Aqueduct: Transforming Snapshots of Cloud Data Into Event Streams

ANDREW BURRIS, Brown University, United States

Aqueduct is a library that takes snapshot-style methods data access (like web APIs and data exports) and turns them into streams of events. It periodically (but efficiently) polls a new snapshot, then reconciles it with the previously stored snapshot, calculates the events based the difference between the two, and emits those as a stream. After investigating why API access and right to data portability (RtDP) regulations haven't made meaningful differences in data interoperability, it became clear that one-time snapshots of data were insufficient for modern applications. With Aqueduct, and the ecosystem of extensible pre-built integrations that will hopefully grow around it, developers will be able to integrate cloud data into their applications more easily and richly.

CCS Concepts: • **Software and its engineering → Software libraries and repositories**; Software configuration management and version control systems; *Development frameworks and environments.*

Additional Key Words and Phrases: web APIs, data exports, data portability, data interoperability, event streams, data integration

1 INTRODUCTION

In the early days of computing, some of the first software developed was business-critical applications: calendar, email, word processing, etc. Looking at those applications today, they share a characteristic–the concept of the "client". Users can interact with their email interoperably from any and all of the iOS Mail app, the Gmail website, or a specialized email client for people with visual impairment, to name a few examples. The same goes for calendars, and even word processing: `.docx` files can be opened in Microsoft Word, but also LibreOffice, Google Docs, and many more.

The 2000s brought a new paradigm: cloud computing. Cloud applications have clear benefits: the ability to access them from any device, multiplayer functionality, the reliability of data being constantly backed up off-site, and more. But in exchange for the convenience of having data in the cloud, users gave up the option to switch between clients. Because cloud data is stored on a company's servers, users have to use that company's client app to access it.

As an example, compare the older technology of MP3 clients with one of their modern cloud counterparts: Spotify. If a user wanted to switch between traditional MP3 clients, all they had to do was install a new one, point it at the same folder containing all their `.mp3` music files, and it would work. Users could even use multiple at the same time. On the other hand, users constantly complain about changes Spotify makes to their app, but can't do anything about it. Switching to an alternative music streaming service requires hand-migrating all of their data (playlists, saved songs, etc.), not to mention it simply puts them at the whims of a different company's client app.

Aqueduct's goal is to enable developers to interact with 3rd party data so easily that it becomes possible to make client-style software for the new generation of cloud apps.

2 CURRENT SOLUTIONS

It isn't the case that there's no way to access cloud data. Many cloud apps allow 3rd party developers to access their data through application programming interfaces, abbreviated as "APIs". Even cloud apps that don't offer API access usually give users ways to export data from them (if only because they're required to for regulatory reasons). [9]

However, these APIs are fundamentally very challenging to make clients with. In order to get data from an API, a developer has to have their app request a webpage containing machine-readable text from the cloud server that contains a snapshot of the cloud data at that time. Often, each snapshot only includes a subset of the overall data, so multiple

Author's address: Andrew Burris, Brown University, United States, andrew_burris@brown.edu.

requests need to be stitched together to get the full picture. [2] (To further complicate the process, if something updates in between those stitched requests, those discrepancies need to be handled.)

This request-each-snapshot model isn't really how the official clients work. [16] Most have a constant two-way connection open with the backend server, and after an initial download of a full copy of the data, any changes from either side are sent as a notification about the specific change. The benefit is both that the notifications are real-time, and that they're specific, which makes them much more efficient on resources like network and database bandwidth. Since traditional API access lacks this event-stream style of communication, it's too unwieldy to easily create clients with. This isn't an intentional barrier; snapshots like APIs or data exports just aren't well-suited for modern application design.

One solution would be to run these snapshots through a distributed version control system, such as Git. There are a few downsides to this, though. One is that version control systems are usually not automatic, so would require manual interventions from users at times. Additionally, they mainly only handle finding differences between text. Understanding data as it "moves around" across time and formats is outside of their scope, as is any of the logic for actually getting that data.

Another solution that has been proposed is to change the cloud paradigm entirely. The local-first movement is the biggest player in this space: local-first apps store data primarily on-device rather than on-server, using conflict-resolution algorithms to automatically (and optionally) sync changes to data between different devices. [8] While there have been notable success stories, like Obsidian and Linear, the grander vision of the local-first movement is one where all of a user's data is local-first, and thus accessible to them. The issue is that in order to get someone to use, say, a maps application that allows users to store their personal data local-first, it first needs to be comparable in quality to the Google or Apple Maps app they're already using. [18] (As Apple's decade-long journey to compete with Google Maps shows, that's not a trivial task. Also, it's impractical: personal data like a user's location history is very important, but only a fraction of what Google Maps does.) In order for a local-first ecosystem to thrive, it needs to be able to interact with data from the cloud applications people already use.

Finally, there is the concept of API "integrations" (also often called SDKs)–libraries that provide helper code that makes it easier for developers to interact with data from other apps' APIs in their own apps. Most of these are written by the cloud companies that provide the APIs, and have all of the limitations described above. There are a number of libraries that attempt to standardize API integrations to give developers a unified interface for accessing them. [5] [2] [3] Some are just a standardization of traditional snapshot-style requests, but a few sync web API data with a 3rd-party developer's database, creating a much more reactive interface. [4] However, using a standardized integration library, even an open source one, locks developers into the APIs that it supports. If a developer needs to interact with a more bespoke service that isn't in the library (or like in Aqueduct, wants to reconcile with other sources like data exports), they either have to build it themselves outside of the standardized ecosystem, or they have to get in contact with the maintainers of the library to try to get them to write the new integration.

