The InfoQ eMag / Issue #108 / June 2023

Architectures You've Always Wondered About

InfoQ

Magic Pocket: Dropbox's Exabyte-Scale Blob Storage System

Tales of Kafka at Cloudflare: Lessons Learnt on the Way to 1 Trillion Messages

Banking on Thousands of Microservices

Architectures You've Always Wondered About

IN THIS ISSUE

PRODUCTION EDITOR Ana Ciobotaru / COPY EDITOR Susan Conant / DESIGN Dragos Balasoiu, Ana Ciobotaru
GENERAL FEEDBACK feedback@infoq.com / ADVERTISING sales@infoq.com / EDITORIAL editors@infoq.com

CONTRIBUTORS

Facundo Agriel

is currently the tech lead for Dropbox's exabyte-scale blob storage system. This team manages everything from customized storage machines with many petabytes of capacity to the client APIs other teams use internally. Prior to working at Dropbox, Facundo worked at Amazon on a variety of scheduling problems for Amazon's last mile delivery team.

Suhail Patel

is a staff engineer at Monzo focused on building the Core Platform. His role involves building and maintaining Monzo's infrastructure which spans nearly two thousand microservices and leverages key infrastructure components like Kubernetes, Cassandra, Etcd and more. He focuses specifically in investigating deviant behaviour and ensuring services continue to work reliably in the face of a constantly shifting environment in the cloud.

Matt Boyle

is an experienced technical leader in the field of distributed systems, specializing in using Go. He has worked at huge companies such as Cloudflare and General Electric, as well as exciting high-growth startups such as Curve and Crowdcube. Matt has been writing Go for production since 2018 and often shares blog posts and fun trivia about Go over on Twitter (@MattJamesBoyle).

Andrea Medda

is a Senior Systems Engineer at Cloudflare. He loves using cutting edge technology and approaches to solve real customer problems. Andrea loves Golang and distributed systems. He has a pet friend Maru that loves to pop by in his meetings.

Bilgin Ibryam

is a technical product manager at Diagrid, working on developer productivity tools. Prior to this role, he served as a consultant and architect at Red Hat and has also been a committer and member of the Apache Software Foundation. Bilgin has also co-authored two books on Kubernetes Patterns and Camel Design Patterns. In his spare time, Bilgin enjoys writing and sharing his learnings through blogging and other means.

Thomas Betts

is the Lead Editor for Architecture and Design at InfoQ, a co-host of the InfoQ Podcast, and a Laureate Software Architect at Blackbaud. For over two decades, his focus has always been on providing software solutions that delight his customers. He has worked in a variety of industries, including social good, retail, finance, health care, defense and travel. Thomas lives in Denver with his wife and son, and they love hiking and otherwise exploring beautiful Colorado.

A LETTER FROM THE EDITOR

As the flagship track at QCon, Architectures You've Always Wondered About showcases real-world examples of innovator companies pushing the limits with modern software systems. At recent conferences in San Francisco and London, the speakers clearly showed what "scalable" can really mean, from a trillion messages to exabytes of data. This eMag brings together several of these stories and hopefully provides advice and inspiration for your future projects.

At Dropbox, it is crucial to provide high performance and availability for the data storage solutions they provide. Facundo Agriel goes deep into the details about Dropbox's exabyte-scale blob storage system for Magic Pocket. In addition to all the software that handles millions of queries per second, the physical hardware design must also be considered. Because each storage device contains over 100 drives, and hardware devices can fail, the system has to be self-healing. The architecture has to consider this and many other factors to achieve Dropbox's performance targets while remaining cost-effective.

Monzo serves 7 million banking customers daily, on an architecture of thousands of microservices that they deploy hundreds of times a day. Suhail Patel covers some of the specific technologies they use, including Cassandra and Kubernetes, and how a focus on platform engineering and developer experience has allowed them to scale up, even with a relatively lean engineering team. Because nothing ever goes entirely as planned, there were some mistakes and incidents along

the way, but those led to some valuable lessons and a better system in the end.

Cloudflare provides DDoS protection for websites around the globe, and their services have relied on Kafka to send over 1 trillion messages. However, decoupling those microservices, and the teams that build them, required dedicated effort.

Changing the message format from JSON to Protobuf, and the development of custom tools, led to more structured messages and better fault tolerance, which in turn reduced the cognitive load of teams and increased adoption. Andrea Medda and Matt Boyle explain how balancing technical and socio-technical needs is necessary to create a successful solution.

The eMag wraps up by looking at a new architecture pattern, which Bilgin Ibryam calls cloud-bound applications. This is the evolution of cloud-native architecture, and moves the abstraction of compute-centric concerns to an abstraction of application integration concerns. The CNCF project Dapr provides APIs that speak to these integration needs and allow architects to design a clean separation around core business logic.

We would love to receive your feedback via editors@infoq.com or on Twitter about this eMag. I hope you have a great time reading it!

Magic Pocket: Dropbox's Exabyte-Scale Blob Storage System

by **Facundo Agriel,** Software Engineer / Tech Lead @ Dropbox

At QCon San Francisco, I explained how the exabyte scale blob storage system that stores all of Dropbox's customer data works. At its core, Magic Pocket is a very large key-value store where the value can be arbitrarily sized blobs.

Our system has over 12 9s of durability and 99.99% of availability, and we operate across three geographical regions in North America.

Our systems are optimized for 4 MB blobs immutable writes, and cold data.

Magic Pocket manages tens of millions of requests per second and a lot of the traffic comes from verifiers and background migrations. We have more than 600,000 storage drives currently deployed and we run thousands of compute machines.

Object Storage Device

The main focus of the Magic Pocket is the Object Storage Devices (OSD). These devices have over 2 PB of capacity and are made up of around 100 disks per storage machine, utilizing Shingled Magnetic Recording (SMR) technology.

SMR differs from Conventional Magnetic Recording drives as it performs sequential writes instead of random writes, allowing increased density.

The tradeoff of using SMR is that the head erases the next track when you walk over it, preventing random writes in any place.

However, this is perfect for our workload patterns. SMR drives also have a conventional zone that allows for caching of random writes if necessary, which typically accounts for less than 1% of the total capacity of the drive.

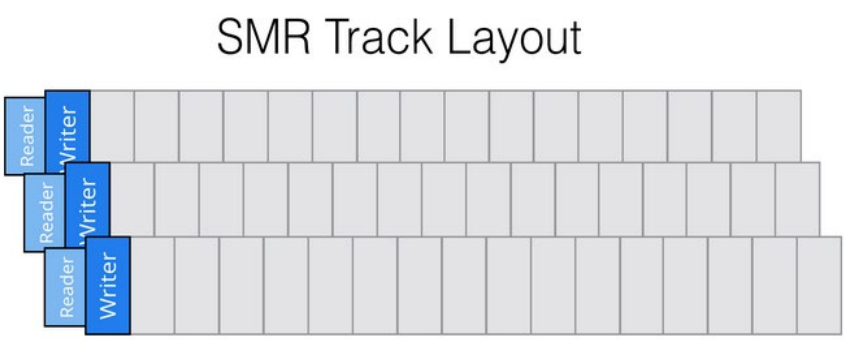


Figure 1: SMR Track Layout

At a high level, the architecture of Magic Pocket consists of three zones: West Coast, Central, and East Coast. The system is built around pockets, which represent logical versions of everything in the system.

Magic Pocket can have multiple instances, such as a test pocket or a stage pocket before the production one. Databases and compute are not shared between pockets, and operate independently of each other.

Zone

These are the different components of the Magic Pocket architecture in each zone.

The first service is the frontend, which is the service that interacts with the clients. Clients typically make PUT requests with keys and blobs, GET requests, delete calls, or perform scans for available hashes in the system.

When a GET request is made, the hash index, a collection of sharded MySQL databases, is queried. The hash index is sharded by the hash, which is the key for a blob, and each hash is mapped to a cell or a bucket, along with a checksum. The cells are the isolation units where all the storage devices are located: they can be over 100 PBs and grow up to specific limits in size. When the system runs low on capacity, a new cell is opened up, allowing for horizontal scaling of the system.

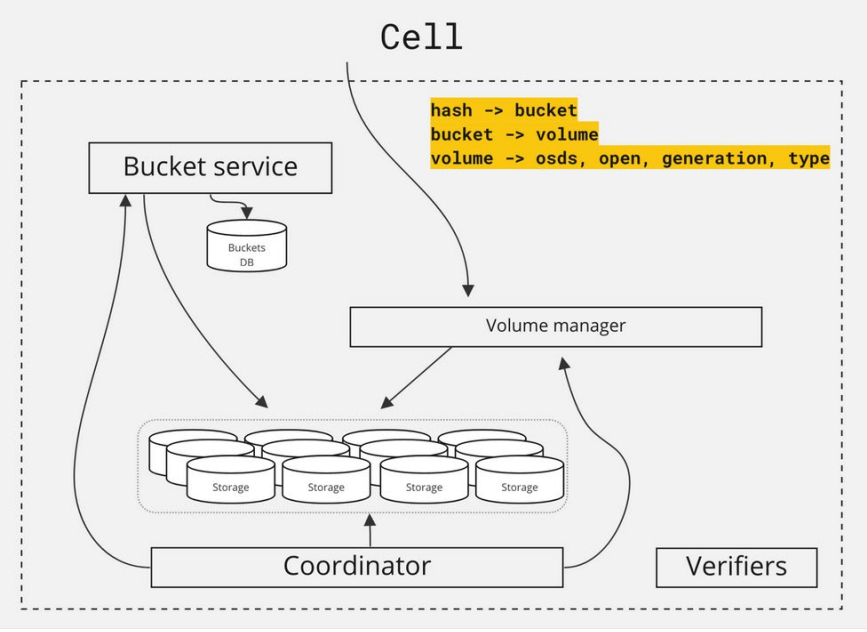


Figure 2: How a zone works

The cross-zone replicator is the component performing cross-zone replication, storing data in multiple regions. The operation is done asynchronously, and once a commit happens in the primary region, the data is queued up for replication to another zone. The control plane manages traffic coordination, generates migration plans, and handles machine re-installations. It also manages cell state information.

Cell

If I want to fetch a blob, I need to access the bucket service that knows about buckets and volumes: when I ask for a bucket, my request is mapped to a volume and the volume is mapped to a set of OSDs.

Once we find the OSD that has our blob, we can retrieve it. For writing data, the frontend service figures out which buckets are open for writing and it commits to the ones that are ready. The buckets are pre-created for us,

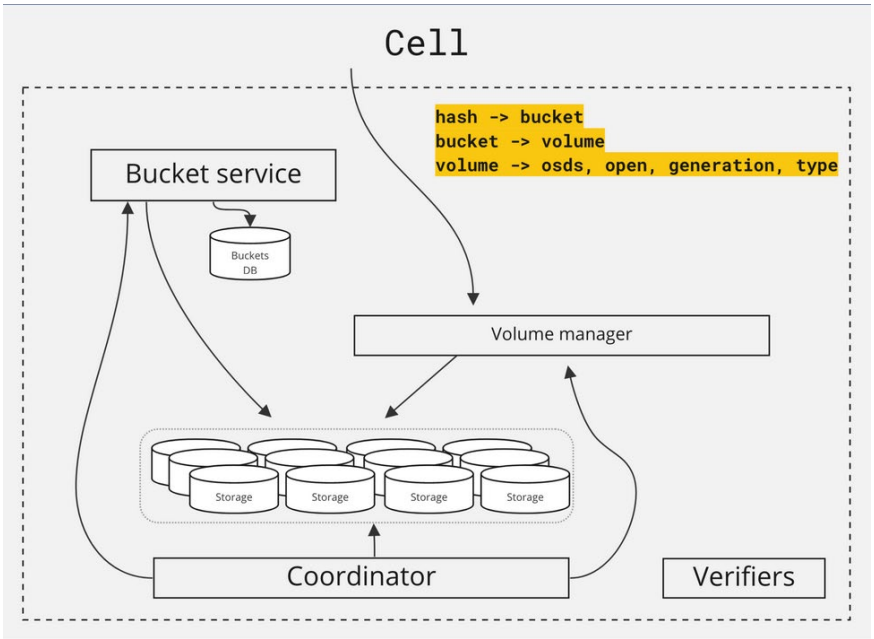


Figure 3: How a cell works

and the data is stored in a set of OSDs within a volume.

The coordinator is an important component in the cell, managing all the buckets, volumes, and storage machines. The coordinator constantly checks the health of the storage machines, reconciles information with the bucket service and database, and performs erasure coding and repairs: It optimizes data by moving things around within the cell and takes care of moving data to other machines when it is necessary.

The volume manager handles the reading, writing, repairing, and erasure encoding of volumes. Verification steps happen both within and outside of the cell.

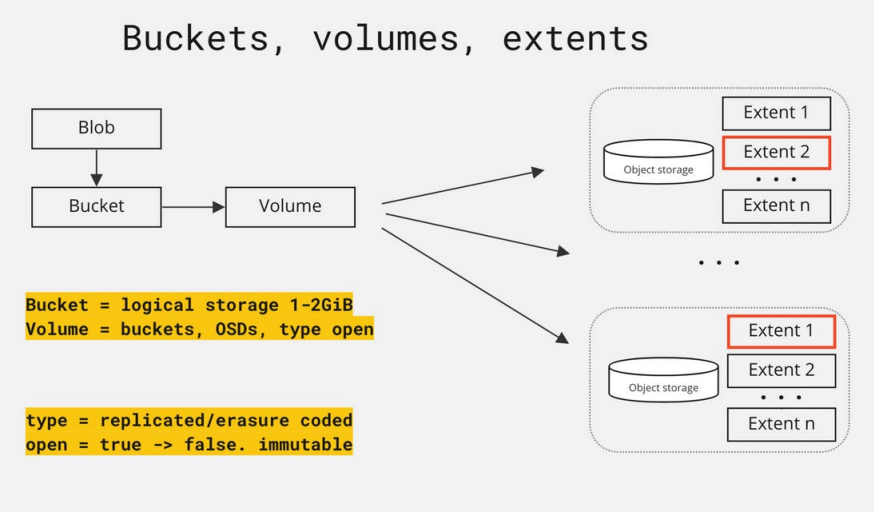


Figure 4: Buckets, volumes, and extends

Buckets, Volumes, and Extents

We can now dive deeper into the components of the Magic Pocket storage system, namely buckets, volumes, and extents.

A bucket is a logical storage unit associated with a volume and extent, which represents 1-2 GBs of data on a disk. When we write, we identify the open buckets and the associated OSDs and then write to the extents.

The coordinator manages the bucket, volume, and extent information, and can ensure that data is not lost by finding a new placement for a deleted extent.

A volume is composed of one or more buckets, it is either replicated or erasure coded, and it is open or closed. Once a volume is closed, it is never opened up again.

