Caching Entire Systems without Invalidation

SREcon EMEA 2022
October 27, 2022

Peter Sperl
Engineering Manager, Structured Products

TechAtBloomberg.com

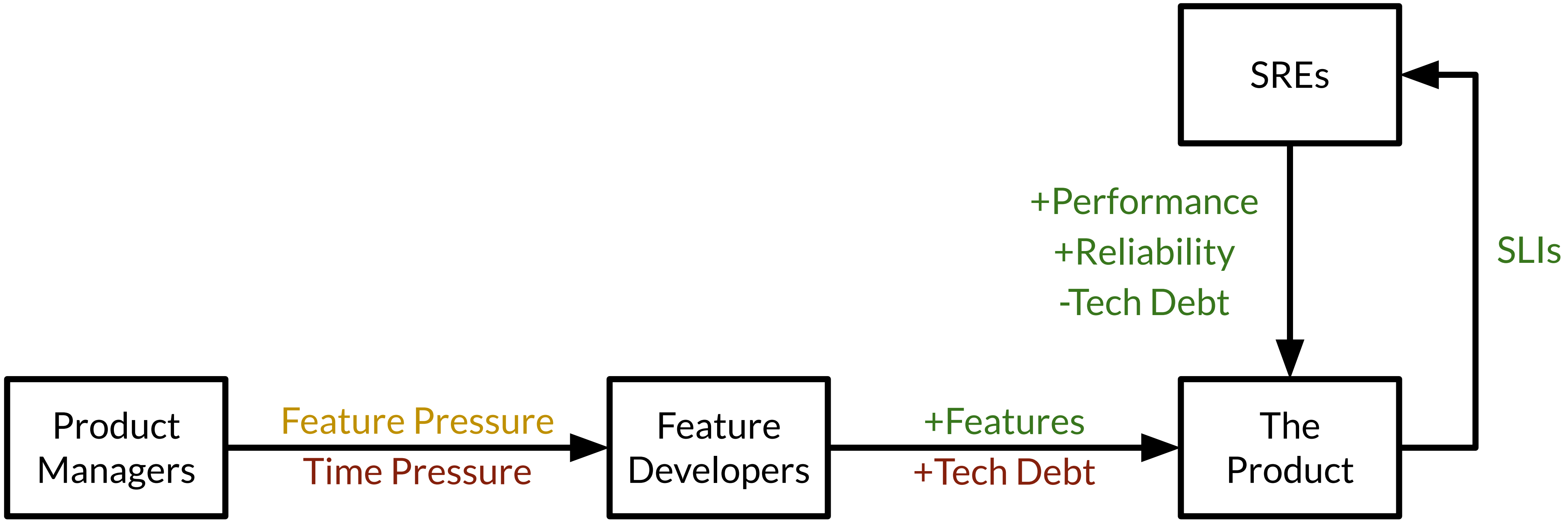
Bloomberg Engineering

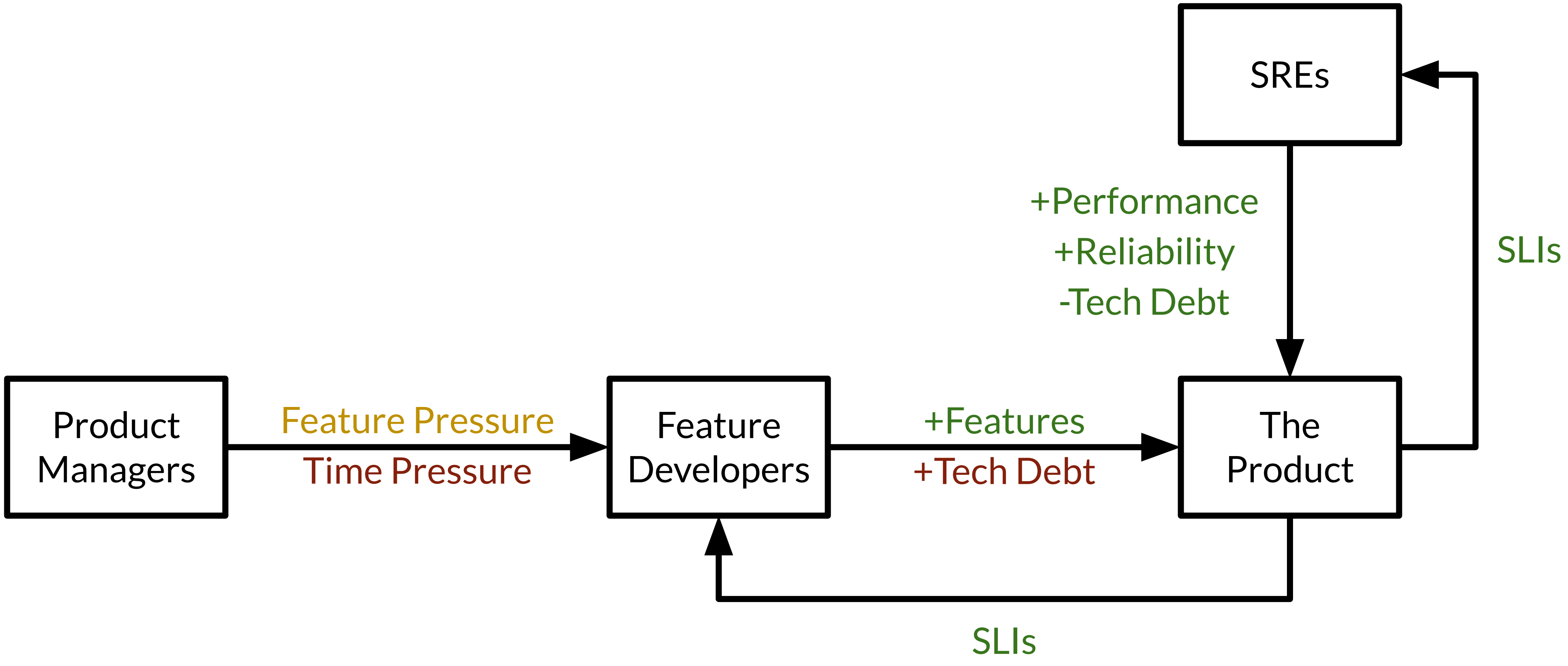
Hello

This is a talk about distributed systems architecture

“What could SRE be?”


Product Managers
Feature Pressure
Time Pressure
Feature Developers
+Features
+Tech Debt
The Product

