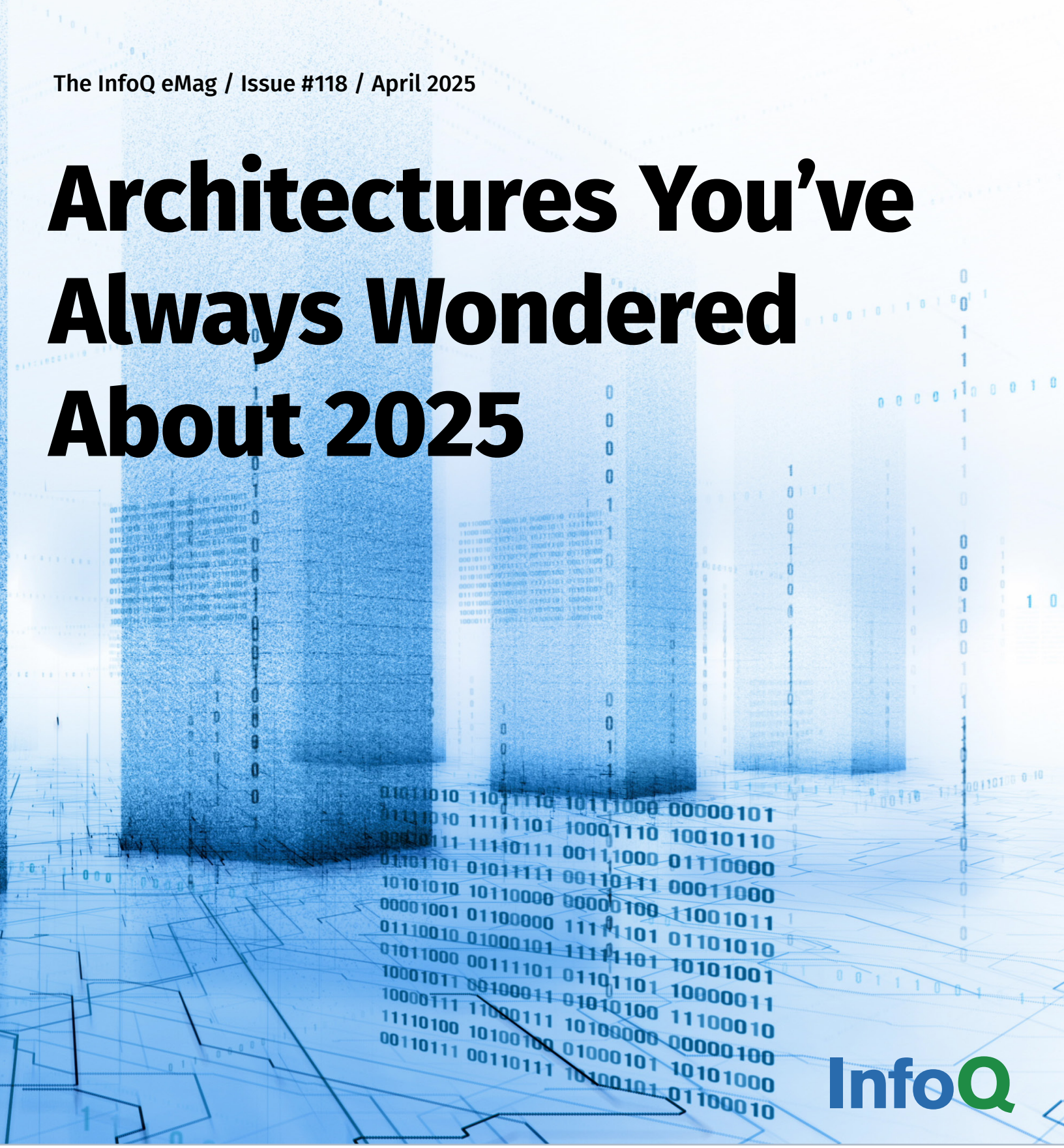
The InfoQ eMag / Issue #118 / April 2025

Architectures You've Always Wondered About 2025



InfoQ

Uber's Blueprint for Zero-Downtime Migration of Complex Trip Fulfillment Platform

How to Minimize Latency and Cost in Distributed Systems

Legacy Modernization: Architecting Real-Time Systems Around a Mainframe

FACILITATING THE SPREAD OF KNOWLEDGE AND INNOVATION IN PROFESSIONAL SOFTWARE DEVELOPMENT

Architectures You've Always Wondered About 2025

IN THIS ISSUE

PRODUCTION EDITORS Anamaria Bota & Roxana Bacila / COPY EDITORS Keith Conant & Susan Conant
GENERAL FEEDBACK feedback@infoq.com / ADVERTISING sales@infoq.com / EDITORIAL editors@infoq.com

CONTRIBUTORS

Madan Thangavelu

is an experienced engineering leader with expertise across fintech, mobile security, and large-scale consumer applications. In his current role, he serves as Senior Director of Engineering at Uber, where he leads the development of the flagship Uber Rider app and leads major business platforms including trip fulfillment platform, fares, and the founder of Uber's API platform.

Amir Souchami

is Chief Architect Aura from Unity. With a great passion for technology, he's constantly learning the latest to stay sharp and create, scale, and drive a positive business ROI. Follow up with him to chat about Gen AI, Machine Learning, Stream Processing, Microservices, and AdTech.

Jason Roberts

With over 15 years of experience in software development, Jason Roberts is passionate about creating innovative solutions that deliver value. Jason has a strong background in systems thinking, event driven programming, cloud architecture, and evolutionary systems. Jason uses these skills to lead the design and implementation of scalable, reliable, and maintainable systems and applications.

Sonia Mathew

has spent over 20 years experience in Software Development leading teams, she is passionate about using technology to deliver value. She has a background in leading large transformational projects. Sonia has broad and deep knowledge of web technologies, Integration (API/ Webservice/Microservices), Angular, API Gateway, Elastic Search, Kafka EDI, data warehousing and BI, Java, Angular, C# and RDBMS.

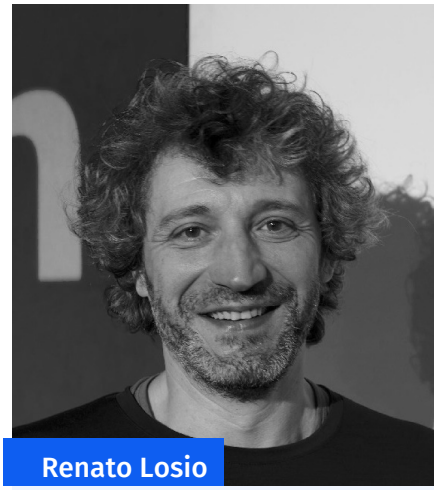
Leander Vanderbijl

is an engineer and architect that has worked across the entire stack and has been working at Kry/Livi for the past number of years. He has developed large enterprise applications, migrated cloud platforms, designed data query frameworks, architected integration systems and built lots and lots of APIs.

Rashmi Venugopal

has a track record of building and operating reliable distributed systems at scale that power mission-critical applications. She spent the last decade designing and developing complex software systems enabling consumers to seamlessly pay for their subscription at Netflix, compute tiered prices for the different ride options at Uber, and manage Azure cloud networking configurations at Microsoft.

A LETTER FROM THE EDITOR



Renato Losio

has extensive experience as a cloud architect, tech lead, and cloud services specialist. Currently, he lives in Berlin and works remotely as a principal cloud architect. His primary areas of interest include cloud services and relational databases. He is an editor at InfoQ and a recognized AWS Data Hero. You can connect with him on LinkedIn.

How did Uber migrate critical systems to a hybrid cloud architecture with zero downtime? How can we architect software to reduce costs and latency in the cloud? What are the major scaling challenges in legacy modernization and mainframe systems? And why is technological renovation so often overlooked?

While the scale of some of these migration and modernization efforts may seem extreme, posing challenges rarely faced by smaller organizations, there are valuable lessons to be drawn. Legacy systems are an inevitable part of success in long-standing companies, but transforming them is essential for future innovation. This eMag explores the challenges of modernizing, migrating, and scaling applications, spotlighting architectures that are redefining what is possible in software today.

In 'Uber's Blueprint for Zero-Downtime Migration of Complex Trip Fulfillment Platform,' Madan Thangavelu discusses how Uber migrated critical systems from on-prem to a hybrid cloud architecture with zero downtime and business impact. Highlighting how Uber operates one of the most intricate real-time fulfillment systems globally, Thangavelu stresses the importance of multiple precautionary and mitigation mechanisms before executing large migrations. The author highlights that a backward compatibility layer and developing mechanisms to drain traffic to new or old systems with heavy business context are critical to de-risking large migrations.

'How to Minimize Latency and Cost in Distributed Systems' by Amir Souchami emphasizes that distributed systems spanning multiple availability zones can incur significant data transfer costs and performance bottlenecks. Souchami discusses how organizations can reduce costs and latencies by applying zone-aware routing techniques, a strategy designed to optimize network costs and latency by directing traffic to services within the same availability zone whenever possible.

While designing systems that take advantage of modern platforms is critical for building scalable, evolvable applications, sometimes legacy modernization is part of the equation. Jason Roberts and Sonia Mathew discuss how National Grid built an event-driven system using change data capture on top of a DB2 mainframe to power a cloud-native GraphQL API hosted in Azure.

Covering another legacy modernization scenario, this edition shows how Livi, a digital healthcare service, managed data synchronization and decided between serverless and containerized solutions. Leander Vanderbijl examines the challenges and strategies involved in transitioning legacy healthcare systems to cloud-native architectures. The author shows how a hybrid approach, combining lift-and-shift strategies with cloud-native rewrites, can help businesses meet customer needs while building a future-proof platform.

Finally, Rashmi Venugopal explains why technological renovation is often overlooked and shares learnings on gracefully outgrowing technology and architectural choices, scaling payment orchestration at Netflix to 250M members and preparing for the next 250M. The author discusses various strategies to tackle such technical renovation initiatives, like evolutionary architecture, deprecation-driven development, and intentional organization design.

We hope you find value in the articles and resources in this eMag and are inspired by these stories. We would love to receive your feedback via editors@infoq.com, on Bluesky, or on X (formerly Twitter). We hope you enjoy reading it!

Uber's Blueprint for Zero-Downtime Migration of Complex Trip Fulfillment Platform

by **Madan Thangavelu**, Sr. Director of Engineering @Uber

In large-scale distributed systems, migrating critical systems from one architecture to another is technically challenging and involves a delicate migration process. Uber operates one of the most intricate real-time fulfillment systems globally. This article will cover the techniques to migrate such a workload from on-prem to a hybrid cloud architecture with zero downtime and business impact.

System Complexity

Uber's fulfillment system is real-time, consistent, and highly available. Users consistently engage with the app, initiating, canceling, and modifying trips.

Restaurants continually update order statuses while couriers navigate the city to deliver packages. The system executes over two million transactions per second to manage the states of our users and trips on the platform.

I was the engineering leader driving the last redesign of this fulfillment system from an on-prem system to a hybrid cloud architecture. Beyond the code and validation required to develop a new system, the most challenging part of the project was to design a migration strategy that involved zero downtime or customer impact.

Architecture of the Previous System

Let me simplify the previous fulfillment system. It consisted of a few services maintaining the state of different user and trip entities in memory. Additionally, there were supporting services to manage locking, searching, and storage systems to support cross-datacenter replication.

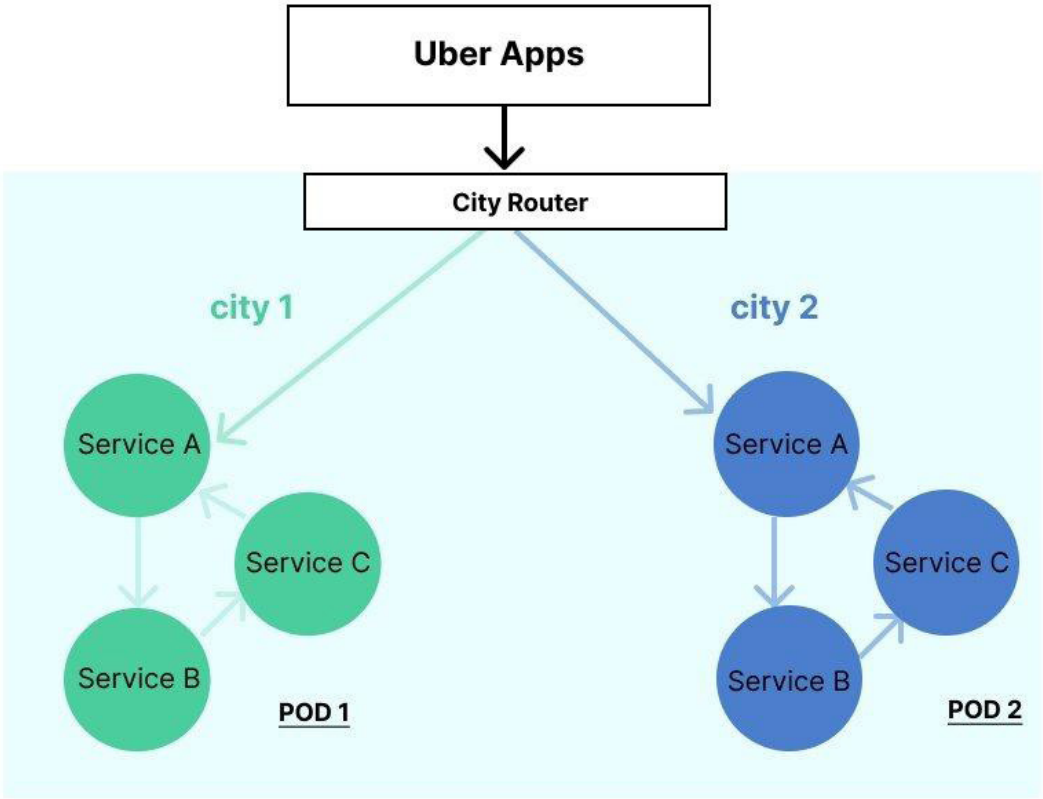
All rider transactions were routed to the "demand" service, and all driver transactions were routed to the "supply" service. The two systems were kept in sync with distributed transaction techniques.

These services operated as a single unit called a pod. A pod is a self-sufficient unit with many services interacting with each other to power fulfillment for a single city. Once a request enters a pod, it remains within the services in the pod unless it needs access to data from services outside the pod.

In the image below, the green and blue service groups represent two pods, with a replica of each service within a pod. Traffic from city 1 is routed to pod 1, and traffic from city 2 is routed to pod 2.

Furthermore, the system facilitated distributed transactions among services A, B, and C to ensure synchronization of entities stored within them, employing the saga pattern. Entity consistency was maintained within each service through in-memory data management and serialization.

At the same time, this system scaled in the early phase of Uber. It has some architectural choices that could have worked better with Uber's growing demands. Firstly, The system was designed to be available rather than consistent. This meant that



when multiple systems store different entities, the system is eventually consistent without a lack of a true acid-compliant system to make changes across them. Since all the data was maintained in memory, the system also inherently has limitations on vertical scaling and the number of nodes for a given service.

Redesigned System

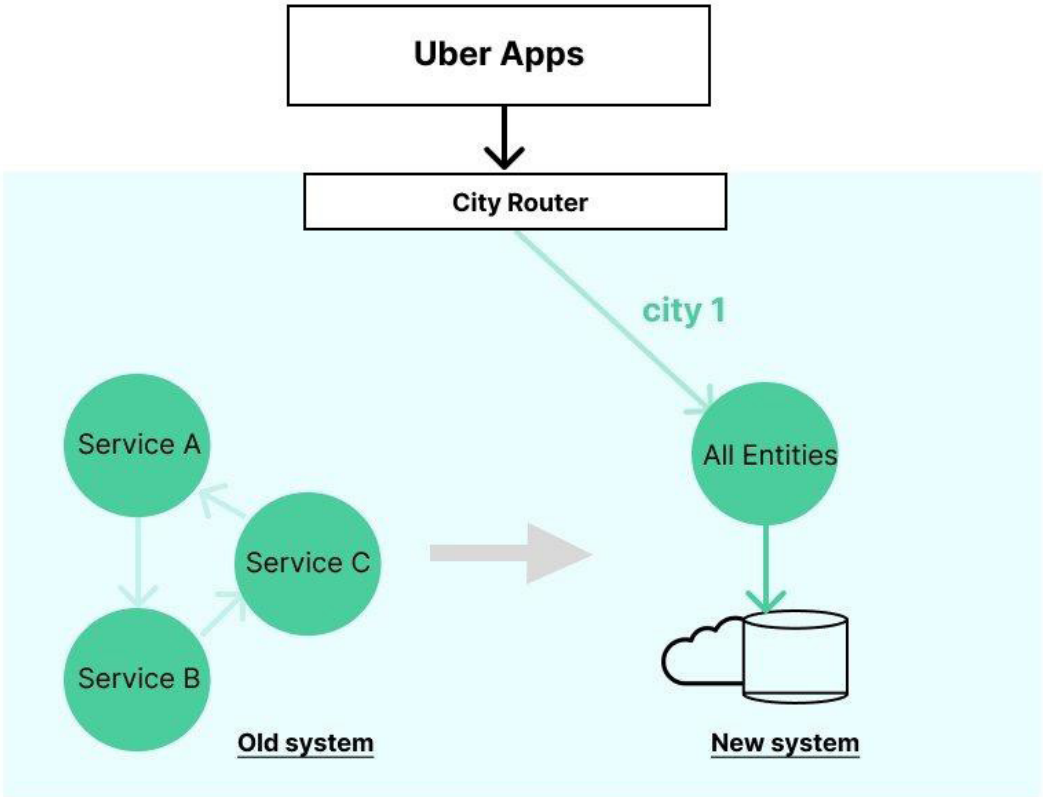
The new system consisted of fewer services. The services that stored entities like “demand” and “supply” were consolidated into a single application backed by a cloud datastore. All the transactional responsibilities were moved down into the datastore layer.

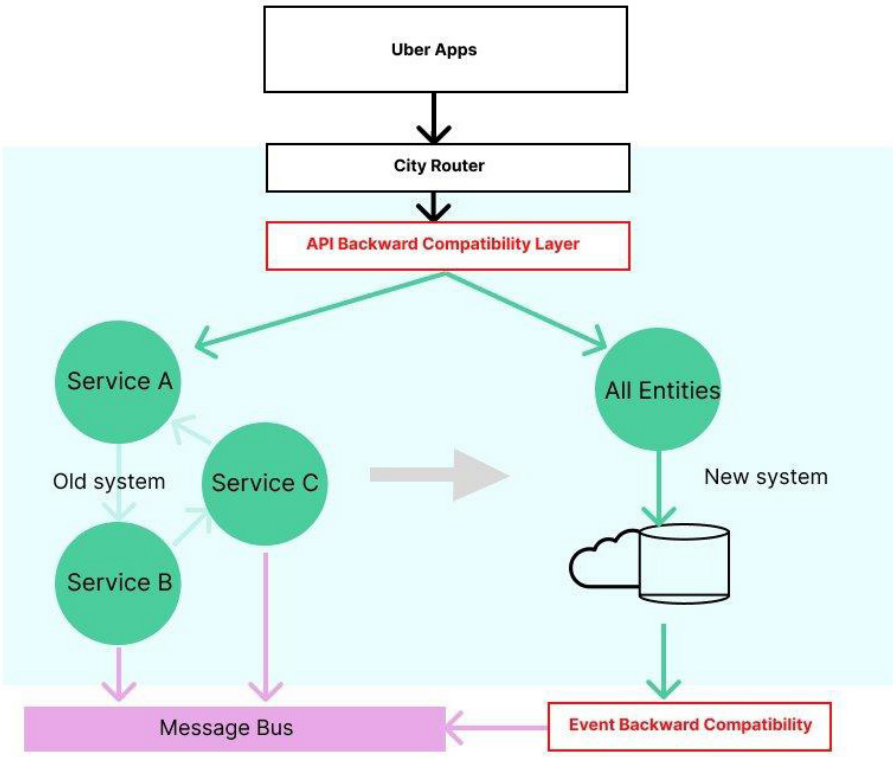
New System and Existing Service Consumers

A multi-dimensional strategy covering various aspects of migration is needed for such critical migrations. In microservices environments, every system interacts with many systems around it. The two primary ways of interaction are APIs and events published to a message bus by the services.

The new system modified all the core data models, API contracts, and event schemas. With 100s of API callers and consumers of the system, it is not an option to migrate them all simultaneously. We adopted the strategy of a “backward compatibility” layer that maintained existing API and events contracts.

Creating a backward compatibility layer allows a system to rearchitect while all consumers consume the old interfaces. Over the next few years, consumers could gradually move to new APIs and events without coupling them to this redesign. Similarly, any consumer of events from the old system would continue receiving those events in the old schema published by the backward compatibility layer.