How to Find a Blob in Object Storage Devices

In this chapter, we learn how to find a blob in a storage machine. To do this, we store the address of the OSDs with the blob, and we talk directly to those OSDs.

The OSDs load up all the extent information and create an in-memory index of which hashes they have to the disk offset. If we want to fetch the block, we need to know the volume and which OSDs have the blob. For a PUT, it's the same process, but we do a write to every single OSD in parallel and do not return until the

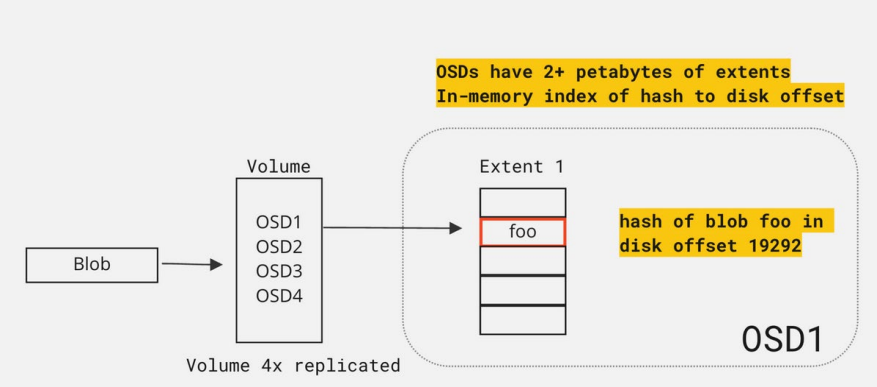


Figure 5: Finding a Blob

write has been completed on all storage machines. As the volume is 4x replicated, we have the full copy available in all four OSDs.

Erasure Coding

While failures are happening all the time, 4 copies by 2 zones replication is costly. Let's see the difference between a replicated volume and an erasure-coded volume, and how to handle it.

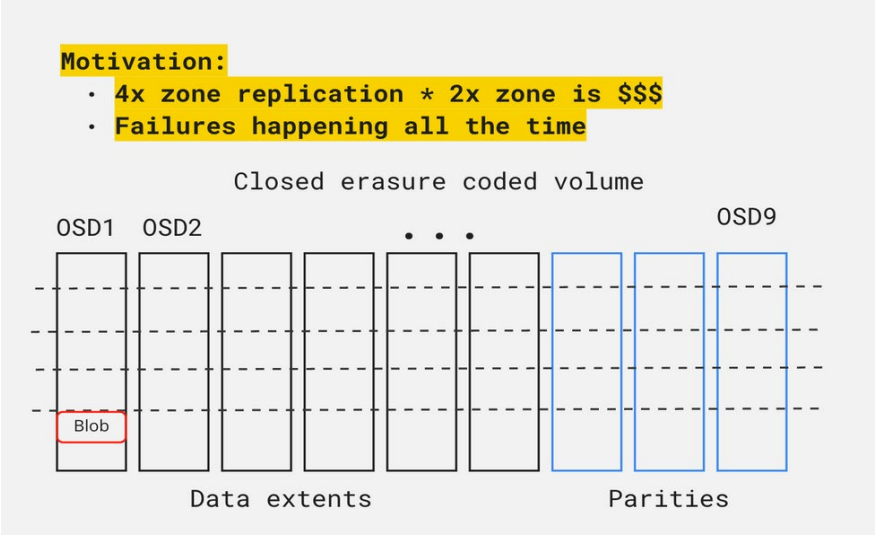


Figure 6: Erasure Coding

Erasure coding is a way to reduce replication costs while maintaining similar durability as replication. In our system, when a volume is almost full, it is closed and eligible to be erasure coded. We use an erasure code, like Reed Solomon error correction 6 plus 3, with 6 OSDs and 3 parities in a volume group. This means there is a single blob in one data extent, and if one OSD fails, it can be reconstructed.

Reconstructions can happen on live requests for data or done in the background as part of repairs.

There are many variations of erasure codes with different tradeoffs around overhead: for example, using XOR as an erasure code can be simple, but custom erasure codes can be more suitable.

The paper "Erasure Coding in Windows Azure Storage" by Huang and others is a useful resource on the topic, and we use similar techniques within our system.

I previously mentioned an example of Reed Solomon 6, 3 codes with 6 data extents and 3 parities. Another option is called local reconstruction codes, which optimizes read cost. Reed's 6, 3 codes result in a read penalty of 6 reads when there are any failures. However, with the local reconstruction codes, you can have the same read costs for one type of data failure but with a lower storage overhead of roughly 1.33x compared to Reed Solomon's 1.5x replication factor. Although this may not seem like a huge difference, it means significant savings on a larger scale.

The local reconstruction codes optimize for one failure within the group, which is usually what you encounter in production. Making this tradeoff is acceptable because more than 2 failures in a volume group are rare.

Even lower replication factors are possible with these codes: the LRC-(12,2,2) code can tolerate any three failures within

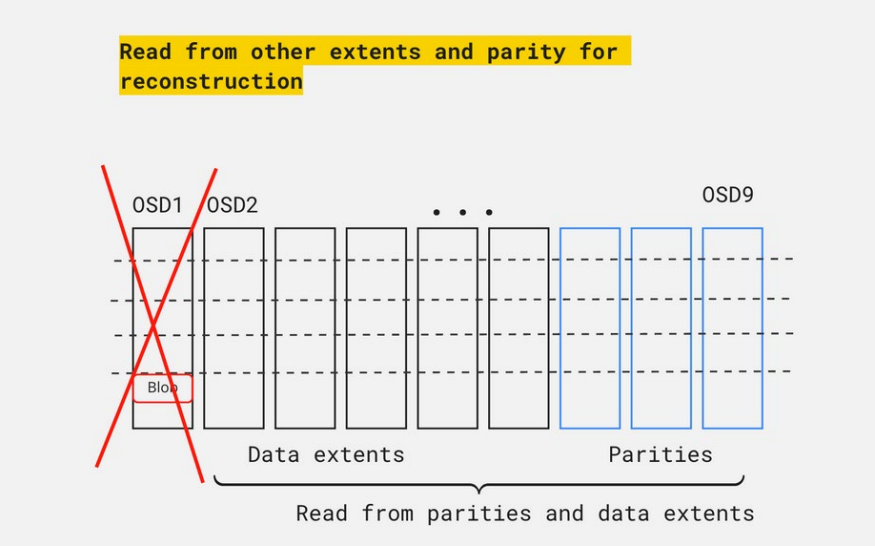


Figure 7: Failure and erasure coding

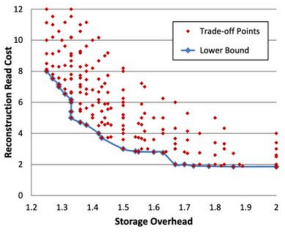


Figure 4: Overhead vs. Recon. Cost.

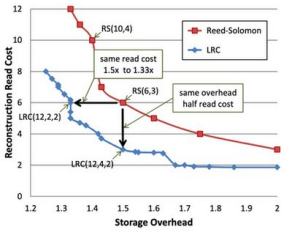


Figure 5: LRC vs. RS Code.

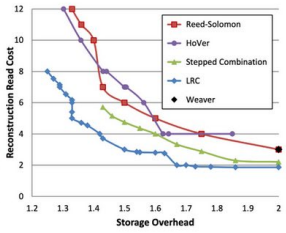


Figure 6: LRC vs. Modern Codes.

Figure 8: Reed Solomon error correction by "Erasure Coding in Windows Azure Storage" by Huang et al

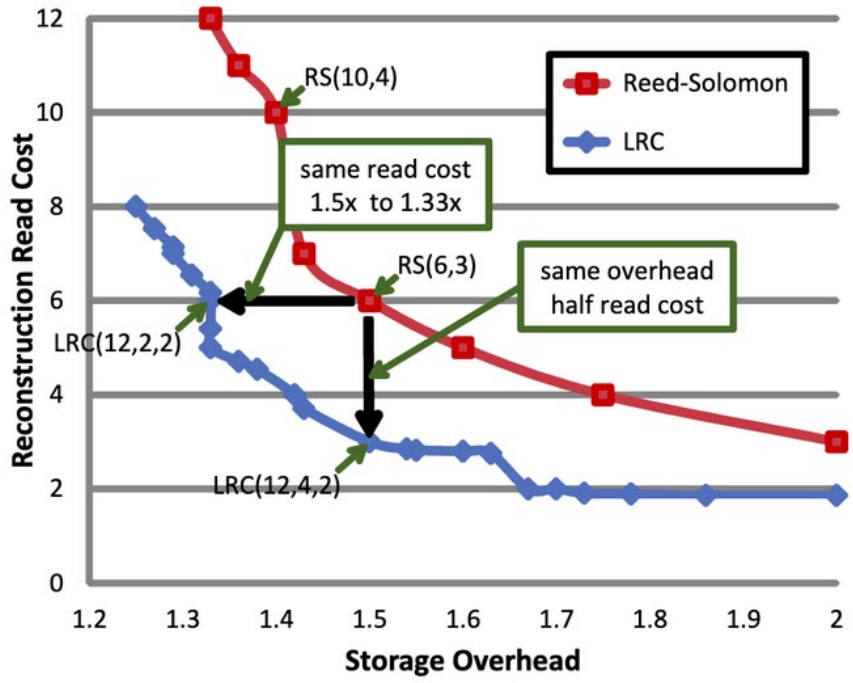


Figure 9: Reconstruction code comparisons from "Erasure Coding in Windows Azure Storage" by Huang et al

the group, but not four, with only some failures that can be reconstructed.

The Cold Storage System

Can we do better than this for our system? As we have observed that 90% of retrievals are for data uploaded in the last year and 80% of retrievals happen within the first 100 days, we are exploring ways to improve our cross-zone replication

As we have a large amount of cold data that is not accessed frequently, we want to optimize our workload to reduce reads and maintain similar latency, durability, and availability. To achieve this, we observe that we do not have to do live writes into cold storage and can lower our replication factor from 2x by utilizing more than one region.

Let's see how our cold storage system works, with the inspiration coming from Facebook's warm blob storage system. The f4 paper suggests a method to split a blob into two halves and take the XOR of those two halves, which are stored individually in different zones. To retrieve the full blob, any one combination of blob1 and blob2 or the XOR must be available in any two regions. However, to do a write, all regions need to be fully available. Note that as the migrations happen in the background and asynchronously, they do not affect the live process.

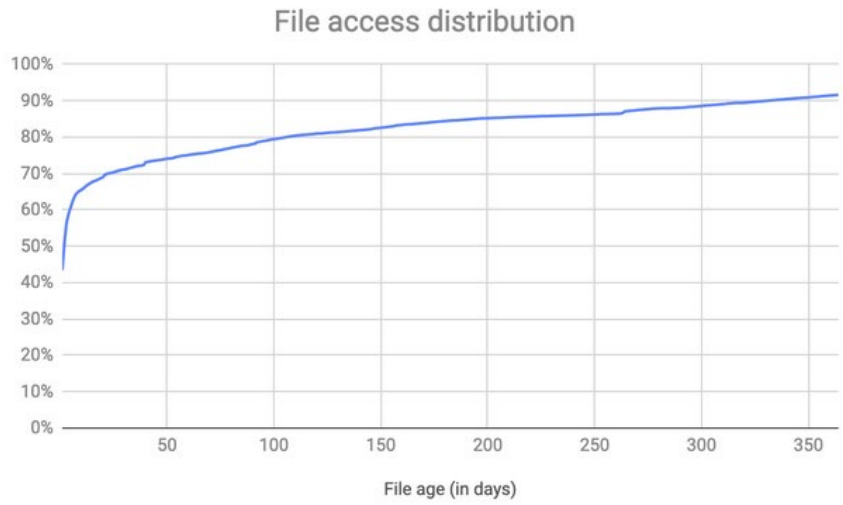


Figure 10: File access distribution

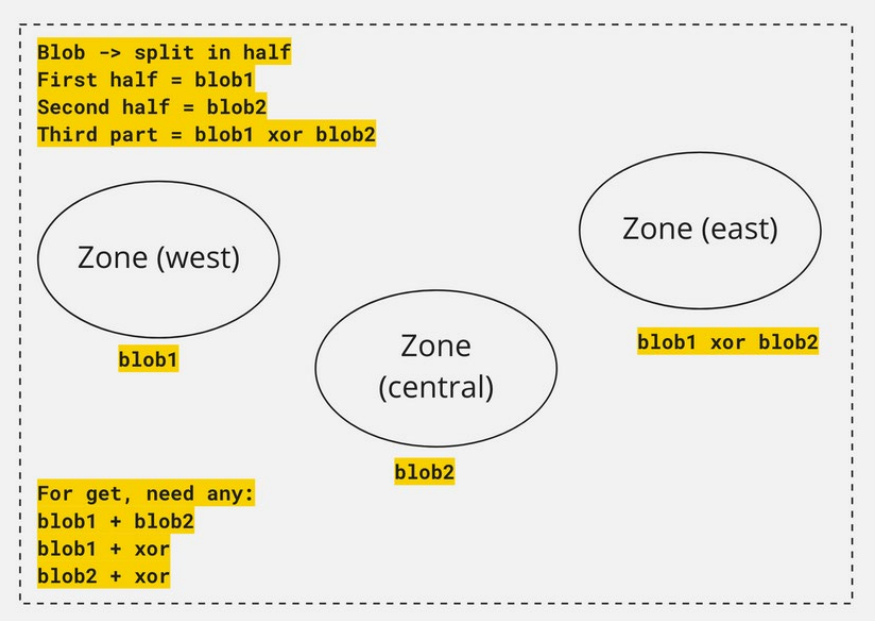


Figure 11: Splitting blobs and cold storage

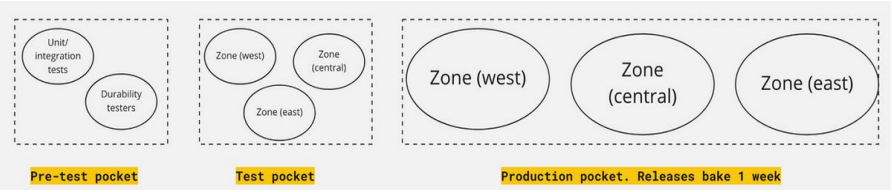


Figure 12: Magic Pocket's release cycle

What are the benefits of this cold storage system? We have achieved a 25% savings by reducing the replication factor from 2x to 1.5x.

The fragments stored in cold storage are still internally erasure-coded, and migration is done in the background. To reduce overhead on backbone bandwidth, we send requests to the two closest zones and only fetch from the remaining zone if necessary. This saves a significant amount of bandwidth as well.

Release Cycle

How do we do releases in Magic Pocket? Our release cycle takes around four weeks across all staging and production environments.