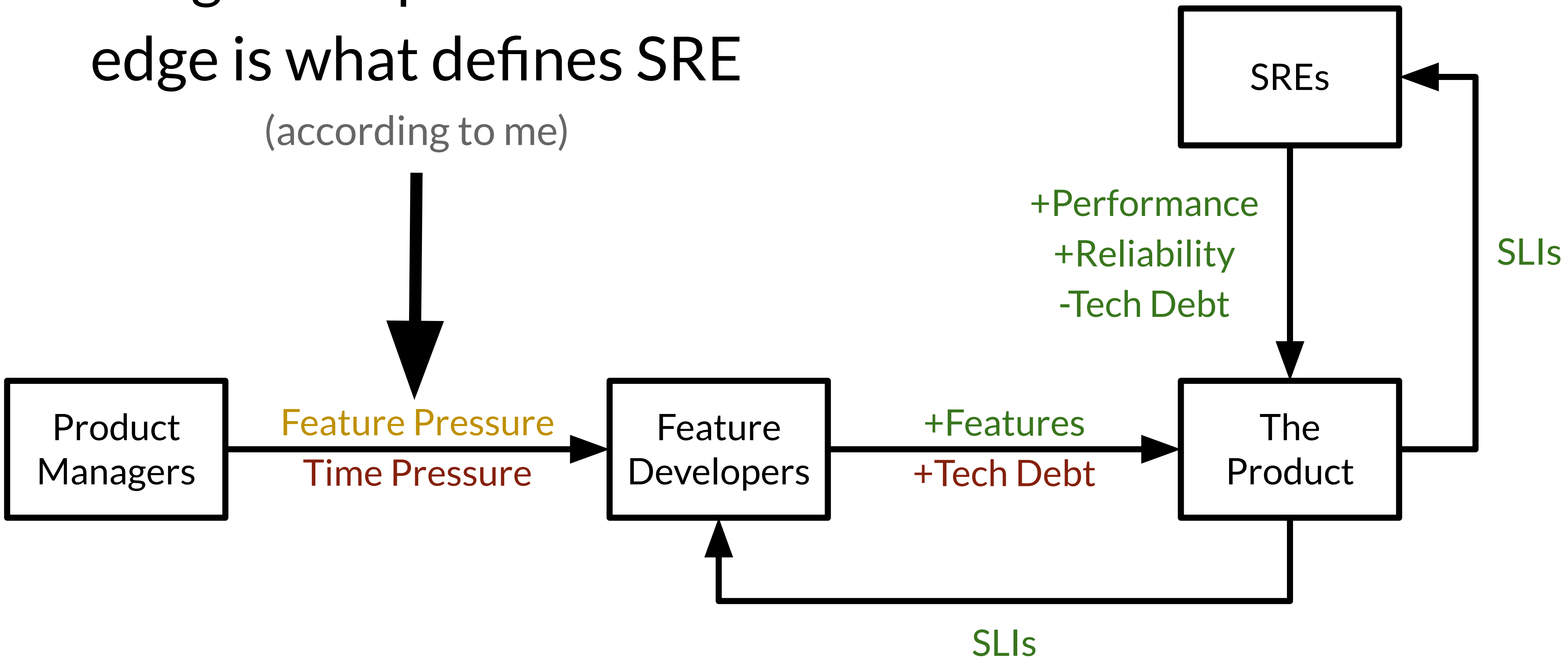

SREs
+Performance
+Reliability
-Tech Debt
SLIs
Product Managers
Feature Pressure
Time Pressure
Feature Developers
+Features
+Tech Debt
The Product



SREs
+Performance
+Reliability
-Tech Debt
SLIs
Product Managers
Feature Pressure
Time Pressure
Feature Developers
+Features
+Tech Debt
The Product
SLIs

Being decoupled from this edge is what defines SRE

(according to me)



I’m here to show you how to design and build truly stateless systems

A stateless component always returns the same output for a given input

- Statelessness
  - Cacheability
    - Performance
    - Scalability
    - Cost Reduction
  - Testability
    - Rapid Iteration
    - Reliability

Caching Entire Systems without Invalidation

Caching $\underbrace{\text{Entire Systems}}_{\text{Topic \#2}}$ $\overbrace{\text{without Invalidation}}^{\text{Topic \#1}}$

Caching without Invalidation

TechAtBloomberg.com

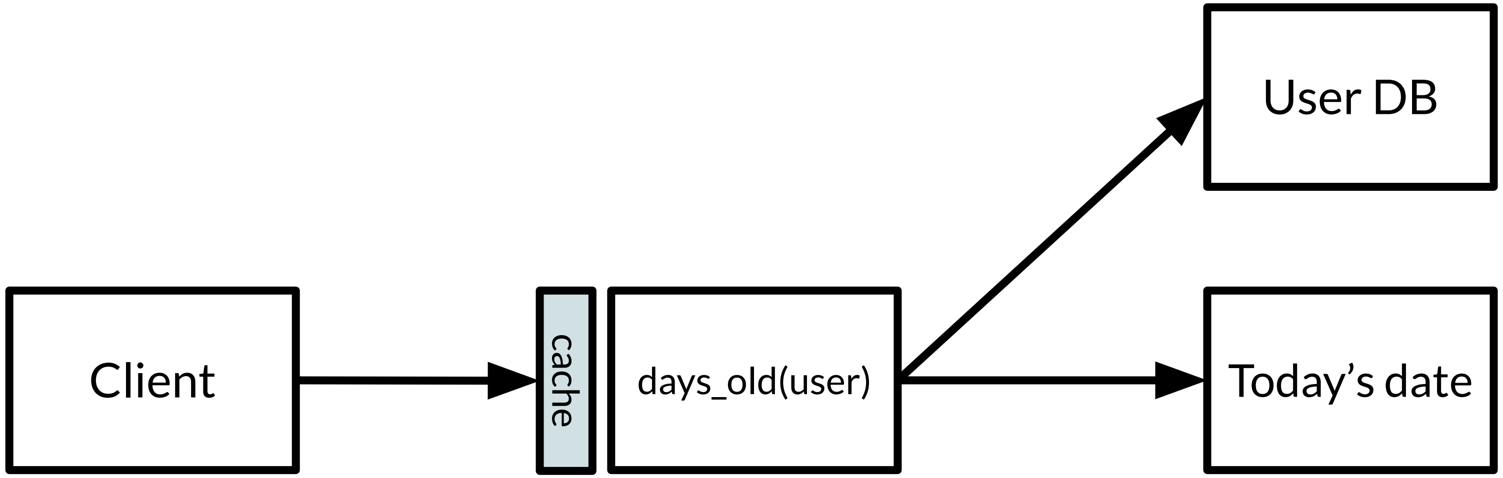
Bloomberg

Engineering

"There are 2 hard problems in computer science: cache invalidation and naming things."

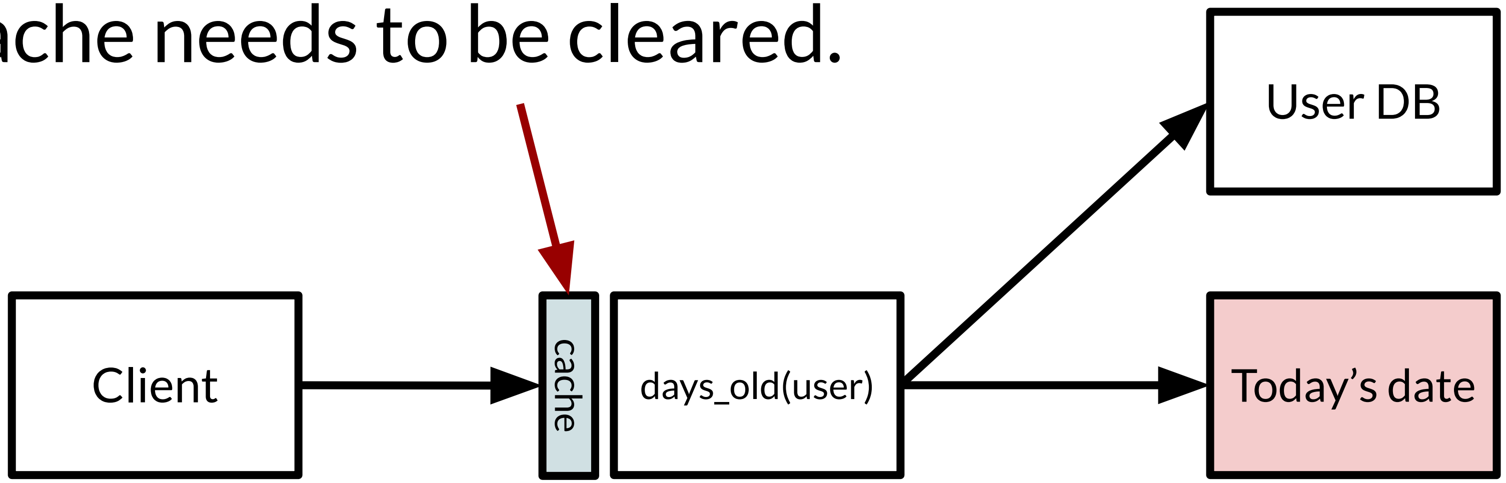
– Phil Karlton

```
days_old(username)
```