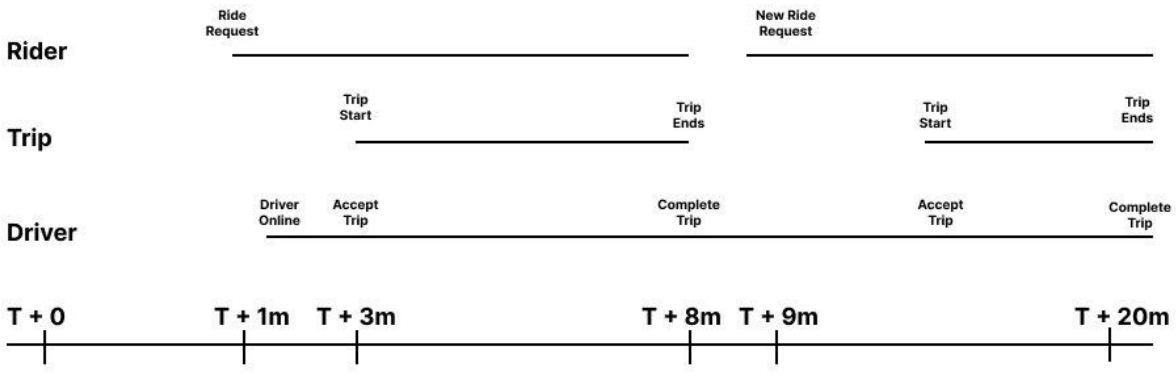
Lifecycle of Entities on the Platform

What makes this transition complex is that the entities are continuously changing states. If we consider the three entities in the system - a rider, a driver, and a trip - they can start and stop at different points in time. The previous system had these entities in different in-memory systems, and the new system has to reflect it all in a database.

The migration challenge is to ensure that every entity state and its relationships are preserved in this transition while actively changing at a high rate.

Migration Strategy

Multiple strategies must be adopted at each stage - pre-rollout, during rollout, and post-rollout—to move traffic from the existing system to the new one. I will go over each in detail in the following section of the article.



Before Rollout

Shadow validation

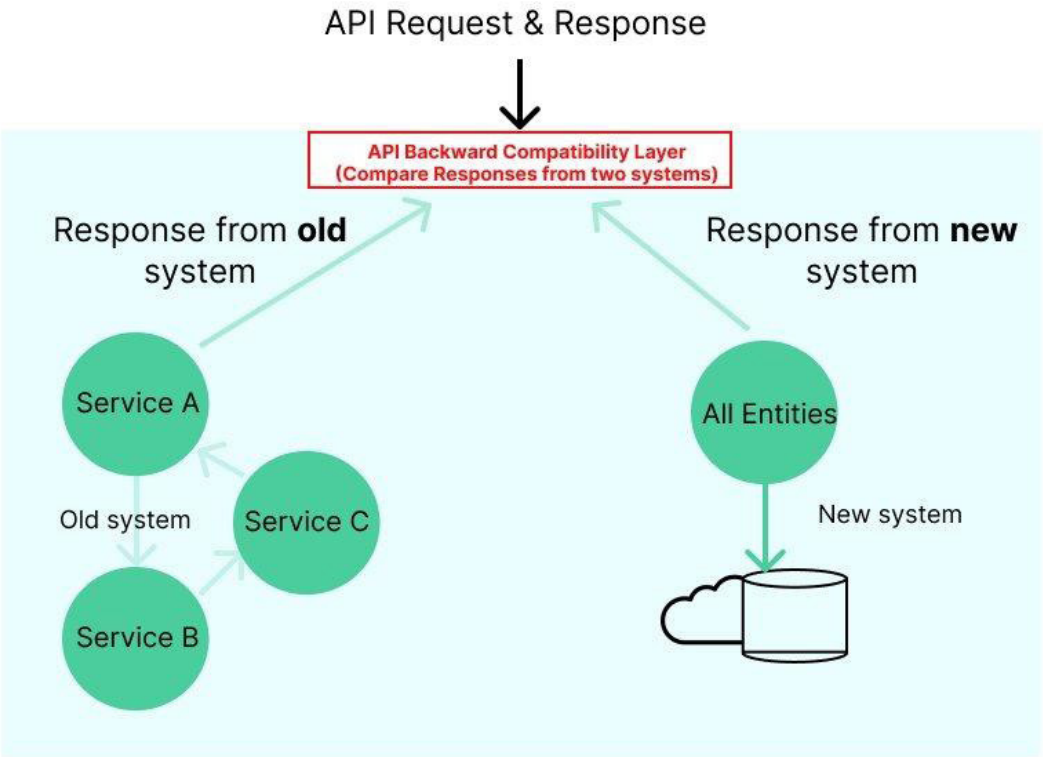
Having a system to validate the API contracts between the old and the new system is critical to gaining confidence that the new system is in parity with the existing system. The backward compatibility layer provides a key functionality of validating API and events parity between the old and the new system.

Every request is sent to both the old and the new system. The responses from both systems are compared per key and value, and the differences are published into an observability system. Before each launch, the goal is to ensure that there are no differences between the two responses.

There are some nuances in a real-time system. In a transient system, not all calls will succeed with exact responses. One strategy is to review all successful calls as a cohort with high match rates and ensure that most API calls are successful.

There are valid situations where calls to the new system will be unsuccessful. When a driver requests an API to go offline on their app, but the new system has no prior record of the user going online, it will generate a client error. In these edge cases, the parity mismatch can be ignored but with caution.

Further, read-only API parity is the easiest because there are no side effects. On write APIs, we introduced a system where the old system would record the request responses in a shared cache, and the new system would replay the response. Since we have a trace header, the new system will transparently pick up the responses originally received by the old system and put them into the new system from the cache instead of making a full external call to the outside system. This allows more consistency in responses and fields from dependent systems, allowing us to better match the new and the old systems without interferences from external dependencies.



E2E Integration testing

Large rewrites are the right moments to improve end-to-end (E2E) test coverage. Doing so will protect the new system as it scales its traffic and new code routinely gets launched in the new system. During this migration, we ramped up our tests to over 300 tests.

E2E tests should be run at multiple points of a development lifecycle: when an engineer creates code, during pre-deploy validation of the build, and continuously in production.

Blackbox testing in canary

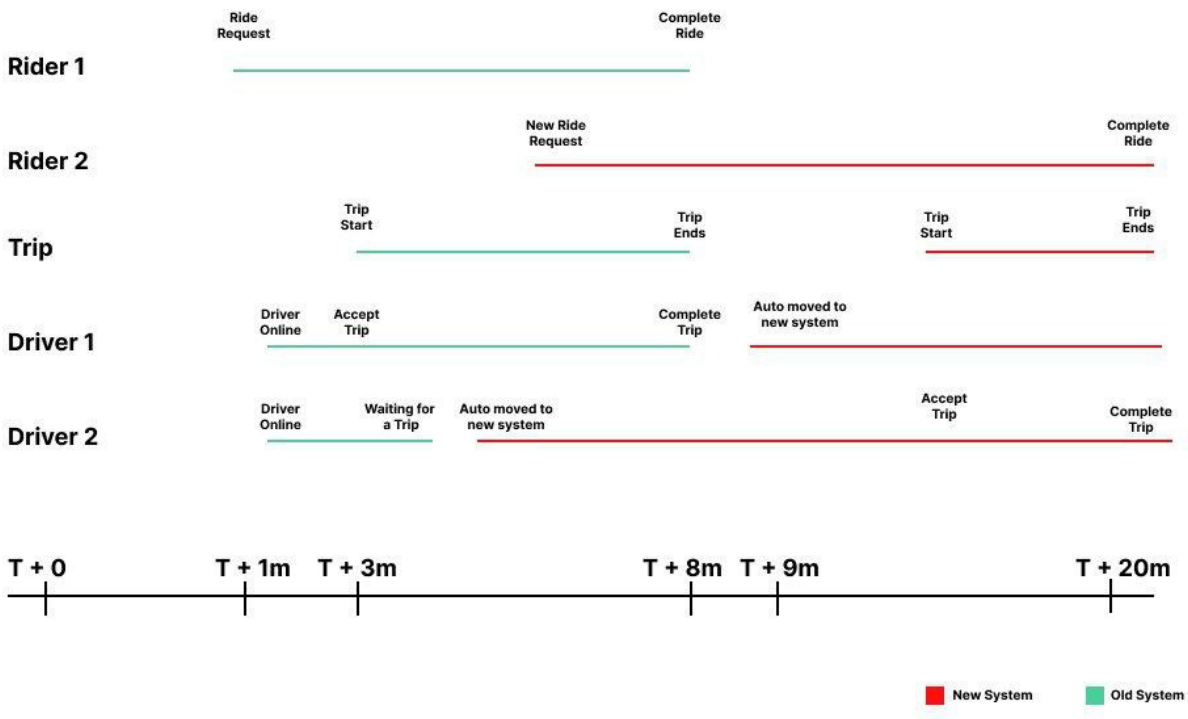
Preventing bad code in critical systems can be achieved by exposing the code only to a small subset of the requests, reducing the blast radius. To further improve, it can expose the code only to a pre-production environment with continuously running E2E tests. Only if the canary environment passes successfully can the deploying system proceed further.

Load testing

A comprehensive load test must be undertaken before a significant production load is placed on the new system. With our ability to redirect traffic to the new system via a backward-compatible layer, we can enable full production traffic to the new system. Custom load tests were written to validate unique integrations like the database network systems.

Warming up database

Database or compute scaling with cloud providers is not instantaneous. When large loads are added, the scaling system does not catch up quickly enough. Data re-balancing, compaction, and partition splitting in distributed data stores can cause hotspots. We simulated synthetic data load against these systems to achieve a certain level of cache warming and splits. This ensured that when production traffic was pointed at these systems, they operated without the sudden scaling needs of the system.



Fallback network routes

Since the application and the database systems are across two different cloud providers, one on-prem and another in the cloud, special attention has to be paid to managing network topology. The two data centers had at least three network routes established to connect to the database across the internet through dedicated network backbones. Validating and load testing these networks at a capacity that is over 3X was critical to ensure their reliability in production.

During Rollout

Traffic Pinning

Multiple apps are interacting with each other on the same trip in a marketplace. A driver's and rider's trips represent the same trip entity. When moving an API call from a rider to the new system, the corresponding trip and driver's states can't be left behind in the old system. We implemented routing logic that enforces trips to continue and end in the same system where the trip started. To make this determination, we start recording all our consumer identifiers around 30 minutes before executing a migration. The requests for corresponding entities continue to be pinned to the same system.

Rolling out in phases

The first set of trips migrated for a given city are the E2E tests. Next, idle riders and drivers not on an active trip are migrated to the new system. After this migration, any new trips were initiated only into the new system. This duration has two systems serving two marketplaces in parallel. As ongoing trips are completed in the old system, the riders and drivers who become idle are progressively migrated to the new system. Within an hour, the entire city is drained to the new system. In the event of an outage in the new system, we can execute the same sequence in reverse.

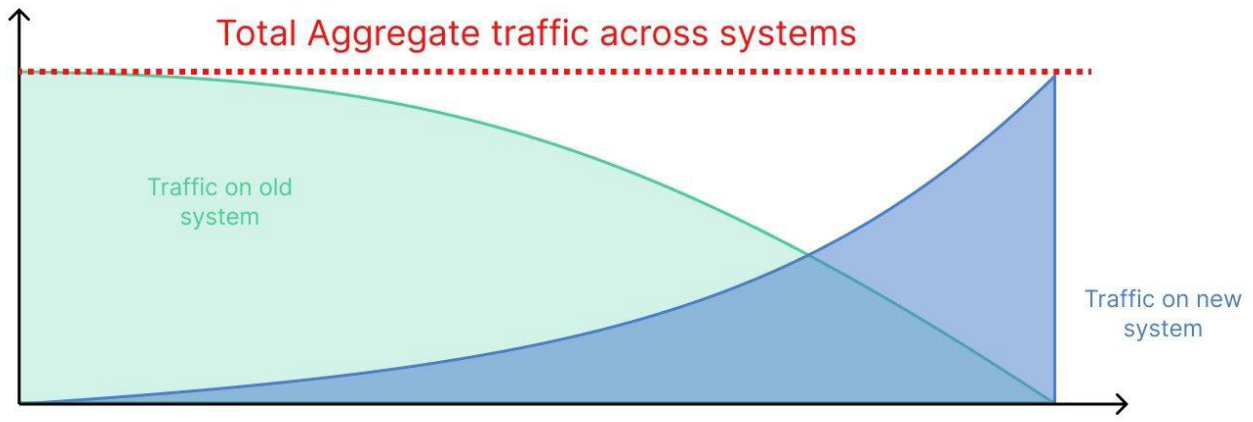
After Rollout

Observability and rollback

Robust observability across both systems is critical while executing a switch between two systems. We developed elaborate dashboards that show traffic draining from one system while the traffic picks up on the new system.

Our goal was to ensure that the key business metrics of trip volume, available supply, trip completion rates, and trip start rates remained unchanged throughout migration. The aggregate metrics should remain flat, while the mix of old vs new systems would change at the moment of migration.

Since Uber operates in thousands of cities with numerous functionalities, it is crucial to observe these metrics at the granularity of a city. It was not possible to observe all the metrics manually when migrating 100s of cities. Specialized tools must



be built to observe these metrics and highlight the most deviant ones. Alerting was also futile since a surge of alerts improperly tuned to different city traffic would not help with confidence in migration. A static tool that allowed an operator to analyze the health of cities was crucial in ensuring that all cities were operating well.

If one city seemed unhealthy, the operator could roll just the single city back to the old system while leaving all the other cities in the new one.

Blackbox tests in production

Similar to the "Blackbox testing in canary" strategy, the same tests should be run against production systems continuously to discover problems.

Executing Successful Migrations

It is crucial to have enough runway in the old system so it does not distract us from developing the new system. We spent four months hardening the reliability of the old system before starting development on the new system. Planning across projects is vital such that features are initially developed in the old system, but mid-way through the development of the new system, features are dual developed in the old and the new system in parallel, finally only in the new system towards later stages of the new system. This juggling of stakeholders is extremely critical to avoid being stuck maintaining two systems eternally.

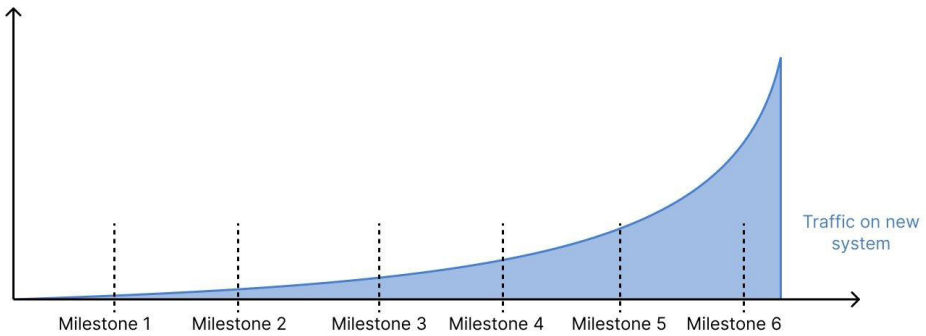
When migrating systems of absolute criticality, which need strong consistency guarantees and are primary to business uptime, almost 40% of the engineering effort should go into observability, migration tooling, and robust rollback systems when building the new stack.

Vital technical elements involve complete control of API traffic at an individual user level, strong field-level parity and shadow-matching machinery, and robust observability across the systems. In addition, the moment of migration is a key engineering challenge fraught with race conditions and special handling that may require explicit business logic or product development.

Traffic migration to the new system typically remains flat, with slow growth in the initial iterations. Once there is confidence in the new system, it usually accelerates exponentially, as shown in the image below.

The challenge of accelerating earlier is that you are stuck with maintaining two systems with equally important traffic volumes for longer periods. Any outages occurring due to scale in the new system will have an outsized impact on the production users. It is best to validate all features against the least number for an extended period before scaling traffic migration to the new system in a relatively shorter duration.

At Uber, we executed this migration with less than a few thousand users impacted throughout the development and migration. The techniques discussed were cornerstone methods to help with a smooth transition from our system to a brand-new fulfillment system.



SPONSORED ARTICLE

AKKA

Agentic AI: Transforming Enterprise Architecture

As enterprises grapple with the limitations of conventional AI tools, agentic AI emerges as a transformative approach to building autonomous, intelligent systems that can reason, plan, and act while being governed and controlled. Unlike traditional AI assistants that follow static workflows, agentic AIs can independently set goals and execute tasks. Architecturally, this marks a fundamental shift from the stateless, transactional design of SaaS applications to a stateful, conversational, and dynamic system model.

From N-tier to N+A-tier Architectures

Traditional enterprise applications follow an N-tier architecture optimized for deterministic, transactional workflows. Agentic AI, by contrast, operates in a different paradigm—requiring an "A-tier" (Agentic Tier) to support its conversational, adaptive, and context-aware behaviors. This A-tier co-exists with the N-tier infrastructure and provides essential services such as:

- **Agent Orchestration**: Multi-agent coordination across long-lived, multi-step workflows to produce a singular outcome.
  Memory Systems: Offering persistent short- and long-term memory to agents is essential due to LLMs' statelessness.
- **Streaming Support**: Enabling agents to process real-time, high-volume multi-modal data.
- **Integrations**: Allowing seamless connectivity with enterprise APIs, tools, data sources, and other agents using protocols like OpenAPI,Model Context Protocol and Agent-to-Agent.

This N+A-tier approach enables enterprises to add intelligent automation without overhauling their existing systems.

Engineering Around LLM Limitations

Large Language Models (LLMs) are core to agentic AI but come with constraints: they are slow, stateless, and limited to only the data available when it was trained. Architectures must therefore include:

- **Execution reliability layers** that ensure reliability and performance, even when an LLM is overloaded.
- **Real-time context assembly mechanisms** that enrich prompts with memory, documents, events, and structured data.
- **Knowledge grounding systems** like RAG (retrieval-augmented generation) to integrate proprietary or up-to-date information.

In effect, agents must *construct* the right prompt—curating context intelligently to guide LLMs toward reliable outputs.

Agentic Workflow Patterns

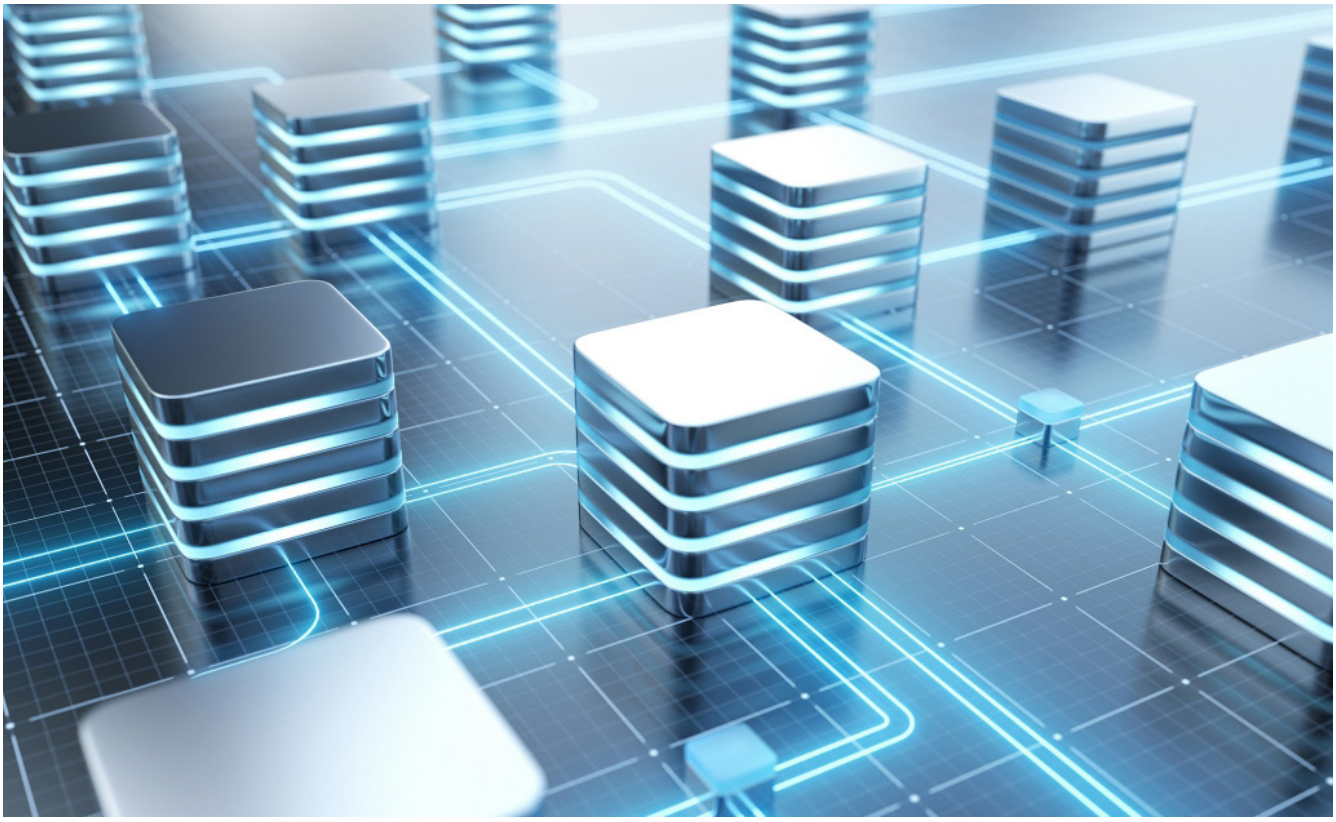
Agentic AI executes tasks using workflow patterns derived from an agentic enrichment loop, where prompts are iteratively refined based on the LLM's output to a preceding prompt. Moreover, these patterns will coordinate multiple agents with specific subtasks in order to execute the complete task. Common patterns include:

- **Prompt Chaining:** Sequential subtasks (e.g., write, then translate).
- **Delegation:** Dynamic task routing / delegation based on classification (e.g., route to the sales LLM vs the support LLM)
- **Parallelization:** Executing subtasks concurrently for performance or diversity (e.g., give a response from three different personas)
- **Synthesizer:** Decomposing unknown tasks via an orchestrator LLM and reassembling results (e.g., identify the information sources needed to answer the question, and then execute the search)
- **Service Agents:** Multi-step planning grounded in user feedback (e.g., travel planning and booking).

