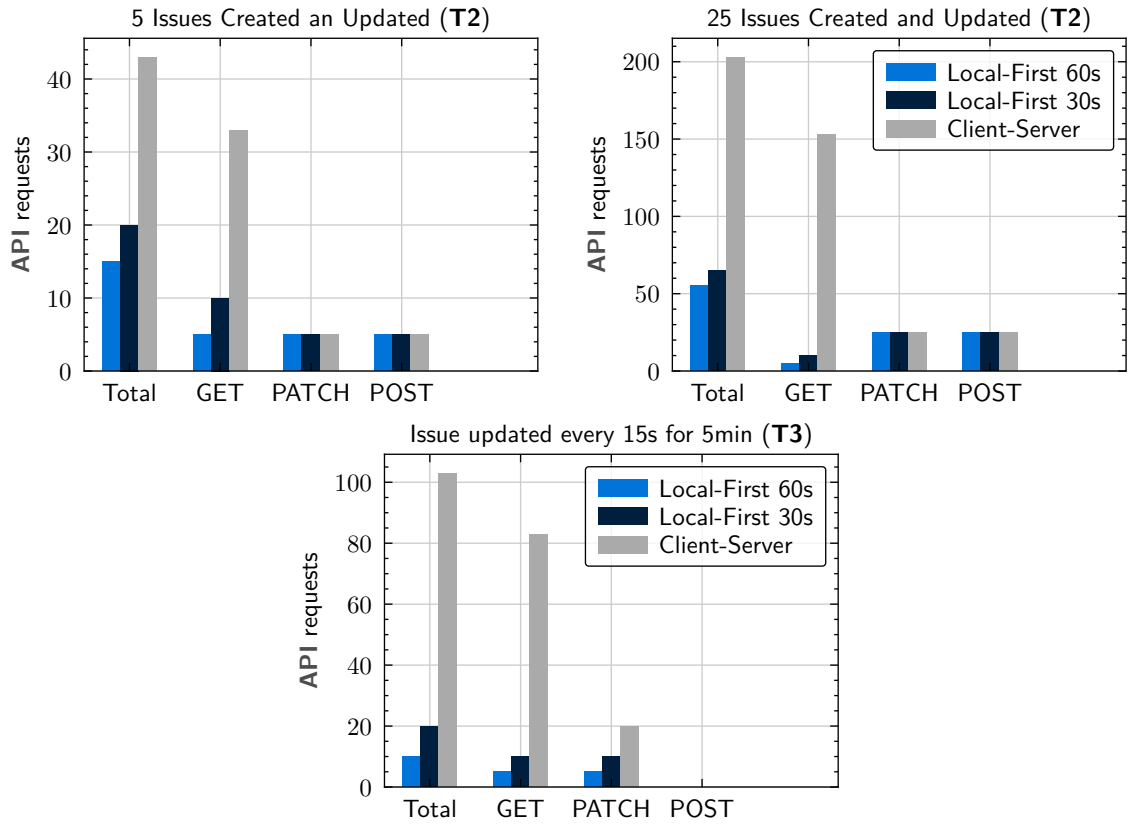

*Figure 14: **API** requests per application*

Use case	Local-First (60s)			Local-First (30s)			Client-server		
	GET	PATCH	POST	GET	PATCH	POST	GET	PATCH	POST
Create and Update 5 issues	5	5	5	10	5	5	33	5	5
Create and Update 25 issues	5	25	25	10	25	25	153	25	25
Update **one** issue 20 Times	5	5	0	10	10	0	83	20	0

*Table 6: Amount of **API** requests per use case*

The total number of requests per interval is defined by the following formular:

Let

- U be the number of remote updates since last synchronization
- P be the page-size limit of the API (items per GET)
- W be number of locally modified items since last synchronization

then the total number of requests per interval is therefore:

$$N_{\text{req}} = N_{\text{GET}} + W = \lceil \tfrac{U}{P} \rceil + W$$

Because updates are fetched only at each synchronization interval, the maximum Age of Information (AoI) seen by a user is bounded by the interval whenever online. This trade-off between data freshness and efficiency maps directly to requirement R5: the reduced network load comes at a bounded, predictable staleness.