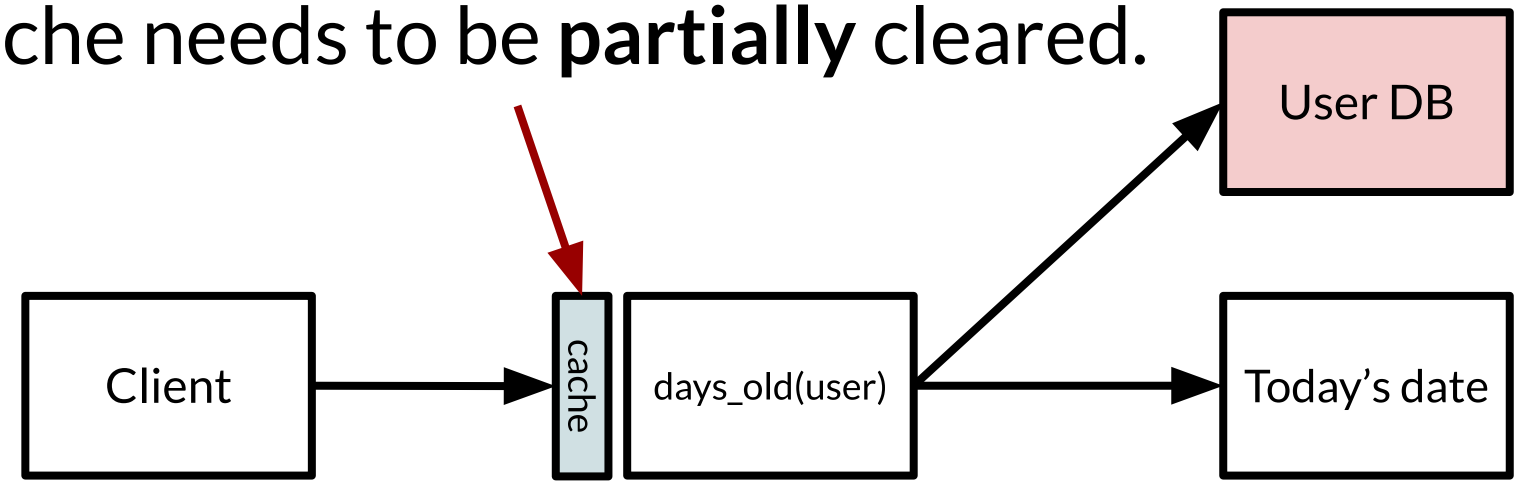


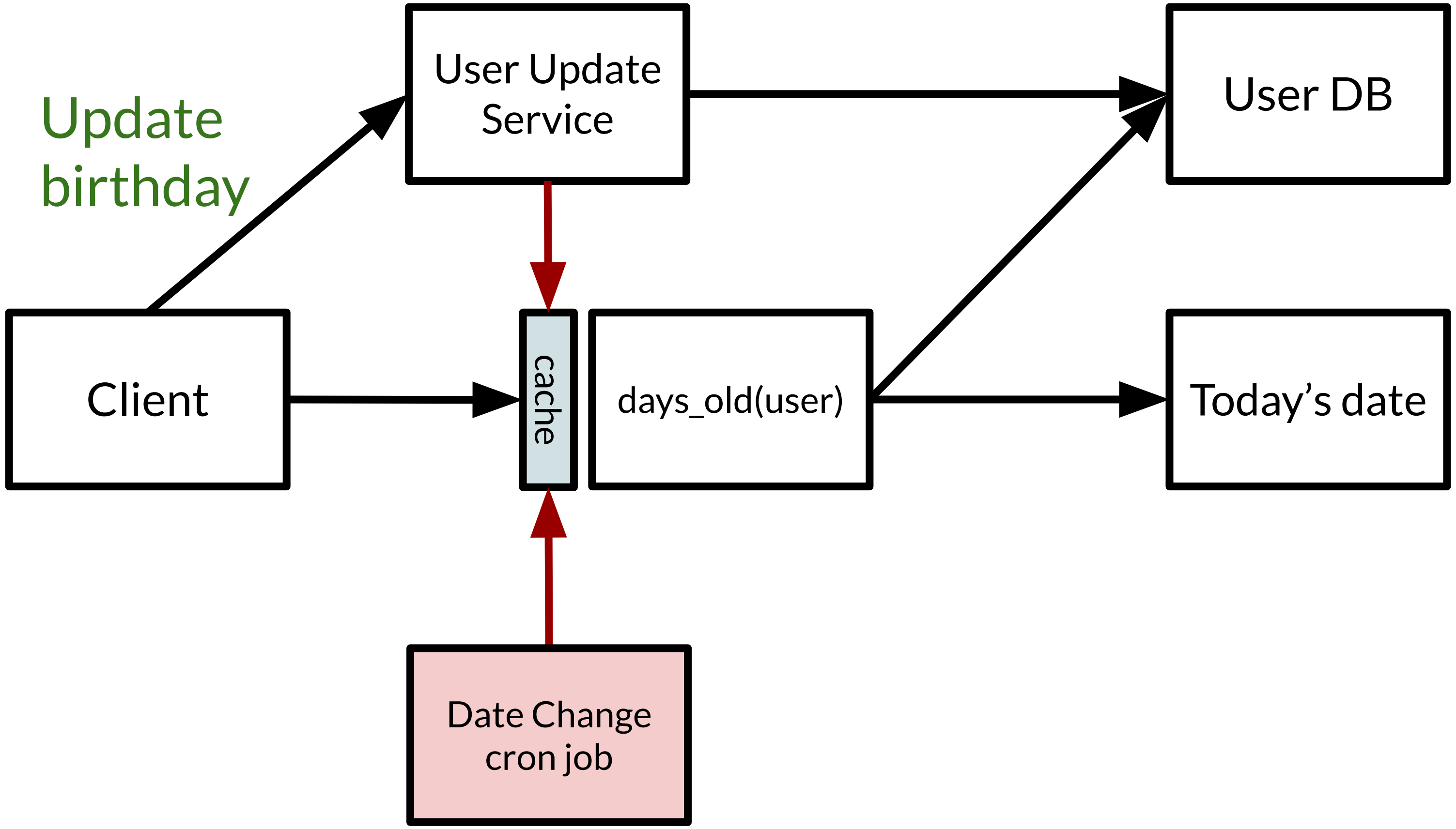
Client
cache
days_old(user)
User DB
Today's date

Whenever the date changes, this cache needs to be cleared.



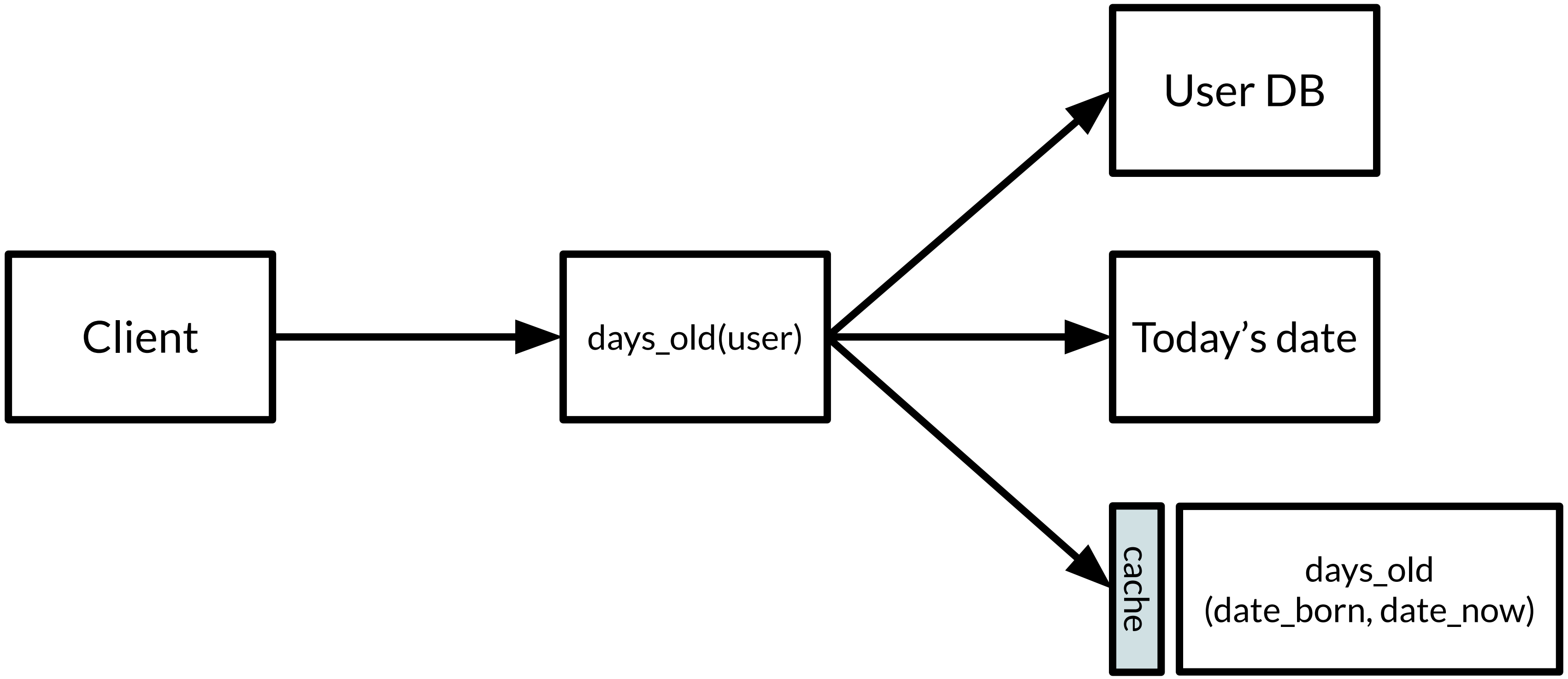
Whenever a user's birthday is updated, this cache needs to be **partially** cleared.