These patterns reflect the shift from hard-coded logic to emergent reasoning driven by LLMs, with system design focused on orchestration rather than prediction alone.

In summary, agentic AI requires rethinking enterprise architectures to support intelligent, autonomous agents at scale. The key is integrating conversational AI with orchestration, memory, streaming, and tool support in a hybrid N+A-tier model—ushering in a new era of adaptive, context-rich systems.

Read more here.



How to Minimize Latency and Cost in Distributed Systems

by **Amir Souchami**, Chief Architect Aura from Unity

The microservices architectural approach has become a core factor in building successful products. It was made possible by adopting advanced cloud technologies such as service mesh, containers, and serverless computing. The need to rapidly grow, create maintainability, resilience, and high availability made it standard for teams to build "deep" distributed systems: Systems with many microservices layers. Systems that span across multiple Availability Zones (AZs) and even regions. Those systems are often characterized by dozens of chatty microservices that must frequently communicate with each other over the network.

Distributing our resources across different physical locations (regions and AZs) is crucial for achieving resilience and high availability. However, significant data transfer costs and performance bottlenecks can be incurred in some deployments. This article aims to create awareness of those potential issues and provide guidelines on maintaining resilience while overcoming these challenges.

The Problem

Using multiple AZs is a best practice and an essential factor in maintaining service availability. Whether those are our microservices, load

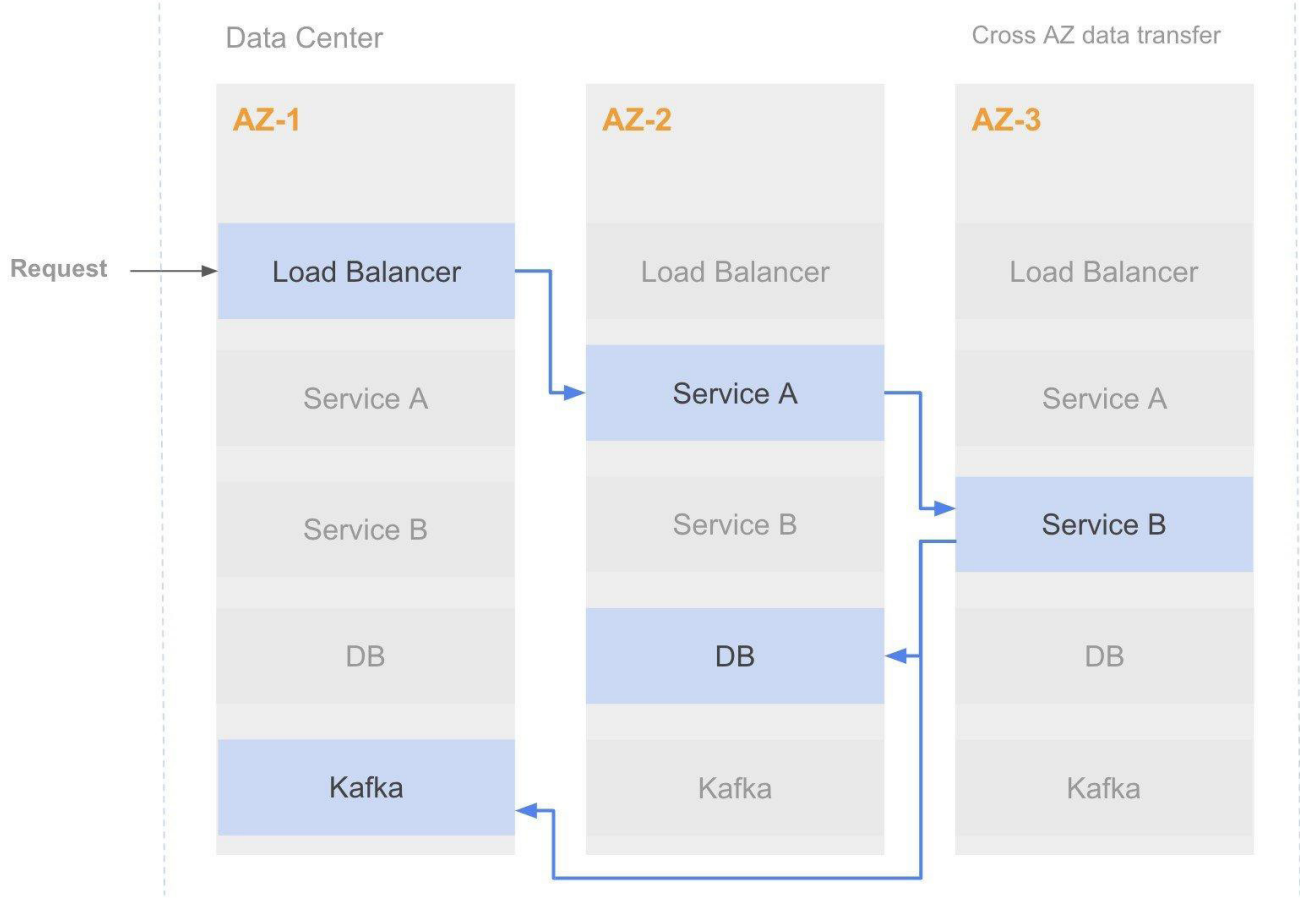


Figure 1: Illustration of a request flow that spans across multiple availability zones

balancers, DBs, or message queues, we must provision them across multiple AZs to ensure our applications are geographically distributed and can withstand all sorts of outages.

In some critical systems, it is also common to distribute the resources and data across multiple regions and even multiple cloud providers for the sake of fault tolerance.

Distributing our services across multiple AZs requires us to transfer data between those AZs. Whenever two microservices communicate with each other or whenever a service reads data from a distributed DB, the communication will likely cross the AZ boundary. This is because of the nature of load balancing.

Usually, load balancers distribute the traffic evenly between instances of the upstream service (the service being called) without awareness of the AZ in which the origin service is located. So, in reality, whenever systems are provisioned over multiple AZs, cross-AZ traffic is likely to happen a lot.

But why is it a cloud cost burden and even a performance bottleneck? Let's break it down.

The Cost Burden

Every time data is transferred between two AZs, it is common among the big cloud providers to apply data transfer charges in both directions for incoming cross-AZ data transfer and for outgoing cross-AZ data transfer. For example, if it costs $0.01/GB in each direction, a single terabyte of transferred data between two same-region

availability zones will cost you $10 for the egress and $10 for the ingress, a total of $20. However, those costs can quickly get out of control on distributed systems. Imagine that as part of a request, your Load Balancer instance on AZ-1 may reach out to Service A in AZ-2, which in turn calls Service B on AZ-3, which in turn reads a key/value record from a DB on AZ-2, and often consumes updates from a Kafka broker on AZ-1 (see figure 1).

So, while this architectural choice prioritizes availability, it is prone to extreme situations where all the communication between services on a single request is done across AZs. So, the reality is that in systems with various DBs, message queues, and many microservices that talk to each other, data must frequently move around and will likely become extremely expensive. A burden that tends to grow every time you add services to your stack.

The Performance Bottleneck

AZs are physically separated by a meaningful distance from other AZs in the same data center or region, although they are all within up to several miles. This generally produces a minimum of single-digit millisecond roundtrip latency between AZs and sometimes even more. So, back to our sample where a request to your Load Balancer instance on AZ-1 may reach out to Service A in AZ-2, which in turn calls Service B on AZ-3, which in turn reads a key/value record from a DB on AZ-2, and often consumes updates from a Kafka broker on AZ-1. That way, we could easily add over a dozen milliseconds to every request. A precious time that can drop to sub millisecond when services are in the same AZ. And like with the cost burden, the more services you add to your stack, the more this burden grows.

So, how can we gain cross-AZ reliability without sacrificing performance?
And is there anything to do about those extra data transfer costs?
Let's dive into those questions.

Zone Aware Routing to the Rescue

Zone Aware Routing (aka Topology Aware Routing) is the way to address those issues. And in Aura from Unity, we have been gradually rolling this capability for the past year. We saw that it saved 60% of the cross-AZ traffic in some systems. We understood how to leverage it to optimize performance and reduce bandwidth costs significantly. We also uncovered where it didn't fit and should be avoided. So, let's describe Zone Aware Routing and what needs to be considered to utilize it well.

What Is Zone Aware Routing?

Zone Aware Routing is a strategy designed to optimize network costs and latency by directing traffic to services within the same availability zone whenever possible. It minimizes the need for cross-zone data transfer, reducing associated costs and latencies.

Zone Aware Routing aims to send all or most of the traffic from an originating service to the upstream service over the local zone. Ideally, routing over the local zone or performing cross-zone routing should depend on the percentage of healthy instances of the upstream service in the local zone of the origin service. In addition, in case of a failure or malfunction with the upstream service in the current AZ, a Zone Aware Routing component should be able to automatically reroute requests across AZ to maintain high availability and fault tolerance.

At its core, a Zone Aware Router should act as an intelligent load balancer. While trying to maintain awareness of zone locality, it is also responsible for balancing the same number of requests per second across all upstream instances of a service. Its purpose is to avoid cases where specific instances are bombarded with traffic. Such a traffic skew can cause localized overload or underutilization, which can cause inefficiency and potentially incur additional costs.

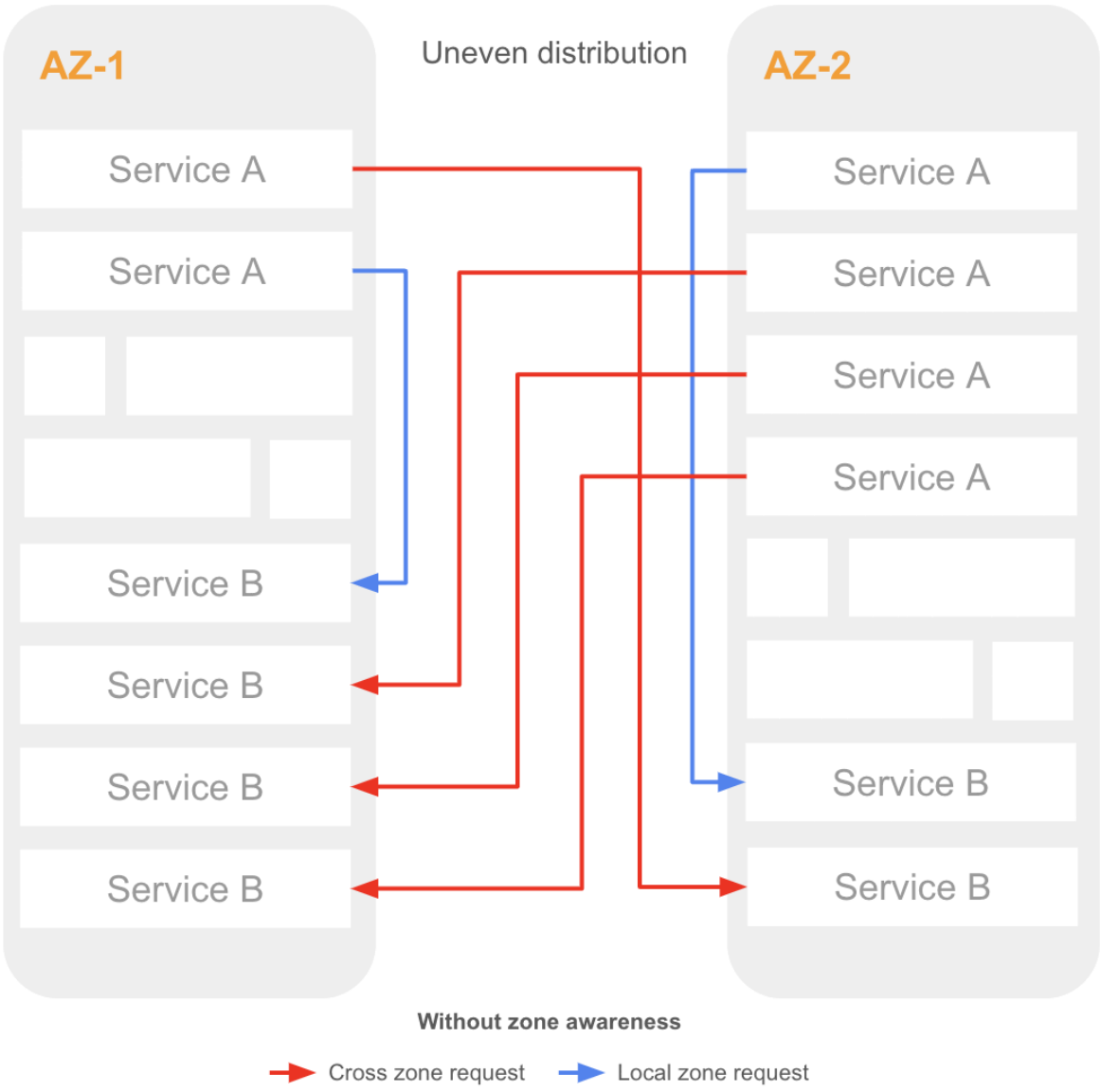


Figure 2: Illustration of uneven zonal distribution. Two services across two zones. Will Service B hold up when it will serve all requests coming from AZ-2?

Due to that nature, Zone Aware Routing can be very beneficial in environments with evenly distributed zonal traffic and resources (for example, when you distribute your traffic 50/50 over two AZs with the same amount of resources). However, for uneven zonal distributions, it could become less effective due to the need to balance the traffic across multiple zones and overcome hotspots (bombarded and overloaded instances—see figure 2).

How Can I Adopt Zone Aware Routing?

The concept of Zone Aware Routing has various implementations. So, adopting it depends on finding the implementations that fit your setup and tech stack. Let's review some of the options:

Istio Locality Load Balancing

Istio is an open-source service mesh platform for Kubernetes. Istio probably has the most promising

Zone Aware Routing implementation out there: Locality Load Balancing. It is a feature in Istio that allows for efficient routing of requests to the closest available instance of a service based on the locality of the service and the upstream service instances. When Istio is installed and configured in a Kubernetes cluster, locality can be defined based on geographical region, availability zone, and sub-zone factors.

Istio's Locality Load Balancing enables requests to be preferentially routed to an upstream service instance in the same locality (i.e., zone) as the originating service. If no healthy instances of a service are available in the same locality as the origin, Istio can failover and reroute the requests to instances in the nearest available locality (e.g., another zone and even another region). This ensures high availability and fault tolerance.

In addition, Istio allows you to define weights for different localities, enabling you to control the distribution of traffic across AZs based on factors such as capacity or priority. This makes it easier to overcome uneven zonal distributions by letting you adjust the amount of traffic you keep localized in each zone vs. the amount of traffic you send across zones.

Topology Aware Routing (AKA Topology Aware Hints)

If you work with Kubernetes without Istio, Topology Aware Routing is a native feature of Kubernetes, introduced in version 1.17 (as topology aware hints). While somewhat simpler than what Istio offers, it allows the Kubernetes scheduler to make intelligent routing decisions based on the cluster's topology. It considers the topology information, such as the node's geographical location (e.g., region) or the availability zone, to optimize the placement of pods and the routing of traffic.

Kubernetes also provides some essential safeguards around this feature. The Kubernetes control plane will apply those safeguard rules before using the feature. Kubernetes won't localize the request if these rules don't check out. Instead, it will select an instance regardless of the zone or region to maintain high availability and avoid overloading a specific zone or instance. The rules can evaluate whether every zone has available instances of a service or if it is indeed possible to achieve balanced allocation across zones, preventing the creation of uneven traffic distribution on the cluster.

While Topology Aware Routing enables optimizing cross-zone traffic and handling the hidden burden of performance and costs of working with multiple AZs, it is a little less capable than Istio's Locality Load Balancing. The main drawback is that it doesn't handle failovers like Istio. Instead, its safeguards approach shuts down the zone locality entirely for the service, which is a more rigid design choice requiring services to spread evenly across zones to benefit from those rules.

End-to-End Zone Aware Routing

Using Istio's Locality Load Balancing or Kubernetes Topology Aware Routing to maintain Zone Aware Routing is a significant step forward in fighting our distributed systems' data transfer costs and performance bottlenecks. However, to complete the adoption of Zone Aware Routing end-to-end and bringing cross-AZ data transfer to a minimum, we also have to figure out whether our services are capable of reading their data from DBs and Message Queues (MQ) over the local zone (see figure 3).

DBs and MQs are typically distributed across AZs to achieve high availability, maintaining copies of each piece of data in every AZ. This is why DBs and MQs are prone to being accessed cross zones, and as such, exposing the system to the performance and cost burden. So, can we optimize our DB reads latencies and reduce data transfer costs by accessing their data over the local zone without compromising resilience?

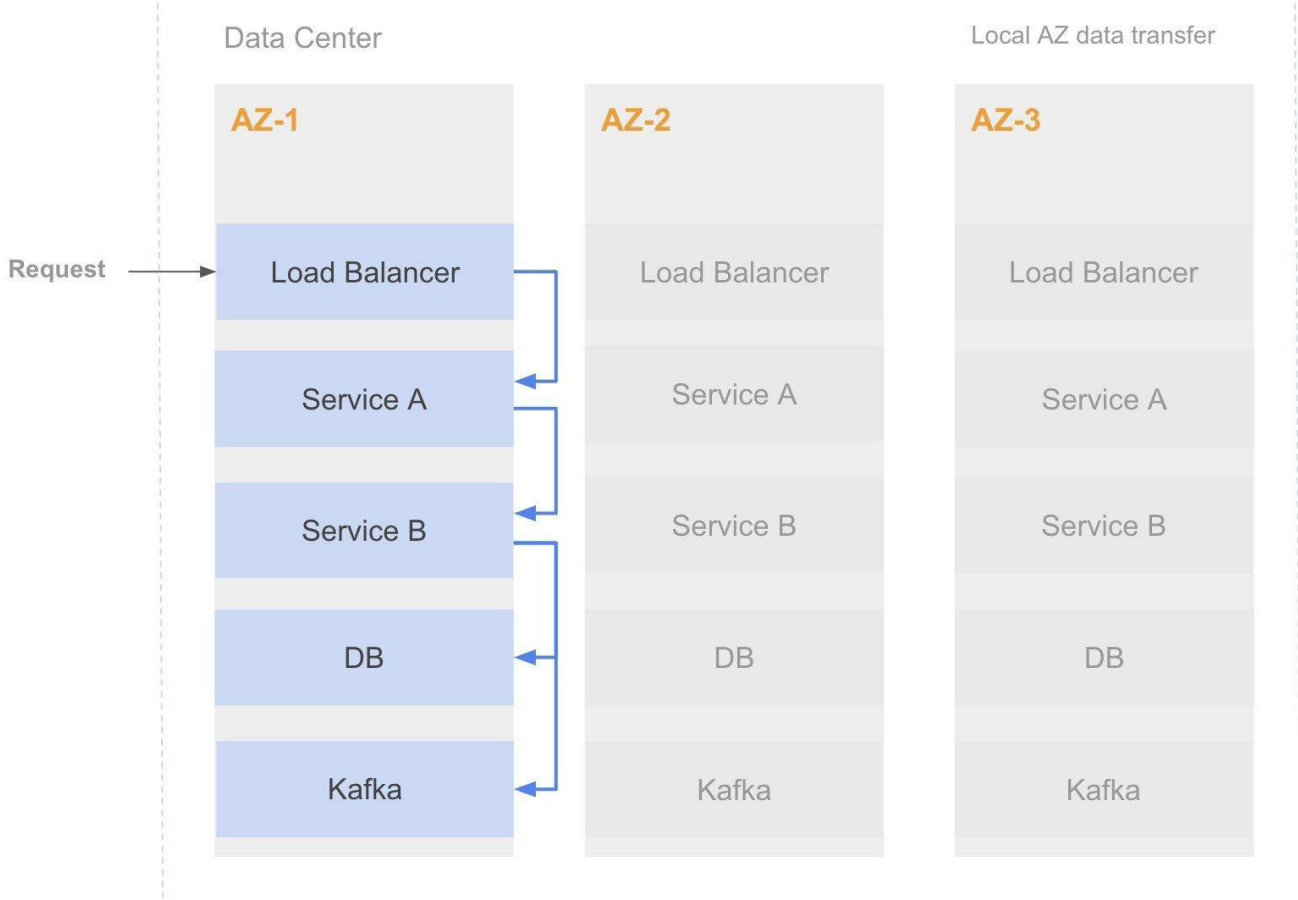


Figure 3: Illustration of a request flow localized on a single AZ end-to-end

Here are some examples of DBs and MQs that can support Zone Aware Routing:

Kafka's Follower Fetching feature

Apache Kafka is an open-source distributed event streaming platform for high-performance data pipelines, streaming analytics, and data integration. Like other MQs, it is typically used to connect between microservices and move data across distributed systems.