3 RELATED WORK

Much research has been done around the concept of data portability in the cloud, which is very helpful in elucidating the ways cloud apps expose data. Krämer et al. has a comprehensive listing of the ways that cloud apps have made their data available under the EU's directive of "right to data portability" (RtDP). One helpful differentiation is the idea of relatively "synchronous" access, like APIs, and relatively "asynchronous" access, like data exports that take weeks to compile. It also covers why projects aiming to change the siloed cloud paradigm, like SOLID, haven't been effective, partially because they have no interoperability with existing cloud providers. [9]

Several studies have explored users' experience with RtDP, mostly finding that it has not had the desired impact due to its unintuitive nature. Kuebler-Wachendorff et al. finds that users don't know about RtDP or what it could be used for, and thus don't feel comfortable with it. [10] Luzsa et al. finds that only technologically adept people have been using RtDP to achieve "digital sovereignty". [11] Syrmoudis et al., 2021 provides sources for a lot of the regulations that establish RtDP, and notes that they require "interoperability", which essentially hasn't been seen so far. It also questions why current applications largely do not take advantage of being able to import from their competitors. [18] This is partially explained in Syrmoudis et al., 2024 which finds that users who do take advantage of RtDP use it to transfer data between existing accounts rather than create new ones, showing that the use case is mainly this "interoperability". [17]

There is a lot of literature about the relationship between snapshots and events in the domain of event sourcing, but this is in the opposite direction of my research. Event sourcing explores reconstructing a snapshot from a series of events, whereas Aqueduct tries to reconstruct those events from snapshots. However, Overeem et al. describes the challenges of changing schema within event-based systems in general, which is relevant because Aqueduct deals with multiple schema inherently, all of which might change on a whim. [13]

Some, but much less research has been done in the snapshot-to-events direction, especially because Aqueduct considers the surrounding components of this translation as well, such as reactivity, and reconciling and filling in data with different information from multiple sources. Cao et al. has many insights on converting snapshots of JSON data into events that track specifically what has changed about the data, which is very helpful in the diffing process specifically. [6] Vesdapunt and Garcia-Molina explores how to (theoretically) poll an API for up-to-date data with a minimally efficient number of actual calls to that API, which is quite relevant for Aqueducts task of fetching data efficiently. [19]

4 RESEARCH QUESTIONS

Setting out with a goal of creating a library to make it easier to interact with cloud data, I decided on three central research questions to define what "making it easier" actually looks like:

(1) What are the ways apps expose data? What are the barriers to accessing it?
(2) What does a great ecosystem of integrations look like? How should data be represented and stored?
(3) Based on the ”inputs” from Q1 and the ”outputs” from Q2, what should the process be to transform the former to the latter? (Those transformations being the integrations themselves.) What building blocks are most useful to help developers build these integrations?

5 THE LIMITATIONS OF SNAPSHOT-STYLE APIS

The first part of my research consisted of exploring my first research question: "What are the ways apps expose data?" It was in this process that I started recognizing the concept of (and limitations of) the "snapshot-style" API. After reviewing the data access options for dozens of cloud applications, [9] I began to group them into five general categories (though there were, of course, outliers):

(1) **OAuth** (or similar login-based API)
Most large cloud apps (Google, Spotify, etc.) have some kind of official login-based API, almost all of which use a protocol called OAuth. Often they have several different OAuth-type flows to authenticate different clients based on what platform they're on. Websites, unlike servers, can be reliably identified due to the domain name the request originates from. But servers can store things securely, whereas any data in a web browser is user-accessible and thus insecure. This leads to different authentication flows based on what information can be shared.

(2) **API Key**
Some simpler or developer-focused apps offer users API keys for 3rd-party access. (This also occurs in unofficial APIs, like Notion, where the initial data download on a webpage is made through traditional API requests; it's just not documented and the user has to get their API key from their browser cookies.). This method is usually the easiest for the developer but it can be complex and confusing for the user to obtain their key.

(3) **Export**
Both large and small apps offer full data export options, though some (e.g. Google) have more complete information than others (e.g. Notion). These data exports are also often complex for the user to obtain. One benefit, however, is that once the file is obtained, it requires no authentication to parse it. (Though if the export has incomplete data, it's often necessary to use that app's API to fill out the full data.)

(4) **Scraping**
Sometimes, a website offers no official way to obtain data, so it must be scraped from the web page that displays it. This is the hardest for developers to get right, because the website's structure can be changed on a whim, breaking the scraping code until the developer updates it. Thus, any solution involving scraping needs to be flexible, and should be a last resort.

(5) **Platform-specific**
Some data is easily attainable if the integration is running on a specific platform, but complex otherwise. For example:
- An integration in an Android app can easily ask for permission and access text messages, which would otherwise require a complex exporting process
- An integration running on a Mac can easily access Apple Reminders and Notes through AppleScript, which would otherwise require very complex scraping from the iCloud website.

Notably, all of these access methods are one-time snapshots of data. They quickly stop being up-to-date (and in fact, exports can often take weeks to be created, making them inherently outdated).

Another major friction point I found with these current APIs was authentication. One issue is that there are so many options, even for a single app's data. Between differing authentication methods per platform, different scopes of data being requested, and the fact that every app's API handles authentication slightly differently, it can take a lot of effort for developers figure out how to properly request data, before they can even begin to build on top of that data.

In addition to the plethora of methods of authentication, there's a sequencing problem. A typical API interaction goes like this:

(1) The app tries to request data from an API.
(2) If it returns successfully, great. If not, the app next checks if the authentication token needs to be refreshed before retrying.
(3) If that wasn't the problem, it then checks whether the user is perhaps logged out, and redirects them to do that if not.

The issue is that most apps that interact with API data do so in many places, often on pages far away from the one where the user is prompted to log in. That means that any time a developer wants to write code that accesses API data at some place in their application, they also have to write all of this checking and redirection logic. (In reality, a lot of apps assume the user has logged in in one location and if it doesn't work, they reset the entire transaction and return to the home page to have the user log back in.)

To recap, this line of questioning led me to a few major issues with the current methods of interactions with APIs that Aqueduct needed to address. First, the snapshot problem: it's brittle to work with one-time captures of data. Second, the resource problem (an offshoot of the snapshot problem): it takes considerable resources to fetch and save all of the data needed from a web API each time it needs to be updated. Third, the authentication problem: it's hard for developers to even get started with accessing data due to complex and diverse authentication methods.