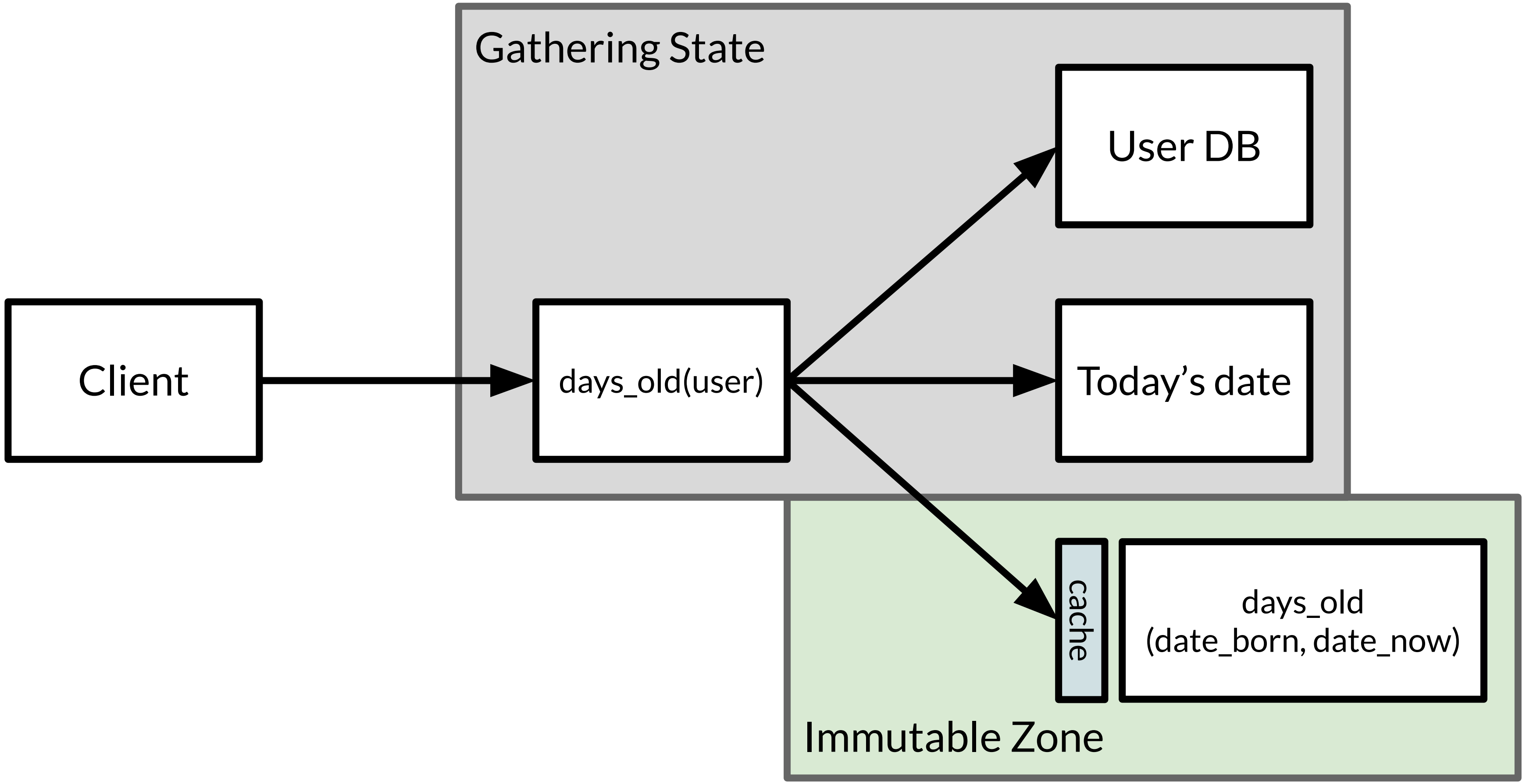



Update
birthday
User Update
Service
User DB
Client
cache
days_old(user)
Today's date
Date Change
cron job

"The only good cache invalidation strategy is no strategy."

– Me, maybe


Client
days_old(user)
User DB
Today's date
cache
days_old
(date_born, date_now)


Gathering State
User DB
Client
days_old(user)
Today's date
cache
days_old
(date_born, date_now)
Immutable Zone

	Cache Lifetime	Invalidation Strategy
`days_old(date_born, date_now)`	Forever	None
`days_old(date_born)`	Until tomorrow	TTL
`days_old(username)`	Until birthday changes	Active & TTL

Cache

days_old(1982-09-06, 2022-10-27) = 14,661

days_old
(date_born, date_now)

What happens if I change my birthday?

Cache

days_old(6/9/82, 27/10/22) = 14,661

days_old
(date_born, date_now)

Old birthday is still in cache

Cache

days_old(6/9/82, 27/10/22) = 14,661

days_old(1/9/82, 27/10/22) = 14,666

days_old
(date_born, date_now)

New entry with new birthday

Cache

days_old(pete) = 14,661

days_old(username)

What happens if I change my birthday?

Cache

days_old(pete) = 14,661

days_old(username)

This is now wrong!
It should be 14,666

Cache

days_old(pete) = 14,661

days_old(username)

We need to erase the old entry first
to allow the new value to be written

Cache

~~days_old(pete) = 14,661~~

days_old(pete) = 14,666

days_old(username)

Cache

days_old(6/9/82) = 14,661
(expires at midnight)

days_old(1/9/82) = 14,666
(expires at midnight)

days_old(date_born)

True statelessness reduces total complexity

Any cache invalidation is bad

Interface design drives caching characteristics (among other things)