Follower Fetching is a feature of the Kafka client library that allows consumers to prefer reading data over the local availability zone, whether from the leader or replica node. This capability is based on Kafka's Rack Awareness feature, which is designed to enhance data reliability and availability by leveraging the physical or logical topology of the data center.

Kafka's Rack Awareness requires each broker within the cluster to be assigned its corresponding rack information (e.g., its availability zone ID) so that it will be possible to ensure that replicas of a topic's partition are distributed across different AZs to minimize the risk of data loss or service disruption in case of an AZ or node failure. While rack awareness doesn't directly control the locality of client reads, it does enable the Kafka broker to provide its location information via the rack tag so that the client libraries can use this to attempt to read from replicas located in the same rack or zone, or either fallback to another zone if the local broker isn't available. Thus, it follows the concept of Zone

Aware Routing, which optimizes data transfer costs and latency. To utilize follower fetching, we must assign the consumer the rack tag, indicating which AZ it is currently running from so the client library can use it to locate a broker on that same AZ.

Notice that when using Kafka's follower fetching feature and a local broker's replication lags behind the leader, it will increase the consumer's waiting time within that local zone instead of impacting random consumers across zones, thus limiting the blast radius of some issues to the local zone. Also, like other Zone Awareness implementations, it is very sensitive to uneven zonal distributions of consumers, which may cause certain brokers to become overloaded and require the consumers to handle failover.

Redis and other Zone Aware DBs

Redis is a popular key-value DB used in distributed systems. It is one of those DBs used in high throughput, large-scale apps for caching and other submillisecond queries. A DB that typically would be distributed across multiple AZs for redundancy reasons. As such, any geographically distributed application that reads from Redis would have a great performance and cost-benefit from reading over the local zone.

Redis does not have built-in support to automatically route read requests to the local replica based on the client's availability zone. However, we can gain this capability with certain Redis client libraries. For example, when using Lettuce (a Java lib), setting the client's "ReadFrom" setting to LOWEST_LATENCY will configure the client to read from the lowest-latency copy of the data, whether it is on a replica or the master node. This copy will usually reside in the local zone, therefore reducing the cross-zone data transfer.

If you use a client library that does not support selecting the nearest replica, you can implement custom logic that achieves the same by retrieving the data from the local Redis endpoint. And preferably fallback to a cross-AZ endpoint when needed.

There are more popular database technologies that support Zone Aware Routing; here are two that are worth mentioning:

Aerospike—A distributed key-value database designed for high-performance data operation. It has a rack awareness feature, which, like Kafka, provides a mechanism for database clients to prefer reading from the closest rack or zone.

Vitess—A distributed scalable version of MySQL originally developed by Youtube, enabling Zone Aware Routing with its Local Topology Service that enables it to route queries over the local zone.

While we may notice that the concept of Zone Aware Routing is named a little bit differently in every piece of technology, all of those implementations have the same goal —to improve read latencies and data transfer costs while not sacrificing service reliability.

Another cost burden of distributing our DBs and MQs across multiple availability zones is that every piece of data we write needs to be replicated within the DB cluster itself to every zone. The replication, by itself, can cause a significant amount of data transfer cost. While technically there is nothing to do about it, it is essential to note that usually managed DB services such as AWS MSK (Amazon's Managed Streaming for Apache Kafka), AWS RDS (Amazon Relational Database Service), Elasticache (Amazon's managed Redis), and more, aren't charging their users for cross AZ data transfer within the cluster but only for data getting in and out of the cluster. So, choosing a managed service can become a significant cost consideration if you plan to replicate large amounts of data.

Handling Hotsp/ots and Uneven Service Distribution

While it is highly suggested that we maintain even zonal distributions of our services, it is not always as trivial or possible. Handling hotspots is a complex topic, but one way to address it would be to consider a separate deployment and a separate auto-scaling policy for each zone. Allowing a service to independently scale based on the traffic within a specific zone can ensure that if the load is concentrated in a specific zone, it will be adequately scaled to handle that load with dedicated resources, eliminating the need to fall back to other zones.

Conclusions

Fighting data transfer costs and performance in highly available distributed systems is a real challenge. To handle it, we first need to figure out which microservices are prone to frequent cross-AZ data transfer. We need to be data-driven about it, measure cross-AZ costs, and trace latencies of microservices communication to ensure we put our efforts in the right direction.

Then, we must adopt the proper tooling to eliminate or minimize those cross-AZ calls. Those tools and optimizations depend on our technology stack. To adopt Zone Aware Routing end-to-end, we must adopt different tools for different use cases. Some tools, like Istio and Topology Aware Routing rules, are meant to unlock zone awareness for Kubernetes pods communicating with each other. However, they won't apply to cases where your microservices consume data across AZ from Databases or Message Queues. So, we must choose the right DBs and MQs with zone-aware routing baked in their client libs.

Finally, we must ensure we do not break our system's high availability and resilience by deploying those tools and optimizations in an uneven zonal distribution setup that causes hotspots (instances bombarded with traffic). Such deployment may defeat the purpose, causing us to fix our cost and performance problem by creating a new one.

Overall, optimizing cross-AZ traffic is crucial; it can affect our applications' performance, latency, and cost, but it has to be handled carefully to ensure that we do not sacrifice our app's reliability and high availability.

Legacy Modernization: Architecting Real-Time Systems Around a Mainframe

by **Jason Roberts**, Lead Software Consultant @Thoughtworks
Sonia Mathew, Sr Director Technical Area Lead @Boston Consulting Group

Introduction: "Where Did Half the Room Go?"

This article is based on our talk at QCon San Francisco in November 2024. It was mid-year in 2024, and our team was in a PI planning session with about 50 or 60 people, including our primary business stakeholders. In the middle of that session, half the room got up and left. The billing mainframe, which drives the customer web portal and many other applications, had gone down, causing outages.

The half that remained in the room was primarily composed of our team. We were about halfway completed with an incremental multi-year program to replace that customer web portal. Thanks to edge routing, the application dynamically routed users to new pages as we developed and released them. Those pages were still being served, despite the fact that the system of record for it was down, thanks to our cloud-based streaming architecture.

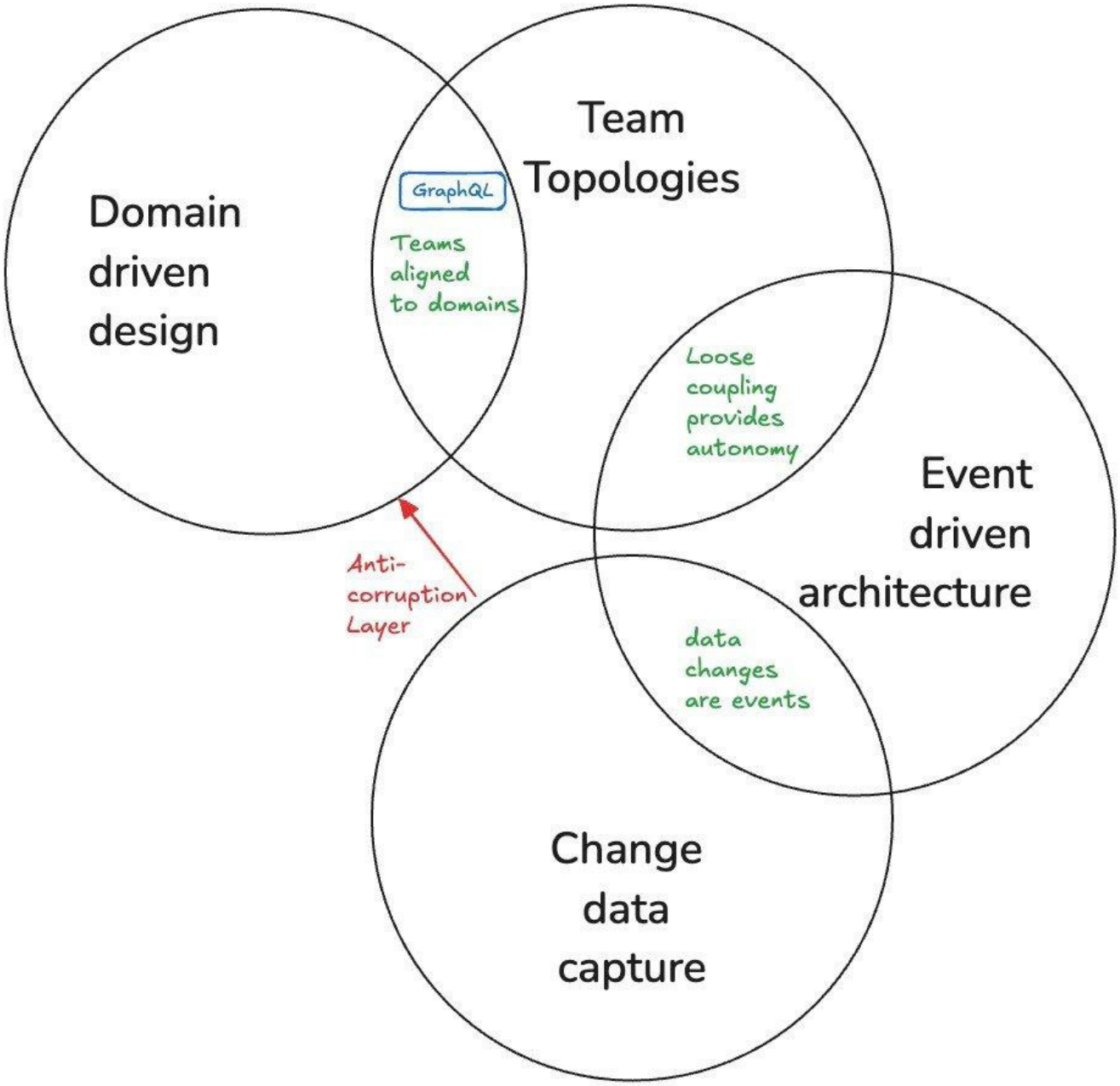
The customer self-service portal provides crucial functionality to customers such as viewing bills, making payments, starting or stopping utility services, and monitoring energy usage patterns. The system we were replacing failed to deliver

a reliable experience, but the new system had just proven itself to be resilient. This is the story of how we transformed our architecture through a comprehensive decoupling strategy spanning technical systems, organizational structures, and semantic understanding.

This Is a Story About Decoupling

At its heart, our transformation journey is about breaking dependencies at multiple levels. Many enterprises face similar challenges with legacy systems: tightly coupled architectures that are difficult to scale, change, or maintain. For us at National Grid, the solution came through four complementary paradigms that worked together to enable different forms of decoupling:

1. **Domain-Driven Design (DDD)**: A software development methodology focusing on modeling complex systems around business



domains. DDD gave us tools to reshape legacy mainframe data into business-friendly concepts that were easier to understand and evolve. By creating a common language between technical and business stakeholders, DDD enabled semantic decoupling from legacy constraints.

2. **Team Topologies**: An organizational design approach that structures teams according to different types and interaction patterns. For us, this meant aligning teams to business value streams rather than technical layers, creating stream-aligned teams that could own entire business capabilities, and complicated subsystem teams that could handle the mainframe integration complexity. This organizational decoupling reduced coordination overhead and accelerated delivery.
3. **Event-Driven Architecture**: A system design pattern using asynchronous events as the primary means of communication between components. Unlike synchronous APIs with direct request-response patterns, events provide a stronger abstraction that allows for multiple or even no responses. This architectural style shifted our system from strong consistency to eventual consistency, a deliberate tradeoff that enabled technical decoupling between front-end and back-end systems.
4. **Change Data Capture (CDC)**: A technique that identifies and captures changes in a database, allowing those changes to be propagated to other systems. For us, CDC formed the foundation of our new architecture, enabling the creation of a "system-of-reference" that could serve applications without direct mainframe dependency.

These four paradigms, while powerful individually, created an even stronger solution when applied together. Domain-Driven Design shaped how we modeled the business. Team Topologies aligned our organizational structure with those domains. Event-Driven Architecture provided the technical patterns to decouple components. Change Data Capture created the bridge from legacy systems to modern, domain-aligned data stores.

Where We Started: The Unified Web Portal 1.0

National Grid is a regional utility provider delivering both gas and electric service. Over years of mergers and acquisitions, we had accumulated multiple systems-of-record for billing and customer information. The resulting customer experience was fragmented - customers with both gas and electric service often needed to use different portals for each utility type.

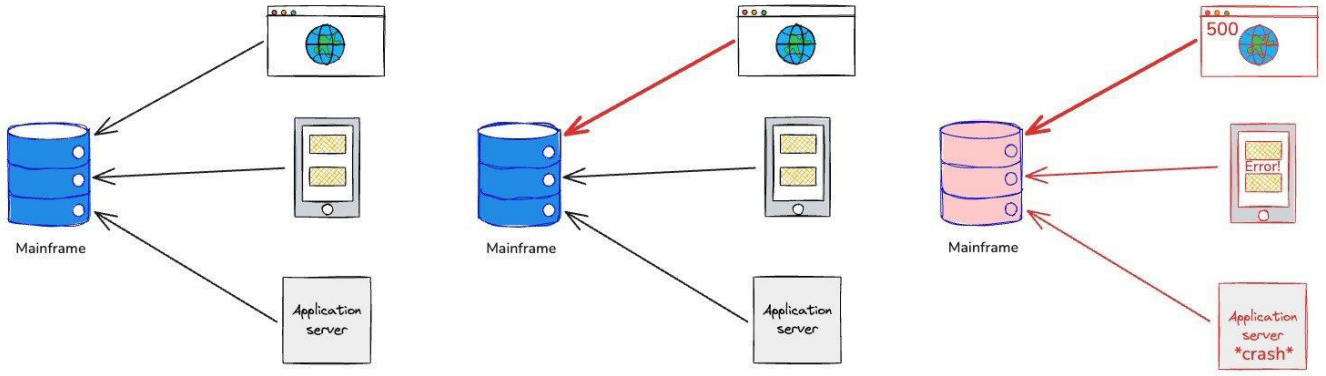
The first version of the Unified Web Portal (UWP 1.0) attempted to solve this fragmentation by consolidating data from multiple mainframes into a single customer experience. The technical approach was straightforward:

1. Extract data from multiple mainframe systems using ETL (Extract, Transform, Load) processes
2. Transform this data into a unified data model
3. Load it into a SQL database
4. Finally, move it into a SaaS platform with its own datastore

However, this approach had significant limitations:

- Our ETL processes ran in batches only a few times per day, creating data freshness issues
- Combining data from multiple sources led to data quality problems
- For critical operations where real-time data was essential, applications had to make synchronous calls directly to the mainframe, bypassing the unified model

ETLs are perfectly fine if you're looking at analytical data. However, in our case, we were dealing with operational data, and the batch nature of ETL was ill-suited to give customers the most recent information about their accounts. Worse still the synchronous connection to the mainframe was



an inelastic resource serving an inherently elastic system, a web site.

Why That Didn't Work: The Emergent Problems

The UWP 1.0 architecture created multiple interconnected problems that compounded over time:

Technical Problems

1. **Data Currency Issues**: With ETL batch processes running only a few times daily, customers would see outdated information. A payment made in the morning might not appear on their account until evening, creating confusion and support calls.
2. **Synchronous Mainframe Dependencies**: For critical operations requiring current data, applications needed direct mainframe access. This created a scenario where a highly elastic web application was tightly coupled to an inelastic mainframe system.
3. **Backend-for-Frontend Proliferation**: To handle different use cases with current data, we created dozens of Backend-for-Frontend (BFF) APIs, each tightly coupled to specific frontend needs and mainframe endpoints. These multiplied over time, creating a maintenance nightmare.
4. **Distributed Transaction Complexity**: Our API integration platform used synchronous calls to achieve distributed transactions across multiple systems, increasing coupling and failure points.

Organizational Problems

That technical architecture was shaped by the organizational structure of the business' technology groups (Conway's Law):

- Mainframe Architects managing the core legacy systems
- SQL Database Administrators handling the intermediate datastore
- Integration Engineers creating and maintaining the API platform and BFFs
- Web Developers building the frontend experience using a SaaS web platform

This led to a change and release process that required high coordination effort to release even minor features, requiring siloed deliverables across multiple teams with different priorities and release cycles.

Cascading Failures

The most dramatic consequence of this architecture was the potential for cascading failures due to inelasticity.

When traffic spikes hit our web portal, those requests would flow through to the mainframe. Unlike cloud systems, mainframes can't elastically scale to handle sudden load increases. This created a bottleneck that could overload the mainframe, causing connection timeouts. As timeouts increased, the mainframe would crash, leading to

complete service outages with a large blast radius, hundreds of other applications which depend on the mainframe would also be impacted.

This is a perfect example of the problems with synchronous connections to the mainframes. When the mainframes could be overwhelmed by a highly elastic resource like the web, the result could be failure in datastores, and sometimes that failure could result in all consuming applications failing.

Setting Our New Goals: The Path Forward

Understanding these challenges, we had clear objectives for our architectural transformation:

Technical Goals

1. **Decoupling**: Separate the frontend and web services from direct mainframe dependencies
2. **Reducing Dependencies**: Eliminate brittle point-to-point integrations, especially the proliferation of BFFs
3. **Empowered Engineering**: Create a structure where teams could own and deliver end-to-end slices of functionality

Business Goals

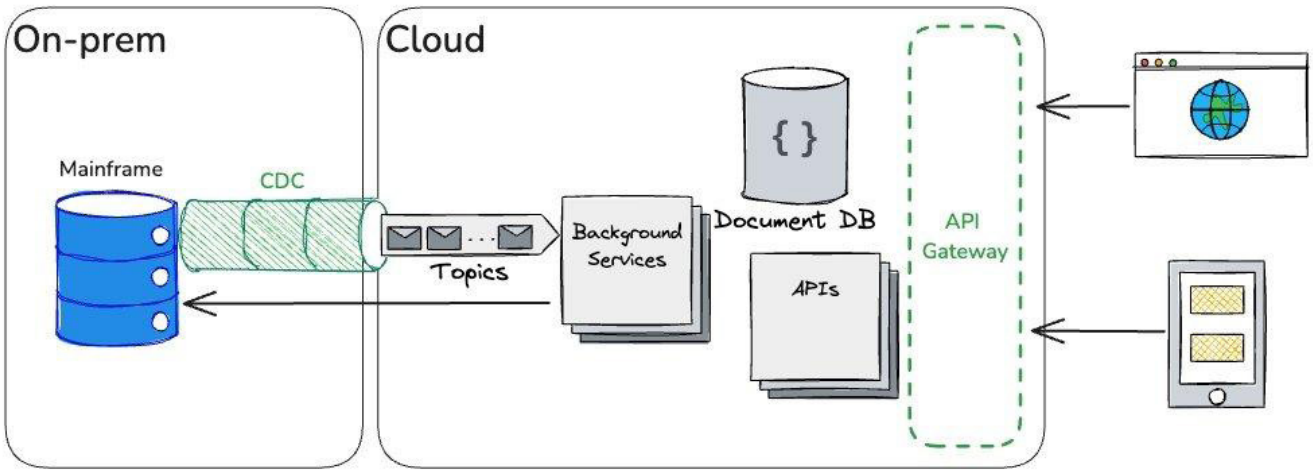
1. **Reduced Call Center Volumes**: Decrease support calls resulting from portal outages or stale data
2. **Lower Licensing Costs**: Eliminate expensive third-party middleware and integration platforms
3. **Improved Customer Satisfaction**: Provide a more reliable experience with fresher data

Our plan required a fundamental rethinking of both the architecture and team structure - not just technical changes but also a shift in how people collaborated and what they owned.

Our New Architecture: A Bird's Eye View

Our new architecture represented a complete departure from the previous approach:

- **On-premises**: The mainframe remained as the system-of-record with CDC configured to capture changes
- **Cloud**: A modern, cloud-native architecture with several key components:
- Streaming CDC events into Event Hubs (Kafka)
- Background services that processed events, both upstream and downstream
- Document databases (Cosmos DB for MongoDB) storing the processed domain entities
- Public APIs (GraphQL and REST) exposing these entities through an API gateway



- Web, mobile, and other applications consuming these APIs

The specific technology here isn't necessarily important. It's the strategies and patterns that we use. You could replace Azure with AWS. We used widely adopted standards wherever we could, further increasing decoupling and reducing lock-in.

This architecture created a clear separation between our legacy systems and modern applications, with change data capture serving as the engine and event-driven architecture as the highway.

Step One: Change Data Capture – Creating a System-of-Reference

Change Data Capture became the foundation of our new architecture. Instead of batch ETLs running a few times daily, CDC streamed data changes from the mainframes in near real-time. This created what we called a "system-of-reference" - not the authoritative source of truth (the mainframe remains "system-of-record"), but a continuously updated reflection of it. The system of reference is not a proxy of the system of record, which is why our website was still live when the mainframe went down.

Additionally, this system of reference was not constrained by the schema, structure or semantics of the mainframe. We modeled this system using domain-driven design, allowing the API to evolve at its own pace.