6 TOWARDS BETTER INTEGRATIONS

The second part of my research involved thinking through what a great developer experience for developers to integrate with 3rd party data would look like. One goal from the start was to make the ecosystem extensible by anyone. Other projects that brought together lots of integrations weren't easily extensible, which meant that when inevitably a developer needed an integration with an unsupported application, it became much more difficult to keep using that project. So I began by thinking about how to make it easy to build new integrations.

6.1 Building integrations

In order to make building integrations easier, a big factor was making them composable. Historically, composability has been extremely beneficial for building out robust developer ecosystems, and that applies here equally. [14] Consider a developer wanting to build an integration that uses data from Google Takeout. They first must build an integration with Google Drive, which is where the Takeouts are exported to (or Gmail, or Microsoft OneDrive, or a few other platforms). Only once they've done all of the work to authenticate with Drive, check it for new Takeout `.zip` files, and download those `.zip` files can they begin to work on the actual parsing of said files.

Instead, imagine they could compose their integration on top of an existing Google Drive integration, where another developer had already handled all of those steps. All the current developer would have to do would be to consume the `.zip` file output by the Drive integration and parse the Takeout data–no authentication handling, no refresh handling, just a simple function mapping one type of data to another.

One key aspect of this ecosystem is that integrations have to be open source packages in order to be built on top of. How open-source should they have to be? I didn't want to preclude developers from building integrations that were scoped locally to their private projects. On the other hand, integrations can't be built on if they're private. Open source is also important for transparency: if code is running on users' personal data, anyone should be able to inspect that it's not doing anything malicious. This is still an ongoing question for Aqueduct, but the general goal is to maximize developer freedom so long as a healthy ecosystem is maintained.

Another overarching goal was to make it possible for these integrations to be flexible towards their disparate data sources. A single integration might take in different pieces of data from a virtually instantaneous web API, a data export that takes a month to compile, a web scraping utility that might fail when a website changes their layout, or a database accessible only on a specific operating system. All of these sources might have differing levels of richness, and data from one source might often overlap with data from another source but in a different format. Some sources might be less reliable, and fail often, or deliver out of date data (especially with exports). These all would need to be reconciled into a single source of truth in order to effectively create an API that could be relied upon. Moreover, each integration has a different landscape of sources, so Aqueduct needed to be flexible in defining this reconciliation process per-integration.

6.2 Using integrations

Most of my early work on Aqueduct revolved around trying to define a unified structure for developers to build integrations on top of, generalizing the methods of authentication and data access I'd defined into a cohesive system. While I had good explorations, I made very little progress in terms of actually implementing the system. I had been so focused on making it easy for developers to *create* integrations that I'd forgotten all about the developers actually *using* the integrations. No matter how easy it is to build an integration with Aqueduct, if other developers don't then want to use that integration, it won't matter. And if they aren't going to be used, no one will want to build these integrations in the first place.

So I started focusing on a new question: "What does a significantly better developer experience (DX) for integrating with 3rd party data look like?" It was in grappling with this question that the "snapshot problem" finally clicked: real clients are working with streams of events, not one-off requests. This feels obvious in retrospect but it really wasn't from the start. I began to conceptualize a new kind of API in the form of an event stream. Each step in the chain of composition wouldn't just run once, once the past value had completed; it would listen continuously for the previous step to emit values at any time, re-run on each of those, and emit values of its own. (These kinds of streams aren't new–they exist as primitives or in libraries for languages like Kotlin, Java, Rust, and in fact JavaScript–but they have almost never been used for calling APIs, a task seemingly antithetical to the concept of a continuous stream.) Furthermore, the stream wouldn't have to output the entirety of the data each time. Instead, it could output just the parts of the data that had changed from the previous snapshot, emulating as closely as possible the event-driven architecture of most real clients.

With the end goal principles in mind, the design of more specific features began to fall into place. Before covering those, another recap. I ended up with a few major goals for what using Aqueduct should entail. First, it should be easy to build new integrations in a composable manner, ideally as part of an open-source ecosystem. Second, it needs to be able to reconcile differences in data richness, timing, and even reliability of success. Third, it should have great DX in the form of an event stream, so developers will actually build and use these integrations.

7 ARCHITECTURE

At a high level, the goal of Aqueduct is to make it easy to transform a snapshot-style API into an event stream-style API. Those event stream APIs need to be much better to use than the traditional ones, so developers will actually adopt them. And since no one developer can build every possible integration, it needs to be easy for anyone to build an integration and (ideally) contribute it to the ecosystem.

To turn snapshots into reactive event streams, Aqueduct has two core primitives: the `Stream` class and the `fetchReconcileDiff` function.

`Stream`, as one might expect, is responsible for constructing the reactive stream. `fetchReconcileDiff` is responsible for turning snapshots into a set of events, in a way that is as resource-efficient as possible. Together they combine to solve the entire transformation. (There are other helper functions that provide further help on top of `Stream` and `fetchReconcileDiff` for common use cases, but they are optional and used inside of these two primitives.

7.1 `Stream`

`Stream` chains together a sequence of functions, each one of which runs every time the previous one emits a value. If the previous function hasn't emitted a value yet, the next function won't run. Each step of the `Stream` can be one of many helper functions, which might transform data, handle asynchronous requests, or repeat a request every `n` seconds, to name a few.

`Stream` was originally conceived as a solution to the "authentication problem", where each data request relied on a trust that the current authentication state was valid, and required complex series of logic to deal with the times it inevitably wasn't. `Stream` inverts this: instead of cascading down authentication checks when a data request fails, the data requests only run once the authentication checks have succeeded.

This is especially useful when there are multiple steps to authenticate with a service. Consider the authentication flow for the Google Drive API, which begins with a verification code returned from the authorization screen on the Google Drive website. That verification code is then exchanged with the Google Drive server for an access token, which must be attached to every data request. To make it more challenging, that access token will eventually expire, and must be regenerated using the refresh token that was also returned from the server, so that refresh check also has to be run before making any data request. [1]

It is immensely complicated to keep track of which of these many prerequisites have been fulfilled, which is what `Stream` solves. With it, this mess of authentication becomes much easier to follow. Any new code value gets exchanged for an access token, each token value is periodically validated and refreshed if needed, and every validated token value can have data requests run on it. If the token becomes invalid at some point, the data requests simply stop until it's back.