Before committing changes, we run a series of unit and integration tests with all dependencies and a durability stage with a full verification of all data. Each zone has verifications that take about a week per stage: our release cycle is fully automated, and we have checks in place that will abort or not proceed with code changes if there are any alerts. Only in exceptional cases do we stop the automatic deployment process and have to take control.

Verifications

What about verifications? Within our system, we conduct a lot

of verifications to ensure data accuracy.

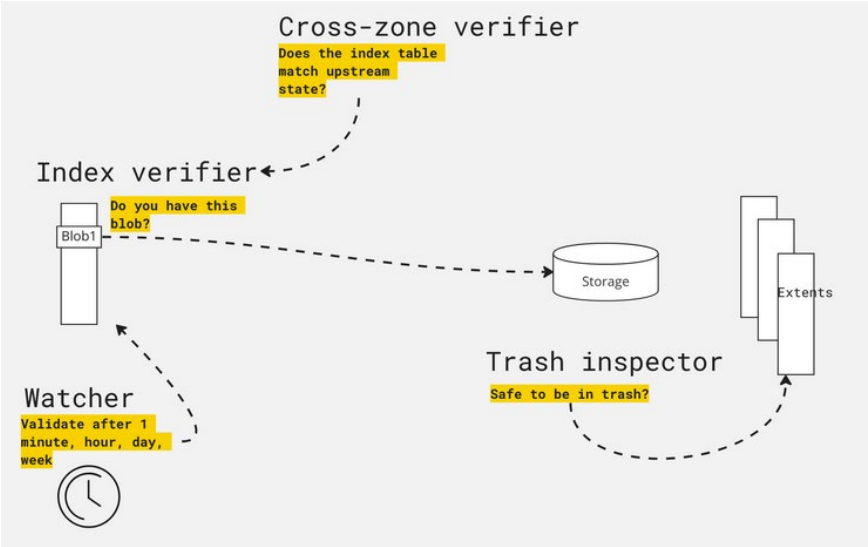


Figure 13: Verifications

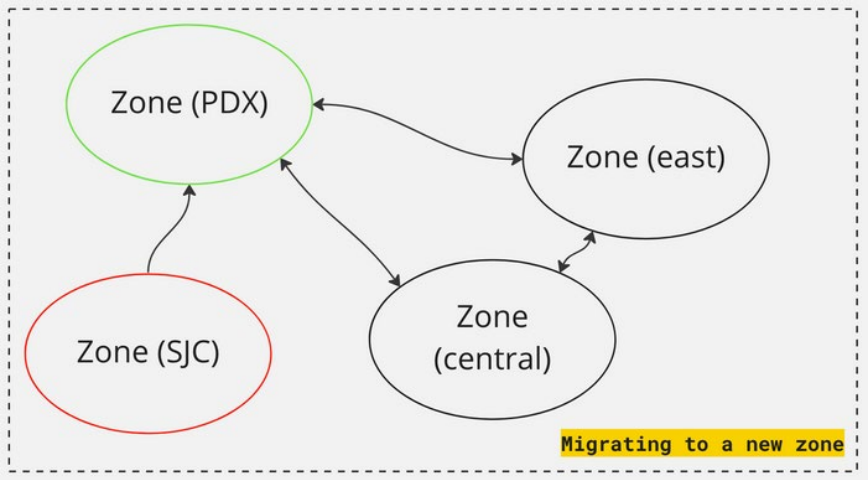


Figure 14: Migrations

One of these is performed by the cross-zone verifier, which synchronizes data mappings between clients upstream and the system.

Another is the index verifier, which scans the index table to confirm if specific blobs are present in each storage machine: we simply ask if the machine has the blob based on its loaded extents, without actually fetching the content. The watcher is another component that performs full validation of the blobs themselves, with sampling done after one minute, an hour, a day, and a week. We also have the trash inspector, which ensures that all hashes within an extent are deleted once the extent is deleted.

Operations

With Magic Pocket we deal with lots of migrations since we operate out of multiple data centers. We manage a very large fleet of storage machines, and it's important to know what's happening all the time. There's a lot of automated chaos taking place, so we have tons of disaster recovery events to test the reliability of our system: upgrading at this scale is just as difficult as the system itself. Managing background traffic is one of our key operations since it accounts for most of our traffic and disk IOPS. The disk scrubber constantly scans through all of the traffic and checks the checksum for the extents. We categorize traffic by service into different tiers, and live traffic is prioritized by the network.

The control plane generates plans for a lot of the background traffic based on forecasts we have about a data center migration: we take into account the type of migration we are doing, such as for cold storage, and plan accordingly.

We deal with a lot of failures in our system: we have to repair 4 extents every second, which can be anywhere from 1 to 2 GBs in size. We have a pretty strict SLA on repairs (less than 48 hours) and, as it is part of our durability model, we want to keep this repair time as low as possible. Our OSDs get allocated into the system automatically based on

the size of the cell and current utilization.

We also have a lot of migrations to different data centers.

Two years ago, we migrated out of the SJC region, and it took extensive planning to make it happen. For very large migrations, like hundreds of PBs, there is significant preparation going on behind the scenes, and we give ourselves extra time to make sure that we can finish the migration in time.

Forecasting

Forecasting is a crucial part of managing our storage system at this scale. We are constantly dealing with the challenge of storage growth, which can sometimes be unexpected and require us to quickly adapt and absorb the new data into our system.

Additionally, we may face capacity issues due to supply chain disruptions like those caused by the COVID pandemic: as soon as we identify any potential problems, we start working on backup plans as it takes a considerable amount of time to order and deliver new capacity to the data centers. Our forecasts are directly integrated into the control plane, which helps us execute migrations based on the information provided by our capacity teams.

Conclusion

Managing Magic Pocket, four key lessons have helped us maintain the system:

- Protect and verify
- Okay to move slow at scale
- Keep things simple
- Prepare for the worst

First and foremost, we prioritize protecting and verifying our system. It requires a significant amount of overhead, but it's crucial to have end-to-end verification to ensure consistency and reliability.

At this scale, it's important to move slowly and steadily. We prioritize durability and take the time to wait for verifications before deploying anything new. We always consider the risks and prepare for worst-case scenarios.

Simplicity is also a crucial factor. We aim to keep things simple, especially during large-scale migrations, as too many optimizations can create a complicated mental model that makes planning and debugging difficult.

In addition, we always have a backup plan in case of failures or issues during migrations or deployments. We ensure that changes are not a one-way door and can be reversed if necessary. Overall, managing a storage system of this scale requires a careful balance of protection, verification, simplicity, and preparation.

SPONSORED ARTICLE

Serverless for Survival

by **Michelle Gienow**

When new technologies arise, we first adopt them for their technical value. If that value proves out, a technology may then jump to widespread adoption through proven business value.

Some technologies, a very select few, then make one more leap forward — from mainstream to existential imperative.

These paradigm-shifting technologies are things that just about every business needs in order to survive. Relational databases made it possible to store and retrieve massive amounts of information quickly and easily. Graphic interfaces, which (along with word processing and spreadsheet software) made personal computers not only possible, but quickly impossible to do business without. And the rise of the internet gave us email, e-commerce and mobile computing on devices that fit into the palm of your hand.

All this progress occurred over a mere handful of decades. But if we step back for a moment to take it all in, a common thread becomes clear:

Our modern world is composed of the applications that make it possible.

Serverless computing

Over time all these new applications, from SQL databases to native mobile apps, needed a new architecture to truly realize their promise: cloud computing.

The cloud, of course, has more than proven its business value. Unfortunately, cloud native architecture is challenging to implement due to the inherent complexity of distributed systems, and its fullest potential has been truly accessible only for organizations with deep technical talent.

Every problem invites a solution, however, and thus we find ourselves in the middle of a new transformation. Serverless computing has emerged as the logical next evolution of cloud native — the ultimate delivery on the principles of cloud, container, and microservices architectures.

How does serverless drive innovation?

Serverless shifts complex operational responsibilities like server or cluster provisioning, patching, system maintenance, and capacity management to your cloud provider. By automating tedious but necessary IT work, serverless computing unblocks technical teams to use their time for innovation instead. With their DevOps teams freed up, engineering organizations are able to rapidly prototype and trial new products or services, then pivot based on market response.

Simplicity as a service: Serverless can automate the complex cognitive work of provisioning, predicting capacity, config, updating, security, networking. This democratizes cloud native, opening the door to small and medium enterprises with their small and medium-sized tech teams.

Please read the full-length version of this article here.

Banking on Thousands of Microservices

by **Suhail Patel,** Staff Engineer @ Monzo

In this article, I aim to share some of the practical lessons we have learned while constructing our architecture at Monzo. We will delve into both our successful endeavors and our unfortunate mishaps.

We will discuss the intricacies involved in scaling our systems and developing appropriate tools, enabling engineers to concentrate on delivering the features that our customers crave.

Monzo's Goal

Our objective at Monzo is to democratize access to financial services. With a customer base of 7 million, we understand the importance of streamlining our processes and we have several payment integrations to maintain.

Some of these integrations still rely on FTP file transfers, many with distinct standards, rules, and criteria.

We continuously iterate on these systems to ensure that we can roll out new features to our customers without exposing the underlying complexities and restricting our product offerings.

In September 2022, we became direct participants in the Bacs scheme, which facilitates direct debits and credits in the UK. Monzo had been integrated with Bacs since 2017, but through a partner who handled the integration on our behalf.

Last year we built the integration directly over the SWIFT network, and we successfully rolled it out to our customers with no disruption.

This example of seamless integration will be relevant throughout this article.

Our Tech Stack

A pivotal decision was to build all our infrastructure and services on top of AWS, which was unprecedented in the financial services industry at the time.

While the Financial Conduct Authority was still issuing initial guidance on cloud computing and outsourcing, we were among the first companies to deploy on the cloud. We have a few data centers for payment scheme integration, but our core platform runs on the services we build on top of AWS with minimal computing for message interfacing.

With AWS, we had the necessary infrastructure to run a bank, but we also needed modern software. While pre-built solutions exist, most rely on processing everything on-premise. Monzo aimed to be a modern bank, unburdened by legacy technology, designed to run in the cloud.

Adoption of Microservices

The decision to use microservices was made early on. To build a reliable banking technology, the company needed a dependable system to store money. Initially, services were created to handle the banking ledger, signups, accounts, authentication, and authorization. These services are context-bound and manage their own data. The company used static code generation to marshal data between services, which makes it easier to establish a solid API and semantic contract between entities and how they behave.

Separating entities between different database instances is also easier with this approach. For example, the transaction model has a unique account entity but all the other information lives within the account service. The account service is called using a Remote Procedure Call (RPC) to get full account information.

During the early days of Monzo, before the advent of service meshes, RPC was used over RabbitMQ, which was responsible for load balancing and deliverability of messages, with a request queue and a reply queue.

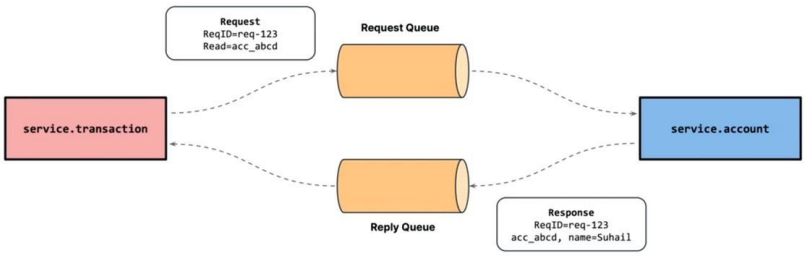


Figure 1: Rabbit MQ in Monzo's early days

Today, Monzo uses HTTP requests: when a customer makes a payment with their card, multiple services get involved in real-time to decide whether the payment should be accepted or declined. These services come from different teams, such as the payments team, the financial crime domain team, and the ledger team.

Monzo doesn't want to build separate account and ledger abstractions for each payment scheme, so many of the services and abstractions need to be agnostic and able to scale independently to handle different payment integrations.

Cassadra as a Core Database

We made the decision early on to use Cassandra as our main database for services, with each service operating under its ownkeyspace. This strict isolation between keyspaces meant that a service could not directly read data from another service. Cassandra is an open-source NoSQL database that distributes data across multiple nodes based on partitioning and replication, allowing for dynamic growth and shrinking of the cluster. It uses timestamps and quorum-based reads to provide stronger consistency, making it an eventually consistent system with last-write wins semantics.

Monzo set a replication factor of 3 for the account keyspace and defined a query with a local

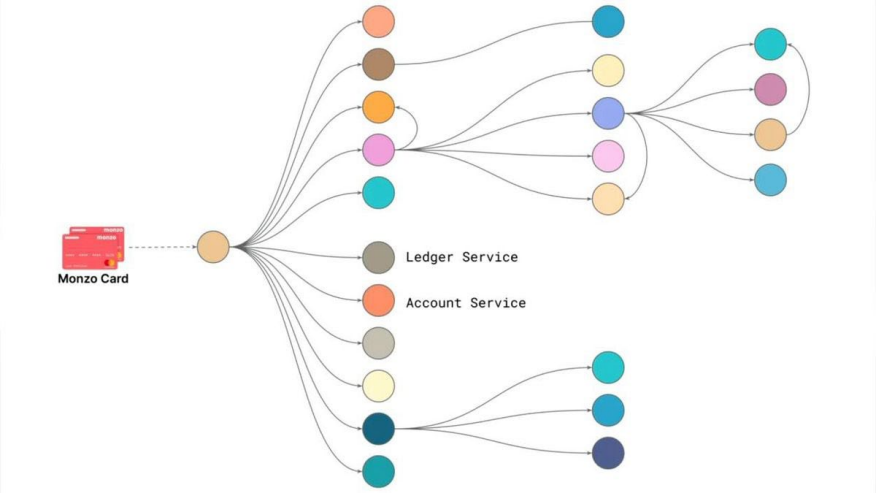


Figure 2: A customer paying for a product with a card

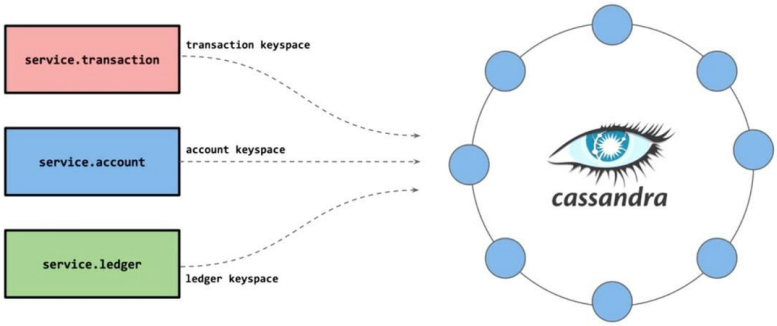


Figure 3: Cassandra at Monzo

```
CREATE TABLE hotel.hotels (
    id text PRIMARY KEY,
    name text,
    phone text,
    address frozen<address>,
    pois set
);

CREATE TABLE hotel.hotels_by_poi (
    poi_name text,
    hotel_id text,
    name text,
    phone text,
    address frozen<address>,
    PRIMARY KEY ((poi_name), hotel_id)
) AND CLUSTERING ORDER BY (hotel_id ASC);


CREATE TABLE hotel.pois_by_hotel (
    poi_name text,
    hotel_id text,
    description text,
    PRIMARY KEY ((hotel_id), poi_name)
);
```

Figure 4: Hotel example, with the hard-to-read point of interests table

quorum to reach out to the three nodes owning the data and return when the majority of nodes agreed on the data. This approach allowed for a more powerful and scalable database, with fewer issues and better consistency.