Our data flow worked like this:

1. Mainframe data changes were captured through CDC
2. These changes were published to Kafka topics
3. Data processors formed an anti-corruption layer, transforming mainframe concepts to domain concepts
4. The transformed data populated domain-specific entity databases
5. APIs exposed these entities to applications

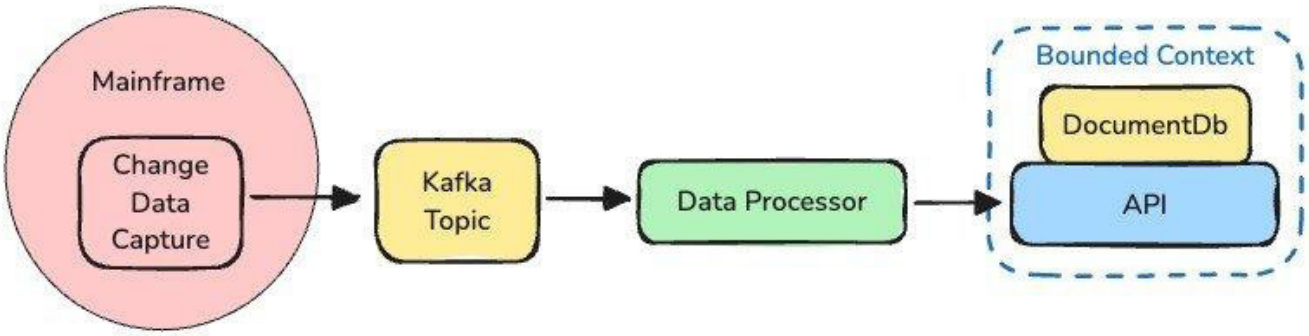
This approach allowed us to decouple the frontend experience from the constraints of the mainframe. Applications could now access data in business-friendly formats without being coupled to mainframe semantics.

Scaling Our System-of-Reference

Our system needed to handle massive throughput - processing several million events daily. This required a horizontal scaling strategy with multiple components:

We implemented:

- Kafka topics with multiple partitions to distribute event processing and streaming
- Kubernetes for horizontal scaling of data processors and APIs
- Document databases optimized for the API usage patterns (both internal and public)



We can scale the topics via partitions to accommodate any kind of change volume that we have, and the same with the data processors. We've proven this works at the scale we require. Our application was not only rolled out incrementally, but our user base was also scaled up incrementally. We started with dozens of users being routed to the new experience. Then hundreds, then thousands, then hundreds of thousands, and eventually millions. This occurred over a year and a half, with releases every two weeks, and we never once had a scaling problem.

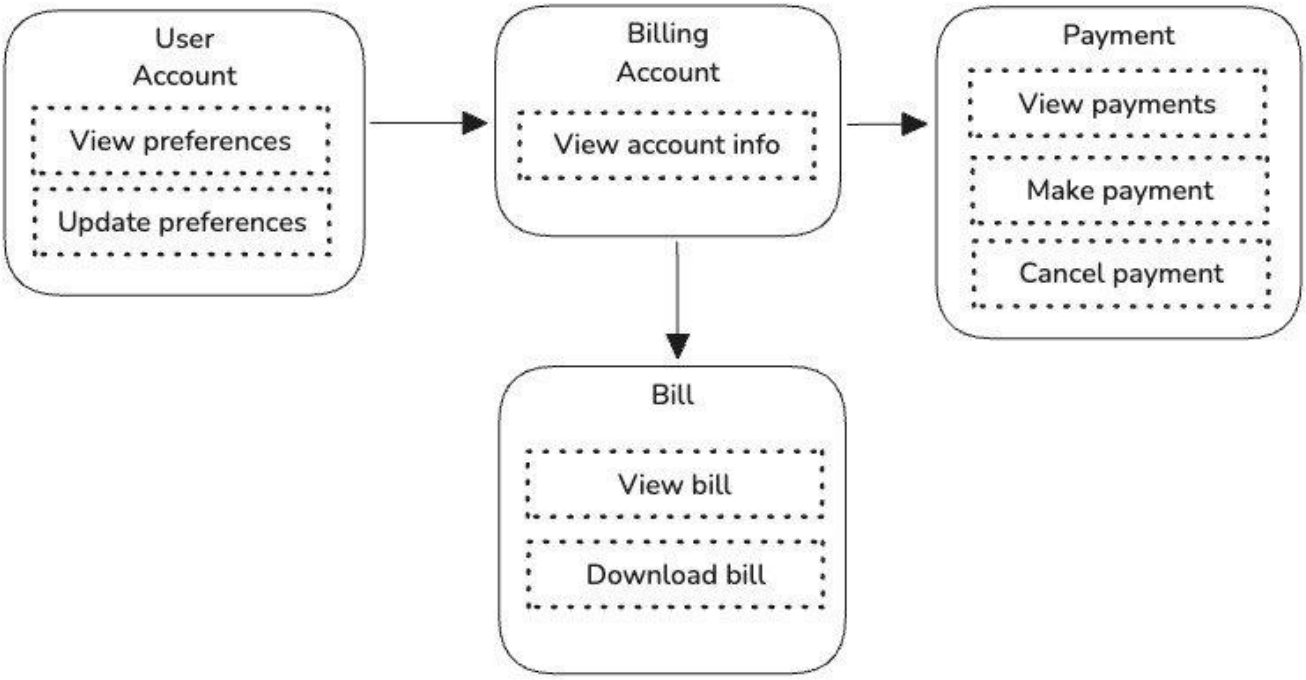
Step Two: Domain-Driven Design with GraphQL

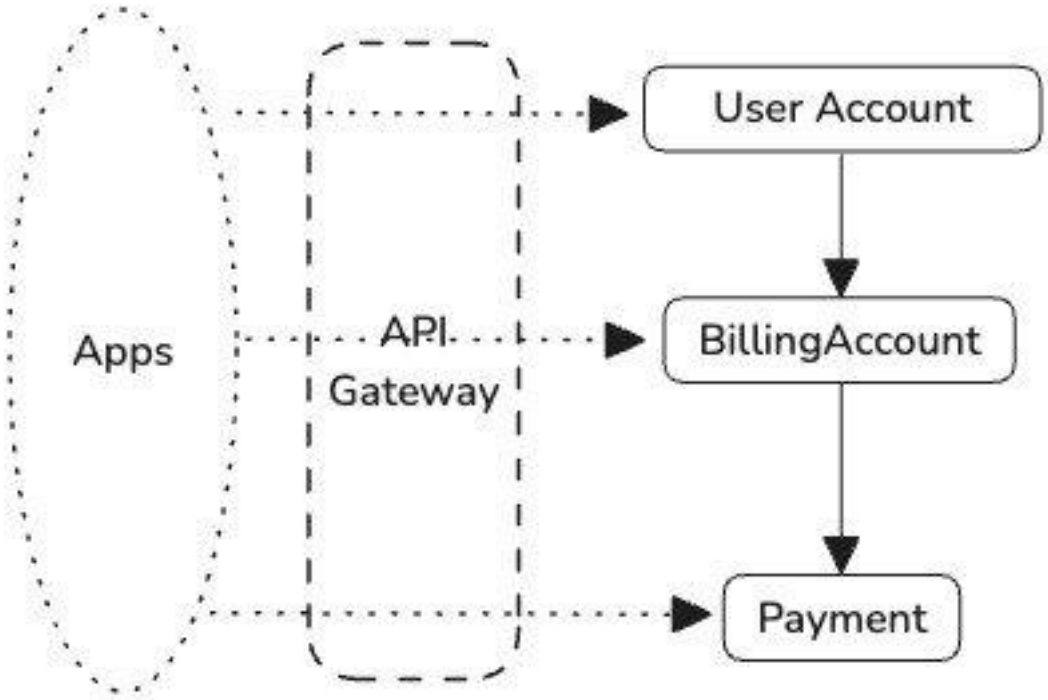
With Change Data Capture providing the data pipeline, we needed to model this data effectively to create an API that was wide enough to power many applications, but avoid APIs that were tailored to the specifics of the customer portal. We turned to Domain-Driven Design (DDD), starting with event-storming workshops, to identify:

- Bounded Contexts: solution spaces that tie together related data (Entities) and operations (Commands)
- Entities: objects defined by continuity and identity (e.g., Billing Account, Payment)
- Commands: operations that change state (e.g., Submit Payment)
- Events: things that happen as a result of commands (e.g., Payment Submitted) or when our reference data changes (via CDC)

Importantly, we modeled these domains based on business needs first, independent of mainframe constraints. Only after establishing the model did we map it back to the mainframe data.

Our next challenge was to create a productized API that could be scalable and maintainable.





GraphQL: Avoiding Backend-for-Frontend Proliferation

To prevent recreating the dozens of Backend-for-Frontend (BFF) APIs that UWP 1.0 used, we chose GraphQL. Each bounded context acted as a node in a GraphQL graph, and like all graphs, the edges defined the relationships between those nodes.

We used schema stitching to compose these domain nodes into a supergraph. This allowed applications to write queries that traversed multiple domains without knowing the underlying complexity.

This approach offered us several advantages:

- **Preventing over-fetching**: Frontend developers could request only the specific fields they needed
- **Eliminating under-fetching**: Complex nested queries could fetch related data across domains in a single request
- **Flexible traversal**: Queries could start from any node in the graph, allowing for query optimization

Team Topologies – Organizational Design is Architecture

We know that organizational structure has a major impact on architecture. Execution of a successful software delivery is as much about scaling teams as it is about scaling hardware and software. Without the ability to scale teams, you are limited in how fast you can deliver.

With Team Topologies, we restructured from highly interdependent technology-centric teams to autonomous, empowered, domain-oriented teams. Team Topologies suggests several ‘shapes’ for teams, what we used was:

1. **Stream-aligned teams**: Focused on specific business functionality like Payments, Move Service, or Billing. These teams owned entire features end-to-end within the scope of their bounded contexts.

2. **Enablement teams**: Provided technical foundations like observability frameworks and DevOps tooling to support other teams.
3. **Complicated subsystem teams**: Focused on complex technical areas like mainframe integration. Our Enterprise Integration Team handled the complexity of data streaming out of the mainframe, as well as state-changing operations to the mainframe. Stream-aligned teams interact with integration services asynchronously, allowing them to focus on customer value.

This structure established team boundaries as system boundaries, giving each team full ownership.

Event-Driven Architecture – Beyond CDC

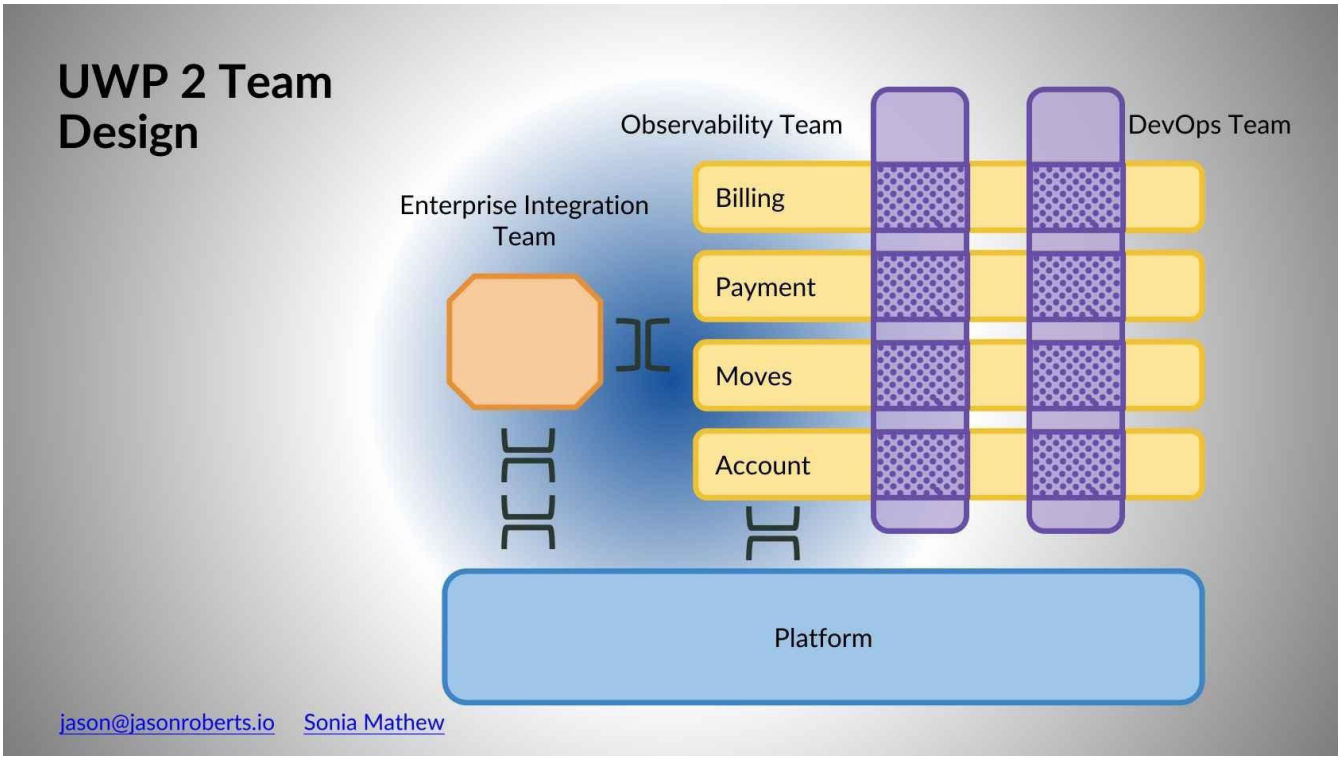
While Change Data Capture formed the foundation of our system-of-reference, we implemented additional event-driven patterns for communication within and between bounded contexts

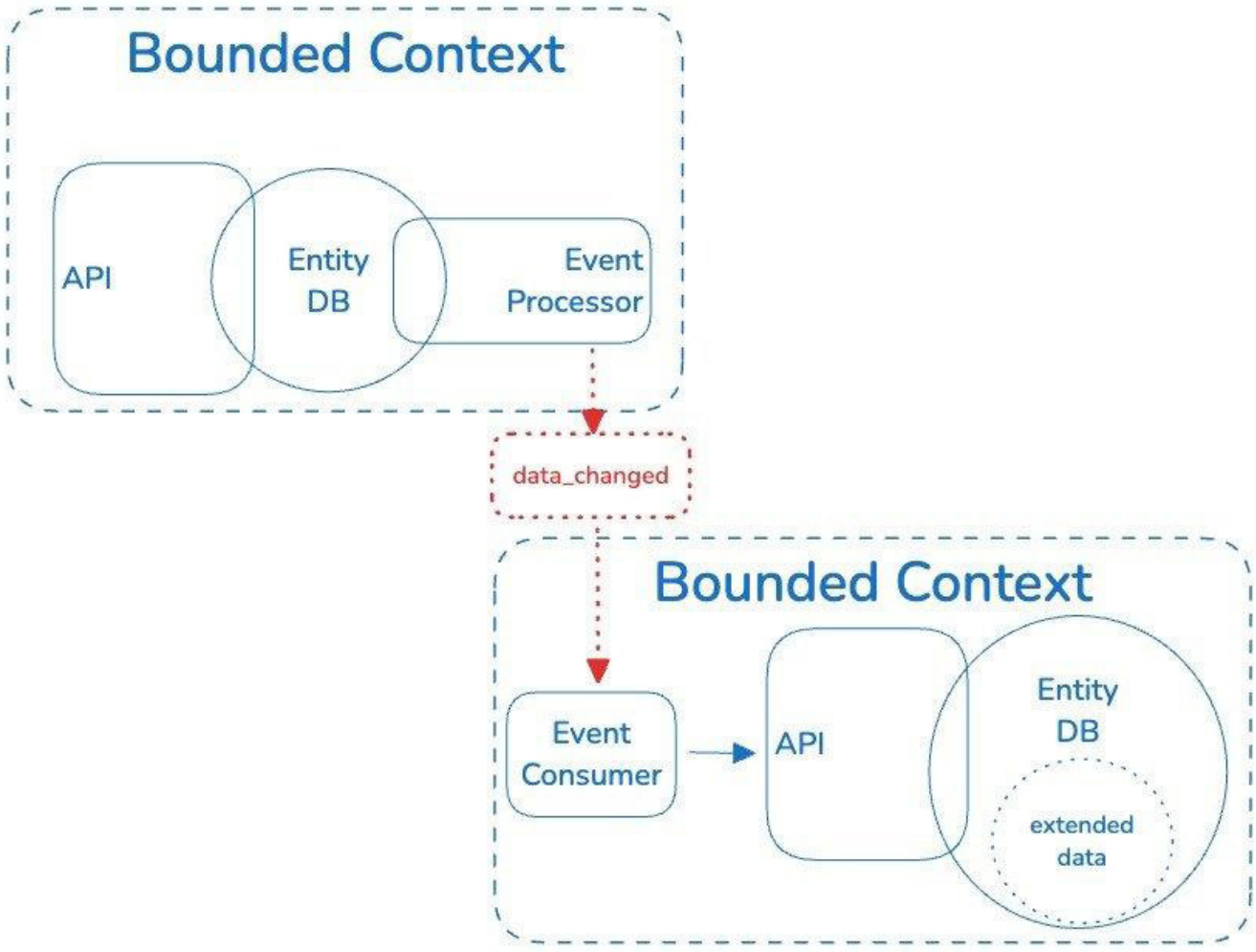
Internal Domain Events

Even within our system-of-reference, bounded contexts are needed to communicate changes. We implemented the Outbox Pattern - another form of change data capture - to publish events when data changed within a bounded context.

Some example usage scenarios include:

- **Performance optimization**: Pre-compute and cache frequently accessed data
- **Derived value calculation**: Pre-calculate complex values that would otherwise require mainframe logic
- **Cross-domain consistency**: Keep related data in sync across bounded contexts



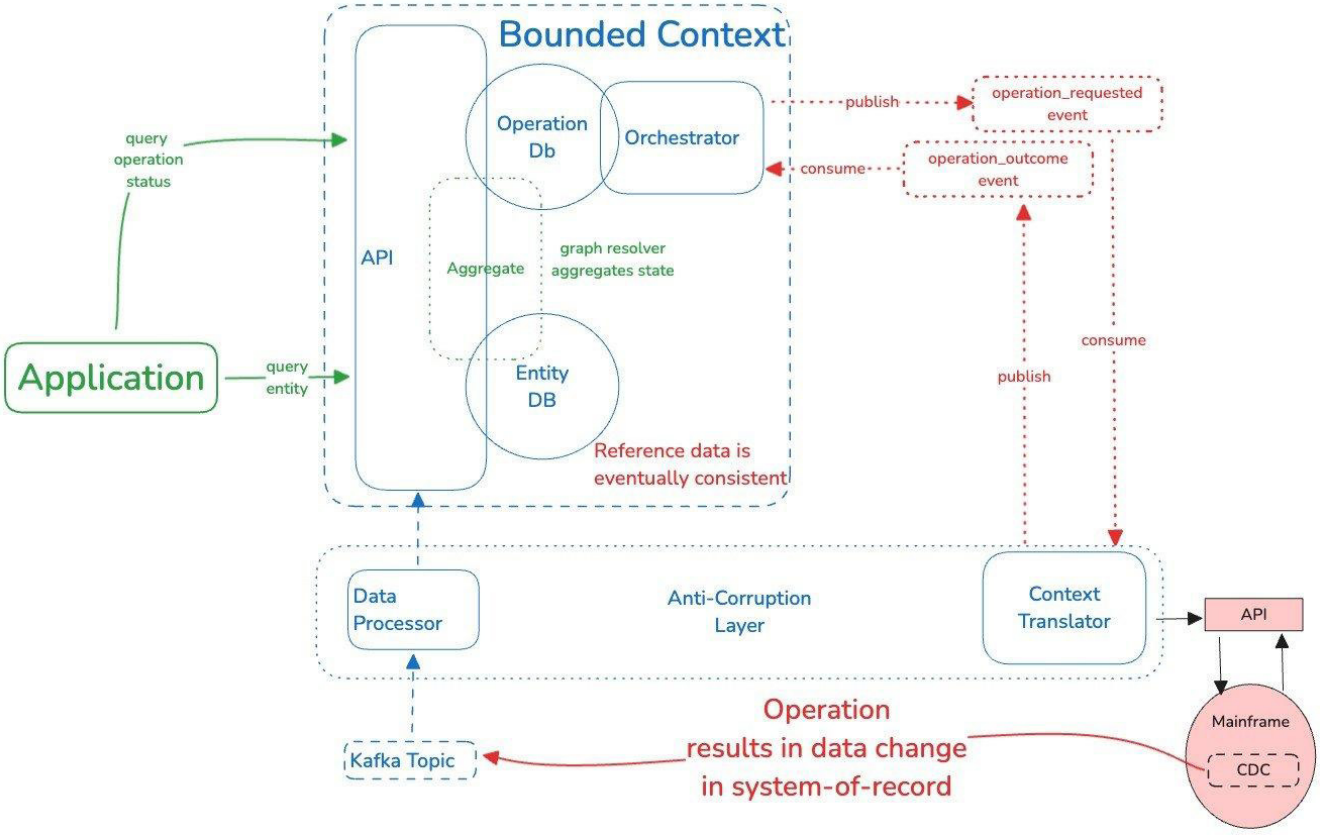


The Anti-Corruption Saga

For operations like payment submissions that required changes to the mainframe, we implemented the Parallel Saga pattern (asynchronous communication, eventual consistency, and orchestration). However, that alone was not quite enough. Our CDC-based system-of-reference needed a way to synchronize with these delegated requests. To make this work, we needed two data collections, one that would only be updated by the CDC, one that tracked the state machine of the requests, and an API that could aggregate both.