Table 7 summarizes the resource-usage data for these same tests. For the sustained-update scenario (Test T3), the local-first prototype consumes on average 1.20 % $\pm$ 0.53 % CPU and 6.58 $\pm$ 0.62 MB of memory at a 30 s interval (1.19 % $\pm$ 0.56 % and 6.40 $\pm$ 0.56 MB at 60 s), compared to 1.43 % $\pm$ 0.96 % CPU and 6.43 $\pm$ 1.09 MB of memory for the client-server baseline. During **T2**, comparable results were recorded (Table 10 in the appendix). The slightly lower CPU and memory usage in the local-first case was not anticipated, likely reflecting minor differences in UI rendering and event loop scheduling rather than an intrinsic advantage of the synchronization layer. Crucially, these measurements remain well below levels that could impact interactive responsiveness, confirming **R7**.

Application	mean CPU (% $\pm$ σ)	peak CPU (%)	mean RAM (MB $\pm$ σ)	peak RAM (MB)
Local-First (30s)	$\mathbf{1.20} \pm 0.53$	5.08	$\mathbf{6.58} \pm 0.62$	8.00
Local-First (60s)	$\mathbf{1.19} \pm 0.56$	5.25	$\mathbf{6.40} \pm 0.56$	8.00
Client Server	$\mathbf{1.43} \pm 0.96$	9.00	$\mathbf{6.43} \pm 1.09$	9.00

*Table 7: resource usage for **T3***

Storage Efficiency

Table 8 shows the size of the data extracted from the database during an active test scenario with 50 issues and 50 writes. Each issue has a title, description, priority, and status. As expected, the storage requirement before synchronization is larger then after synchronization, with a drop of around 22.8%. This was to be expected as the write-log stores the old and new data for each update. However, differences this large would only occur after large offline periods, as the write-log would be cleared after every synchronization. Additionally, as the write-log data might be repetitive, the actual on disk size could be smaller if the database employs compression, as the extracted data is uncompressed.

State	Data (kB)	Write-Log (kB)	State Metadata (kB)	**Total Size (kB)**
Without Remote Data	25.62	-	-	**25.62**
Before Synchronization	25.77	9.10	0.17	**35.04**
After Synchronization	26.87	0.00	0.17	**27.04**

Table 8: Storage Requirements. Database consists of one project with 50 issues and 50 write operations

The synchronization layer introduces storage overhead through three components:

- Write-log entries: around 180 bytes per operation, but only kept until next synchronization
- `remote-id` fields: around 25 bytes per item
- Synchronization state metadata: around 170 bytes.

For the evaluated scenario (50 issues, 50 writes), this totals 1.42 kB overhead on 25.62 kB of application data (around 5.5%) with 8 fields in the schema (including `remote_id`). Overhead percentage decreases as application data size increases. For example, if the schema would scale linearly from 8 to 16 fields, the overhead would still only be 1.42 kB on then 51.24 kB of application data (around 2.8%).

7.3 Discussion

In this section the key findings are synthesized and related back to the requirements and research questions, as well as identifying remaining limitations of both the prototype and evaluation.

Synthesis of Results

- Performance and Responsiveness (R1, R7): The Local-First prototype consistently delivers sub-100 ms visual feedback for all interactive operations (subsequent loads and writes), fully satisfying R1. Background synchronization incurs no perceptible UI stalls (R7). The one exception is the Initial Load, which exceeds 100 ms but occurs only once per project and therefore does not affect overall responsiveness.

- Offline Capability (R2, R3, R4): The local-first app passes every essential offline use case, complying with R2, R3, and R4 once the JavaScript bundle has been loaded. True "first-load" offline would require a service worker, but this responsibility lays outside of the synchronizations layer.

- Synchronization Efficiency (R5): By batching remote reads and local writes at each synchronization interval, the synchronization layer can reduce traffic by a factor of 2-5 compared to client-server approach, without altering write semantics.

- Resource and Storage Overhead (R6, R7): The CPU and memory measurements show a low resource usage even during active synchronization, not impairing interactivity, therefore meeting R7. The uncompressed storage overhead after synchronization in the tested scenario is around 25 bytes for each database item.