This allows developers using these integrations to write much simpler code. Code requesting data only has to handle the actual request, instead of also handling checking and storing authentication, and vice versa.

It also imposes constraints on authentication when building the integrations, which counterintuitively make that process significantly simpler. Because each step of the chain can run an arbitrary amount of times, the function for each step must be idempotent (have no side effects). This is unusual. Many SDKs' authentication flows involve setting credentials once and assuming they'll still be there in future steps. But by having each step depend only its inputs, it removes the ability for stored credentials to silently get out of date and cause bugs. It also makes it clear what data is being used for which purposes, increasing the developer using it's understanding of how the authentication flow actually works.

With how simple these idempotent functions made authentication, the natural next step was to extend it to the other parts of the sync process. (This took a long time to figure out, but it was the breakthrough that finally solved how to make a legitimately better DX for developers using these integrations.) Each data request function is similarly idempotent, depending only on the prerequisites it takes as inputs, such as authentication tokens, previous results, and search parameters. This again has the benefit of making it very clear what information is needed to make these requests; no subtle bugs from mistakenly modified global state.

Another benefit of the `Stream` design is that it enables composability in a simple way. Instead of an integration having to consider where it gets its inputs from, it just has to take in any `Stream` of its inputs. Returning to the Google Takeout example from above, that integration's developer doesn't have to worry if it's getting its files from Google Drive or Gmail or OneDrive or a simple file upload. The Takeout integration simply needs to accept `.zip` files as input, and it can be added as a step on top of any existing `Stream` handling those other integrations. (For this example, it's worth noting that `Streams`–e.g. ones from different data sources all returning `.zip` files–can be easily combined.)

Yet another issue `Stream` helps to solve is the "snapshot problem": where traditional APIs only access the state of data at a single point in time. By providing steps in the process that run future fetch steps every *n* seconds, it emulates the experience of having a constant stream of notifications about changes to data. (While simple in concept, constant refresh is another area where non-`Stream` style APIs often run into the complex error and authentication handling issues from above.) Having to manually handle constant refreshing with updated data is a major complication in a world where almost all APIs are built to be used once per interaction, so this removes a major barrier.

All of these amount to a developer experience that is uniquely well-suited to the problem of keeping cloud data from various sources in sync, hopefully achieving the final goal of building a user-base that enables the ecosystem of integrations.

7.2 `fetchReconcileDiff`

`fetchReconcileDiff` is a function that takes as parameters a) new data from a source (an API, a data export, etc.) and b) the currently stored snapshot from that integration. It automatically figures out which items have been updated, fetches any additional information needed for those items, reconciles any differing formats, and compares the new snapshot to the old one to determine changes. It returns both the next stored snapshot and a listing of the changes made. These three processes–fetching, reconciling, and determining the changes–are all run simultaneously, since they share many of the same steps, and can be intertwined.

Typically, the input data passed to `fetchReconcileDiff` is incomplete in some way. Most API endpoints that list large amounts of data only include basic metadata like identifiers, which can be passed to more specific API calls to get more detail. (The same goes for data exports.) This is usually a problem–the "resource problem"–because making potentially hundreds of API calls to access all the details is enormously inefficient. But because `fetchReconcileDiff` usually knows which items have been edited since the last snapshot (by comparing metadata like a "last modified" timestamp), it only has to make those detailed requests for the portions of the data that have changed. It also has an optional cache system, so that if different items have some subset of their data that is shared, it only has to be fetched once. All of the logic for identifying which items have been changed and how to fetch more details are easily customizable by the developer, but are segmented so that the configuration is still very simple to understand.

When data from a source is run through `fetchReconcileDiff`, it is separated into three types: new items, overlapping items, and stale items (the saved data not present in the new data.) Any new items go through the full process of fetching additional data, reconciling it, and calculating events. For overlapping items, it compares the new data with the stored data to determine if it's been updated, and only runs the fetch, reconcile, and diff processes if it has. For stale data, there are simple configuration options to determine if they've been deleted, or if the input data is just incomplete.

To mark which parts of the data need to be updated, `fetchReconcileDiff` associates every item in the new data with its counterpart in the currently stored data if it exists. Having both the new and stored versions of an item during the fetch process allows for certain fetches to be cached, preventing redundant requests, but in the process, does most of the work of figuring out how to reconcile that data, and what's changed. So essentially, it gets the "reconcile" and "diff" parts of the process for free by figuring out an efficient "fetch" process. (This phenomenon is why all three processes are run at the same time.)

This reconciliation pattern also helps with temporal reconciliation. A big part of the snapshot problem is the fact that snapshots like data exports can be asynchronous, and thus any given snapshot might be out of date by the time it's processed. By reconciling with the existing data rather than replacing it, there isn't the risk of losing new data.

`fetchReconcileDiff` returns a data structure that includes both a snapshot of the newly updated source of truth, and more detailed information delineating what has been added, changed, or removed since the previous snapshot, based on the separated types of data it's already done. These details help reconstruct the specific changes to the data, rather than just the current state, making the "events" half of the "event stream" a reality.

7.3 Other design choices

One major design choice made early in the process is that Aqueduct is designed primarily to run in server environments rather than browser ones (though there are parts of it that can be run in the browser). The main reason for this is simple: due to cross-origin resource sharing (CORS) restrictions, it is often impossible to fetch data from a 3rd party server inside

a webpage with a different domain. [12] For example, it's not possible to access Notion data in-browser from any domain except `notion.so`, but it is possible from a separately-running server.

The original vision was to have everything runnable in-browser, to promote the idea of a client where users had all the data on-device, rather than it being stored on yet another server. But Aqueduct integrations should be able to be used by anyone, on any project, and most of those projects involve some central server. Even local-first projects like my original vision have increasingly been moving towards centralized servers that partially sync data to clients, but offload compute and storage. [15] And really, there is too much data being stored from these cloud services to have a copy of all of it on every device. Lastly, integrations running on servers allows them to access scraping libraries, which are key for getting data that isn't accessible through traditional APIs or exports (or even for automating that export process.)