How it works:

1. A user initiates an action, such as making a payment, which marks the initial state of a state machine, which is then managed by an orchestrator.
2. The orchestrator transitions to the “Requested” state and emits an event of that state transition
3. Another component listens for that event and upon receiving it converts the domain event into the supporting mainframe operation
4. The mainframe processes the request and returns a success/failure response, and our component wraps that response in a ‘state change reaction’ event
5. The orchestrator listens for state change reactions and makes another state transition, to “succeeded” or “failed”.
6. In parallel, the CDC captures the resulting data change from the request



7. The API uses an aggregate to reconcile both signals (state machine state and CDC) to present a consistent view

This pattern handled the reality that two signals would return from the mainframe - the user-initiated operation response and the CDC-captured data change - potentially in any order. It's a race condition. The aggregate reconciles those two things through the two data stores and presents a unified state for that request to an application.

Composable Workflows

The great thing about state-machine-based workflows is that they're inherently composable. A state change can trigger a fully independent workflow, and the completion of a workflow can trigger a state change. Each sub-workflow could be tested and deployed independently while still supporting composite operations. An example: one user flow on the web portal allows a user to make a payment while simultaneously storing bank account information. The "Make a Payment and Add Bank Account" workflow is simply first running "Add Bank Account" and then running "Make a Payment".

Challenges and Tradeoffs We Faced

Our transformation wasn't without difficulties. We encountered several challenges:

Event-Driven Complexity

Event-driven architecture is hard. People don't understand it. It takes a paradigm shift most of the time. Moving from request-response thinking to event-based patterns required significant education and mindset changes within our teams.

Observability Requirements

With asynchronous processing and eventual consistency, traditional debugging approaches became insufficient. We had to implement comprehensive observability from the start, with correlated logs, metrics, and traces across services and workflows.

Mainframe Batch Processes
Large mainframe batch jobs (like monthly billing) could flood our CDC pipeline with events. This causes data latency. Worse still, if I happen to have an orchestrator that's waiting on one of those statuses to change, it's now backed up because of the lag created by the batch. This was an early bug we identified, and soon realized we needed to handle our CDC and Saga topics differently (hence the Anti-Corruption Saga).

GraphQL Schema Management
With schema stitching, we ended up with a shared supergraph schema that all nodes referenced. It's not fun to maintain. It also more tightly couples the nodes in the graph to each other. We came up with strategies to work around that, such as versioning specific iterations of the supergraph, but that was a bandaid.

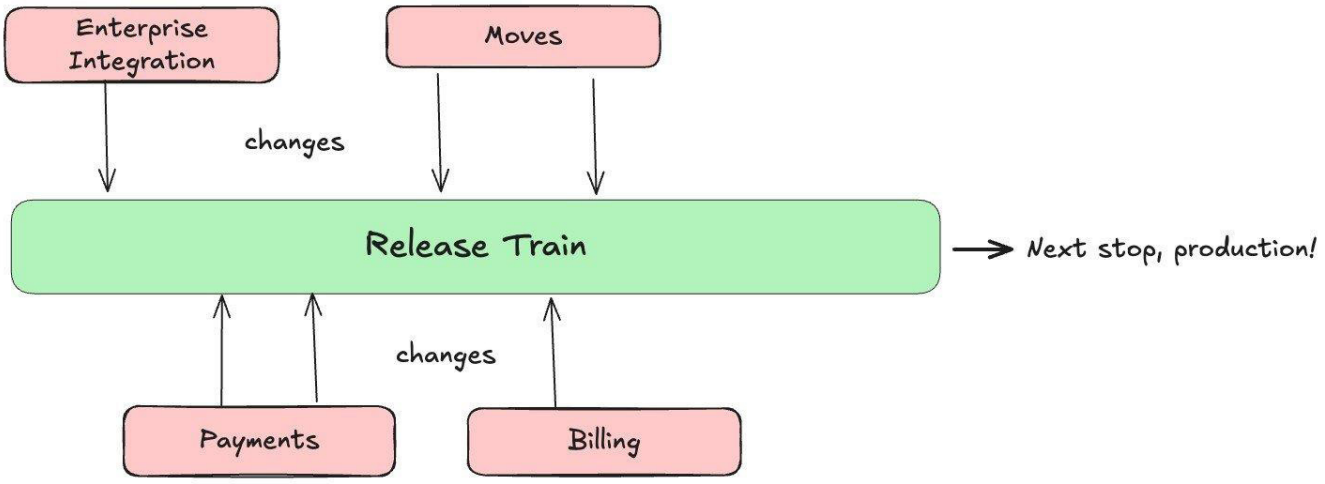
An alternative would have been to use a federated composition, which would compute schemas at runtime rather than at build time. The federated approach uses dependency inversion. Instead of a node having knowledge of its outgoing relationships, nodes define *what they extend*. This breaks the potentially inexhaustible chain of transitive dependencies. It does require a supergraph gateway to do that runtime composition, which is why we shied away from starting with federation. However, having experienced the challenges of maintaining a single graph schema, it may have been the better choice.

Cross-Cutting Concerns
While team boundaries aligned with domains, some concerns cut across all teams - authentication, observability, and error handling, for instance. We also chose a multi-repo approach, rather than a monorepo. This made things a bit difficult, but we addressed it through inner sourcing and automation:

- Private libraries provide foundational capabilities like API patterns, network communication, persistence, service hosting, distributed tracing, and more
- Those libraries are published as semantically versioned packages, and a weekly "patch Tuesday" job publishes them
- We used Dependabot to automate the dependency updates on the consuming services' repositories.

Our Release Strategy - All aboard the Release Train!

Organizationally, we follow an Agile Release Train process. As mentioned, we also have autonomous teams who own multiple repos. Each of those repos has its own CI/CD pipelines and produces an



artifact (a Docker image). To coordinate releasing those artifacts in a single release train, we created a release train automated pipeline.

The "train station" in this analogy is merely another repository, our deployment repository. Our Kubernetes manifests live here, which describe the shape of our cluster. When a CI/CD job would run on one of the application repos, after the Docker image was published, that same job would update a Kustomize file in the deployment repo, incorporating the latest change into the full system structure. The Kustomize file changes would then get promoted to higher environments. First, one-by-one to QA, when stories were ready for QA. Then all at once, when "the train leaves the station" (i.e., at the end of a sprint/start of UAT).

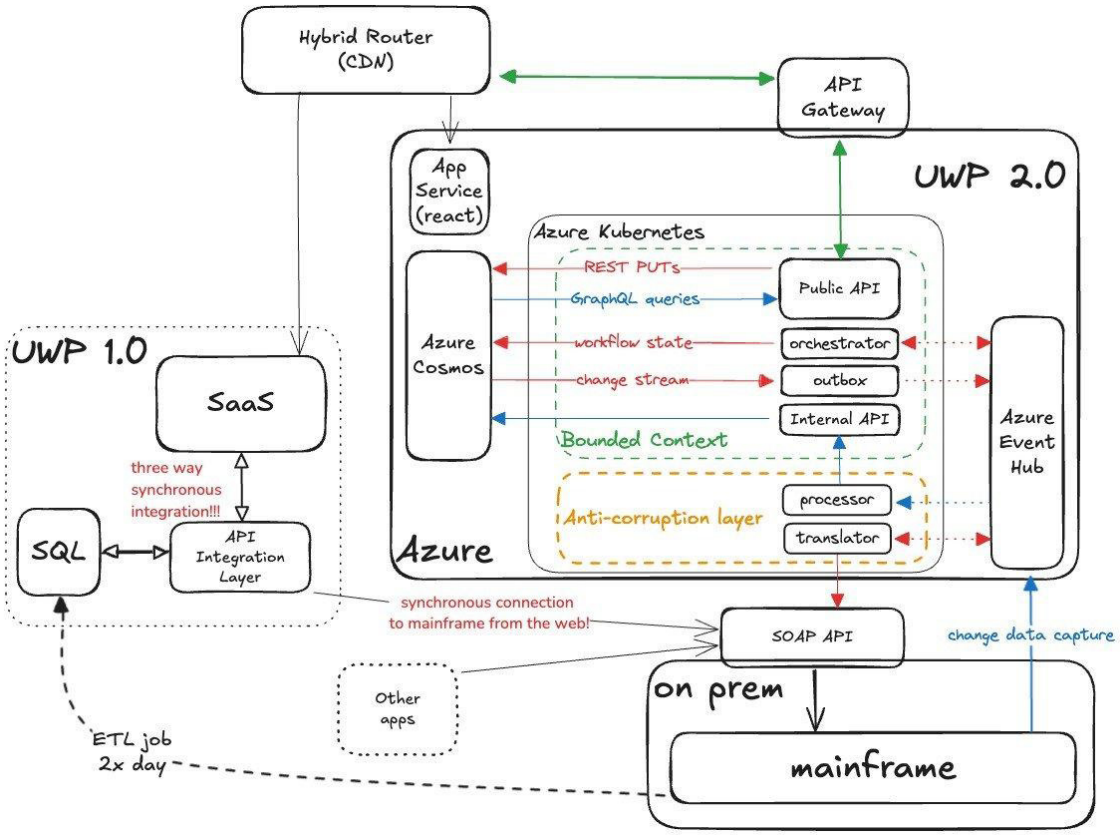
The final piece of that puzzle is feature flags. Feature flags enabled us to release security updates and bug fixes to services, but hold back feature work in progress. Because of all this, we were able to use trunk-based development across the board, avoiding potentially messy and painful merge scenarios.

Overall, this greatly reduced coordination cost (of 5 teams and 30+ repositories) for releases, as well as the support needed to release. Which in turn increased our velocity, stability, and predictability.

The "Hybrid Architecture"

To avoid a risky "big bang" cutover, we implemented a gradual transition from UWP 1.0 to UWP 2.0:

- Edge routing to direct traffic between old and new implementations (Hybrid Router)
- Context awareness between systems for a seamless user experience



This approach allowed us to incrementally replace features while maintaining a consistent user experience, reducing risk, and enabling continuous delivery of value.

Putting all that together, this is the big picture:

Conclusion – Building a Platform That Can Evolve

Looking back on our transformation, we were able to achieve our technical and business goals:

- We decoupled the frontend from the mainframe
- We eliminated brittle point-to-point integrations
- We empowered teams to own their domains end-to-end
- We reduced call center volumes
- We lowered licensing costs
- We improved customer satisfaction through stability

Here's what some of that looks like from a product and performance metrics perspective:

Achievements

Product Highlights

10%
Increase in web payments via UWP2.0 Portal

81%
Self-service transaction rate, increase by 12%

100%
Of eligible users can now access the site

0
P1 incidents

$789M+
One-time Payments
Scheduled
(Up from $756M in the last 90 days)

$27.5M+
Bulk Payments
Scheduled
(Up from $26.8M in the last 90 days)

>2.3M
Unique users have logged in. That's 61% of all eligible users

>95%
Reduction in the UWP production incident

Scale & Performance

200+
Services in k8s

700+
Pods in k8s

32M
Requests in k8s

6.6M
GraphQL Web Queries Per Day

3B
Document Count

350K
Daily Incoming Volume from Mainframe

45ms
Avg GraphQL response time

215ms
95 percentile GraphQL response time

5
Delivery Teams

~50
Developers

>50
Releases to production

Plus, we built a foundation that could evolve over time, serve diverse applications, and isn't reliant on a centralized system or technology. These patterns can facilitate future modernization efforts, including potential mainframe replacement using the Strangler Fig pattern.

By thoughtfully applying Change Data Capture, Domain-Driven Design, Event-Driven Architecture, and Team Topologies, we created not just a better portal, but an innovative way of building software with legacy systems - one that embraces decoupling at technical, organizational, and semantic levels and unlocking speed, scale, and sanity.

Transforming Legacy Healthcare Systems: a Journey to Cloud-Native Architecture

by **Leander Vanderbijl**, Staff Software Engineer @Livi

The main objective of this article is to explore the challenges and strategies involved in transitioning legacy healthcare systems to cloud-native architectures. Our company, Livi, is a digital healthcare service. I will walk through the challenges we faced when modernizing our Mjog patient messaging product including how we integrated with legacy Electronic Medical Records (EMRs), managed data synchronization and decided between serverless and containerized solutions.

By highlighting the lessons we learned along the way, I hope to offer insights that can help others navigate similar cloud migrations while ensuring scalability and adaptability in an ever-changing tech environment. This article is a summary of my presentation at QCon London 2024.

Livi is at the forefront of revolutionizing healthcare delivery. As a healthcare services provider, we facilitate online GP appointments, connecting patients with General Practitioners (GPs) through our innovative platform. This blend of technology and healthcare enables us to serve National Health Service patients and those seeking private consultations.

What is MJog?

A pivotal development in our journey was the acquisition of MJog, a patient relationship management system designed to enhance communication between GPs and their patients. MJog empowers healthcare providers to send everything from batch messaging and questionnaires to crucial appointment reminders.

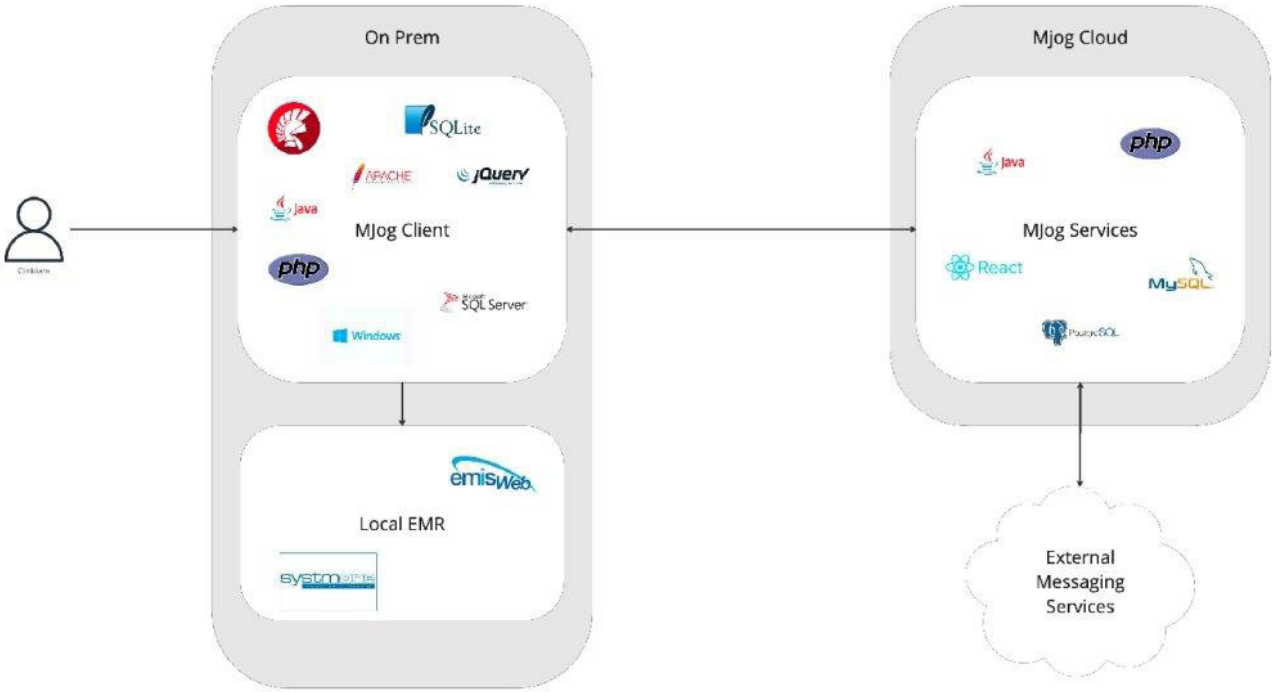
An integral feature of our application is its ability to integrate seamlessly with Electronic Medical Records (EMRs), which are essential for storing patient data. In the UK, the landscape of EMRs is dominated by two primary systems, but numerous others exist globally. Our application is designed to connect with these systems, allowing for efficient data exchange and enhancing the overall patient experience.

Transitioning from on-premises systems to cloud-based architectures poses numerous challenges, especially in industries as complex as healthcare. Our journey was no exception. As we navigated the intricacies of this transformation, several vital lessons emerged that are valuable for any team facing similar hurdles.

The migration of our legacy MJog application from on-premises to the cloud was a comprehensive, five-step process. First, we assessed the necessity for migration, identifying the strategic need to modernize our infrastructure for scalability and improved performance. Next, we redesigned our architecture to transition from the on-premise model to a cloud-optimized framework. With the architecture in place, we made critical decisions around implementing cloud-native features while balancing legacy system requirements. We then focused on observability, embedding robust monitoring tools to ensure visibility and reliability across the cloud environment. Finally, we tackled the challenge of defining clear requirements by mapping the functionality of the legacy system to its cloud counterpart, a process more complex than anticipated but essential for ensuring consistency and functionality in the new environment.

MJog Architecture

When I joined the MJog project, the system was a complex two-layer architecture with most of its logic housed on-premise at GP surgeries across the UK. The on-premise system had grown over 15-20 years into a tangled mix of outdated technologies, including Delphi, Java, PHP 5, and both SQLite and MS SQL databases. It was entirely Windows-based, with services directly accessing databases through raw SQL connections. Business logic was scattered across services, creating inefficiencies. We faced enormous technical debt.



Following COVID, GPs wanted remote access to our applications without VPNs and compatibility with non-Windows devices. Compounding this, our key Delphi developer was leaving the company, forcing us to urgently address the system's modernization.

The Necessity of Change

Early on, we recognized that evolving our application was not just an option; it was a necessity. The technology stack was outdated, and we were getting requests from all sides: our user experience was aging, our libraries needed updating, and customers were demanding cloud solutions. On top of that, we were facing a crucial deadline: some of our biggest customers had contracts up for renewal and were clear they would not stay with us unless we offered a cloud version. To complicate things further, our only Delphi developer who knew the ins

• A separate ec2 for every customer • Standard 'on-prem' installation	• 1-2 month time frame • Lose money on every installation • Doesn't solve our lack of knowledge problem
• Cloud native services for syncs and data access • Refactor the web app to work against any tenant	• 6-9 month time frame • Much cheaper • Doesn't solve our contract renewal problem

and outs of our legacy code, was preparing to leave. These two issues meant we had no choice but to move forward and modernize.

We faced two primary options: a quick lift-and-shift to the cloud or a complete cloud-native rewrite. The lift-and-shift would have been fast, keeping us operational, but it was expensive and didn't solve the long-term issues, particularly the reliance on Delphi. The cloud rewrite, on the other hand, would reduce costs and future-proof our platform but would take too long, risking customer churn. Realizing that neither approach alone would work, we did both. We swiftly moved to the cloud to keep customers happy, then focused on the rewrite to ensure scalability and sustainability. This hybrid approach allowed us to meet immediate demands while laying the foundation for future growth.

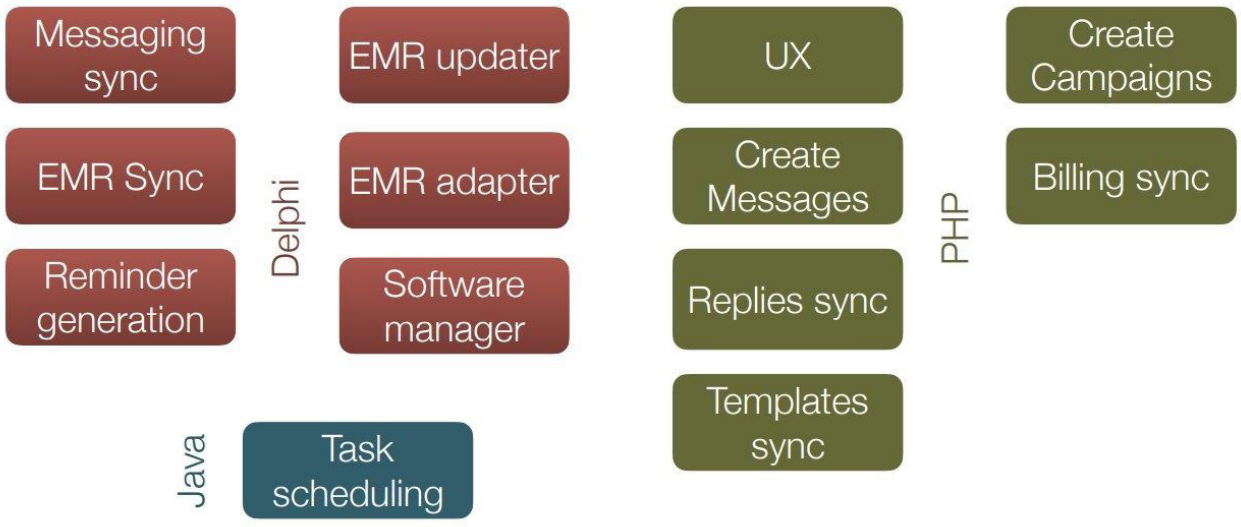
Reimagining the Architecture for the Cloud

After completing the lift-and-shift migration to the cloud, we focused on the more complicated task: the cloud rewrite. Our legacy system was a mess of various technologies, including Delphi, Java, and PHP, which lacked a clear separation of responsibilities. Before we could build a more maintainable, cloud-friendly architecture, we had to understand each component's underlying functionality.