Limitations

There are several limitations to this evaluation that must be acknowledged. Firstly, the scope of the tests was limited to just tens of items, workloads involving thousands of records or more may reveal different performance characteristics or synchronization issues. Secondly, all measurements were taken on a high-end desktop with a fibre-optic connection, which may not reflect the behaviour of lower-powered devices or high-latency networks. Each test scenario was repeated only five times, longer or more varied runs could reveal additional variability or edge-case behaviours. Offline functionality was assessed using a qualitative scorecard rather than automated metrics, so a more rigorous quantitative approach could be used in future. Finally, the evaluation only targeted the GitHub **REST API**. Other **API** types, such as GraphQL, were deliberately excluded and may exhibit different rate-limit policies, pagination schemes or schema-mapping requirements. The storage requirements were only evaluated with one exported dataset, however the true total storage requirements of the synchronization layer will vary considerably depending on the different application, while being around 25 bytes per synchronized item. These constraints should be borne in mind when interpreting the results and planning subsequent extensions.

Beyond the evaluation methodology, the current implementation of the synchronization layer also possesses specific inherent limitations:

- Write-log growth: Although the local-first ideal promises unlimited offline operation, an unbounded write-log can eventually exhaust storage if the **API** becomes unavailable. A compaction mechanism or the disablement of the write-log is required to fully fulfil the long-now requirement of local-first.
- Interrupted synchronization: If connectivity drops mid-sync, in-flight state is lost and the next reconnect triggers a full re-sync. Without persisted synchronization cursors, this can waste bandwidth and, in extreme cases, replay the same writes to the remote.
- **API** rate-limits for large offline batches: The synchronization was only measured with sub hundred amounts of local writes per interval. Additionally, the synchronization currently cannot be paused and finished at a later time. Exceedingly long offline periods, queuing thousands of local writes, will exceed GitHub rate limits (especially for creation requests), requiring multiple hours of synchronization and therefore additional strategies.

8 Conclusion

The research defined and validated a client-side synchronization layer, paving the way for a local-first application to interoperate with a non-collaborative external an **API** (GitHub **REST**). The seven concrete requirements (R1-R7) encompass responsiveness, offline functionality, efficiency, storage overhead, and resource usage. A prototype of the synchronization layer, with a write-log, two-pass reconciliation algorithm, and a bidirectional data-lens, was implemented inside a SvelteKit prototype application and evaluated through quantitative tests (T1-T3) and a qualitative offline use case scorecard.

In summary, this work demonstrates that a dedicated client-side synchronization layer can bridge the gap between **REST** endpoints and a seamless, resilient local-first user experience. The architectural blueprint and prototype together provide a solid foundation for future extensions and confirm that the core local-first ideals can be achieved without compromising performance or efficiency.

This work addresses the four research questions as follows:

1. Architectural Patterns (RQ1): Comparing the proxy pattern and the synchronization-layer pattern demonstrated that a dedicated synchronization layer with a write-log and two-pass reconciliation outperforms a simple proxy in responsiveness, offline resilience, and API efficiency.

2. Offline mechanisms (RQ2): A combination of a local write-log for capturing offline modifications and a bidirectional data lens (with `toLocal` and `toRemote` transforms) was sufficient for preserving correctness across schema differences while maintaining immediate local reads.

3. Conflict resolution (RQ3): The two-pass categorization algorithm, coupled with a **LWW** policy, reliably detects and resolves conflicts without requiring any support from the server, as validated by concurrent update tests.

4. Trade-offs (RQ4): The key trade-off is between data freshness (maximum staleness equal to the synchronization interval) and network efficiency, by choosing an appropriate synchronization interval for the application. Application responsiveness is not part of this trade-off, as all interactivity happens below 100 ms, no matter the synchronization interval.

8.1 Realized Goals

The synchronization layer meets all target requirements in a compact, efficient design. It delivers sub-100 ms response times for subsequent data loads and all write operations, supports offline use and reflects local edits immediately in the UI. By batching remote reads and writes, it reduces API calls, and its only storage overhead after synchronization is a single extra schema field. Crucially, background synchronization runs unobtrusively, preserving interactive performance. These capabilities are embodied in a working local issue-tracker prototype featuring a GitHub synchronization adapter, bidirectional schema transforms, and empirical validation.

