Death Stars & River Deltas

Toward a Functional Programming Analogy for Microservices

Hi, I’m Bobby

I’m on the Technology Fellows team at 

I dislike accidental complexity

bobby.calderwood@capitalone.com
@bobbycalderwood
https://github.com/bobby

Disclaimers

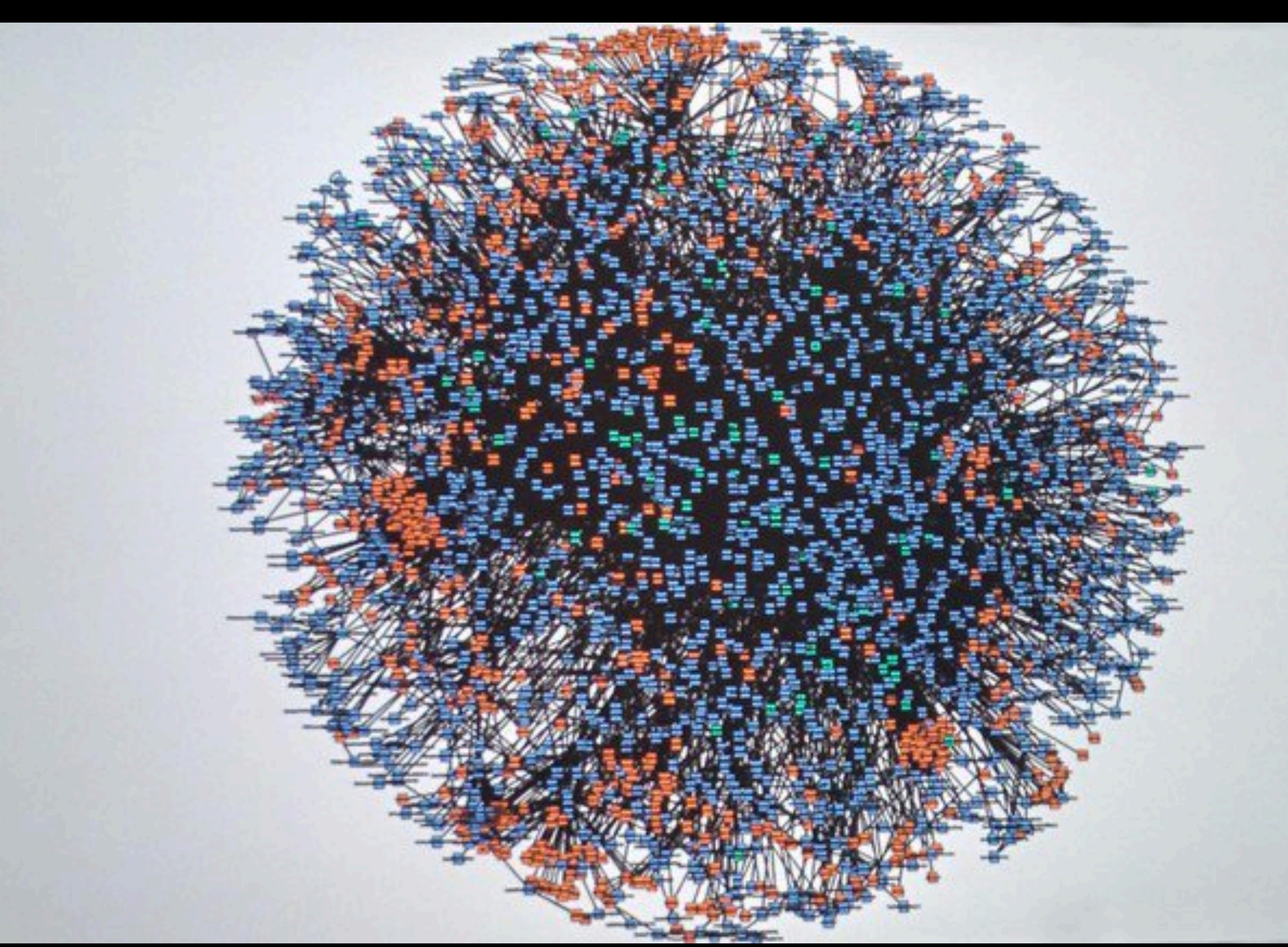
- What follows is not a rigorous attempt to extend the formal models of OO and FP to the distributed case

- Rather, it is an informal extension of basic intuition and principles

- Plus I’m intentionally trolling a little, relax :-)

Microservices!