In order to distribute data evenly across nodes and prevent hot partitions, it's important to choose a good partitioning key for your data.

However, finding the right partitioning key can be challenging as you need to balance fast access with avoiding duplication of data across different tables. Cassandra is well-suited for this task, as it allows for efficient and inexpensive data writing.

Iterating over the entire dataset in Cassandra can be expensive and transactions are also lacking. To work around these limitations, engineers must be trained to model data differently and adopt patterns like canonical and index tables: data is written in reverse order to these tables, first to the index tables, and then to the canonical table, ensuring that the writes are fully complete.

For example, when adding a point of interest to a hotel, the data would first be written to the pois_by_hotel table, then to the hotels_by_poi able, and finally to the hotels table as the canonical table.

Migration to Kubernetes

Although scalability is beneficial, it also brings complexity and requires learning how to write data reliably. To mitigate this, we provide abstractions and autogenerated code for our engineers. To ensure highly available services and data storage, we utilize Kubernetes since 2016. Although it was still in its early releases, we saw its potential as an open-source orchestrator for application development and operations. We had to become proficient in operating Kubernetes, as managed offerings and comprehensive documentation were unavailable at the time, but our expertise in Kubernetes has since paid off immensely.

In mid-2016, the decision was made to switch to HTTP and use Linkerd for service discovery and routing. This improved load balancing and resiliency properties, especially in the event of a slow or u≠nreliable service instance.

However, there were some problems, such as the outage experienced in 2017 when an interaction between Kubernetes and *etcd* caused service discovery to fail, leaving no healthy endpoints. This is an example of teething problems that arise with emerging and maturing technology. There are many stories of similar issues on k8s.af, a valuable resource for teams running Kubernetes at scale. Rather than seeing these outages as reasons to avoid Kubernetes, they should be viewed as learning opportunities.

We initially made tech choices for a small team, but later scaled to 300 engineers, 2500 microservices, and hundreds of daily deployments. To manage that, we have separate services and data boundaries and our platform team provides infrastructure and best practices embedded in core abstractions, letting engineers focus on business logic.

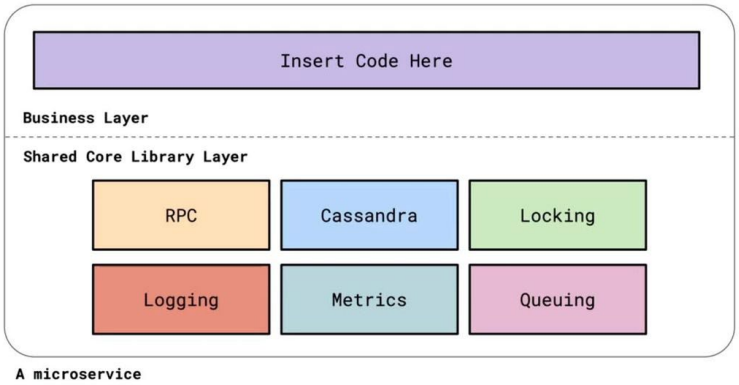


Figure 5: Shared Core Library Layer

We use uniform templates and shared libraries for data marshaling, HTTP servers, and metrics, providing logging, and tracing by default.

The Observability Stack

Monzo uses various open-source tools for their observability stacks such as Prometheus, Grafana, OpenTelemetry, and Elasticsearch. We heavily invest in collecting telemetry data from our services and infrastructure, with over 25 million metric samples and hundreds of thousands of spans being scraped at any one point. Every new service that comes online immediately generates thousands of metrics, which engineers can view on templated dashboards. These dashboards also feed into automated alerts, which are routed to the appropriate team.

For example, the company used telemetry data to optimize the performance of the new customer feature Get Paid Early.

When the new option caused a spike in load, we had issues with service dependencies becoming part of the hot path and not being provisioned to handle the load. We couldn't statically encode this information because it continuously shifted, and autoscaling wasn't reliable. Instead, we used Prometheus and tracing data to dynamically analyze the services involved in the hot path and scale them

appropriately. Thanks to the use of telemetry data, we reduced the human error rate and made the feature self-sufficient.

Abstracting Away Platform Infra

Our company aims to simplify the interaction of engineers with platform infrastructure by abstracting it away from them. We have two reasons for this: engineers should not need to have a deep understanding of Kubernetes and we want to offer a set of opinionated features that we actively support and have a strong grasp on.

Since Kubernetes has a vast range of functionalities, it can be implemented in various ways. Our goal is to provide a higher level of abstraction that can ease the workload for application engineering teams, and minimize our personnel cost in running the platform. Engineers are not required to work with Kubernetes YAML.

If an engineer needs to implement a change, we provide tools that will check the accuracy of their modifications, construct all relevant Docker images in a clean environment, generate all Kubernetes manifests, and deploy everything.

We are currently undertaking a major project to move our Kubernetes infrastructure from our self-hosted platform to Amazon EKS, and this transition has also been made seamless by our deployment pipeline.

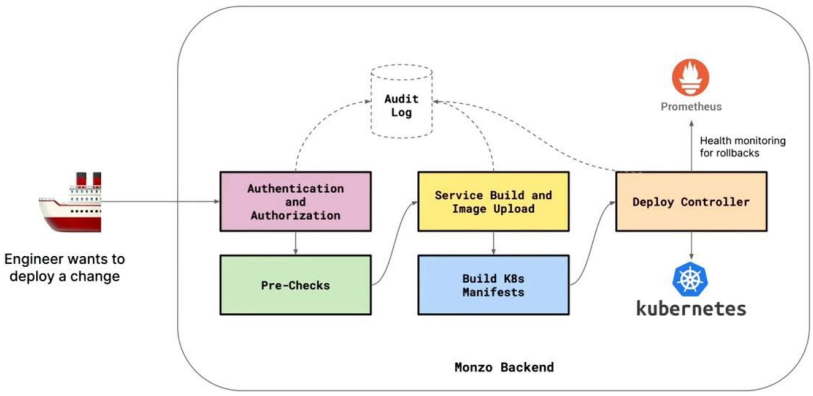


Figure 6: How an engineer deploys a change

If you're interested in learning more about our deployment approach, code generation, and our service catalog, I gave a talk at QCon London 2022 where I discussed the tools we have developed, as well as our philosophy towards the developer experience.

Embracing Failure Modes of Distributed Systems

The team recognizes that distributed systems are prone to failure and that it is important to acknowledge and accept it. In the case of a write operation, issues may occur and there may be uncertainty as to whether the data has been successfully written.This can result in inconsistencies when reading the data from different nodes, which can be problematic for a banking service that requires consistency. To address this issue, the team has been using a separate service running continuously in the background that is responsible for detecting and resolving inconsistent data states. This service can either flag the issue for further investigation or even

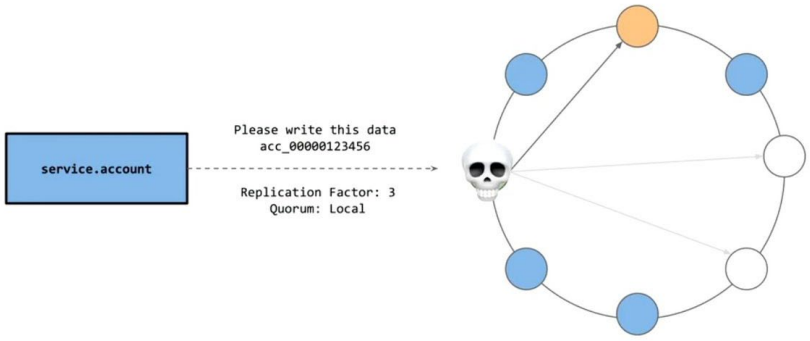


Figure 7: Handling failures on Cassandra

automate the correction process. Alternatively, validation checks can be run when there is a user-facing request, but we noticed that this can lead to delays.

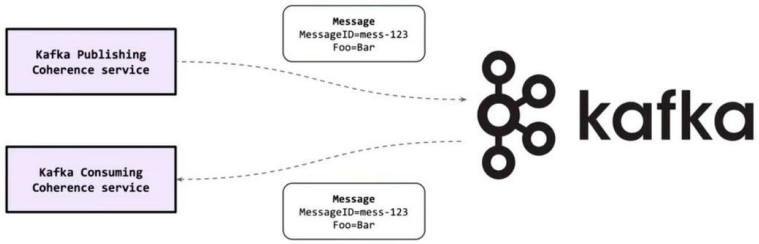


Figure 8: Kafka and the coherence service

Coherence services are beneficial for the communication between infrastructure and services: Monzo uses Kafka clusters and Sarama-based libraries to interact with Kafka. To ensure confidence in updates to these libraries and Sarama, coherence services are continuously run in both staging and production environments. These services utilize the libraries like any other microservice and can identify problems caused by accidental changes to the library or Kafka configuration before they affect production systems.

The Organizational Aspect

Investment in systems and tooling is necessary for engineers to develop and run systems efficiently: the concepts of uniformity and "paved road" ensure consistency and familiarity, preventing the development of unmaintainable services with different designs.

From day one, Monzo focuses on getting new engineers onto the "paved road" by providing a documented process for writing and deploying code and a support structure for asking questions. The onboarding process is defined to establish long-lasting behaviors, ideas, and concepts, as it is difficult to change bad habits later on. Monzo continuously invests in onboarding, even having a "legacy patterns" section to highlight patterns to avoid in newer services.

While automated code modification tools are used for smaller changes, larger changes may require significant human refactoring to conform to new patterns, which takes time to implement across services. To prevent unwanted patterns or behaviors, Monzo uses static analysis checks to identify issues before they are shipped. Before making these checks mandatory, we ensure that the existing codebase is cleaned up to avoid engineers being tripped up by failing checks that are not related to their modifications. This approach ensures a high-quality signal, rather than engineers ignoring the checks. The high friction to bypass these checks is intentional to ensure that the correct behavior is the path of least resistance.

Handling Incidents

In April 2018, TSB, a high-street bank in the UK, underwent a problematic migration project to move customers to a new banking platform. This resulted in customers being unable to access their money for an extended period, which led to TSB receiving a large million fine, nearly £33 million in compensation to customers, and reputational damage. The FCA report on the incident examines both the technological and organizational aspects of the problem, including overly ambitious planning schedules, inadequate testing, and the challenge of balancing development speed with quality. While it may be tempting to solely blame technology for issues, the report emphasizes the importance of examining organizational factors that may have contributed to the outage.

Reflecting on past incidents and projects is highly beneficial in improving operations: Monzo experienced an incident in July 2019, when a configuration error in Cassandra during a scale-up operation forced a stop to all writes and reads to

the cluster. This event set off a chain reaction of improvements spanning multiple years to enhance the operational capacity of the database systems. Since then, Monzo has invested in observability, deepening the understanding of Cassandra and other production systems, and we are more confident in all operational matters through runbooks and production practices.

Making the Right Choices

Earlier I mentioned the early technological decisions made by Monzo and the understanding that it wouldn't be an easy ride: over the last seven years, we have had to experiment, build, and troubleshoot through many challenges, and this process continues. If an organization is not willing or able to provide the necessary investment and support for complex systems, this must be taken into consideration when making architectural and technological choices: choosing the latest technology or buzzword without adequate investment is likely to lead to failure. Instead, it is better to choose simpler, more established technology that has a higher chance of success. While some may consider this approach to be boring, it is ultimately a safer and more reliable option.

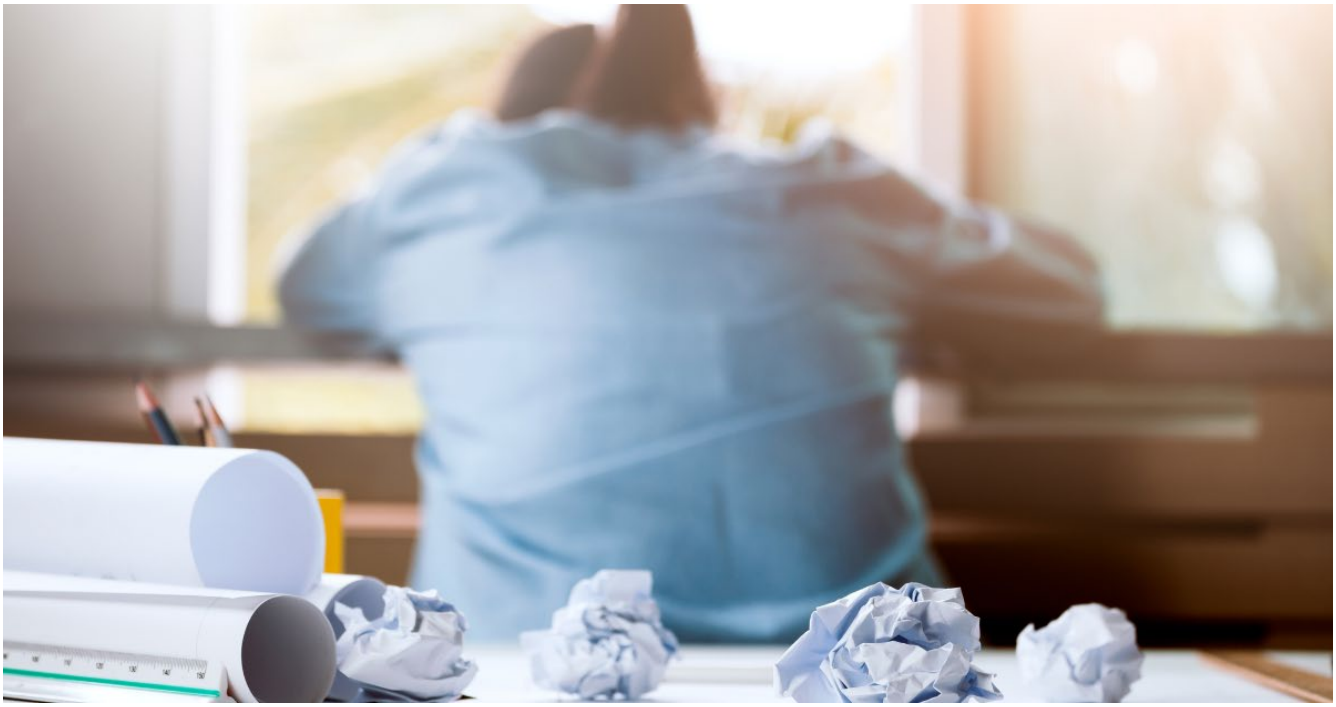
Conclusions

Teams are always improving tools and raising the level of abstraction. By standardizing on a small set of technological choices and continuously improving these tools and abstractions, engineers can focus on the business problem rather than the underlying infrastructure. It is important to be conscious when systems deviate from the standardized road.