8.2 Open Goals

However, there still are open goals this research set out that remain to be addressed in subsequent studies. The compaction of the write-log, to allow the full long-now requirement, set out by the ideals of local-first, was not implemented. Additionally, the synchronization layer does not currently support thousands of local writes, as the synchronization currently must be finished in on session and is limited by **API** rate limits.

8.3 Future Work

Future work will involve broadening the empirical evaluation to encompass a wider variety of devices, network conditions and workloads. It will also involve adding support for alternative storage backends, such as SQLite, as well as other specific **APIs** and paradigms, such as GraphQL.

Offloading synchronization logic into dedicated background workers and integrating the PWA Background Synchronization API[20] could further improve performance and ensure non-blocking operation.

An additional server-hosted webhook service could be created for **APIs** that support Webhooks, to extend the synchronization layer and enable push-based remote updates, while falling back to pull-based synchronization if unavailable.

[20] https://developer.mozilla.org/en-US/docs/Web/API/Background_Synchronization_API

First-class integrations for popular frameworks such as React, Vue and Angular can simplify adoption, while adaptive rate-limit handling will guard against API throttling under high offline write volumes.

Enabling true multi-user, multi-device collaboration will require a robust cross-device synchronization protocol and richer conflict resolution, potentially via semantic merge strategies, CRDTs or the integration with already existing local-first libraries and synchronization engines.

To limit storage growth during extended offline periods, write-log compaction or snapshotting mechanisms must be introduced. Additionally, sensitive API credentials should get stored securely using the secure storage provided by the platform. Finally, persisting the in-flight synchronization state will make the layer interruption-safe and fully resumable, thus completing its support for seamless offline functionality.

9 Appendix A: Additional Listings

Listing 14: Initializing the synchronization objects

```
export const sync = new GitHubSync({
  db,
  token: GITHUB_TOKEN,
  syncConfig: {
    issues: {
      syncInterval: 60,
    },
    repositories: {
      mode: 'manual',
      path: 'r',
    },
  },
  schema: {
    issues: {
      tableName: 'issues',
      initialData: {
        priority: 1,
      },
      repos_to_fetch: () =>
        db.repositories
          .filter((repo) => repo.project_id !== undefined)
          .toArray()
          .then((repos) => {
            return repos.map((repo) => ({
              full_name: repo.full_name,
              id: repo.project_id as number,
            }));
          }),
      getRepo: (issue) =>
        db.projects
          .get(issue.project_id)
          .then((repo) => db.repositories.get(repo?.repository_id ?? 0))
          .then((repo) => repo?.full_name ?? ''),
      //@ts-ignore
      toLocal: (remote, args?: { full_name: string; id: number }) => ({
        title: remote.title,
        description: remote.body,
        status: remote.state === 'open' ? 0 : 2,
        github_number: remote.number,
        remote_id: String(remote.id),
        //@ts-ignore
```

```
          project_id: args.id,
        }),
        toRemote: (
          local,
        ): Partial<{ title: string; body: string; state: string }> => {
          const remoteDelta: Partial<{
            title: string;
            body: string;
            state: string;
          }> = {};
          if (local.title) {
            remoteDelta.title = local.title;
          }
          if (local.description) {
            remoteDelta.body = local.description;
          }
          if (local.status !== undefined) {
            remoteDelta.state = local.status === 2 ? 'closed' : 'open';
          }
          return remoteDelta;
        },
      },
      repos: {
        tableName: 'repositories' as keyof Dexie,
        toLocal: (remote) =>
          ({
            name: remote.name,
            description: remote.description ? remote.description : '',
            remote_id: remote.id.toString(),
            full_name: remote.full_name,
          }) as Required<Repository> & { remote_id: string },
      },
    },
  });
```