- Adopt!
- Abandon!
- Assumptions…



AWS Death Star diagram, circa 2008 as per Werner Vogels tweet https://twitter.com/Werner/status/741673514567143424

Object-Oriented

- Encapsulated data access via synchronous calls
- Mutable state change via synchronous calls
- Dependency web
- Imperative, sequence-oriented orchestration
- Referentially opaque

Functional Programming

- Data access via ubiquitous access to immutable values
- State change via mapping an identity to different immutable values over time
- Data Flow graph
- Declarative orchestration
- Referential transparency

Contrasting Principles

	Object Oriented	Functional
Data Access	Encapsulated	Ubiquitous
State Change	Mutable, in-place	Immutable, values over time
Organization	Dependency graph	Data-flow graph
Orchestration	Imperative, sequential	Declarative, parallelizable
Referentially…	Opaque	Transparent

Object-Oriented

- Might be fine within single memory space (let's talk…)

- **Does not scale well to distributed case!**

Distributed Objects

- Latency grows with depth of dependency graph
- Temporal liveness coupling
- Synchronous, pull-oriented by default
- Cascading failure modes
- Inconsistent, imperative orchestration
- Hidden narrative: can only see the nouns, verbs ephemeral
- Complexity: how to reason about state of system at any point in time?

Source: NASA, Public Domain

Deltas and Lambdas

- OO : Death Star :: FP : River Delta

- Functional Programming analogy scales ***well*** to the distributed case!