While there's a lot of focus on infrastructure in organizations, such as infrastructure as code, observability, automation, and Terraform, one theme often overlooked is the bridge between infrastructure and software engineers. Engineers don't need to be experts in everything and core patterns can be abstracted away behind a well-defined, tested, documented, and bespoke interface. This approach saves time, promotes uniformity, and embraces best practices for the organization.

Showing different examples of incidents, we highlighted the importance of introspection: while many may have a technical root cause, it's essential to dig deeper and identify any organizational issues that may have contributed. Unfortunately, most post-mortems tend to focus heavily on technical details, neglecting the organizational component.

It's essential to consider the impact of organizational behaviors and incentives on the success or failure of technical architecture. Systems don't exist in isolation and monitoring, and rewarding the operational stability, speed, security, and reliability of the software you build and operate is critical to success.



Tales of Kafka at Cloudflare: Lessons Learnt on the Way to 1 Trillion Messages

by **Matt Boyle,** Engineering Manager @ Cloudflare, and **Andrea Medda**, Senior Systems Engineer @ Cloudflare

Cloudflare has generated over 1 trillion messages to Kafka in less than six years just for inter-service communication. As the company and application services team grew, they had to adapt their tooling to continue delivering fast.

We will discuss the early days of working in distributed domain-based teams and how abstractions were built on top of Kafka to reach the 1 trillion message mark.

We will also cover real incidents faced in recent years due to scalability limitations and the steps and patterns applied to deal with increasing demand.

What Is Cloudflare?

Cloudflare provides a global network to its customers and allows them to secure their websites, APIs, and internet traffic.

This network also protects corporate networks and enables customers to run and deploy entire applications on the edge.

Cloudflare offers a range of products, including CDN, Zero Trust, and Cloudflare Workers, to achieve these goals, identifying and blocking malicious activity and allowing customers to focus on their work.

Looking at the Cloudflare network from an engineering perspective, there are two primary components: the Global Edge network and the Cloudflare control plane.

A significant portion of the network is built using Cloudflare's

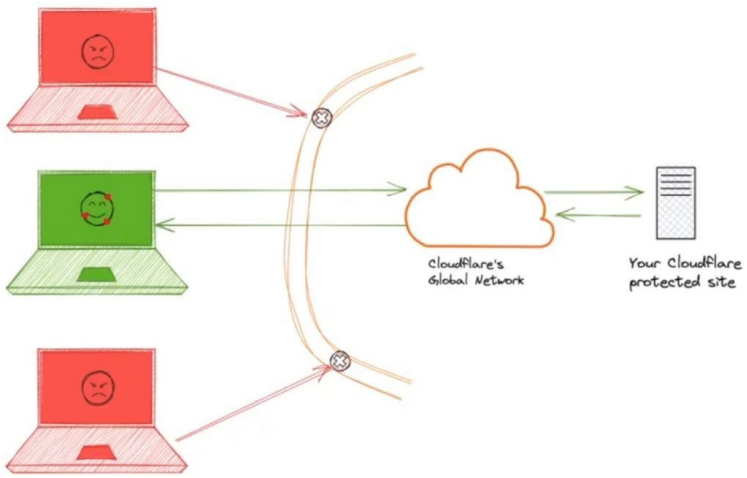


Figure 1: Cloudflare's Global Network

products, with Workers deployed and used on the edge network. The control plane, on the other hand, is a collection of data centers where the company runs Kubernetes, Kafka, and databases on bare metal. All Kafka producers and consumers are usually deployed into Kubernetes, but the specific deployment location depends on the workload and desired outcomes.

In this article, we will focus on the Cloudflare control plane and explore how inter-service communication and enablement tools are scaled to support operations.

Kafka

Apache Kafka is built around the concept of clusters, which consist of multiple brokers, with each cluster having a designated leader broker responsible for coordination. In the diagram below, *broker 2* serves as the leader. Messages are categorized into topics, such as user events, for example, user creation, or user information updates. Topics are then divided into partitions, an approach that allows Kafka to scale horizontally. In the diagram, there are partitions for topic A on both brokers, with each partition having a designated leader to determine its "source of truth". To ensure resilience, partitions are replicated according to a predetermined replication factor, with three being the usual minimum. The services that send messages to Kafka are called producers, while those that read messages are called consumers.

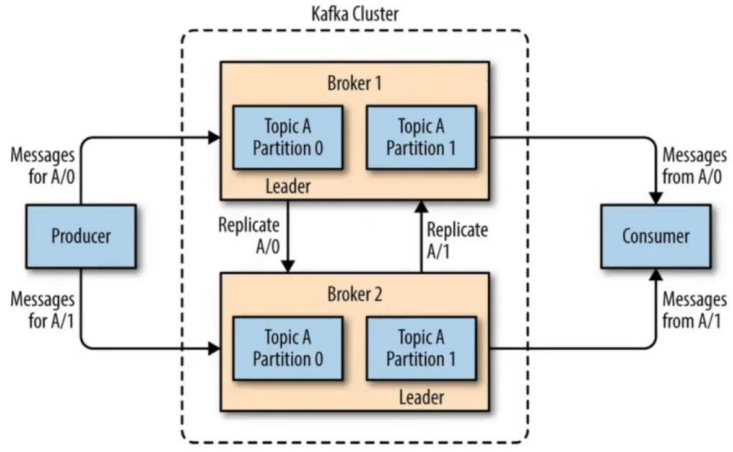


Figure 2: Kafka Cluster

Cloudflare Engineering Culture

In the past, Cloudflare operated as a monolithic PHP application, but as the company grew and diversified, this approach proved to be limiting and risky.

Rather than mandating specific tools or programming languages, teams are now empowered to build and maintain their services, with the company encouraging experimentation and advocating for effective tools and practices. The Application Services team is a relatively new addition to the engineering organization, to make it easier for other teams to succeed by providing pre-packaged tooling that incorporates best practices. This allows development teams to focus on delivering value.

Tight Coupling

With the product offerings growing, there was a need to find

better ways of enabling teams to work at their own pace and decouple from their peers and the engineering team also needed to have more control over backoff requests and work completion guarantees.

As we were already running Kafka clusters to process large amounts of data, we decided to invest time in creating a general-purpose message bus cluster: onboarding is straightforward, requiring a pull request into a repository, which sets up everything needed for a new topic, including the replication strategy, retention period, and ACLs. The diagram illustrates how the Messagebus cluster can help decouple different teams.

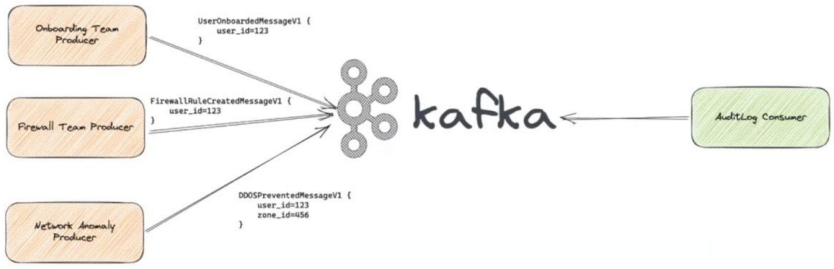


Figure 3: The general-purpose message bus cluster

For example, three teams can emit messages that the audit log system is interested in, without the need for any awareness of the specific services. With less coupling, the engineering team can work more efficiently and scale effectively.

Unstructured Communication

With an event-driven system, to avoid coupling, systems shouldn't be aware of each other. Initially, we had no enforced message format and producer teams were left to decide how to structure their messages. This can lead to unstructured communication and pose a challenge if the teams don't have a strong contract in place, with an increased number of unprocessable messages.


```
message EmailMessageV1 {
    repeated string email = 1;
    string template = 2;
    bytes data = 3;
    bytes meta = 4;
    repeated int64 user_id = 5;
}
```

Figure 4: A protobuf message

To avoid unstructured communication, the team searched for solutions within the Kafka ecosystem and found two viable options, Apache Avro and protobuf, with the latter being the final choice. We had previously been using JSON, but found it difficult to enforce compatibility and the JSON messages were larger compared to protobuf.

Protobuf provides strict message types and inherent forward and backward compatibility, with the ability to generate code in multiple languages also a major advantage. The team encourages detailed comments on their protobuf messages and uses Prototool, an open-source tool by Uber, for breaking change detection and enforcing stylistic rules.

Protobuf alone was not enough: different teams could still emit messages to the same topic, and the consumer may not be able to process it due to the format not being what was expected. Additionally, configuring Kafka consumers and producers was not an easy task, requiring intricate knowledge of the workload. As most teams were using Go, we decided to build a "message bus client library" in Go, incorporating best practices and allowing teams to move faster.

To avoid teams emitting different messages to the same topic, we made the controversial decision to enforce (on the client side)

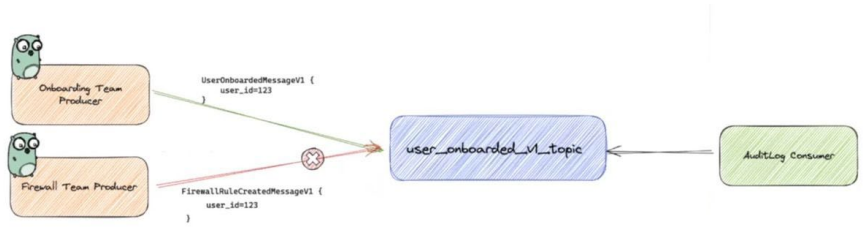


Figure 5: Switching to Protobuf

one protobuf message type per topic. While this decision enabled easy adoption, it resulted in numerous topics being created, with multiple partitions replicated, with a replication factor of at least three.

Connectors

The team had made significant progress in simplifying the Kafka infrastructure by introducing tooling and abstractions, but we realized that there were further use cases and patterns that needed to be addressed to ensure best practices were followed: the team developed the connector framework.

Based on Kafka connectors, the framework enables engineers to create services that read from one system and push it to another one, like Kafka or Quicksilver, Cloudflare's Edge database. To simplify the process, we use Cookiecutter to template the service creation, and engineers only need to enter a few parameters into the CLI.

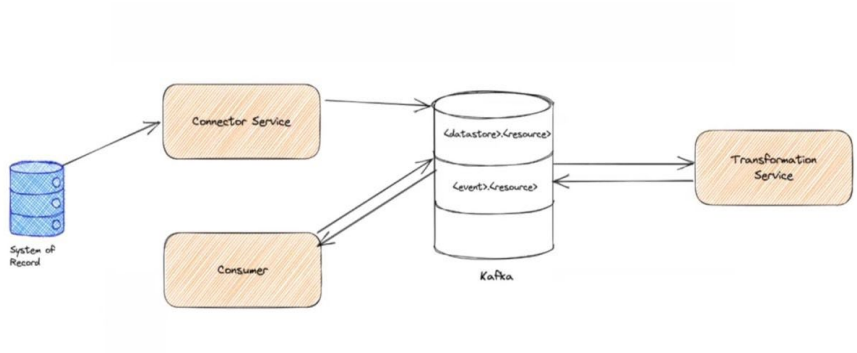


Figure 6: The connector framework

```
READER=kafka
TRANSFORMATIONS=topic_router:topic1,topic2|pf_edge
WRITER=quicksilver
```

Figure 7: A simple connector

The configuration process for the connector is simple and can be done through environment variables without any code changes.

In the example below, the reader is Kafka and the writer is Quicksilver. The connector is set to read from topic 1 and topic 2 and apply the function pf_edge. This is the complete configuration needed, which also includes metrics, alerts, and everything else required to move into production, allowing teams to easily follow best practices. Teams have the option to register custom transformations, which would be the only pieces of code they would need to write.

For example, we utilize connectors in the communication preferences service: if a user wants to opt out of marketing information in the Cloudflare dashboard, they interact with this service to do so. The communication preference upgrade is stored in its database, and a message is emitted to Kafka. To ensure that the change is reflected in three different source systems, we use separate connectors that sync the change to a transactional email service, a customer management system, and a market email system. This approach makes the system eventually consistent and we leverage the guarantees provided

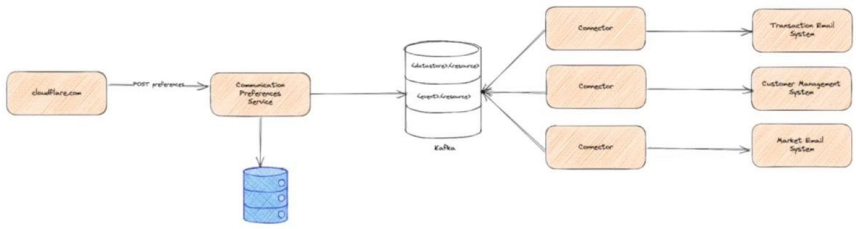


Figure 8: Connector and communication preferences

by Kafka to ensure that the process happens smoothly.

Visibility

As our customer base grew rapidly during the pandemic, so did the throughput, highlighting scalability issues in some of the abstractions that we had created.

One example is the audit logs, which we handle for our Kafka customers: we built a system to manage these logs, allowing producer teams to produce the events, while we listen for them, recording the data in our database.

We expose this information through an API and an integration called log push that enables us to push the audit log data into various data buckets, such as Cloudflare R2 or Amazon S3.