Listing 14: Initializing the synchronization objects

Listing 15: Evaluate Storage Requirements

```
import { db } from "$lib/db";
import type { Table } from "dexie";

async function getTableSizeBytes(table: Table) {
  try {
```

```
    const tableData = await table.toArray();
    if (tableData.length === 0) return 0;
    return new TextEncoder().encode(JSON.stringify(tableData)).length;
  } catch (error) {
    console.warn(`Error getting size for table ${table.name}:`, error);
    return 0;
  }
}

export async function evaluateStorage() {
  const sizes = {
    // Application Data
    issues: await getTableSizeBytes(db.issues),
    repositories: await getTableSizeBytes(db.repositories),
    // Sync Layer Data
    writeLog: await getTableSizeBytes(db.table('_writeLog')),
    syncState: await getTableSizeBytes(db.table('_syncState')),
  };

  console.log("Detailed Breakdown:", sizes);
  return sizes;
}
```

Listing 15: Readout the storage requirements of the database

Listing 16: Remove Remote Data from IndexedDB

This snippet removes the `_writeLog` and `_syncState` tables, as well as `remote_id` field from every table.

```
(async () => {
  const dbName = 'LocalIssueDB';

  try {
    const db = await new Promise((resolve, reject) => {
      const request = indexedDB.open(dbName);
      request.onsuccess = () => resolve(request.result);
      request.onerror = () => reject(request.error);
    });

    // Clear _writeLog table
    try {
      const writeLogTx = db.transaction(['_writeLog'], 'readwrite');
```

```
      const writeLogStore = writeLogTx.objectStore('_writeLog');
      await new Promise((resolve, reject) => {
        const request = writeLogStore.clear();
        request.onsuccess = () => resolve();
        request.onerror = () => reject(request.error);
      });
    } catch (e) {
      console.log('_writeLog table not found or already empty');
    }

    // Clear _syncState table
    try {
      const syncStateTx = db.transaction(['_syncState'], 'readwrite');
      const syncStateStore = syncStateTx.objectStore('_syncState');
      await new Promise((resolve, reject) => {
        const request = syncStateStore.clear();
        request.onsuccess = () => resolve();
        request.onerror = () => reject(request.error);
      });
    } catch (e) {
      console.log('_syncState table not found or already empty');
    }

    // Function to remove remote_id from a table
    const removeRemoteIdFromTable = async (tableName) => {
      try {
        const tx = db.transaction([tableName], 'readwrite');
        const store = tx.objectStore(tableName);

        // Get all records
        const records = await new Promise((resolve, reject) => {
          const request = store.getAll();
          request.onsuccess = () => resolve(request.result);
          request.onerror = () => reject(request.error);
        });

        // Update records without remote_id
        let updatedCount = 0;
        for (const record of records) {
          if (record.remote_id) {
            delete record.remote_id;
            await new Promise((resolve, reject) => {
              const request = store.put(record);
              request.onsuccess = () => resolve();
              request.onerror = () => reject(request.error);
            });
```

```
            updatedCount++;
          }
        }
      } catch (e) {
        console.log(`${tableName} table not found:`, e.message);
      }
    };

    // Remove remote_id from all main tables
    await removeRemoteIdFromTable('issues');
    await removeRemoteIdFromTable('repositories');
    await removeRemoteIdFromTable('projects');

    db.close();

  } catch (error) {
    console.error('Error during cleanup:', error);
  }
})();
```

Listing 16: Browser console snippet to remove all remote data from the IndexedDB database

10 Appendix B: Additional Tables

Application	Size	First load **TTVF** (ms)		Subsequent load **TTVF** (ms)	
		Mean	σ	Mean	σ
Local-First	10	**779.7**	1.3	**23.8**	1.2
	50	**1683.2**	200.9	**24.8**	0.7
	250	**5794.1**	316.5	**36.2**	1.5
Client-Server	10	**654.5**	73.7	**688.9**	10.4
	50	**716.5**	57.4	**709.6**	58.5
	250	**1506.9**	35.1	**1822.6**	615.4

*Table 9: Mean **TTVF** for **T1***

Application	mean CPU (% ± σ)	peak CPU (%)	mean RAM (MB ± σ)	peak RAM (MB)
Local-First (30s)	**1.20** ± 0.50	5.29	**7.00** ± 0.57	9.00
Local-First (60s)	**1.15** ± 0.51	5.3	**6.80** ± 0.56	8.00
Client Server	**1.41** ± 0.71	9.00	**7.74** ± 1.81	13.40

*Table 10: Resource usage for **T2***

Bibliography

[1] 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*, 2019, pp. 154–178.