Stateful interfaces can be converted into stateless ones internally

Factor systems into stateful and stateless layers

Caching Entire Systems

TechAtBloomberg.com

Bloomberg

Engineering

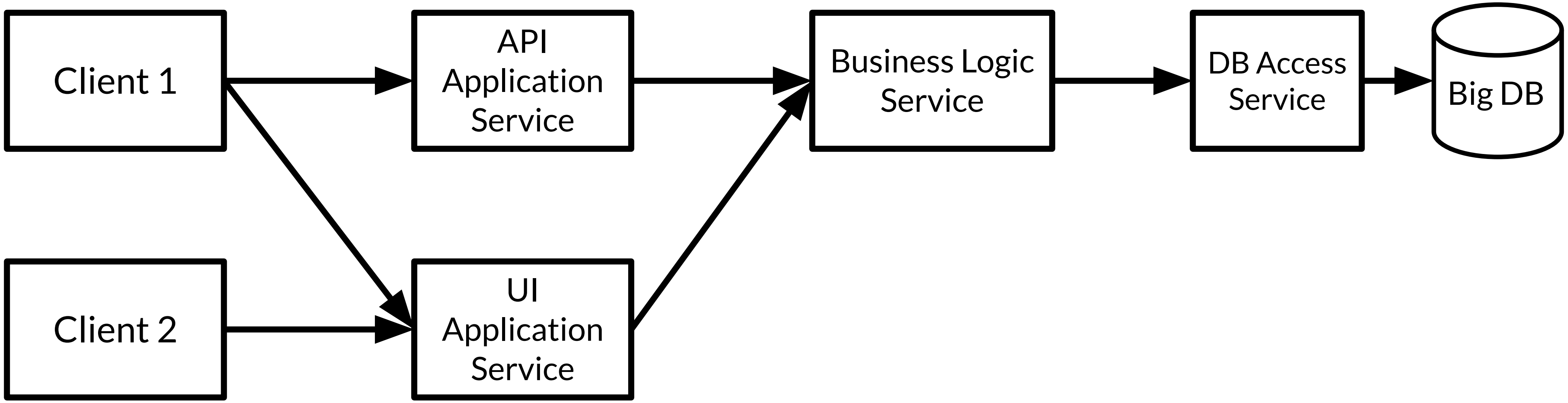
Big Databases

✔ Small Databases — Resolve early into explicit values, replicate to scale

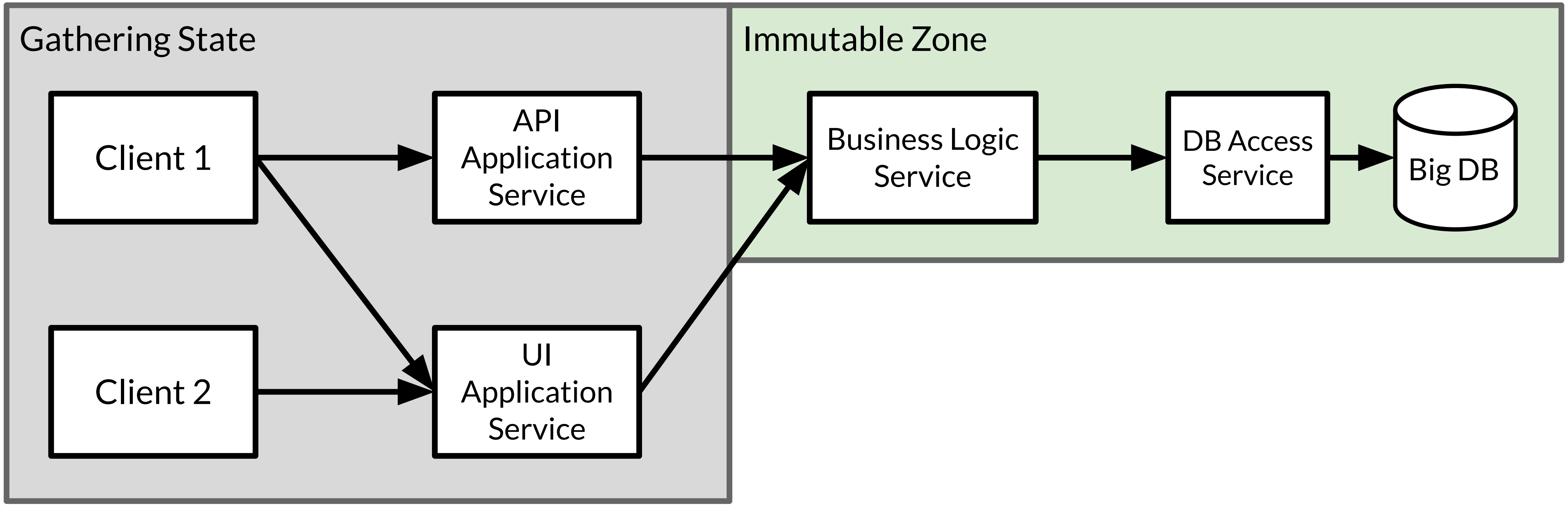
✔ Wall Time — Resolve early into explicit time or date

Software Versions

External Systems



Client 1
API Application Service
Business Logic Service
DB Access Service
Big DB
Client 2
UI Application Service



Gathering State
Immutable Zone
Client 1
API
Application
Service
Business Logic
Service
DB Access
Service
Big DB
Client 2
UI
Application
Service

How can we make access to a large, constantly changing database, stateless?

How can we make access to a large, constantly changing database, stateless?

The timestamped data pattern

-or-

The snapshot pattern

Entity	timestamp	some_data	more_data
A	1		
B	1		
B	2		
B	3		
B	5		

Step 1

get_most_recent_timestamp(B) = 5

Step 2

get_data(B, 5) = some_data

This is immutable!

Step 1: Stateful call to get timestamp

```
select max(timestamp) from table where entity="B"
```

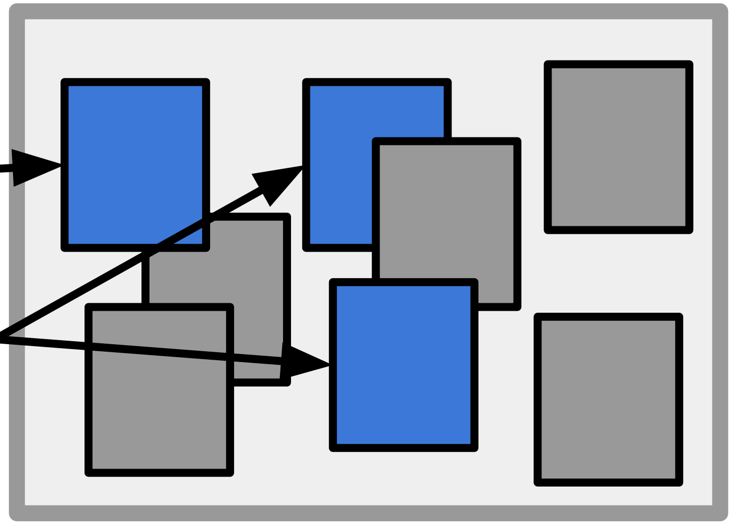
Step 2: Stateless call to get data using said timestamp

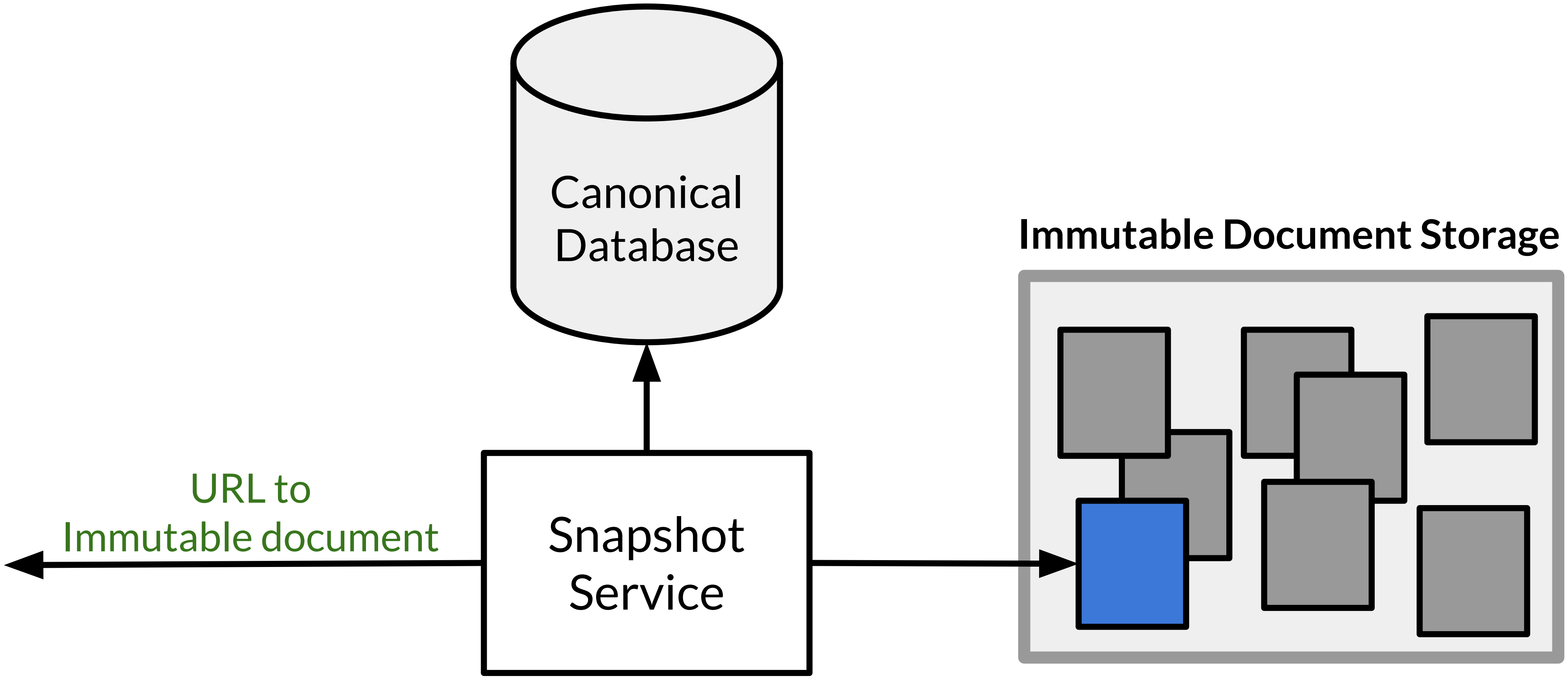
```
select data from table where entity="B" and timestamp <= 24
```