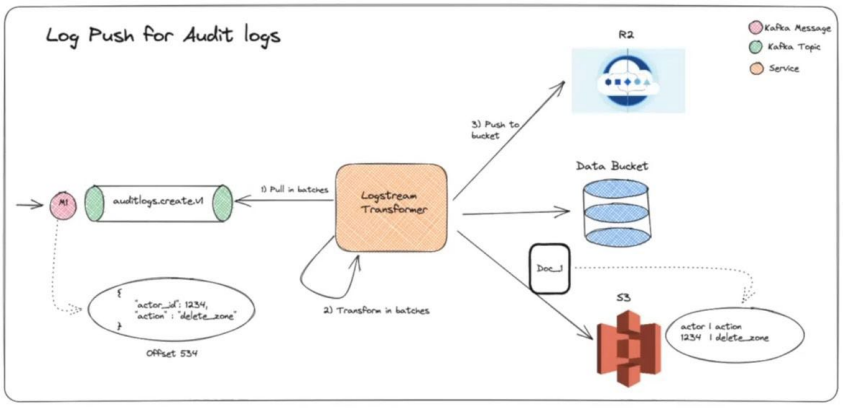


Figure 9: Adding the log push for Audit logs

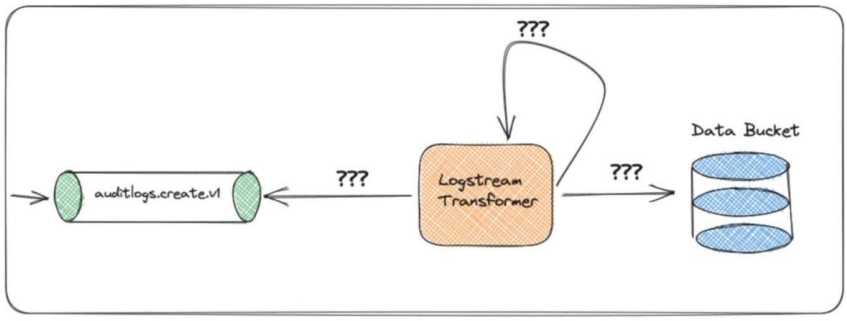


Figure 10: Where is the bottleneck?

During the pandemic, we experienced the registration of many more audit logs and customers started using our APIs to get the latest data. As the approach was not scalable, we decided to develop a pipeline to address the issue, creating a small service that listens for audit log events and transforms them into the appropriate format for direct storage in a bucket, without overloading the APIs.

We encountered further issues as we accumulated logs and were unable to clear them out quickly enough, resulting in lags and breaches of our SLAs. We were uncertain about the cause of the lag as we lacked the tools and instrumentation in our SDK to diagnose the problem: was the bottleneck reading from Kafka, transformation, or saving data to the database?

We decided to address it by enhancing our SDK with Prometheus metrics, with histograms measuring the time each step takes in processing a message. This helped us identify slower steps, but we couldn't tell which specific component was taking longer for a specific message.
To solve this, we explored OpenTelemetry, focusing on its tracing integrations: there were not many good integrations for OpenTracing on Kafka, and it was challenging to propagate traces

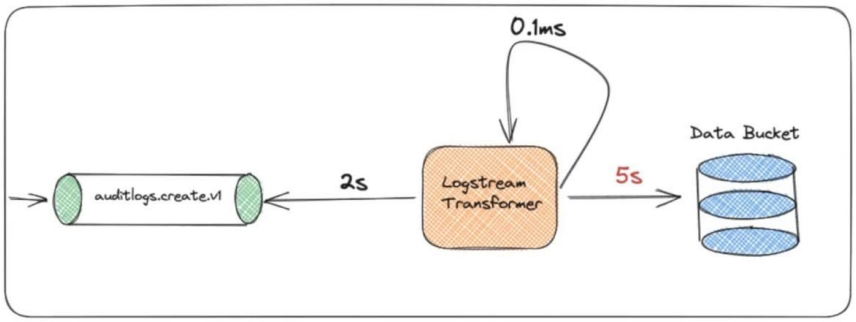


Figure 11: Identifying the bottlenecks

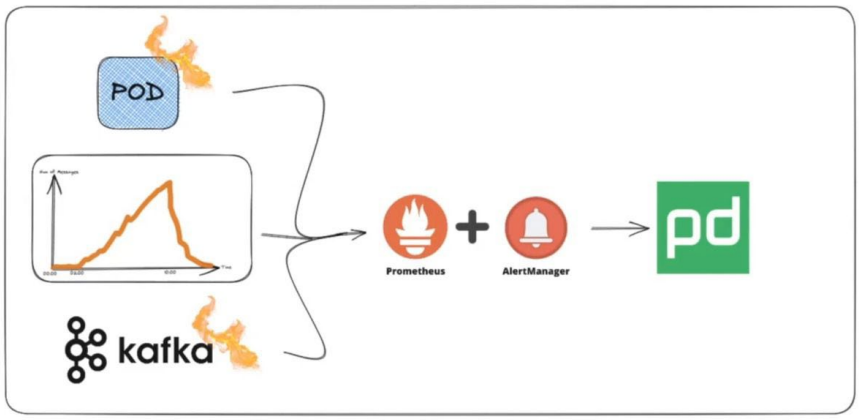


Figure 12: Alerting pipeline

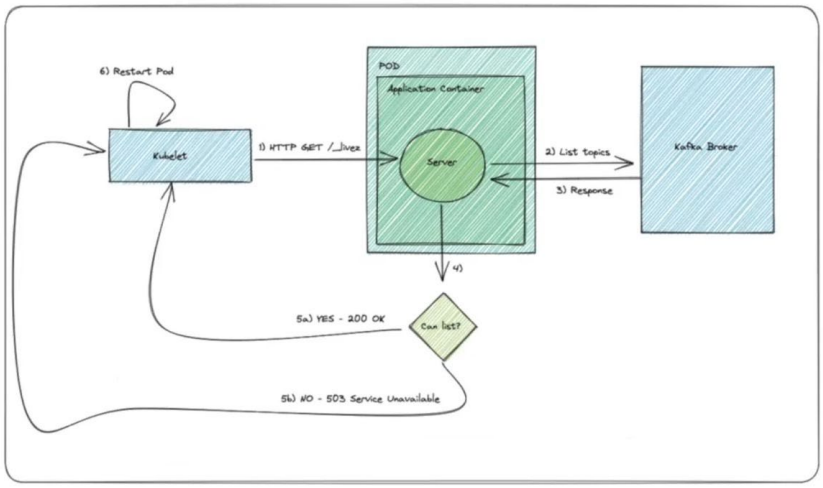


Figure 13: Health checks and Kafka

across different services during a production incident.

With the team enriching the SDK with OpenTracing, we were able to identify that pushing data to the bucket and reading from Kafka were both bottlenecks, prioritizing the fixes for those issues.

Adding metrics into the SDK, we were able to get a better overview of the health of the cluster and the services.

Noisy On-call

We encountered a challenge with the large number of metrics we had collected, leading to a noisy on-call experience, with many alerts related to unhealthy applications and lag issues.

The basic alerting pipeline consists of Prometheus and AlertManager, which would page to PagerDuty. As restarting or scaling up/down services was not ideal, we decided to explore how to leverage Kubernetes and implement health checks.

In Kubernetes, there are three types of health checks: liveness, readiness, and startup. For Kafka, implementing the readiness probe is not useful because usually, an HTTP server is not exposed. To address this, an alternative approach was implemented.

When a request for the liveness check is received, we attempt a basic operation with a broker, such as listing the topics, and if the response is successful, the check passes. However, there are cases where the application is still healthy but unable to produce or consume messages, which led the team to implement smarter health checks for the consumers.

The current offset for Kafka is the last available offset on the partition, while the committed

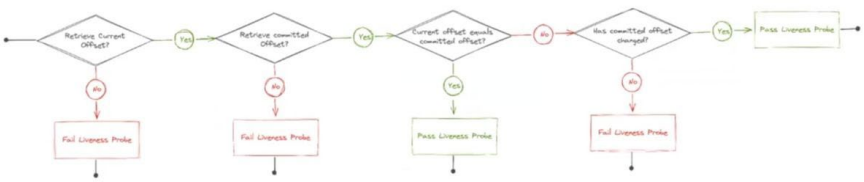


Figure 14: Health checks implementation

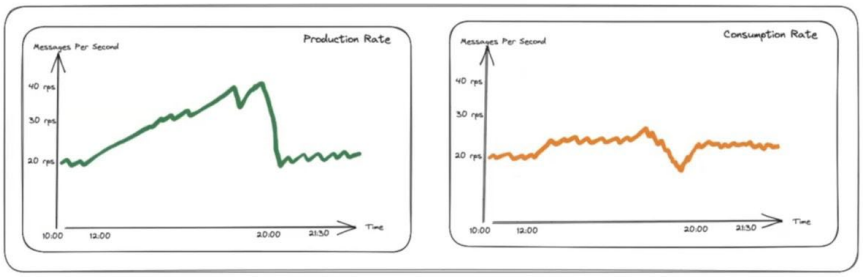


Figure 15: The lag in consumption

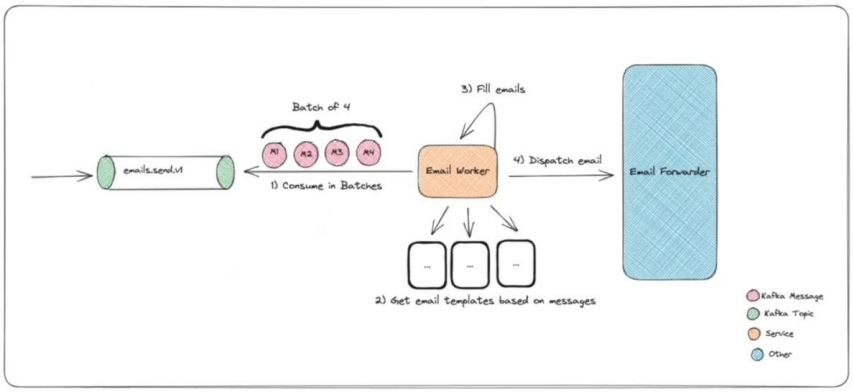


Figure 16: The batching approach

offset is the last offset that the consumer successfully consumed. By retrieving these offsets during a health check, we can determine whether the consumer is operating correctly: if we can't retrieve the offsets, there are likely underlying issues, and the consumer is reported as unhealthy. If the offsets are retrievable, we compare the last committed offset to the current one. If they are the same, no new messages have been appended, and the consumer is considered healthy. If the last committed offset is different, we check if it is the same as the previously recorded last committed offset to make sure that the consumer is not stuck and needs a restart. This process resulted in better on-call experiences and happier customers.

Inability to Keep Up

We had a system where teams could produce events from Kafka for their email system. These events contained a template, for example, an "under attack" template, that includes information about a website under attack and the identity of the attacker, along with metadata.

We would listen for the event, retrieve the template for the email from their registry, enrich it, and dispatch it to the customers. However, we started to experience load issues: the team started to see spikes in the production rate, causing a lag in consumption and impacting important OTP messages and the SLOs.

Batching

We started exploring different solutions to address the problem, with an initial solution of scaling the number of partitions and consumers not providing significant improvement.

We decided to implement a simpler but more effective approach, batch consuming, processing a certain number of messages at a time, applying a transformation, and dispatching them in batches. This proved effective and allowed the team to easily handle high production rates.

Documentation

Developing our SDK, we found that many developers were encountering issues while using it. Some were encountering bugs while others were unsure how to implement certain features

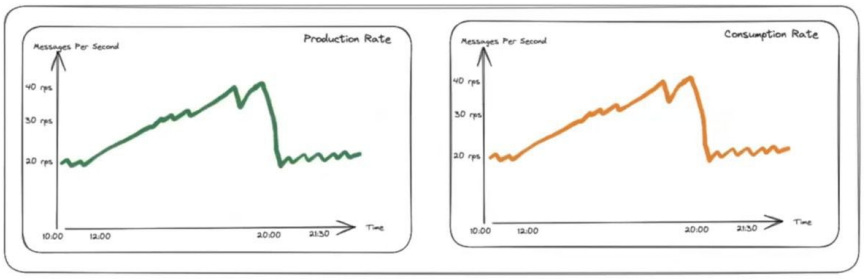


Figure 17: No lag in consumption with batching

or interpret specific errors. To address this, we created channels on our Google Chat where users could come and ask us questions. We had one person on call to respond and spent time documenting our findings and answers in our wiki. This helped to improve the overall user experience for the SDK.

Conclusions

There are four lessons to be learned:

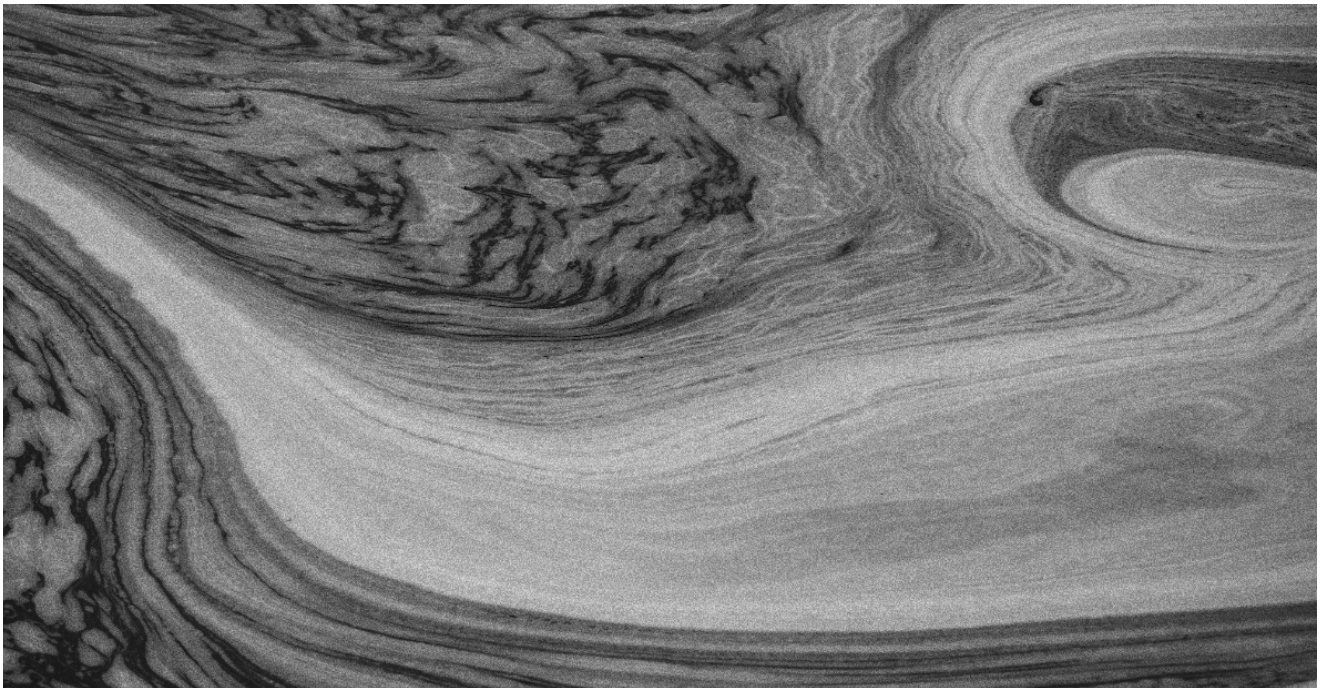
Always find the right balance between flexibility and simplicity: while a configurable setup may offer more flexibility, a simpler one allows standardization across different pipelines.

Visibility: adding metrics to the SDK as soon as possible can help teams understand how the system is behaving and make better decisions, especially during incidents.