[2] M. Sosnowski, R. von Seck, F. Wiedner, and G. Carle, "CRDT Web Caching: Enabling Distributed Writes and Fast Cache Consistency for REST APIs," in *2024 20th International Conference on Network and Service Management (CNSM)*, IEEE, 2024, pp. 1–7.

[3] W. Wingerath *et al.*, "Speed Kit: A Polyglot & GDPR-compliant Approach for Caching Personalized Content," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, IEEE, 2020, pp. 1603–1608. Accessed: Mar. 27, 2025. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9101613/

[4] "ElectricSQL | ElectricSQL." Accessed: Jun. 19, 2025. [Online]. Available: https://electric-sql.com/

[5] "Zero Docs." Accessed: Jun. 19, 2025. [Online]. Available: https://zero.rocicorp.dev/

[6] A. Gadkari, "Caching in the Distributed Environment," vol. 2, no. 1, 2013.

[7] M. van Steen and A. S. Tanenbaum, *Distributed Systems*, Fourth edition, version 4.01 (January 2023). Maarten van Steen, 2023.

[8] J. Zhong, R. D. Yates, and E. Soljanin, "Two Freshness Metrics for Local Cache Refresh," in *2018 IEEE International Symposium on Information Theory (ISIT)*, Jun. 2018, pp. 1924–1928. doi: 10.1109/ISIT.2018.8437927.

[9] B. Abolhassani, J. Tadrous, A. Eryilmaz, and S. Yüksel, "Optimal Push and Pull-Based Edge Caching for Dynamic Content," *IEEE/ACM Transactions on Networking*, 2024, Accessed: Mar. 27, 2025. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10452408/

[10] I. S. MacKenzie and S. K. Card, "Lag as a Determinant of Human Performance in Interactive Systems," in *Conference on Human Factors in Computing Systems - Proceedings*, 1993, pp. 488–493. doi: 10.1145/169059.169431.

[11] I. Mandrigin, “Optimistic UI.” Accessed: May 31, 2025. [Online]. Available: https://uxplanet.org/optimistic-1000-34d9eefe4c05

[12] N. Tolia, D. G. Andersen, and M. Satyanarayanan, “Quantifying Interactive User Experience on Thin Clients,” *Computer*, vol. 39, no. 3, pp. 46–52, 2006, Accessed: Apr. 23, 2025. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/1607949/

[13] J. Nielsen, *Usability Engineering*. Morgan Kaufmann, 1994.

[14] W. Vogels, “Eventually Consistent,” *Communications of the ACM*, vol. 52, no. 1, pp. 40–44, Jan. 2009, doi: 10.1145/1435417.1435432.

[15] J. Bonér, D. Farley, R. Kuhn, and M. Thompson, “The Reactive Manifesto.” Accessed: Jun. 17, 2025. [Online]. Available: https://www.reactivemanifesto.org/

[16] M. Fowler, “Event Sourcing.” Accessed: Jun. 17, 2025. [Online]. Available: https://martinfowler.com/eaaDev/EventSourcing.html

[17] M. Kleppmann, *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. "O'Reilly Media, Inc.", 2017.

[18] E. Murphy, “Distributed Data for Microservices — Event Sourcing vs. Change Data Capture.” Accessed: Jun. 17, 2025. [Online]. Available: https://debezium.io/blog/2020/02/10/event-sourcing-vs-cdc/

[19] N. Preguiça, C. Baquero, and M. Shapiro, “Conflict-Free Replicated Data Types (CRDTs).” Accessed: May 30, 2025. [Online]. Available: http://arxiv.org/abs/1805.06358

[20] 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.