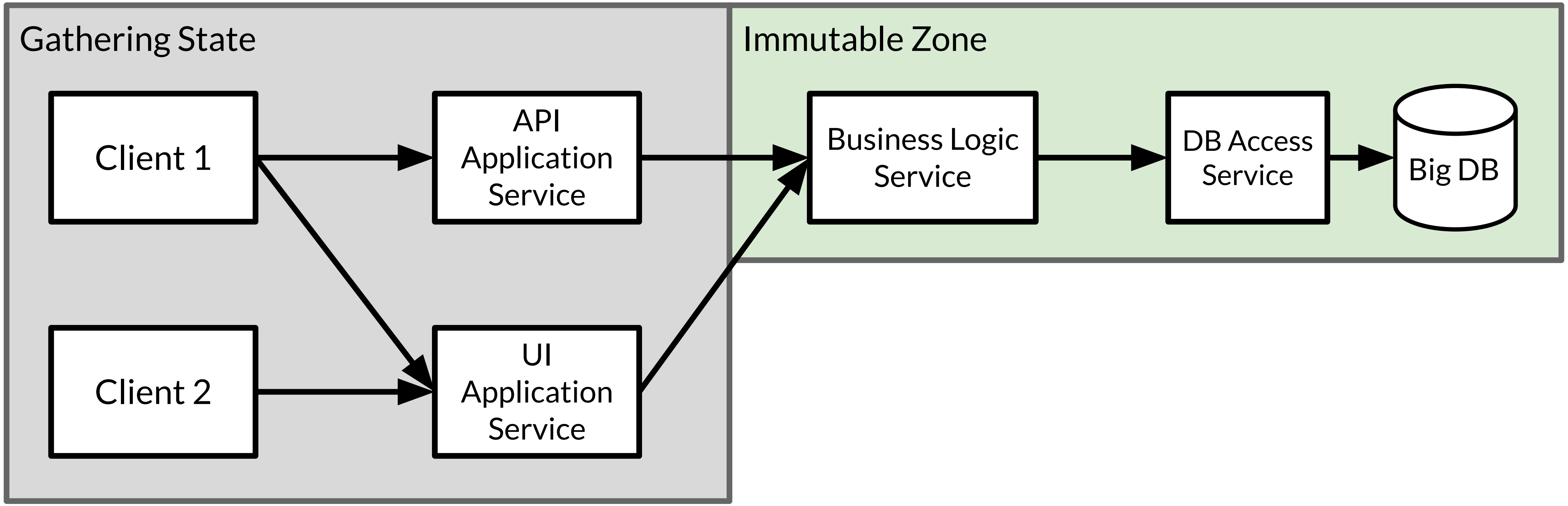
Stateful Relational Table

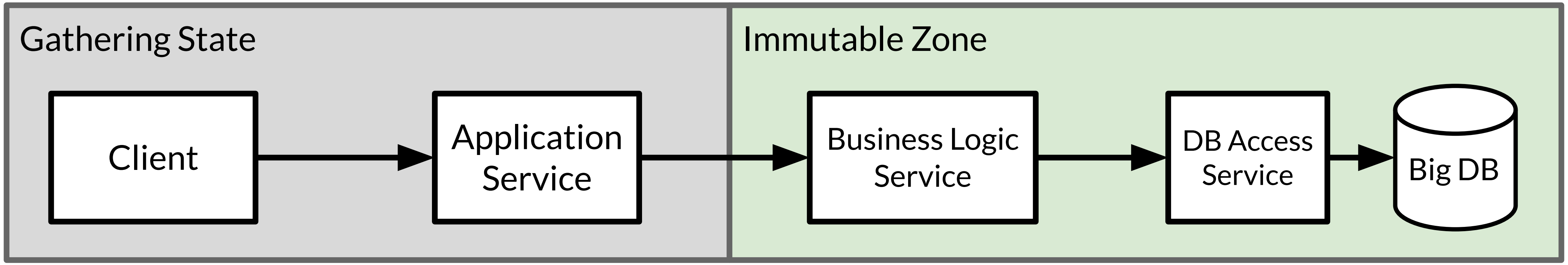
Entity	timestamp	URL
A	1	datastore.com/3fds80mvdy
B	2	datastore.com/7xdf8kasnw
B	1	datastore.com/cjw92kscnsq