Contracts: enforcing one strong, stric contract, gives great visibility into what is happening inside a topic, allowing one to know who is writing and reading from it.

Document the good work that you do so that you don't have to spend time answering questions or helping people debug production issues. This can be achieved through channels like Google Chat and wikis.

By following these rules, we were able to improve our systems and make our customers happy, even in high-stress situations.



What Are Cloud-Bound Applications?

by **Bilgin Ibryam,** Author of Kubernetes Patterns | Product Manager @ Diagrid

The increasing adoption of application-first cloud services is causing applications to blend with the cloud services at levels much deeper than before. The runtime boundaries between the application and the cloud are shifting from virtual machines to containers and functions. The integration boundaries are shifting from database and message broker access only to one where the mechanical parts of the applications are blended and running within the cloud. In this resulting architecture, applications are "cloud bound" and allow developers to focus on the business logic by offloading more application logic and management responsibilities into cloud services.

This article examines the commoditization of the full software stack by binding the application to cloud services using open APIs and standards that preserve flexibility and portability.

Internal architecture evolution

The internal architecture of an application is typically owned and controlled by a single team. Depending on the language of choice and runtime, tools and abstractions such as packages, modules, interfaces, classes, and functions help developers control the inner boundaries. Domain-driven Design (DDD) assists developers in crafting domain models, which serve as abstractions encapsulating complex business logic and mediating the divide between business reality and code.

Hexagonal, Onion, and Clean architectures can complement DDD and arrange application code with distinct boundaries and externalized infrastructure dependencies. While these approaches were innovative at the time of their inception and remain relevant today, they were initially developed for three-tier Java applications comprising JSPs, Servlets, and EJBs deployed in shared application runtimes. The primary focus then was decoupling application logic from the UI and database and enabling isolated testing.

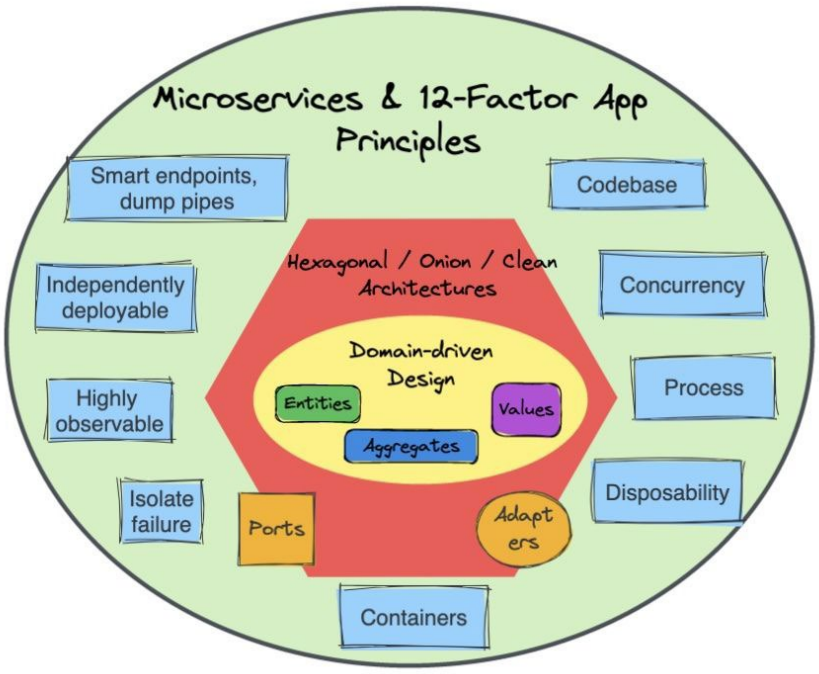


Figure 1: Internal application architecture

Since then, new challenges and concepts, such as microservices and the twelve-factor app, emerged and influenced how we design applications. Microservices center on separating application logic into independently deployable units owned by a single team. The twelve-factor app methodology aims to create distributed, stateless applications that run and scale in dynamic cloud environments. All of these architectures introduced principles and best practices that shape how we structure an application's internal architecture and how we manage it.

Later in the application architecture evolution timeline, the mainstream adoption of containers and the introduction of Kubernetes revolutionized the way applications are packaged and orchestrated. AWS Lambda introduced the concept of highly scalable functions as a service (FaaS), taking the idea of application granularity to the next level and offloading the complete infrastructure management responsibilities to the cloud provider. Other technology trends, such as service mesh and Mecha architecture, have also emerged and commoditized non-functional aspects of the application stack, such as networking and distributed developer primitives, respectively, and extracting them into sidecars. Inspired by microservices, Data Mesh architecture aimed to break down the analytical data architecture of applications into smaller, independent data domains, each with its own product and team. These, and more recent trends, such as application-first cloud services, started reshaping applications' external architecture, which I collectively refer to as "cloud-bound applications" in this article.

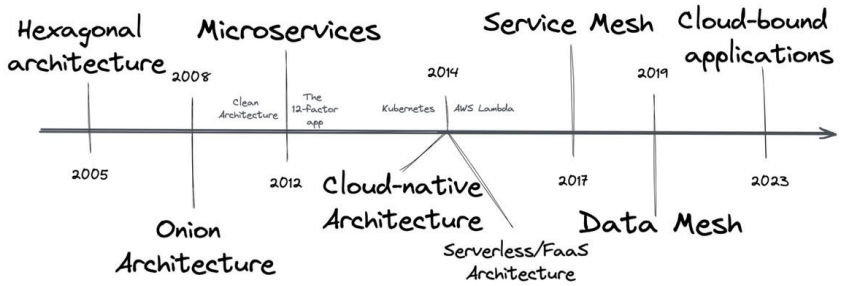


Figure 2: Application architecture evolution timeline

External architecture evolution

The external architecture is where an application intersects with other applications and the infrastructure which other teams and organizations in the form of specialized on-premise middleware, storage systems, or cloud services typically own. The way the application connects to external systems and offloads some of its responsibilities forms the external architecture. To benefit from the infrastructure,

an application needs to bind with that infrastructure and enforce clean boundaries to preserve its agility. An application's internal architecture and implementation should be able to change without changing the other one and also be able to swap outer dependencies, such as cloud services, without changing the internals.

- Compute bindings are all the necessary bindings, configurations, APIs, and conventions used to run an application on a compute platform such as Kubernetes, a container service, or even serverless functions (such as AWS Lambda). Mostly, these bindings are transparent to the internal architecture,

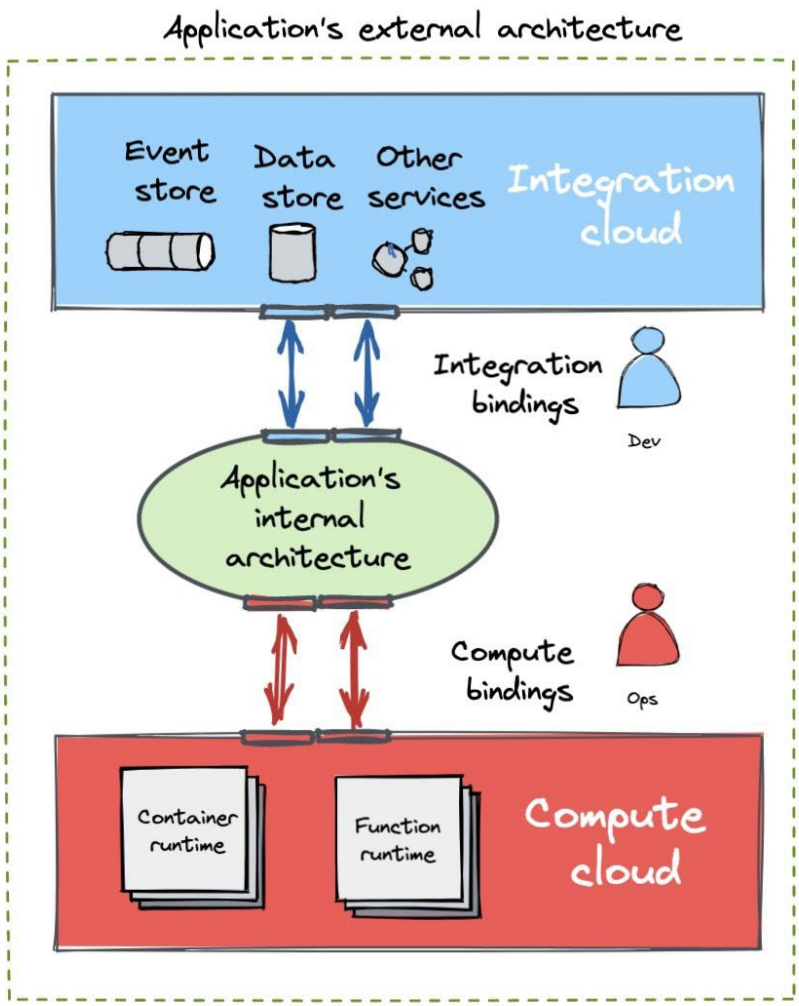


Figure 3: External application architecture

Broadly, we can group the way an application binds with its surroundings into two categories. and configured and used by operations teams rather than developers. The container abstraction is the most widespread "API" for application compute binding today.

- Integration bindings is a catch-all term for all other bindings to external dependencies that an application is relying upon. The cloud services also use these bindings to interact with the application, usually over well-defined HTTP "APIs," or specialized messaging and storage access protocols, such as AWS S3, Apache Kafka, Redis APIs, etc. The integration bindings are not as transparent as the runtime bindings. The developers need to implement additional logic around them, such as retry, TTL, delay, dead-letter queue (DLQ), etc., and bind these to the application's business logic.

Applications run on the cloud and consume other services by using these bindings. Let's see in more detail what exactly is behind these bindings and what is not.

Compute bindings

For the operations teams, ideally, each application is a unit of a black box that needs to be operated on the compute platform. The compute bindings are used to manage an application's lifecycle on platforms such as Kubernetes, AWS Lambda, and other services. These bindings are formalized and defined in the form of a

collection of configurations, and API interactions between the application and the platform running the application. Most of these interactions are transparent to the application, and there is only a handful of APIs that developers need to implement, such as the health endpoints and metrics APIs. This is how far the current CNCF definition and scope of "cloud native" extends, and as long as developers implement cloud-native applications, they can bind and run on a cloud compute platform.

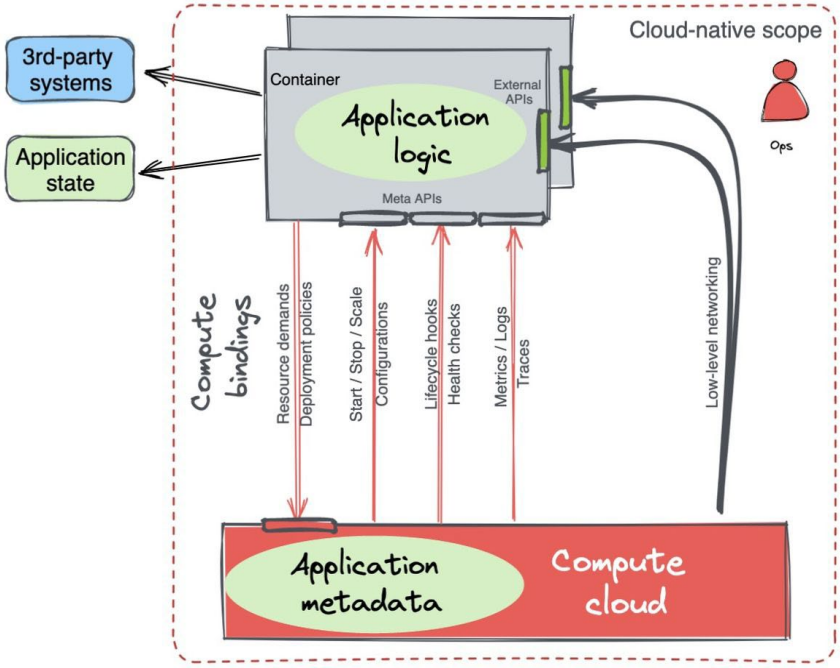


Figure 4: Application and platform compute bindings

To run on a cloud platform reliably, the application has to bind with it on multiple levels, ranging from specifications to best practices. This happens through a collection of industry-standard specifications such as container APIs, metrics APIs, for example, based on Prometheus, health endpoints, or cloud vendor specifications, such as AWS Lambda or AWS ECS specifications. Also, through cloud-native best techniques and shared knowledge such as health checks, deployment strategies, and placement policies. Let's see the common compute bindings used today.

Resource demands

Applications, including microservices and functions, require resources, such as CPU, memory, and storage. These resources are defined differently depending on the platform being used. For example, on Kubernetes, CPU and memory are defined through requests and limits, while on AWS Lambda, the user specifies the amount of memory to allocate at runtime, with a corresponding allocation of CPU. Storage is also handled differently on these platforms, with Kubernetes using ephemeral storage and volumes, while Lambda offers ephemeral scratch resources and durable storage based on Amazon EFS mounts.

Lifecycle hooks

Applications managed by a platform often need to be aware of important lifecycle events. For example, on Kubernetes, concepts such as init containers and hooks like PostStart and PreStop allow the application to react to these events. Similarly, Lambda's extension API allows the application to intercept the Init, Invoke, and Shutdown phases. Other options for handling lifecycle events include wrapper scripts or language-specific runtime modification options, such as a shutdown hook for the JVM. These mechanisms form a contract between the platform and the application, enabling it to respond to and manage its own lifecycle.

Health checks

Health probes are a way for the platform to monitor the health of an application and take corrective action if necessary, such as restarting the application. While Lambda functions do not have health probes due to the request's short lifespan, containerized applications and orchestrators like Kubernetes, AWS EKS, and GCP Cloud Run do include health

probes in their definitions. This allows the platform to ensure that the application runs smoothly and take action if not.

Deployment and placement policies

With knowledge of the required resources, the compute platform can begin managing the application's lifecycle. To do so in a manner that does not compromise the integrity of the business logic, the platform must be aware of the scaling constraints. Some applications are intended to be singletons. For example, they need to maintain the order of processed events and cannot be scaled beyond one instance. Other stateful applications may be quorum-driven and require a specific number of minimum instances constantly running to function properly. And still others, such as stateless functions, may favor rapid scaling to address increasing spikes in the load. Once the scaling guidelines for the application have been established, the platform assumes control of initiating and terminating instances of the application.

Compute platforms also offer a variety of deployment strategies, including rolling, blue-green, canary, and at once, to control the sequence of updates for a service. In addition to the deployment sequence, these platforms may allow the user to specify placement preferences. For example, Kubernetes offers options such as labels, taints and tolerations, affinity, and anti-affinity, while Lambda allows users to choose between regional and edge placement types. These preferences ensure the application is deployed and aligned with compliance and performance requirements.