Instead of enforcing that Aqueduct run on users' devices, as part of the demo application, I began building a self-hosted server that would live on a user's computer, handle most of the non-platform-specific syncing, and be the central store. Then, connected devices can only download the data relevant to them, and upload any data from the integrations specific to their platform. The server isn't a part of this thesis's scope (other than assisting with the demo application), but likely will be an important part of the future of the Aqueduct ecosystem. However, Aqueduct itself is storage-agnostic, again to make it as widely useful as possible.

Aqueduct is written in TypeScript, an interoperable version of JavaScript that encourages static typing of variables. This didn't (and doesn't) have to be the case. Aqueduct is primarily a design specification rather than a heavy implementation, and could easily be ported to other languages and platforms (more on that in the Future Work section). However, I deliberately chose TypeScript for a couple of reasons. First, it and JavaScript combine to be the 2nd most popular programming language in the world, and easily the most popular for user-facing software (the 1st, Python, mainly dominates in the machine learning arena). [7] This makes the population of developers who could use and build integrations with Aqueduct as large as possible. Moreover, web APIs and data exports almost always return their data in JSON format, which maps directly to the object format of JavaScript and TypeScript. (In fact, JSON stands for JavaScript Object Notation). This makes the language incredibly well-suited for working with these payloads. Finally, through projects like React Native, TypeScript can run on almost any device, albeit with worse performance than truly native code.

Finally, the structure of the integrations themselves. These integrations are simply collections of idempotent functions, along with some immutable global state shared among those functions (like client IDs for APIs, which are hardcoded values attached to every request, or SDK helpers for constructing the web requests).

I experimented with forcing more structure upon the integrations (especially at the start of the thesis), but the wide variety of necessary functions among different integrations made imposing one ontology impossible. Additionally, much of the sequencing I was trying to force is inherent in idempotent functions because they don't access stored state. With all of the input parameters and return types visible in each function's header, it's clear an authentication function returning an access token needs to be run before a data request function that takes that access token as a parameter, for example.

A benefit of this flexibility developers building integrations can use whatever other libraries inside of them they want, such as scraping libraries or official SDKs. Because there are no restrictions other than idempotency, it's no more challenging to build an entire integration than it is to write a small script making a single snapshot request. In fact, the design goal for *building* integrations in Aqueduct is simple: a developer should be able to write code in a test script make a regular, single snapshot-style request, then wrap that exact code in 'fetchReconcileDiff', an idempotent function, and a 'Stream', and (other than some light configuration), automatically get an event stream.

There are some ideas I want to explore in the future: adding sections for "flows" that combine commonly-chained steps into one function, and delineating functions that can be run only on-server, only in-browser, or on both platforms.

But these, too, would be design recommendations. At the end of the day, giving developers the flexibility to compose functions as fits their application is the most important factor.

8 DISCUSSION

Aqueduct shows that even with the limited, snapshot-style data access options available today, it's possible to use them in a much more modern, reactive event-driven style. Even differentiating between these two types creates a distinction that is helpful towards understanding how to improve our current methods of data access.

While it isn't yet ready for release, when it does, it will allow developers to get a stream of events of the data from a cloud application with little more work than making a single request to its API (or less, if they build on top of a pre-built integration). In turn, developers can create applications that act as alternative clients for cloud apps, operating off of almost full-featured, real-time data. (As discussed, getting that full-featured data is a barrier that is currently almost impossibly high.) The `demo` folder in the linked repository demonstrates a prototype of this kind of client app that is now possible.

The biggest limitation of this project is that it is, at some level, all theory. Though I iterated on and validated many of the design decisions through building the corresponding demo application, it remains to be seen if other developers see Aqueduct as an improvement over their current interactions with APIs. Hopefully over time, this ecosystem will grow to allow developers to integrate with almost all of the cloud applications users currently have their data siloed in.

8.1 Future work

Most of the future work on Aqueduct revolves around polishing it up in order to release it as a library anyone can use. It currently has some redundant files I used for explorations, some code that's inefficient, and it's lacking documentation on certain elements of its API. I also want to move integrations from one central hub inside the `aqueduct` package to be their own individual projects, so it's possible to pick and choose which ones to use.

In terms of making the code more efficient, as well as improving the developer experience in general, it would be helpful to think about a more advanced diffing helper. Right now, scaffolding is set up to automate diffing at the item level, but within items, it's up to the developer to differentiate the changes that have occurred. Adding a version of the JSON patch API [6], or something similar that easily supports Aqueduct's reconciliation needs would further improve the developer experience.

As illustrated in Overeem et al., another issue future versions of Aqueduct will have to tackle is the issue of data sources changing schema over time. [13] A flexible reconciliation protocol would ideally go a long way towards being flexible towards schema changes as well, but schema migration is one of the most challenging problems in application design, so will require serious effort.

Longer-term, Aqueduct will be most successful if it comes alongside a pre-built server that handles syncing data from various sources and exposes it to client applications. That server, called Reservoir, is something I plan to work on once Aqueduct has been finalized for its initial release. (Truly long-term, an application that displays this aggregation of users' data to them and lets them interact with it will drive adoption even further. The demo application in the `aqueduct` repository, Fountain, is an extremely early prototype of this.)

Also in the long-term, I'd like Aqueduct to be available on more platforms than just JavaScript ones. Since the API of the library is much more important than its current implementation, it could be easily ported into a more cross-platform language in the future.

9 REPOSITORY

For those looking to explore Aqueduct in more detail, the library and a demonstration application is available at `https://github.com/andyburris/aqueduct`.

10 ACKNOWLEDGMENTS

I'd like to thank my advisor, Jeff Huang, for his guidance, advice, and many insights throughout the thesis process, as well as my second reader, Nikos Vasilakis, for his advice and feedback on my work. This is the culmination of all the learning and thinking I've done at Brown about computers, so I'd also like to thank all the professors that have taught me along that journey. Finally, to my parents: thank you for letting me talk to you endlessly about this project, and for unconditionally supporting my love of technology throughout my life.