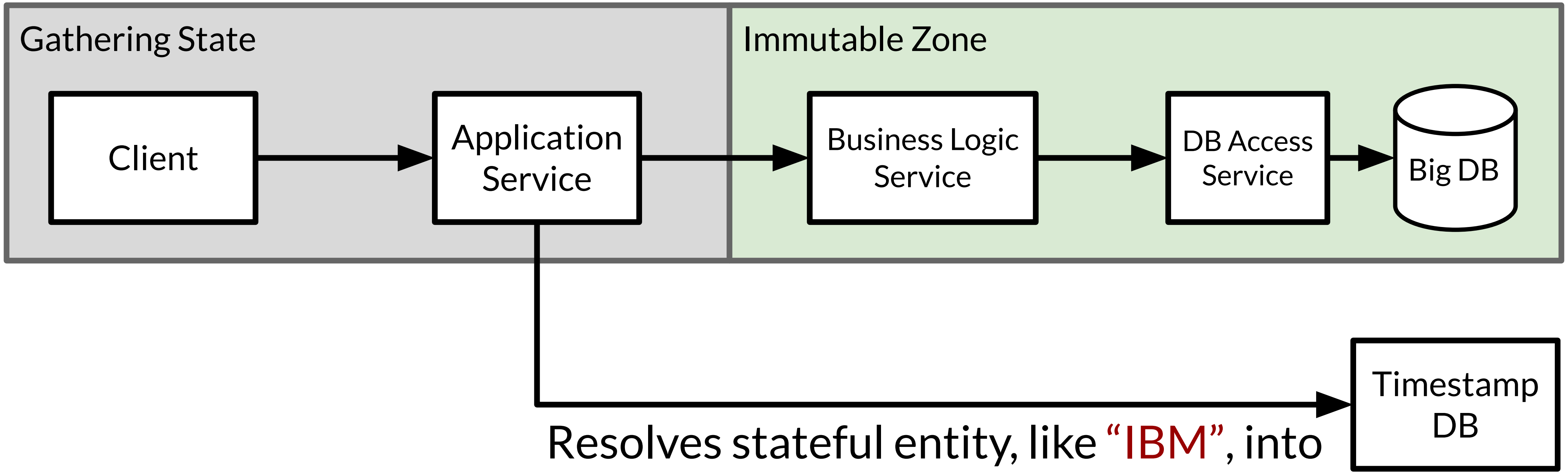
Immutable Document Storage



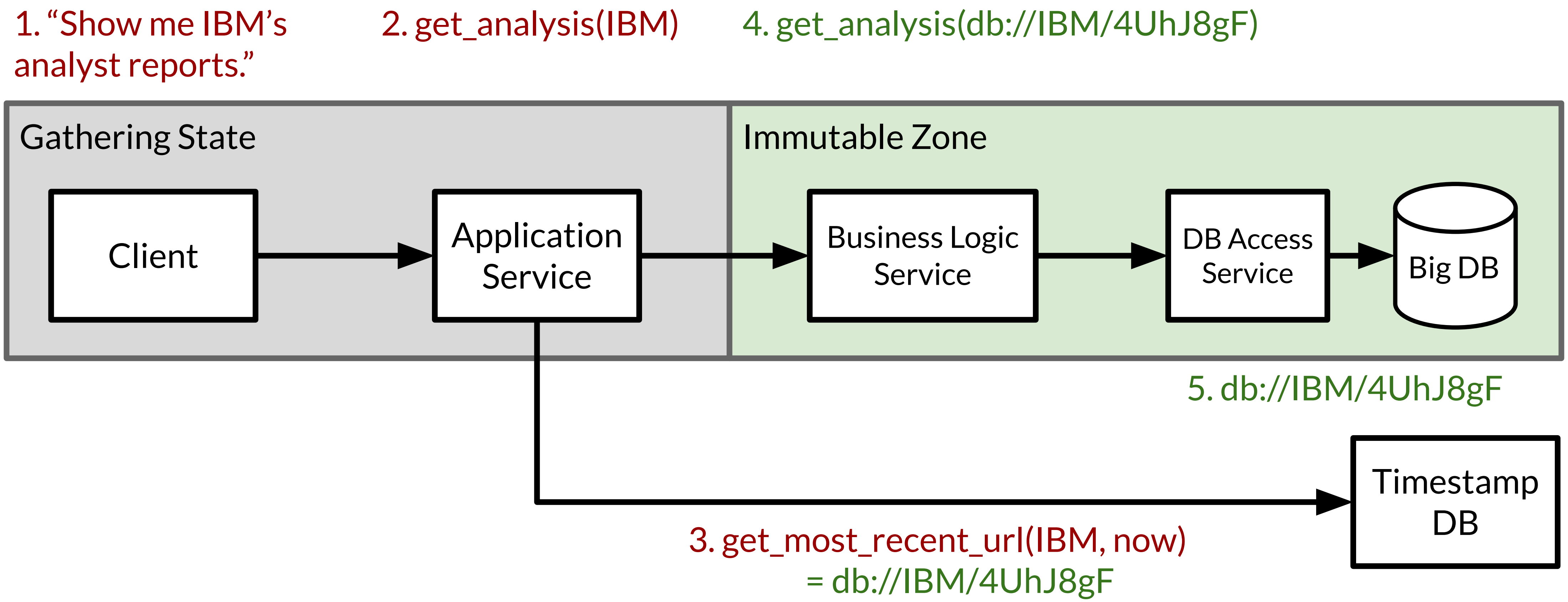

Canonical
Database
Immutable Document Storage
URL to
Immutable document
Snapshot
Service


Gathering State
Immutable Zone
Client 1
Client 2
API Application Service
UI Application Service
Business Logic Service
DB Access Service
Big DB

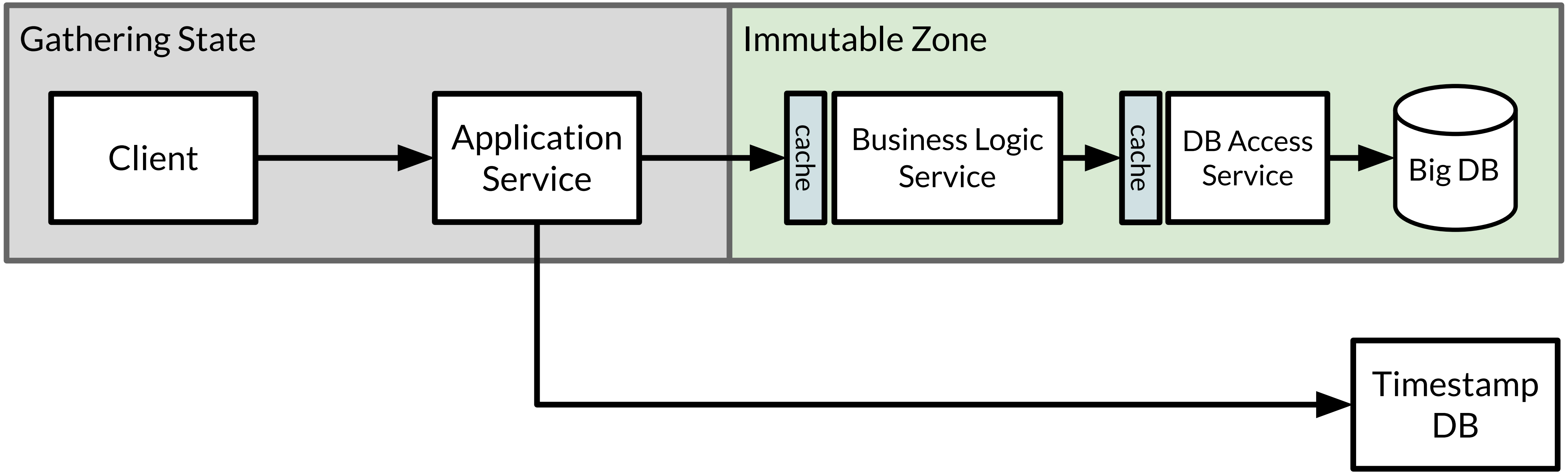

Gathering State
Immutable Zone
Client
Application Service
Business Logic Service
DB Access Service
Big DB



Resolves stateful entity, like "IBM", into the URL of an immutable document (db://reports/IBM/4UhJ8gF)



1. “Show me IBM’s
analyst reports.”
2. get_analysis(IBM)
4. get_analysis(db://IBM/4UhJ8gF)
Gathering State
Immutable Zone
Client
Application
Service
Business Logic
Service
DB Access
Service
Big DB
5. db://IBM/4UhJ8gF
Timestamp
DB
3. get_most_recent_url(IBM, now)
= db://IBM/4UhJ8gF



Gathering State
Immutable Zone
Client
Application Service
cache
Business Logic Service
cache
DB Access Service
Big DB
Timestamp DB

Benefits of Timestamped Data Storage

Database reads can be cached as well as any service call that depends on it

Point-in-time access is trivially supported

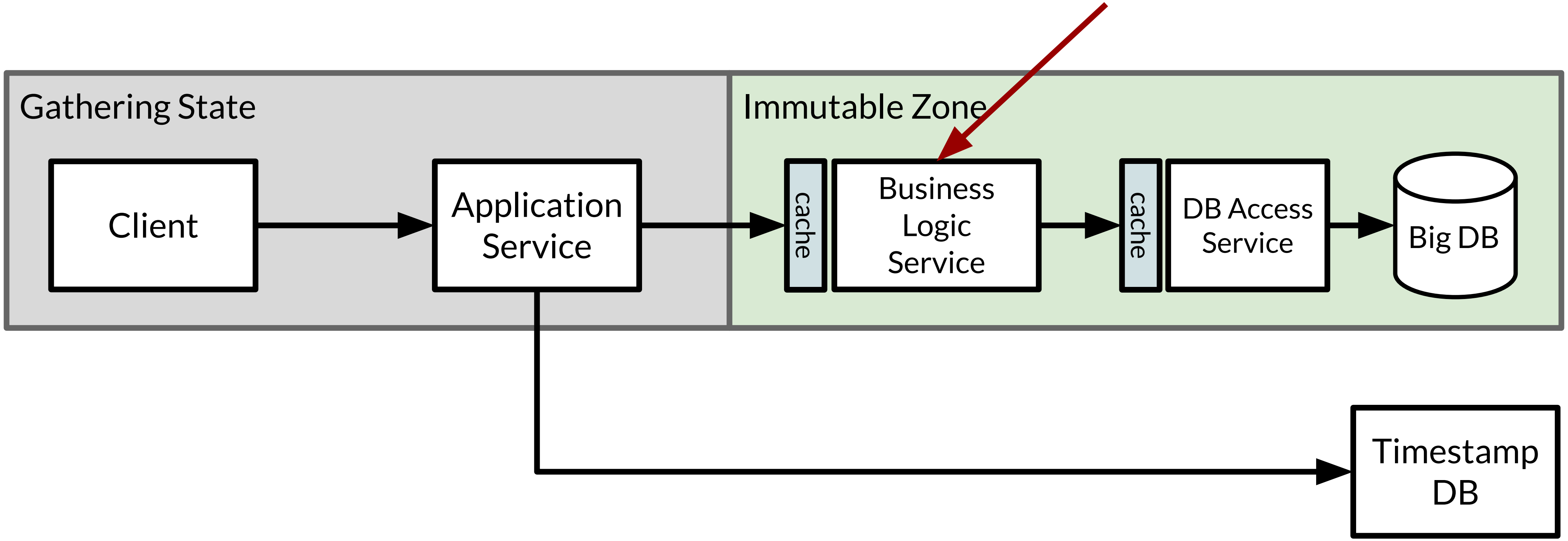
Batch jobs can freeze the timestamp to ensure consistency, while updates continue unaffected

Rollbacks can be performed with a system-wide cap on timestamp

Timed releases are just future-dated timestamps

✔	Big Slow Databases	Use the timestamped data or snapshot pattern
✔	Small Databases	Resolve early into explicit values, replicate to scale
✔	Wall Time	Resolve early into explicit time or date
	Software Versions	
?	External Systems	Resolve early, use the snapshot pattern, or give up

What if you deploy a new version of this service that changes the output?