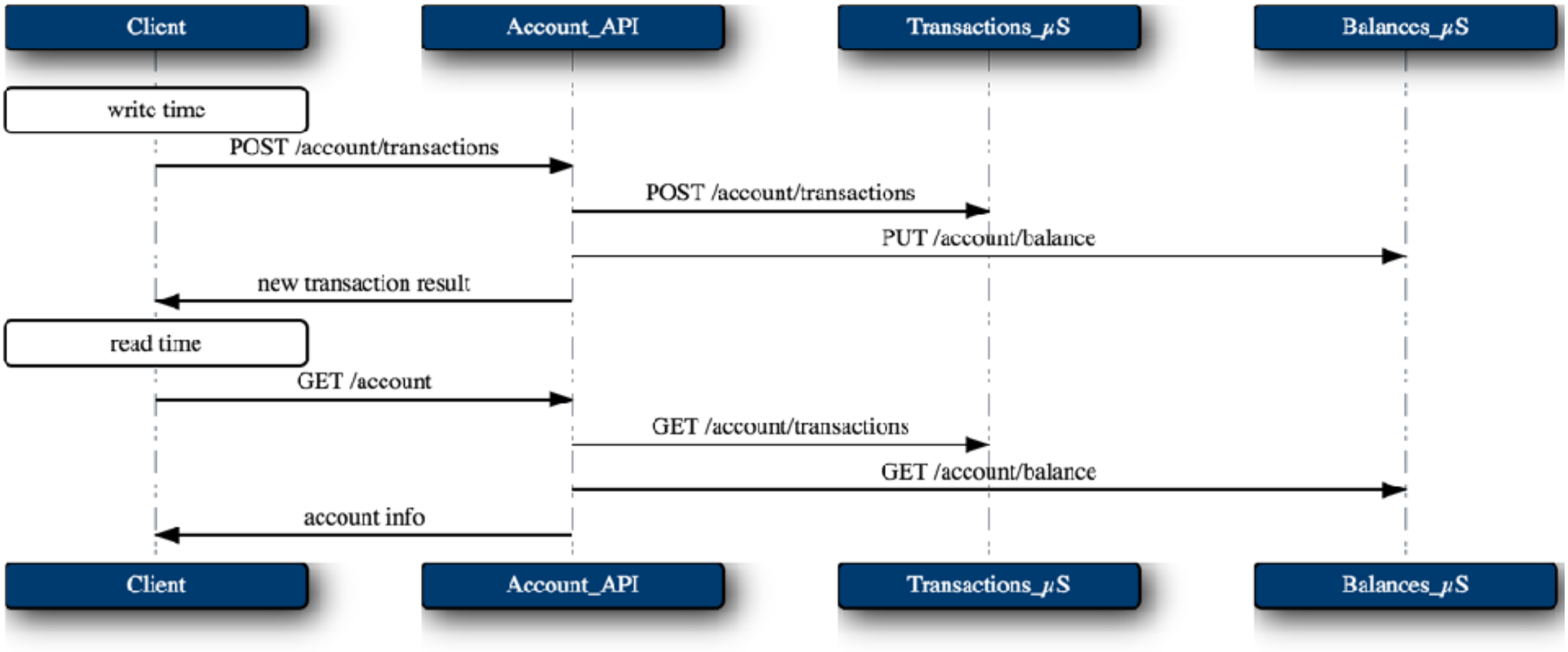
Distributed Functions

- Low latency at both read and write time (with eventual consistency in between)
- Temporal de-coupling
- Isolated failure modes
- (Eventually) consistent, declarative orchestration
- Reified narrative: event stream
- Clear reasoning about (and replay of!) state over time

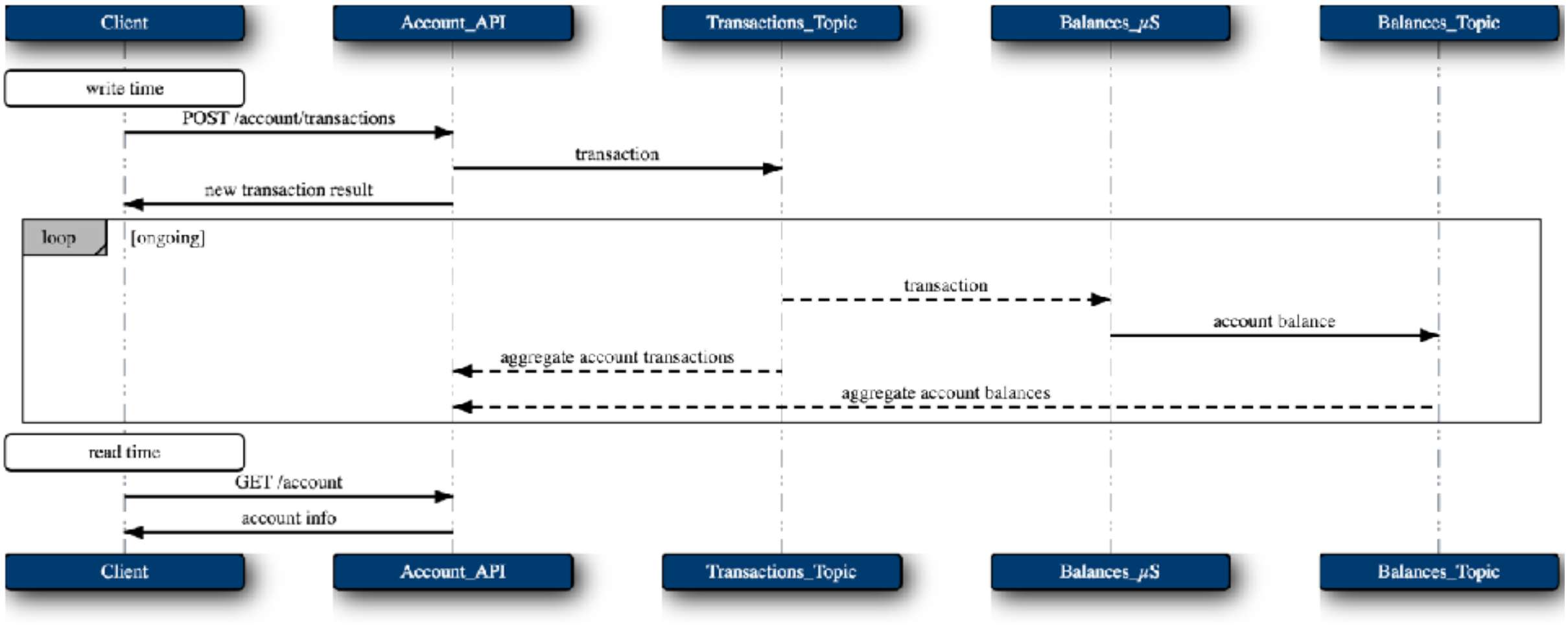
Example: Balance Calculation

- Customer-defined weekly spending limits/ notifications on a particular account
- Aggregate the debits, possibly emit event when balance exceeds limit for time period
- React to event in customer-defined way (prevent additional transactions until end of period, send notification, etc.)
- Allow customer to see transactions and balances