REFERENCES

[1] [n.d.]. Google-Api-Nodejs-Client/README.Md. https://github.com/googleapis/google-api-nodejs-client/blob/main/README.md#oauth2-client.
[2] [n.d.]. Mainframe. https://www.mainframe.so/.
[3] [n.d.]. Nango. https://www.nango.dev/.
[4] [n.d.]. Sequin. https://sequin.io/.
[5] 2025. Surfer-Org/Protocol. Surfer-Org.
[6] Hanyang Cao, Jean-Rémy Falleri, Xavier Blanc, and Li Zhang. 2016. JSON Patch for Turning a Pull REST API into a Push. In *Service-Oriented Computing*, Quan Z. Sheng, Eleni Stroulia, Samir Tata, and Sami Bhiri (Eds.). Vol. 9936. Springer International Publishing, Cham, 435–449. https://doi.org/10.1007/978-3-319-46295-0_27
[7] Stephen Cass. 2024. Top Programming Languages 2024. https://spectrum.ieee.org/top-programming-languages-2024.
[8] Martin Kleppmann, Adam Wiggins, Peter Van Hardenberg, and Mark McGranaghan. 2019. 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*. ACM, Athens Greece, 154–178. https://doi.org/10.1145/3359591.3359737
[9] Jan Krämer, Pierre Senellart, and Alexandre de Streel. [n.d.]. Making Data Portability More Effective for the Digital Economy. ([n. d.]).
[10] Sophie Kuebler-Wachendorff, Robert Luzsa, Johann Kranz, Stefan Mager, Emmanuel Syrmoudis, Susanne Mayr, and Jens Grossklags. 2021. The Right to Data Portability: Conception, Status Quo, and Future Directions. *Informatik Spektrum* 44, 4 (Aug. 2021), 264–272. https://doi.org/10.1007/s00287-021-01372-w
[11] Robert Luzsa, Susanne Mayr, Emmanuel Syrmoudis, Jens Grossklags, Sophie Kübler-Wachendorff, and Johann Kranz. 2022. Online Service Switching Intentions and Attitudes towards Data Portability – The Role of Technology-related Attitudes and Privacy. In *Mensch Und Computer 2022*. ACM, Darmstadt Germany, 1–13. https://doi.org/10.1145/3543758.3543762
[12] Mozilla. 2025. Reason: CORS Header 'Access-Control-Allow-Origin' Missing - HTTP | MDN. https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/CORS/Errors/CORSMissingAllowOrigin.
[13] Michiel Overeem, Marten Spoor, and Slinger Jansen. 2017. The Dark Side of Event Sourcing: Managing Data Conversion. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 193–204. https://doi.org/10.1109/SANER.2017.7884621
[14] Artur Piszek. 2021. Composability Is the Only Game in Town – Roam, Shipping Containers, Lego and Twitter. https://piszek.com/2021/02/07/composability/.
[15] Rocicorp. [n.d.]. Zero. https://zero.rocicorp.dev/.
[16] Spotify Engineering. 2020. Spotify Modernizes Client-Side Architecture to Accelerate Service on All Devices. https://engineering.atspotify.com/2020/05/spotify-modernizes-client-side-architecture-to-accelerate-service-on-all-devices/.
[17] Emmanuel Syrmoudis, Robert Luzsa, Yvonne Ehrlich, Dennis Agidigbi, Kai Kirsch, Danny Rudolf, Daniel Schlaeger, Joelle Weber, and Jens Grossklags. 2024. Unlocking Personal Data from Online Services: User Studies on Data Export Experiences and Data Transfer Scenarios. *Human–Computer Interaction* (March 2024), 1–25. https://doi.org/10.1080/07370024.2024.2325347
[18] Emmanuel Syrmoudis, Stefan Mager, Sophie Kuebler-Wachendorff, Paul Pizzinini, Jens Grossklags, and Johann Kranz. 2021. Data Portability between Online Services: An Empirical Analysis on the Effectiveness of GDPR Art. 20. *Proceedings on Privacy Enhancing Technologies* 2021, 3 (July 2021), 351–372. https://doi.org/10.2478/popets-2021-0051
[19] Norases Vesdapunt and Hector Garcia-Molina. 2016. Updating an Existing Social Graph Snapshot via a Limited API. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. ACM, Indianapolis Indiana USA, 1693–1702. https://doi.org/10.1145/2983323.2983703

A THE AQUEDUCT SYSTEM

Though the general `Stream`/`fetchReconcileDiff` combination addresses the core principles of my research, the specific design choices of details within each of those components, as well as the other helpers, all are important for creating a great ecosystem of integrations.

A.1 'Stream'

First up is `Stream`, which has been described at a higher level above. In my implementation, this is a wrapper around the `xstream` library for JavaScript, but it could be reasonably converted to any of the stream libraries in most major programming languages. (It's even possible to create a custom stream implementation. Originally, Aqueduct had a custom, from-scratch `Stream` class, but moved to the `xstream` wrapper for robustness.) Instead, the design of the API for using `Stream`--the methods available on it, its initializers, and its returns--and that API's suitability for syncing data is the novel area.

Each step in the chain of a `Stream` runs every time the previous step emits any value, and can emit as many or few values of their own as they want (including none).

A `Stream` can be created with a number of initializers. Some of them are basic, like `Stream.from([1, 2, 3])` and `Stream.of(1, 2, 3)`, which emit all of their values essentially instantaneously, and are mainly used for testing purposes. Others are more specific to the task. The data that initializes a `Stream` is usually the first data in the process; often something like an authentication token. That data has to get to the server somehow, and there are two initializers that are specifically designed for the task of starting integrations on a server:

(1) `Stream.fromListener` is used when the initializing data is stored in something that can be listened to (e.g a database with subscription functionality). Then, a stream can be started using, for example, `Stream.fromListener(emit => db.subscribe(data => emit(data))`.
(2) `Stream.fromHandle` is used if the initializing data is a one-off communication (e.g. if the server handles HTTP PUSH requests from the client). A stream can be started using `const [handle, stream] = Stream.fromHandle<Type>()`, and later begun by running `handle(data)`. Crucially, this initializer returns both the `Stream` itself to be built on top of, and the callback handle that can be returned from the function that sets up the entire stream, and then passed around to whatever locations need to call it and begin the stream. There is another function, `Stream.create`, that is the generic initializer provided by the `xstream` library, that these are built on top of, and allows for finer-grained control.