To start, we categorized the existing services by language and purpose. The Delphi code contained critical logic for connecting to Electronic Medical Records and synchronizing data between external systems and our database. Java was used for task scheduling, while PHP managed our user interface, which was tightly coupled with business logic. This examination revealed natural groupings within our system, allowing us to think more clearly about how to redesign it for the cloud.

We recognized that our sync services could be templated into reusable components, streamlining future development. Task scheduling was simplified to a straightforward set of cron jobs managed by AWS EventBridge, which made it easier to orchestrate processes without the complexity of the previous implementation. Additionally, we decided that the EMR connections deserved a standalone service, as this functionality would be valuable for our application and could be used independently by our parent company, which focused on online GP meetings.

By modularizing the architecture, we significantly reduced the complexity of our system. We wrapped data services in APIs, allowing for easier access to data and promoting a clear separation of concerns. The application services, primarily written in PHP, were left largely untouched as they required minimal



modification. Our primary focus was on the sync and EMR services, which became the backbone of our new cloud architecture.

This reimagined system addressed the technical debt accumulated over the years and provided a scalable and maintainable framework for future growth. We created a solid foundation that would allow us to adapt to the evolving needs of our customers while ensuring that our application remained relevant in a fast-changing technological landscape.

Building Cloud-Native Services

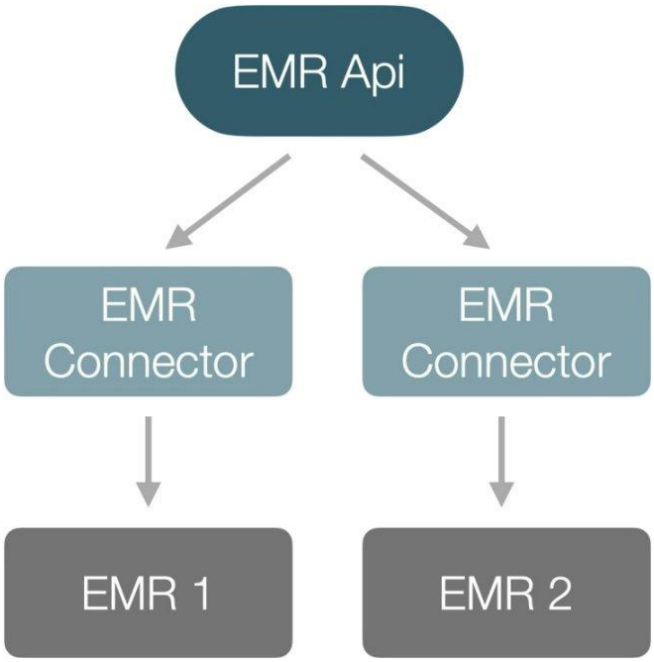
In our journey to build cloud-native services, we had to overcome several challenges, particularly when dealing with outdated legacy systems in the healthcare sector. Transitioning to the cloud required us to rethink how we integrated with external services, managed data synchronization, and handled varying workloads, all while ensuring scalability and preparing for future growth.

Connecting to External EMRs

Integrating with Electronic Medical Record systems (EMRs) poses unique challenges, particularly within the healthcare sector, which can be slow to adapt to technological change. For our migration, we needed to connect two distinct EMRs, each presenting its own set of complexities and limitations that complicated our transition to cloud-native services.

The first EMR allowed us to connect via HTTP over the internet, which initially seemed promising. However, the integration process required us to wrap our requests in a 32-bit Windows DLL to establish a connection. This added an unnecessary layer of complexity and meant we were still tied to an expensive Windows infrastructure. Furthermore, when we received responses, they came back as complicated XML structures requiring additional processing and handling which could slow down our application's performance and responsiveness.

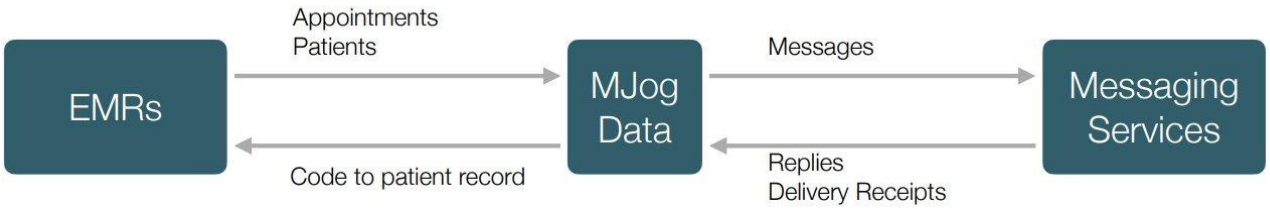
On the other hand, the second EMR offered a different set of challenges. Unlike the first, it



required an on-premise connection and demanded direct TCP calls which complicated the integration by requiring a customer specific vpn. Additionally, while both EMRs managed similar types of content, such as contacts and appointments, they utilized different XML formats and data models, complicating our ability to create a cohesive integration strategy.

As we transitioned to a cloud-native architecture, we faced the significant challenge of managing these disparate systems effectively. We lacked control over the development and monitoring capabilities of the EMRs, which further complicated our ability to ensure seamless operation within our new cloud infrastructure. Ultimately, we had to devise a strategy that allowed us to integrate these two EMRs and ensure that our cloud-native services could handle the complexities and variations inherent in their data models. This required us to think creatively about normalizing the data and streamlining the integration process, setting the stage for a robust and flexible cloud architecture.

To achieve this, we developed a proxy API with adapters that hid the complexities from our calling code. This approach allowed us to standardize data



models and query languages, enabling consistent connections to different EMRs. The system is scalable and independent, making it flexible enough to support additional connectors as needed. Ultimately, this solution decoupled the EMR logic from our core system, allowing for easier integration and growth.

Synchronization Challenges

Synchronization of data was another critical aspect of our architecture. We managed multiple sync processes, including messaging services and EMR data synchronization. Messaging services synchonizing data between our databases and SMS and email providers were, primarily, under our control running efficiently and predictably. In contrast, the EMR integrations presented challenges due to their fragile nature, often dependent on on-premises systems that could be unexpectedly turned off.

We faced a significant disparity in performance and load requirements across these syncs. Some would complete in seconds, while others took hours. Given the unpredictable nature of these demands, we opted for a microservices architecture. This choice allowed us to manage and scale each sync process independently, ensuring that varying loads could be handled efficiently without impacting other services.

Choosing Between Serverless and Containerized Solutions

A crucial decision in our journey was whether to adopt a serverless architecture or utilize containers. Our team's lack of experience with event-driven microservices led us to favor containers for now. Since our processes could take longer than the 15-minute limit imposed by serverless solutions like AWS Lambda, containers provided a familiar, manageable approach that our teams would be comfortable supporting. However, we ensured this choice did not close the door to future serverless options. As our expertise grows and demands evolve, we can transition to serverless architectures when appropriate.

Orchestration vs. Choreography

The discussion around microservices often includes a debate between orchestration and choreography. For our architecture, we primarily implemented orchestration to manage scheduled tasks and trigger processes independently for each customer. In contrast, specific functions like reminder generation operated on an event-driven choreography model, reacting to incoming appointments rather than being triggered on a fixed schedule.

This blend allowed us to leverage the strengths of both methodologies while maintaining simplicity. It also provided us with the foundations of a flexible "plug-in" architecture which would allow us to easily add or remove functionality to or from our system as requirements changed.

Prioritizing Observability

Establishing a robust observability framework became a priority in transitioning to the cloud. Initially, our on-premises systems lacked sufficient observability. We relied on audit logs that were challenging to access and analyze.

To address this, we implemented correlation IDs for every API call, enabling us to track requests across services and understand system behavior comprehensively. Additionally, we adopted structured logging, allowing us to include custom metadata and generate meaningful metrics.

These practices led to the creation of summary metrics at the end of each sync, offering insights into system performance. We recognized that our understanding of observability would evolve as we interacted with the cloud environment, necessitating regular updates to our monitoring tools and dashboards.

Defining Requirements Effectively

One of the most challenging aspects of our transition was defining our requirements accurately. Initially, we approached the task by attempting to replicate legacy code written in Delphi. This approach, however, led to a copy-paste fiasco where subtle errors proliferated across the new codebase.

Recognizing this issue, we shifted our strategy to focus on understanding the underlying requirements of the existing code before attempting to implement them in Java. This initial investment in requirement analysis paid off significantly, allowing us to create more precise user stories that our engineering team could follow. The performance improvement was significant and highlighted how important it is to thoroughly analyze and understand requirements before starting the implementation.

Conclusion

Our experience illustrates that transitioning from legacy systems to cloud-based microservices is not a one-time project but an ongoing journey. We learned that, in the face of competing priorities and pressures, it is crucial to discern the real challenges and opportunities for improvement. Software is a continually evolving entity, and embracing this change can be an advantage. Adaptability, rather than rigidity, will ultimately determine a system's longevity and effectiveness.

In retrospect, the lessons learned from our transition are invaluable for any organization looking to modernize its technology stack. As we navigate this landscape, I encourage teams to prioritize adaptability, invest time in understanding requirements, and recognize the importance of observability in system design. Instead of accepting the notion that technical debt is an unavoidable reality, strive to create inherently adaptable systems. Embrace the challenges, learn continuously, and remember that what is true today may not hold true tomorrow. This philosophy will prepare your organization for whatever changes lie ahead.

Renovate to Innovate: Fundamentals of Transforming Legacy Architecture

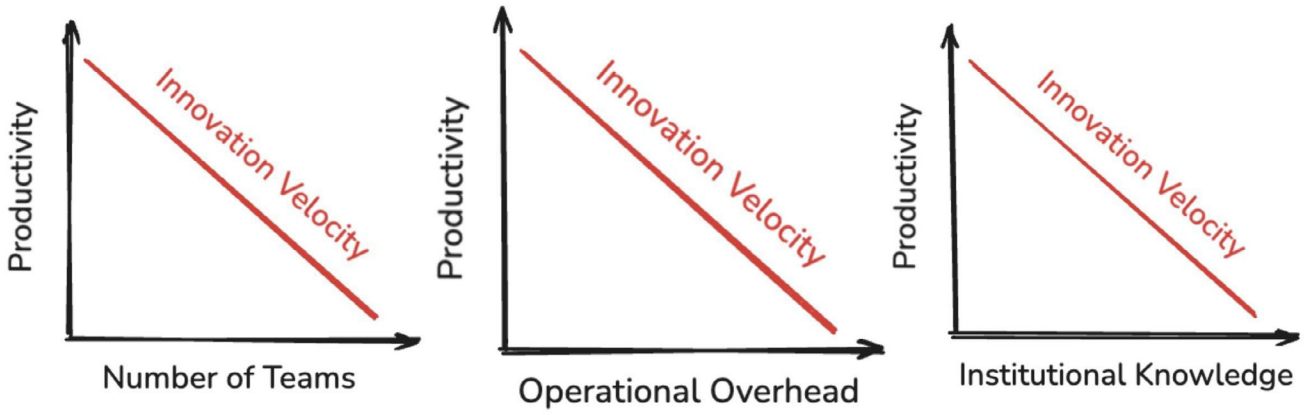
by **Rashmi Venugopal**, Staff Software Engineer @Netflix

This article is based on a talk I gave at QCon San Francisco in November 2024. In this talk, I explored the inevitability of legacy systems in successful companies and the importance of transforming legacy systems to accelerate innovation.

I discussed various strategies to tackle such technical renovation initiatives, like evolutionary architecture, deprecation-driven development, and intentional organization design. The goal of this talk was to provide actionable insights on modernizing legacy systems to ensure continued success in the future.

Understanding Legacy Systems

Successful companies go through different phases of growth, and each phase requires their software systems to evolve differently to meet the business's needs. Software systems that are adequate for validating product-market fit in a company's early stages may not be capable of supporting the business in the next phase of rapid growth. The exponential growth phase requires drastically different software capabilities than earlier phases. Successful companies, like the ones that live to see exponential growth, outgrow their software systems one way or the other.



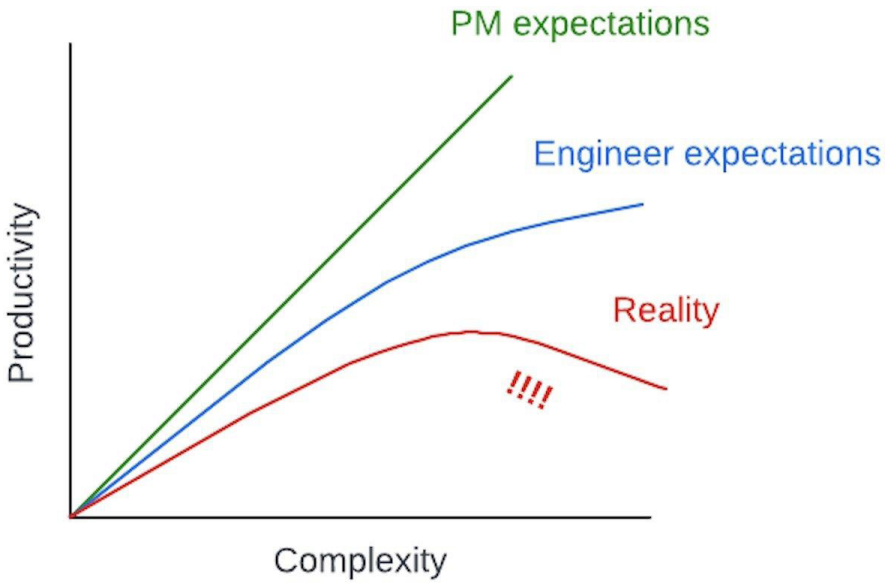
Legacy systems are a byproduct of success. However, they tend to have a negative reputation. The term "legacy" can stir up strong emotions, and understandably so. We often associate legacy systems with technical debt, challenging migrations, high maintenance costs, and a less-than-ideal developer experience. To ensure your company's long-term success, it is crucial to develop the capability to update and improve these legacy systems. Future success hinges on your ability to prevent legacy systems from hindering your company's growth.

What Defines a Legacy System?

The term "legacy" is quite overloaded. Depending on the context, it can mean anything from outdated, unsupported technology to untested code. While these are common associations, in this article, I deem a software system legacy if it can no longer reliably meet a business's evolving needs. Let's review some symptoms of legacy systems.

Substantial Complexity

Legacy systems often exhibit complexity in one or more of the following dimensions: organizational, operational, and cognitive. Organizational complexity occurs when collaboration across numerous teams slows engineers down due to the need for extensive coordination. Operational complexity arises when a lack of automation, testing, monitoring, and observability results in high operational costs. Cognitive complexity develops



when institutional knowledge accumulates, documentation becomes outdated, or key personnel leave the company.

Drop in Innovation Velocity

Why does complexity matter, you ask? As complexity increases, innovation velocity decreases. There is a direct correlation between complexity and innovation velocity.

Product and project managers expect productivity to scale linearly, but engineers know better. It is a symptom of a legacy system when the reality of how long it takes to innovate or build a new feature far exceeds expectations.

Degraded quality of Experience

Quality of Experience(QoE) measures the overall satisfaction of end-users when they interact with a system. Poor user experience, such as slow page loads, high error rates, and poor reliability, impacts the business. Amazon has an infamous study that suggests every 100-millisecond increase in latency impacts their sales by 1%. A dip in the quality of experience despite best efforts to optimize user experience is a symptom of a legacy system.

Why Do Systems Become Legacy?

Software systems become legacy for several reasons. The most apparent is the rapid pace at which technology advances. Systems that were once state-of-the-art quickly fall behind modern industry standards as technology choices become outdated within just a few years. Beyond this obvious reason, there are two schools of thought that further explain the degradation of software.

Bit Rot Theory

The Bit Rot Theory suggests that software gradually deteriorates over time due to incremental changes either within itself or in its environment. Examples of bit rot include unused code paths and code duplication. Even seemingly innocuous issues, such as inadequate documentation or the loss of institutional knowledge, can contribute to this degradation. While, in theory, good software engineering practices can mitigate bit rot, in practice, it often accumulates unnoticed until it becomes a significant hurdle.

The Law of Architectural Entropy

The Law of Architectural Entropy states that software systems lose their integrity when features are added without much consideration of the original architecture. The primary driving factor for architectural entropy is the real and sometimes unintentional tradeoffs that engineers make in order to deliver results faster or meet deadlines.

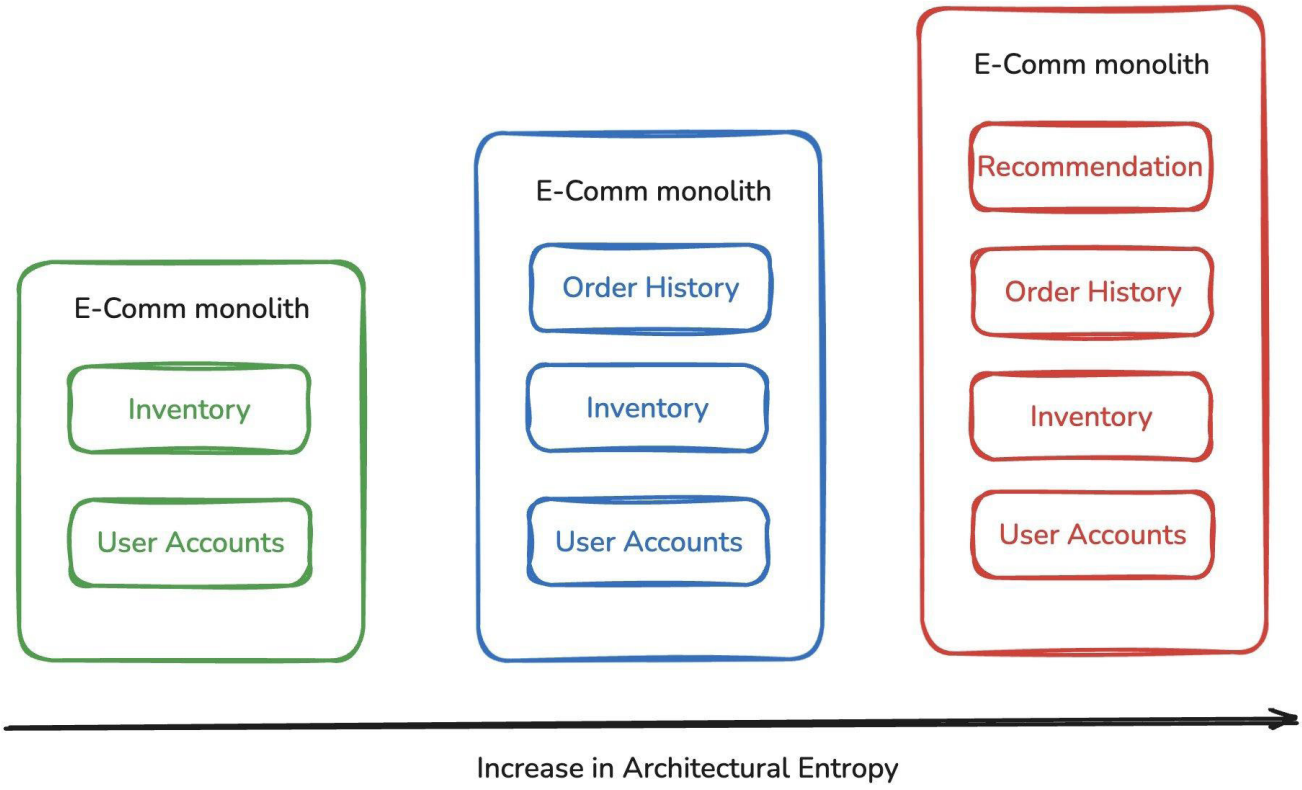
Imagine the growth of a successful e-commerce company. In the early stages, they're focused on establishing a thriving business. Evolving their architecture to be perfect is just not a priority. In fact, changes to the architecture are driven by business needs. In this example, every new feature is added to the existing monolith, steadily increasing the architectural entropy.

In reality, software systems are impacted by all these factors and beyond, which is why outdated and legacy systems are more prevalent than we might prefer. Given the commonality of legacy systems, we must ask ourselves: do we take consistent and proactive steps to renovate them? Unfortunately, the answer is often no.

The inevitability of software degradation and the lack of proactive action to counter it result in systems that are challenging to maintain, comprehend, and extend. These systems can significantly hinder your organization's growth and success, and sometimes, the only path forward is Technical Renovation.

What Is Technical Renovation?

I define technical renovation as the process of upgrading or replacing outdated systems to enhance software capabilities. While it is often confused with refactoring, there is a crucial difference between the two. Refactoring is akin



to reorganizing a closet - rearranging items to optimize for space while keeping the larger architecture intact. It enhances the existing structure and improves maintainability without altering functionality. In contrast, renovation is like transforming a regular closet into a walk-in wardrobe. It is a more extensive endeavor that introduces new capabilities and sets up the system to meet future business needs that it could not otherwise have met. Renovation involves re-architecting systems, updating frameworks, and modernizing infrastructure to align with evolving business needs.