Network traffic

Directing low-level network traffic to service instances is also a responsibility of the compute platform. This is because it is responsible for deployment sequencing, placement, and autoscaling, which all impact how traffic is directed to the service instances. Health checks can also play a role in traffic management, such as the readiness check in GCP Cloud Run and Kubernetes. By handling these tasks, the compute platform helps to ensure that traffic is efficiently and effectively routed to the appropriate service instances.

Monitoring and reporting

Any compute platform for distributed applications must provide deep application insights in the form of logs, metrics, and tracing. And today, there are a few widely accepted de facto standards in this space: logs are ideally in a structured format such as JSON or other industry-specific standards. The compute platform typically collects logs or provides extension points for specialized log draining and analyzing services to access the logs. That can be a DaemonSet on Kubernetes, a Lambda partner extension for monitoring, or a Vercel edge function log Drainer. The compute platform must support the collection and analysis of metrics and tracing data in order to provide comprehensive insights into the performance and behavior of a distributed application. There are several industry-standard formats and tools for handling this data, such as Prometheus for metrics and OpenTelemetry (OTEL) for tracing. The compute platform may offer built-in tools for collecting and analyzing this data or provide extension points for specialized services to access the data. Regardless of the granularity of the code (microservice or function) or the location (edge or not), the compute platform should allow for the capture and export of logs, metrics, and tracing data to other best-of-breed cloud services such as Honeycomb, DataDog, Grafana to name a few.

Compute binding trends

Compute bindings are language and application runtime agnostic and are primarily used by the operations teams for managing applications at runtime rather than developers implementing them.

While the size and complexity of applications can vary from monoliths to functions, they are typically packaged in containers with health check endpoints,

lifecycle hooks implemented, and metrics exposed. Understanding these compute bindings will help you to effectively use any container-based compute platform, whether it is an on-premises Kubernetes cluster, a managed container service such as AWS ECS, Google Cloud Run, Azure Container Apps, or function-based runtimes such as AWS Lambda, GCP Functions, or edge runtimes such as Vercel edge function, CloudFlare workers, or Netlify edge functions, etc. Using open and de facto standard APIs will help you create not only portable applications, but also limit vendor lock-in by using operational practices, and tools that are portable across cloud vendors and service providers.

Integration bindings

Integration bindings, on the other hand, are meant to be used by developers and not operations teams. They are centered around common distributed systems' implementation areas such as service invocation, event-driven interactions, task scheduling, and stateful workflow orchestration.

They help connect the application with specialized storage systems and external systems through cloud-based middleware-like services, which I collectively refer to in this article as the integration cloud. In the same way containers provide compute abstractions, integration cloud services provide language-agnostic integration abstractions as a service. These primitives are independent of the use case, application implementation, runtime, and compute environment. For example, the Retry pattern, the DLQ pattern, the Saga pattern, service discovery, and circuit breaker patterns can all be consumed as a service from the integration cloud.

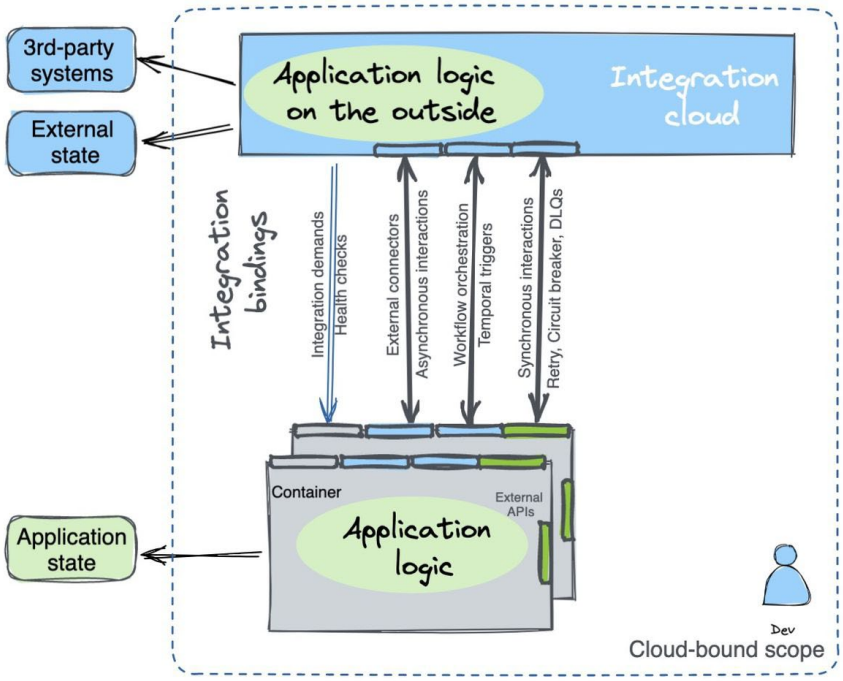


Figure 5: Application and platform integration bindings

Today, a pure integration cloud with all the main patterns exposed as standalone features does not exist yet. Early cloud services are offering some of these integration primitives as features of storage systems such as Kafka, Redis, and the like, but these features can be rarely used on their own or combined with others. Notable exceptions here are services such as AWS EventBridge and Azure Event Grid, which you can use with multiple cloud services from the same vendor, but no other vendors directly. This is a fast-evolving space with some good examples and some gaps that are not filled yet, but I believe they will be in the future. To work, the application must bind with the integration cloud services and offload some of these developer responsibilities. Here are the main types of integration cloud services and binding aspects.

Integration demands

In the same way an application can demand resources and express deployment and placement preferences to the compute platform, the application can also demand and activate specific integration bindings. These bindings can be activated through configurations passed

to the platform declaratively or activated at runtime through programmatic interactions. For example, applications can subscribe to pub/sub topics using declarative and programmatic subscriptions. An AWS Lambda function can subscribe to an event source declaratively through configurations or programmatically through a client library or SDK by asking the integration platform for specific binding to be registered or unregistered. Applications can subscribe to cron job triggers, activate a connector to external systems, make configuration changes, etc., all running on the integration cloud.

Workflow orchestration

A persistent service orchestration logic is a very common necessity and a prime candidate for externalizing and consuming it as a service. As a result, workflow orchestration is among the best-known integration binding types today. Among the common uses of this service are implementations of the Saga pattern for service and business process orchestration, function orchestration with AWS Step Functions, Google Stateful Functions, Azure Durable Functions, task distribution with Google Workflow, and many other services. When such a binding is used, part of the application orchestration state and logic is offloaded into another service. While the application services have internal state and logic to manage that state, other parts are on the outside, potentially in some other cloud service. This represents a shift in how applications are designed and operated as a single self-contained unit today. Future applications will have not only data on the outside but integration on the outside too. With the increasing adoption of integration cloud, more integration data and logic will start living on the outside.

Temporal triggers

Temporal binding represents a time-bound specialization of orchestration binding. It has one single goal, to trigger various services at specific times based on the given policy. Examples in this category are AWS EventBridge Scheduler, Google Cloud Scheduler, Upstash Qstack service, and many others.

Event-driven and messaging services

These bindings act as an event store to offload requests and decouple applications, but they are increasingly not limited to storage and expanding towards providing message processing patterns. They provide developer primitives on top of the event store, such as dead-letter queue, retries, delayed delivery, and message processing patterns such as filtering, aggregation, reordering, content-based routing, wiretap, etc. Examples of this binding would be things such as Confluent Cloud kSQL, AWS EventBridge, Decodable data pipelines, and others.

External connectors

These bindings help connect to external systems. They also perform data normalization, error handling, protocol conversion, and data transformation. Examples are Knative source importers, AWS EventBridge connectors, Confluent Cloud connectors, Decodable Kafka connectors, AWS Lambda source and destinations.

Health checks

Health checks are essential in compute bindings where a failing health check usually causes the application to restart. Integration bindings also need health checks but for a different purpose: an integration health check does not influence the application runtime, but it tells the integration cloud whether the application is capable of handling integration-driven interactions or not.

A failing integration health check can stop the integration binding until the application is back and healthy when the integration bindings are resumed. Very often, you can use the same application endpoints for compute and integration binding checks. A good example is Dapr's application health-check that can temporarily stop consumers and connectors from pushing data into an unhealthy application.

Other bindings

There are even more bindings that are coming up and fall under the category of integration bindings. For example, feeding an application with introspection data such as Kubernetes Downward API and Lambda en ironment variables that provides a simple mechanism for introspection and metadata injection to applications. Configuration and secret bindings where secrets are not only injected into the application at startup time, but any configuration updates are pushed to the application with sidecars such as Hashicorp Vault Sidecar Injector or Dapr's configuration API, service binding specification for Kubernetes. And less common patterns, such as a distributed lock, which is also a type of integration binding that can provide a mutually exclusive access to a shared resource.

Integration binding trends

Containers are becoming the most popular and widely used portable format for packaging and running applications, regardless of whether they are long-running microservices or short-lived functions. Integration bindings, on the other hand, can be grouped into distinct problem areas, such as event-driven interactions, stateful orchestration, and state access, and vary in terms of underlying storage and usage patterns. For example, Apache Kafka is a de facto standard for event logs, AWS S3 API is for document access, Redis is for key-value caching, PostgreSQL is for relational data access, etc. What makes them standards is the growing ecosystems of libraries, tools, and services built around them, giving assurance about the substantial degree of maturity, stability, and future backward compatibility. But these APIs alone are limited to storage access aspects only and often require developers to address distributed system challenges within the application code. Aligned with the software commoditization direction that goes up the stack, the integration bindings are becoming available as a service. There is a growing number of serverless cloud services that offer additional integration capabilities that the application code can bind to in addition to data access.

In this model, a cloud-bound application typically runs on a serverless compute infrastructure, following the cloud native primitives. It binds with other serverless cloud services for service orchestration, event-processing, or synchronous interactions, as shown below.

One project that unites most integration bindings and developer concerns into an open-source API is Dapr from CNCF. It offers synchronous service invocation, stateful service orchestration, asynchronous event-driven interactions, and technology-specific connectors as APIs. Similar to how containers and Kubernetes act as a compute abstraction, Dapr acts as an abstraction for external services. Dapr also offers integration features independent of the underlying

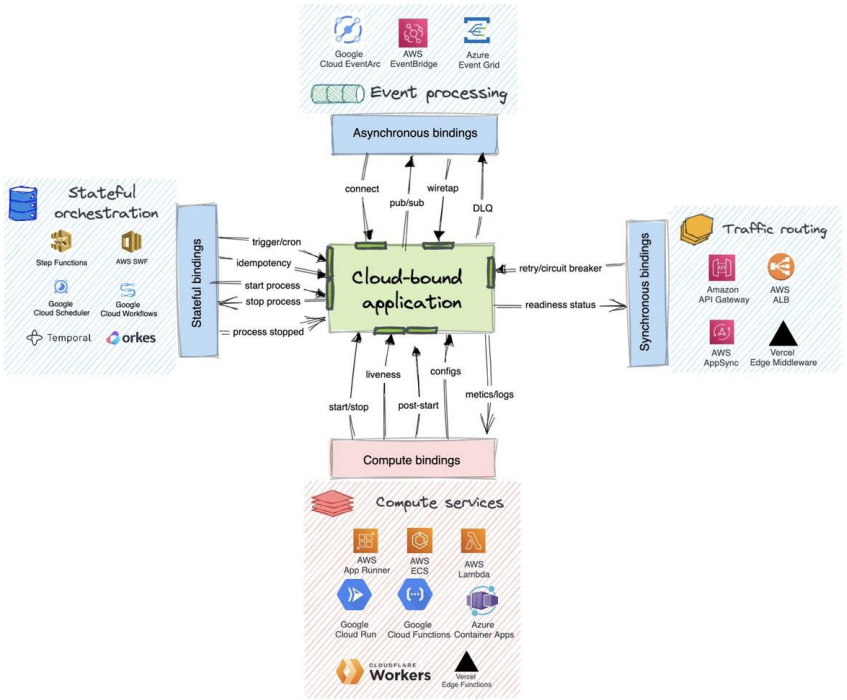


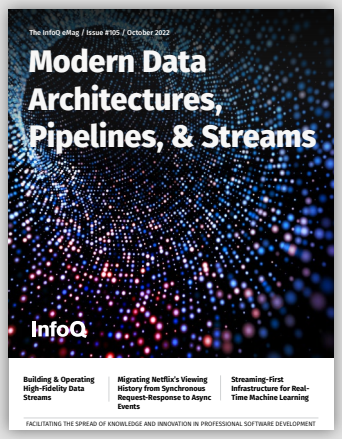
Figure 6: Cloud-bound applications ecosystem

cloud services and very often have to be implemented in the application layer, such as resiliency policies, dead-letter queues, delayed delivery, tracing, fine-grained authorization, and others. Dapr is designed to be polyglot and run outside of the application, making it easy to swap external dependencies without changing the internal architecture of an application as described in the hexagonal architecture. While Dapr is used primarily by developers implementing applications, once introduced, Dapr enhances the reliability and visibility of distributed applications, offering holistic benefits to operations and architect teams. To learn more about this topic, join me in person or virtually at QCon London later this year, where I will speak about "How Application-First Cloud Services Are Changing the Game."

Post-cloud-native applications

Cloud-bound applications represent cloud-native progression from addressing compute-only concerns to managing the application layer needs. This trend is accelerated by the expansion of cloud services up the application stack from infrastructure to application-first services. We can observe this transition in the explosion of developer-centric cloud services for stateful orchestration, event-driven application infrastructure, synchronous interactions, cloud-based development and test environments, and serverless runtimes. This move to application-first cloud services is giving rise to a new application architecture where more and more application logic is running within cloud services. This blending of applications with 3rd-party cloud services allows developers to offload more responsibilities, however, it can limit the flexibility and agility needed by changing business needs. To preserve the independence of an application's internal and external architectures, applications and cloud services need to be decoupled with clean boundaries at development time and deeply bound together at runtime using well-defined open APIs and formats. In the same way containers and Kubernetes have provided open APIs for compute, we need open APIs for application integration abstractions. This will enable the portability and reuse of operational practices and tools and development patterns, capabilities, and practices.

Read recent issues



Modern Data Architectures, Pipelines, & Streams

In this eMag, you'll find up-to-date case studies and real-world data architectures from technology SME's and leading data practitioners in the industry.

The InfoQ Trends Reports 2022

The InfoQ trends reports provide a snapshot of emerging software technology and ideas. We create the reports and accompanying graphs to aid software engineers and architects in evaluating what trends may help them design and build better software. Our editorial teams also use them to help focus our content on innovator and early adopter trends.

The Platform Engineering Guide: Principles and Best Practices

Platform Engineering has quickly become one of the hottest topics in DevOps. The explosion of new technologies has made developing software more interesting but has substantially increased the number of things that development teams need to understand and own.

InfoQ

InfoQ @InfoQ InfoQ InfoQ