Once a `Stream` has been initialized, there are a number of transformation functions that can be run on it. Each returns another `Stream`, just with another step added to it. Again, many of these such as `stream.take(n: number)` and `stream.drop(n: number)` are trivial. There are many, though, that are essential for the ergonomics of chaining together functions from integrations.

- `stream.map`, like the `map` function on lists in many standard libraries, transforms data from one type into another using a given mapping function. There are some important nuances in Aqueduct's implementation, however:
  - Foremost is that rather than having a separate `mapAsync` function, `map` accepts both synchronous and asynchronous transformation functions and treats them equally. Most sync functions in the integrations require some kind of asynchronous web access, so it makes sense to treat them as a first-class citizen. Simultaneously, many of these supposedly asynchronous functions won't actually be that way due to the caching capabilities of `fetchReconcileDiff`, so it further makes sense to treat them interchangeably.

  - Additionally, it accepts a parameter for the number of parallel functions that can be running. If an asynchronous function takes longer to run than the preceding function emits events, requests will be duplicated and unnecessary resources will be used, which this parameter prevents.
- `stream.filter` follows a similar structure, but replaces the `filter` function. Notably, this means that not every event from a source stream will get emitted farther down the chain.
- `stream.dropRepeats` is a key for limiting resource consumption. Because Aqueduct can work with any storage framework, it's hard to tell exactly how many times the `Stream`'s initializer will emit. (In the example with the subscription-capable database, it might notify subscribers any time any data is changed, even if that data isn't the exact subset that's being observed.) Preventing repeats prevents redundant requests that chew up rate limits.
- `stream.onEach` is a version of `map` that runs a given function on every emit, but always returns the input value. It's useful for two things: logging and saving things outside of the `Stream`:
  - Adding a `console.log` that prints on each event is essential while debugging if something in the chain is going wrong.
  - If a developer wants to keep an external source up to date about things going on inside the chain, they can do it in an `onEach`. For example, a developer could add an `onEach` step after a login token has been exchanged, but before it's been used for a web request, that saves it to a database (so that if the server were to restart, the user wouldn't have to log back in).
- `stream.every` is the function that enables the automatic refresh functionality. It takes in an interval to refresh upon, as well as optional functions to save the timestamp of the emit to a database and load that timestamp back (so a server that restarted in the middle of a 10 hour refresh process won't cut the wait time short). Then it emits its most recent input value upon each interval, triggering the sync functions downstream.
- `stream.combine` takes the outputs of two streams and any time one emits, it emits with the latest values of both streams (provided both have emitted at least once). This is key for bringing together different data sources (e.g. an authentication token that is passed from an HTTP handler the first time, but retrieved from a database in subsequent initializations). Since Aqueduct is agnostic to the storage and servers it's being run on, these functions can be combined in any way that is needed to support that specific architecture, the integrations the developer wants to use, and more.

The final component of the `Stream` process is the listener, which contains a callback that runs on every emitted value, but doesn't emit any of its own. In order for an initializer to begin running, a listener *must* be attached. It's called much like the transformation functions, but returns `void` instead of a `Stream`. It takes a function much like the one passed to `onEach`, plus two additional optional functions `onError` and `onComplete` (which is called if one of the basic initializers, like `Stream.of(1, 2, 3)`, runs out of values).

Why is it necessary to attach a listener to initialize rather than just having any created `Stream` automatically initialize and run (replacing the listener with a simple `onEach`)? The basic problem is this: the initializer might start emitting values before all the steps are attached. In a simplified representation, each step of the `Stream` essentially has a handle for passing it values, and a slot that takes in another stream's handle, which it calls when it needs to emit values to. Any time a step is added to a `Stream`, it sets the slot of the previous step to its handle. But if at any point that slot is empty, the value just isn't emitted. There are a number of ways to fix this (all of which I tried):

- One option is to delay every initializer by some number of milliseconds, in order for all of the steps to get attached. This quickly becomes a race condition, however; what if a really slow computer or an asynchronous attachment took more than the arbitrary delay?
- Another option is to store the most recently emitted value alongside the slot, and anytime the slot gets filled, the most recent value gets passed to it. In addition to adding a lot of complexity to the internal workings, this also creates a much more subtle race condition. Imagine a hypothetical function `Stream.delayedBy([1, 2, 3], milliseconds(2))`. By only storing the most recent value, depending on the speed of attachment, subsequent steps might get all of the emissions or only some recent subset. (Storing and replaying the results of every emission is, of course, extremely impractical.)
- The final, and best option, is to only run the initializer once a listener has been attached, marking that there will be no more steps attached on top of that. Only then (which, to be clear, is usually less than a millisecond later) can the initializer be sure that every step will get every value. (This still has a slight issue where if the handle from `Stream.fromHandle` somehow gets returned and called before a listener is attached, it won't emit, but this is an edge case that should be obviously wrong.)

A.2 'fetchReconcileDiff'

Similar to `Stream`, `fetchReconcileDiff` has been described above at a high level, but has a number of interesting details.

The key to understanding `fetchReconcileDiff` is understanding its types. `fetchReconcileDiff` is a generic function that takes in 5 generic types. Any possible type can be assigned to each of those generic types, but those types are then enforced for that instance of `fetchReconcileDiff`. (In real-world usage, these types usually don't have to be set explicitly--TypeScript infers them from the context of the other code.)