[21] B. Zelenka, “Feeling the Local-First Elephant: A Roadmap, Hidden Gems, and New Puzzles from the Field,” Oct. 24, 2023. Accessed: May 08, 2025. [Online]. Available: https://www.youtube.com/live/M8RRZakZgiI?si=gg3HrjoYGSNCLzQ4&t=5971

[22] A. Rotem-Gal-Oz, "Fallacies of Distributed Computing Explained," vol. 20, 2006, [Online]. Available: https://www.se.rit.edu/~se442/doc/fallacies.pdf

[23] Tuomas Artman, "Unexpected Benefits of Going Local-First," Jun. 18, 2024. Accessed: Jun. 09, 2025. [Online]. Available: https://www.youtube.com/watch?v=VLgmjzERT08

[24] J. Yablonski, "Tesler's Law." Accessed: May 08, 2025. [Online]. Available: https://lawsofux.com/teslers-law/

[25] B. Pourebrahimi, K. Bertels, and S. Vassiliadis, "A Survey of Peer-to-Peer Networks," in *Proceedings of the 16th Annual Workshop on Circuits, Systems and Signal Processing*, 2005, pp. 570–577.

[26] H. Caudill, "Alice and Bob in Wonderland." Accessed: May 09, 2025. [Online]. Available: https://herbcaudill.com/words/20240602-local-first-auth

[27] "Why Local-First Software Is the Future and Its Limitations | RxDB - JavaScript Database." Accessed: May 31, 2025. [Online]. Available: https://rxdb.info/articles/local-first-future.html

[28] "Rate Limits for the REST API." Accessed: Jun. 16, 2025. [Online]. Available: https://docs.github.com/en/rest/using-the-rest-api/rate-limits-for-the-rest-api

[29] "Storage Quotas and Eviction Criteria - Web APIs | MDN." Accessed: Jun. 12, 2025. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Storage_API/Storage_quotas_and_eviction_criteria

[30] "GitHub REST API Documentation." Accessed: May 13, 2025. [Online]. Available: https://docs.github.com/en/rest

[31] ""IndexedDB" | Can I Use... Support Tables for HTML5, CSS3, Etc." Accessed: Jun. 01, 2025. [Online]. Available: https://caniuse.com/?search=IndexedDB

[32] "liveQuery()." Accessed: Jun. 16, 2025. [Online]. Available: https://dexie.org/docs/liveQuery()

[33] "Dexie.Js - Minimalistic IndexedDB Wrapper." Accessed: Jun. 01, 2025. [Online]. Available: https://dexie.org/

[34] "$state ▪ Docs ▪ Svelte." Accessed: Jun. 16, 2025. [Online]. Available: https://svelte.dev/docs/svelte/$state

[35] S. Krause, "Krausest/Js-Framework-Benchmark." Accessed: Jun. 16, 2025. [Online]. Available: https://github.com/krausest/js-framework-benchmark

[36] "TinyTick." Accessed: Jun. 16, 2025. [Online]. Available: https://tinytick.org/

[37] "Dexie.Transaction()." Accessed: Jun. 15, 2025. [Online]. Available: https://dexie.org/docs/Dexie/Dexie.transaction()

[38] "Navigator: onLine Property - Web APIs | MDN." Accessed: Jun. 15, 2025. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Navigator/onLine

[39] "Navigator: Connection Property - Web APIs | MDN." Accessed: Jun. 15, 2025. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Navigator/connection

[40] "Network Information API." Accessed: Jun. 15, 2025. [Online]. Available: https://wicg.github.io/netinfo/#dfn-effective-connection-type

[41] "REST API Endpoints for Issues." Accessed: Jun. 16, 2025. [Online]. Available: https://docs.github.com/en/rest/issues/issues

[42] "Loading Data ▪ Docs ▪ Svelte." Accessed: Jun. 17, 2025. [Online]. Available: https://svelte.dev/docs/kit/load

[43] "Form Actions ▪ Docs ▪ Svelte." Accessed: Jun. 17, 2025. [Online]. Available: https://svelte.dev/docs/kit/form-actions

[44] "Chrome DevTools Protocol." Accessed: Jun. 17, 2025. [Online]. Available: https://chromedevtools.github.io/devtools-protocol/tot/Network/#method-emulateNetworkConditions

[45] "Offline and Background Operation - Progressive Web Apps | MDN." Accessed: Jun. 18, 2025. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Guides/Offline_and_background_operation