Cache

biz_logic(url) = X

biz_logic(url)

Cache

biz_logic(url) = X

biz_logic(url)=X

service v1

biz_logic(url)=Y

service v2

Cache

hash(v1, biz_logic(url)) = X

hash(v2, biz_logic(url)) = Y

biz_logic(url)=X

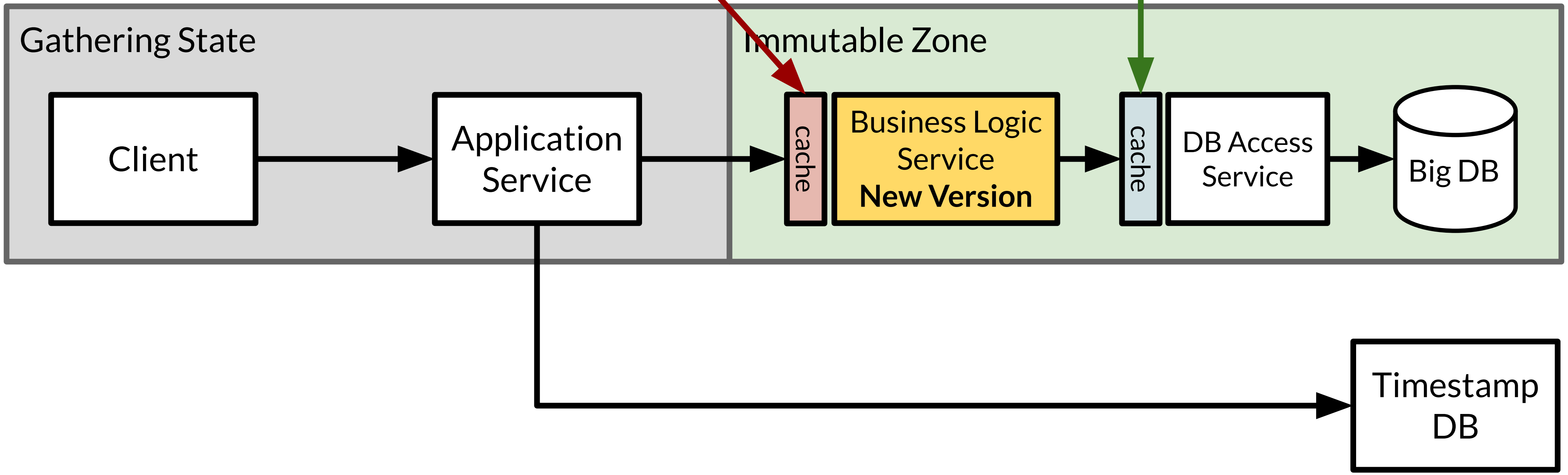
service v1

biz_logic(url)=Y

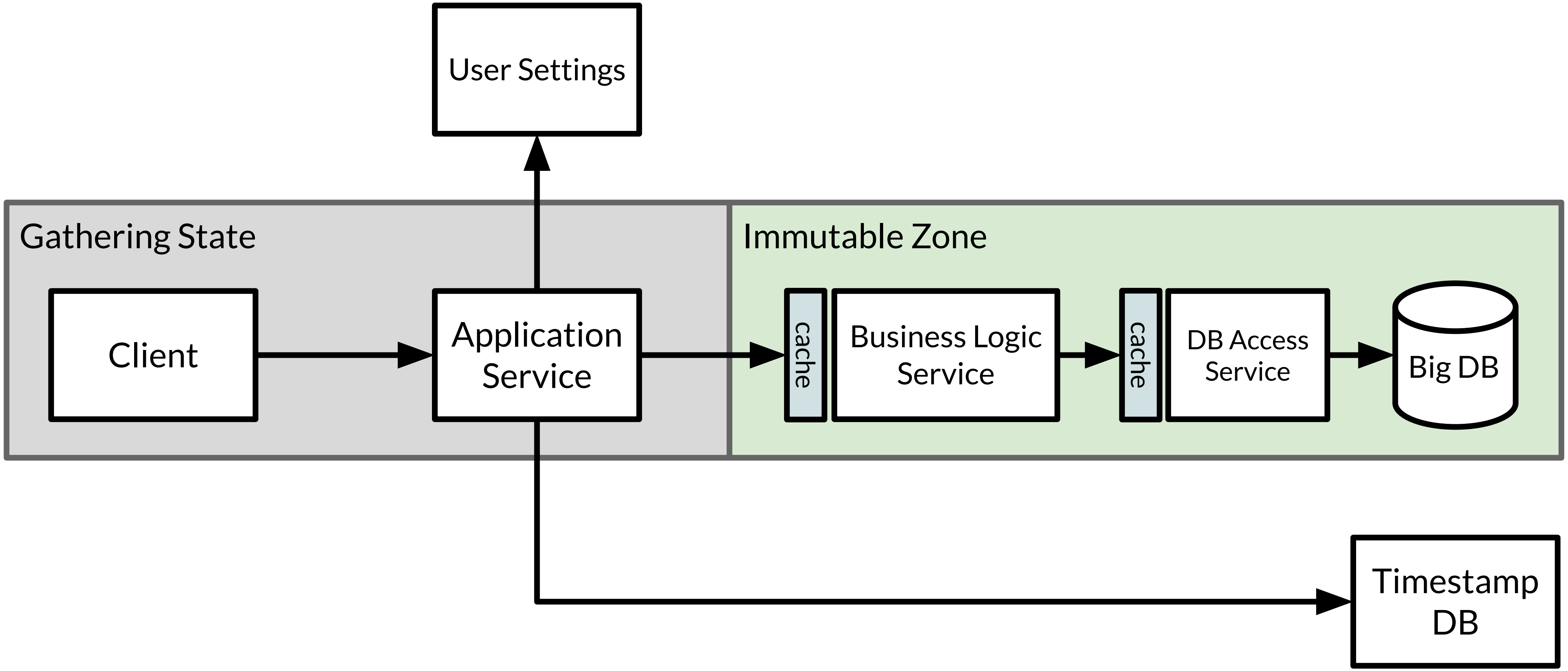
service v2

This cache is “cleared” by the new service version

This cache still works!



✔	Big Slow Databases	Use the timestamped data or snapshot pattern
✔	Small Databases	Resolve early into explicit values, replicate to scale
✔	Wall Time	Resolve early into explicit time or date
✔	Software Versions	Include in cache key
?	External Systems	Resolve early, use the snapshot pattern, or give up


User Settings
Gathering State
Immutable Zone
Client
Application Service
cache
Business Logic Service
cache
DB Access Service
Big DB
Timestamp DB

✔ Big Slow Databases	Use the timestamped data or snapshot pattern
✔ Small Databases	Resolve early into explicit values, replicate to scale
✔ Wall Time	Resolve early into explicit time or date
✔ Software Versions	Include in cache key
? External Systems	Resolve early, use the snapshot pattern, or give up
✖ Write-Heavy DBs	Resolve early, use TTL caching, or give up

We are hiring: bloomberg.com/engineering

True statelessness reduces total complexity

Any cache invalidation should be a non-starter

Interface design drives caching characteristics (among other things)

Stateful interfaces can be converted into stateless ones internally

Factor systems into stateful and stateless layers

Make low-level components stateless and chain upwards

Key generation is the right place to account for state