Although this discussion focuses on renovation, it is important to recognize that refactoring is vital for maintaining system health. Refactoring often reduces technical debt and aids maintainability. However, when systems reach a point where minor improvements are insufficient, a full renovation becomes necessary. Understanding when to refactor and when to renovate is essential to prevent legacy systems from impeding progress.

Let's look at some scenarios that necessitate a technical renovation where just refactoring does not cut it:

- When business requirements undergo a drastic change, such as Netflix's transition from DVD rentals to streaming services, technical renovation becomes unavoidable. The technical capabilities required to deliver DVDs are very different from those required to stream video on demand. This is an example of a scenario where technical renovation is driven by changes in business needs.
- When the system architecture needs a significant evolution, such as Netflix moving from Monolithic to Service Oriented Architecture, necessitated a technical renovation. As Netflix expanded streaming services globally, it became increasingly difficult to scale and deploy new features quickly with a Monolithic architecture, transitioning to Microservices allowed for independent deployment and scaling of services. This is an example of a scenario where

technical renovation is driven by an architecture need.

- Technical debt occurs when you borrow from the future to make a tradeoff for the present. It is a real tradeoff that engineers have to deal with to meet business-driven deadlines. Unexpected longevity of software systems causes tech debt to accumulate to a point of no return despite well-intentioned engineers. This is a scenario where technical renovation is driven by organically outgrowing your existing systems.

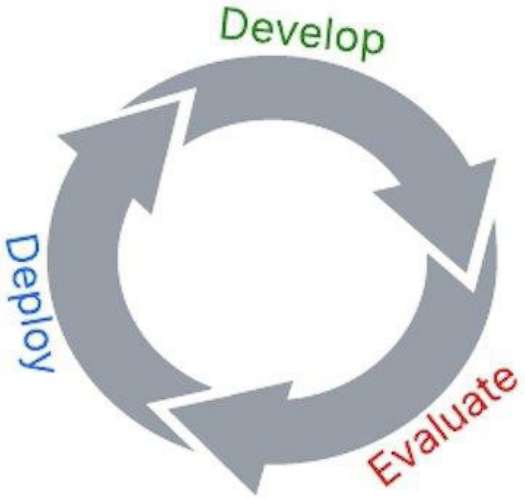
These are some nails for the hammer that is Technical Renovation. Many companies find it difficult to prioritize technical renovation due to competing business priorities, the fear of operational disruptions, and the substantial costs associated with large-scale migrations. Here are some battle tested strategies to adopt for technical renovation projects to mitigate risks of disruption and increase the likelihood of success.

Strategies for Technical Renovation

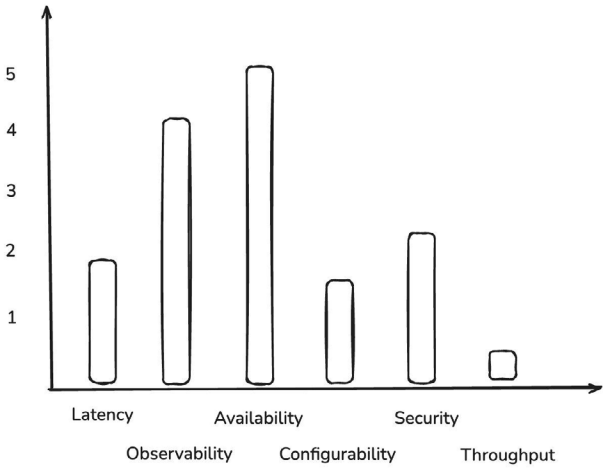
Evolutionary Architecture

The traditional approach to software architecture design is rigid and involves pre-planning. However, modern complex systems cannot be fully designed in advance. In a rapidly changing environment, software development faces numerous uncertainties, necessitating a flexible approach to system architecture. Evolutionary architecture recognizes that software systems must evolve to meet changing business requirements, technological advancements, user needs, and organizational constraints. Here are some key aspects of an evolutionary approach to software architecture:

- **Identify fitness functions** – Begin by clearly defining the business objectives and the quality attributes that are critical to your system. These could include performance, scalability, security, maintainability, and usability. For each quality attribute, establish measurable criteria that can be used to evaluate the system's health. For example, if performance is a key attribute, you might define a fitness function that measures response time or throughput under certain conditions. This is a good exercise to reveal conflicting priorities and recognize that optimizing for one quality attribute may impact others. For instance, enhancing security might affect performance. Use fitness functions to find the right balance that aligns with the overall business priorities.



- **Continuous delivery and evaluation** – Continuous Delivery ensures that software can be reliably released at any time while continuous evaluation involves monitoring the software in production to gather feedback on its performance, reliability, and user experience. This enables teams to deliver high-quality



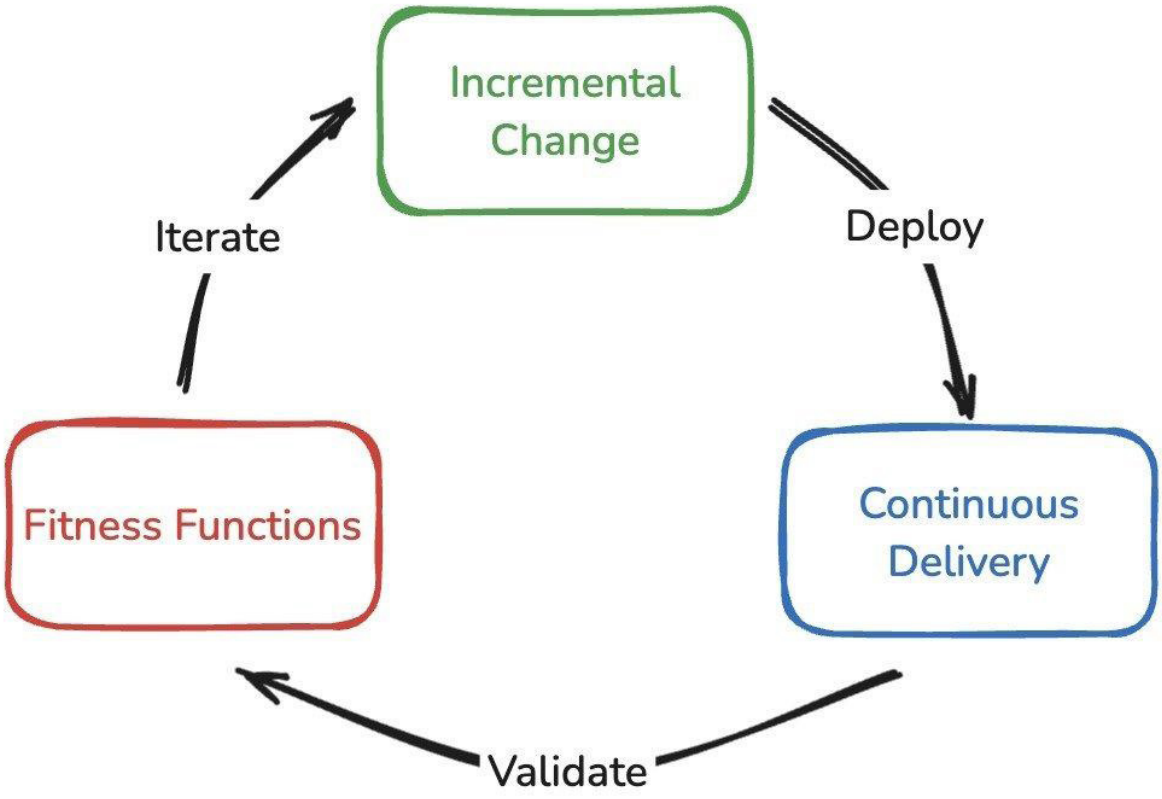
software rapidly and reliably. It encourages a culture of continuous improvement where feedback from fitness function evaluations is used to drive future architectural decisions.

- **Lean into Incremental changes** – Smaller, incremental changes are easier to measure and deploy, reducing the likelihood of introducing significant regressions. Continuous delivery and evaluation are prerequisites to supporting an iterative approach to software development, where feedback from users and monitoring tools is used to make informed decisions about future development. This iterative cycle allows teams to respond quickly to changing requirements and constraints.

Make It Work, Make It Right, Make It Fast

While Kent Beck's quote is commonly applied to writing code, it also applies to software engineering more broadly, including renovation initiatives.

- **Make it work** – This initial phase focuses on gaining confidence that your problem can be solved, one way or the other. From the standpoint of writing code, it is perfectly acceptable to take shortcuts like hard-coded inputs in this phase and allow yourself to write code that may not be pretty or easy to read in the spirit of proving feasibility. In the context of renovation, use this time to validate your assumptions and technology choices by addressing common use cases; and eliminate ineffective solutions along the way. A successful outcome of this phase is a Proof of Concept.
- **Make it right** – Once you've confirmed the viability of your solution, it's time to "Make it right". From a coding perspective, this means focusing on improving readability, refactoring your code, and strengthening your test cases. For your renovation project, now is the ideal time to ensure edge cases are covered and your



fitness functions are satisfied. Now is also a good time to test against a small cohort of real users to see how your solution holds up. An MVP - a minimum viable product that is a natural extension of the proof of concept is a good outcome of this phase.

- **Make it fast** – From a coding perspective, this phase often focuses on enhancing performance like optimizing your code, implementing caching, etc. For renovation initiatives, "make it fast" encompasses more than just performance improvements. It encompasses optimizing the speed of execution by improving documentation, setting up continuous delivery, etc; and improving operational efficiency by establishing monitoring and observability practices. These efforts accelerate your software development process, aligning perfectly with the "make it fast" objective. The outcome of this phase is production-grade software that is ready for prime time.

Here is a visualization of how each of the phases lines up for a renovation initiative. This structured approach to tackling the different aspects of a technical renovation helps break up a daunting endeavor into manageable milestones.

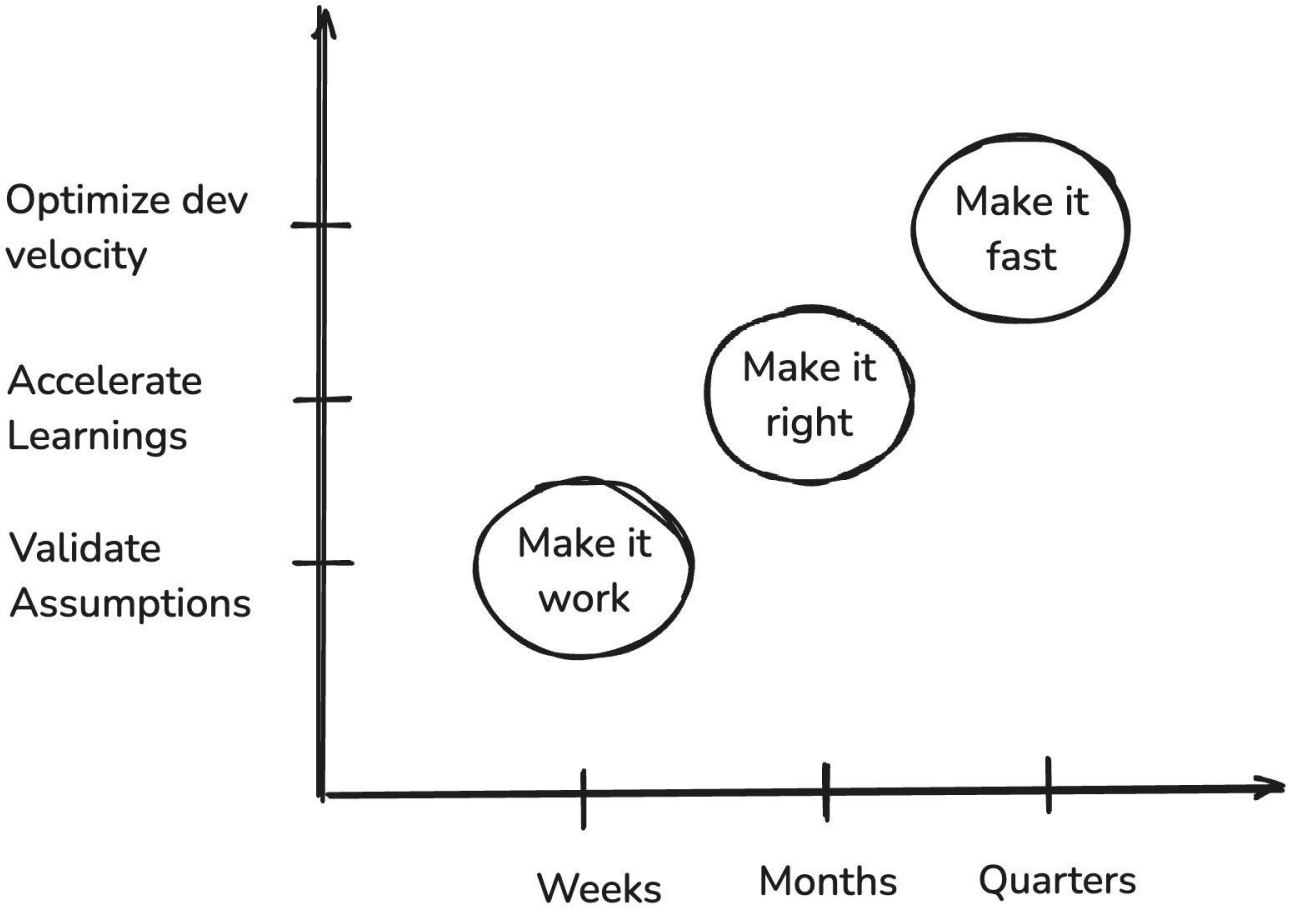
Deprecation-Driven Development

Deprecation-driven development is like spring cleaning, where the focus is on what we gain by clearing out the old rather than what we lose. Just as tidying up your home involves removing clutter to create a healthier living space, systematically eliminating obsolete technology is essential for maintaining healthy software systems. It is important to weigh the tradeoffs before renovating a system and be honest when the return on investment doesn't justify the renovation effort. Not all features are equally important for the business, when you identify a legacy feature that isn't critical to your organization's growth or success, consider scoping it out of your renovation initiative. Better yet, deprecate it entirely! Maintaining and operating software is often the most resource-intensive part of the software development lifecycle, especially for legacy systems. Therefore, I argue that investing in deprecating features is an important step in your renovation journey.

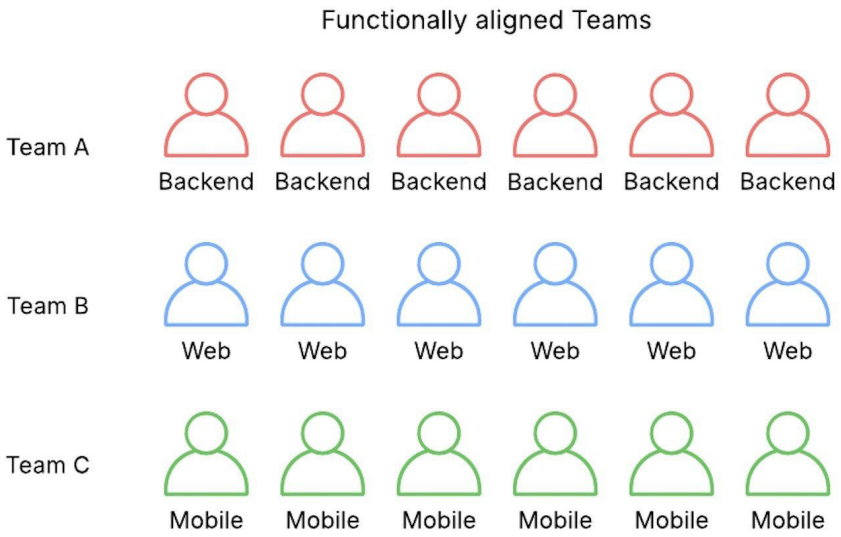
A good example of deprecation-driven development is Netflix discontinuing its DVD service. The key consideration was the balance between the cost of running the DVD service and how much it contributed to the business. As the number of DVD subscribers steadily declined, it became increasingly challenging to justify the cost required to deliver a top-notch service experience for Netflix's DVD users. Once the decision to deprecate the DVD service was made, the focus shifted away from investing in DVD-related technology, allowing resources to be redirected towards more impactful areas of the business that align with the company's long-term vision. This example underscores the significance of making tough decisions to phase out outdated offerings in favor of pursuing more promising business opportunities, ensuring the company's continued growth and relevance in a rapidly evolving entertainment ecosystem.

Intentional Organization Design

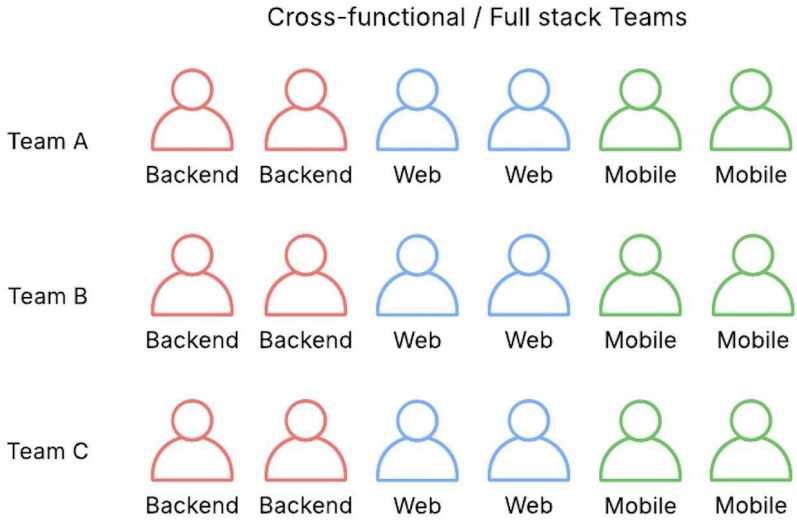
As businesses evolve, so should their organizational structure. An effective structure facilitates seamless communication and collaboration across the organization, beyond that, organizational design also impacts their system design. Conway's Law highlights the connection between team organization and system architecture, suggesting that the structure of teams influences the architecture of the systems they develop. Therefore, for organizations embarking on a technical renovation, it is wise to first step back and evaluate their current structure. This assessment can help identify beneficial changes to streamline communication and reduce the need for cross-team coordination, this is crucial for both innovation and execution velocity. As an example, let's review two common ways of structuring an engineering organization



- Functional Teams – groups engineers by function, such as backend, web developers, and mobile developers. This design allows engineers to quickly ramp up and become experts in their specific functional areas. Each product feature likely requires coordination across multiple teams as changes often span different layers of the stack.



- Cross-functional Teams – organizes engineers into full-stack teams based on common product deliverables. This design minimizes cross-team coordination for features owned within a vertical, as each team can handle all aspects of a product feature. This structure scatters technical experts across different teams, requiring additional coordination for the specialists to align on best practices, share learnings and cross-pollinating ideas relevant for their areas of expertise.



Each has pros and cons, and the right choice depends on whether deep functional expertise or seamless collaboration within a product vertical is the priority. In conclusion, it's essential to strengthen the communication paths that are most critical for your organization and its architecture, as not every communication path can be equally robust.

Conclusion

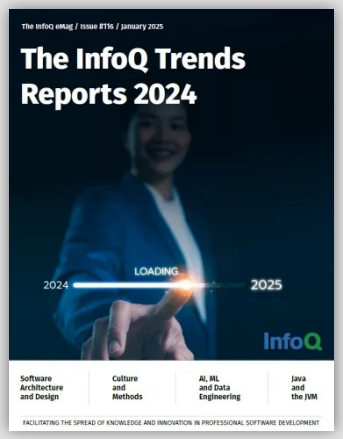
The renovation strategies I discuss above may seem ambitious, but they are intentionally aspirational. There are no one-size-fits-all solutions for technical renovations; the ideal approach is highly context-dependent. Your strategy and decisions should be debated on a case-by-case basis and account for your organization's unique circumstances and goals.

Start with the right perspective, the perspective that legacy systems are inevitable in successful companies because they are a byproduct of success. For continued success, it is crucial to use the right tools for the problem you're facing, while refactoring can address certain issues, a technical renovation might be necessary for others. Invest in technical renovation when it is justified, and when you do, focus on building systems that are evolvable to keep pace with your business needs. Break down the daunting task of renovation into manageable milestones: make it work, make it right, and finally, make it fast. Exclude legacy features from your renovation plan if the return on investment is not justified.

Regardless of your company's growth trajectory, size, or operating margins, transforming legacy systems will present new challenges. So embrace a growth mindset, seek feedback, and learn from your mistakes to make the most of your renovation journey.

Curious about previous issues?

This eMag marks a pivotal moment in the evolution of artificial intelligence, a shift from the familiar realm of chatbots to the more dynamic and autonomous world of AI agents.



This special edition of The InfoQ eMag, which contains a comprehensive collection of our popular InfoQ Trends Reports from 2024. Our reports have delved into the latest advancements, challenges, and future directions.

In this InfoQ emag, we aim to inspire and guide platform engineers into building effective platforms and delivering exceptional developer experience.

InfoQ