These generic types are:

(1) `Current` - the type of the data being passed into the function, which is usually incomplete or different from the already-saved data in some way.
(2) `Stored` - the type of the data that will be output from the function, as well as the type of the previously stored data that is passed into the function as the previous snapshot.
(3) `Identifier` - the type of the subset of the input data that uniquely identifies a certain item. It defaults to a `string`.
(4) `Signature` - the type of the subset of the input data that uniquely identifies a certain *version* of an item. If an item has changed between snapshots, the `Identifier` should be the same, but the `Signature` should be different.
    - As an example, Spotify playlists include a value called `snapshot_id` that changes any time a change is made to a playlist. If the `snapshot_id` of the `Current` and `Stored` versions of the playlist are the same, they'd be identical at the end of the fetch and convert process.
(5) `Cache` - the type of the cache holding relevant stored information that might be shared amongst previously stored and truly new items. Defaults to a `Map` from `Signature` to `Stored`.

With these types in mind, the syntax of `fetchReconcileDiff` can be explored. `fetchReconcileDiff` is a single, asynchronous function that takes in six required parameters, along with a number of optional ones. The six required are as follows:

(1) `currentItems: Current[]` The list of items that will be fetched, converted, and diffed.
    - (At some point, this will accept non-lists as well, but in practice so far almost every single source returns what is essentially a list.)

(2) `storedItems: Stored[]` The list of currently stored items
   - (Again, at some point this could take in other types including `Map`s or even database connections which might be more memory-efficient, but which would provide essentially the same abstraction.)

(3) `currentIdentifier: (currentItem: Current) => Identifier` A function that maps a `Current` item to the value that identifies it uniquely among other `Current` items

(4) `storedIdentifier: (storedItem: Stored) => Identifier` A function that maps a `Stored` item to the value that identifies it uniquely among other `Stored` items. For `Current` and `Stored` items to be properly associated with each other, their identifiers must be equal.

(5) `convert: fetchReconcileDiffConvert<Current, Stored, Cache>` A function that transforms `Current` items into `Stored` ones, often by fetching additional information from an API, reconciling with the corresponding `Stored` item, or getting cached data from other stored items.
   - `fetchReconcileDiffConvert` is a union type that allows for two different kinds of conversions, `each` or `all`
     - `each: (currentChangedItem: Current, cache: Cache, correspondingStoredItem?: Stored) => Stored | Promise<Stored>` has the developer provide a function that runs on each `Current` item and converts it to its `Stored` format with the help of the `Cache` and its corresponding `Stored` item from the source of truth, if it exists.
     - `all: (currentChangedItems: Current[], cache: Cache, correspondingStoredItems: (Stored | undefined)[]) => Stored[] | Promise<Stored[]>` is the same, but the function provides the entire list to the developer to convert all at once. This is useful, for example, to batch data requests if an API lets you get detailed information about multiple items at once. (Later on, the `fetchWindowed` helper will explain this in more detail.)
   - Notably, synchronous or asynchronous functions can be used interchangeably, for the same reason as in `Stream.map`

(6) `keepStaleItems: boolean | ((staleItems: Stored[], nonStaleItems: Stored[]) => Stored[])` A value or function that determines if stale items (ones that weren't included in the current list) are kept or removed.
   - In some cases, missing items in new data doesn't mean stale data should be removed. For instance, Spotify's Recent Listens endpoint only returns the 50 most recent tracks, which shouldn't overwrite all of the previous ones, so `keepStaleItems` should be true.
   - Other times, the opposite is true. If Spotify's Playlists endpoint is missing a playlist that previously was saved, it means that playlist has been deleted, so `keepStaleItems` should be false
   - Finally, there are some times where some stale items should be kept, and some removed. For those situations, `keepStaleItems` also accepts a function that filters down which should be kept. One example of when this might be necessary is discrepancies between different sources in the same integration. If a Spotify user is playing a song, and scrubs to a new position in the track, the Currently Playing endpoint counts it as a brand new "listen". But Spotify's Data Export, the more official source of truth, counts it only as one listen. If listens have been saved from Currently Playing, and now are being imported from an export, any stale listens between the earliest and latest listen of the export should be removed. But since the export takes up to a month to create, Currently Playing will often have picked up new listens after the latest listen, and those, while "stale", are just due to incomplete data, and should be kept. By calculating the time range of `nonStaleItems`, and keeping only the `staleItems` outside that time range, this is possible without leaving the `fetchReconcileDiff` framework.

There are some other optional parameters to `fetchReconcileDiff` for more complicated workflows:

- `currentSignature` and `storedSignature` are the same as `currentIdentifier` and `storedIdentifier`, but for the `Signature` that determines if items are the same version. For example, in any API that returns a timestamp for when an item was last modified, the signature can simply be the item's identifier plus its last modified timestamp.
- `createCache` allows the developer to create a custom cache of information to be checked before fetching new data from an API, which is available in `convert`. If the developer doesn't specify a custom one, it defaults to a `Map` from each item's `Signature` to that item.

Finally there are two helper functions for the actual data fetching process, which might be called inside `convert` or to get the data passed to `currentItems`:

- `fetchPaged` is for situations where a developer is getting the initial list of items from an API, but it has a maximum number of results it returns per request, and to get more, the developer has to request subsequent "pages" of items. This helper automates requesting all of those pages, so they can be treated like one single request that returns all of the items.
- `fetchWindowed` is for situations where a developer has already gotten a list of simple items, but needs to get more detail for each, *and* the API has a mechanism to request details for multiple ids at once (with some maximum batch size). `fetchWindowed` automatically batches the simple data to a certain maximum size, then runs a single request for each batch.

The internal flow of `fetchReconcileDiff` consists of five main steps:

(1) Assign an `Identifier` and a `Signature` to each `Current` item and `Stored` item, and associate them if they're the same
(2) Use those values to classify each item into one of four buckets: new, updated, unchanged, or stale.
(3) Create the cache with `createCache`
(4) Use the cache and associated items to run `convert`
(5) Based on the original buckets (and with the stale items split into stale and removed items by `keepStaleItems`), return the result, consisting of each bucket and a combination of all kept items.

A.3 Other helpers

There are a couple other helper functions in Aqueduct that make it easier to deal with common tasks in the sync workflow:

- `generateUUID` is a function that creates a new UUID (useful for assigning an identifier to data that comes without one) and works on server and browser interchangeably.
- `unzipFiles` and its little brother, `unzipFile`, are for handling data exports, most of which come in `.zip` format