OO-Analogy µServices



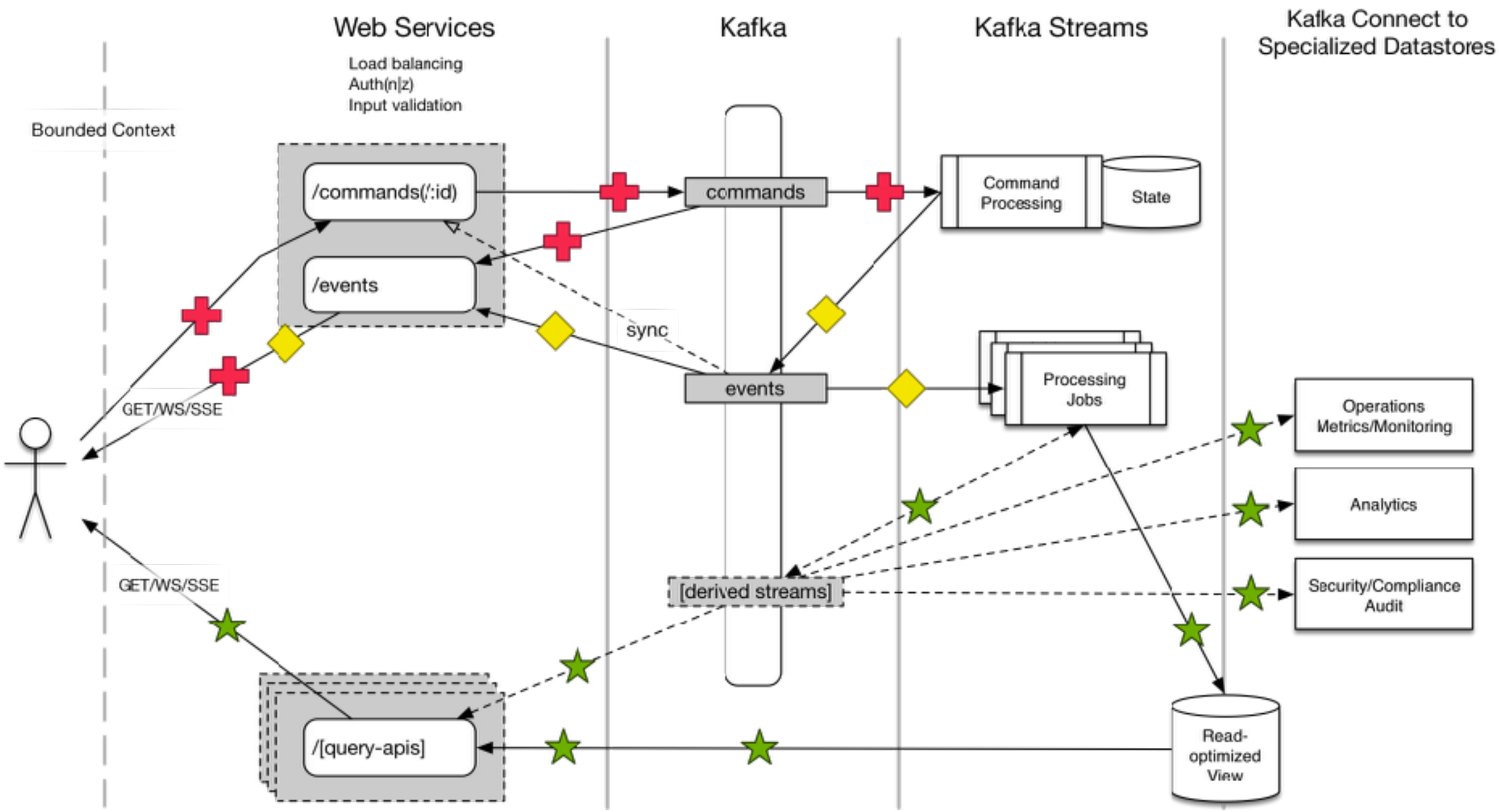
FP-Analogy μServices



But how?

- Architecture informed by
- Techniques
- Rules
- Tools

FP-Analogy μServices



Techniques

- REST (at the edge)
- CQRS
- Event Sourcing (storage)
- Pub/Sub (conveyance)
- Sagas
- Serverless

Rules

1. Capture all observations and changes at the edge (carefully!) to an immutable event stream

2. Reactively calculate streams of derived state from the event stream

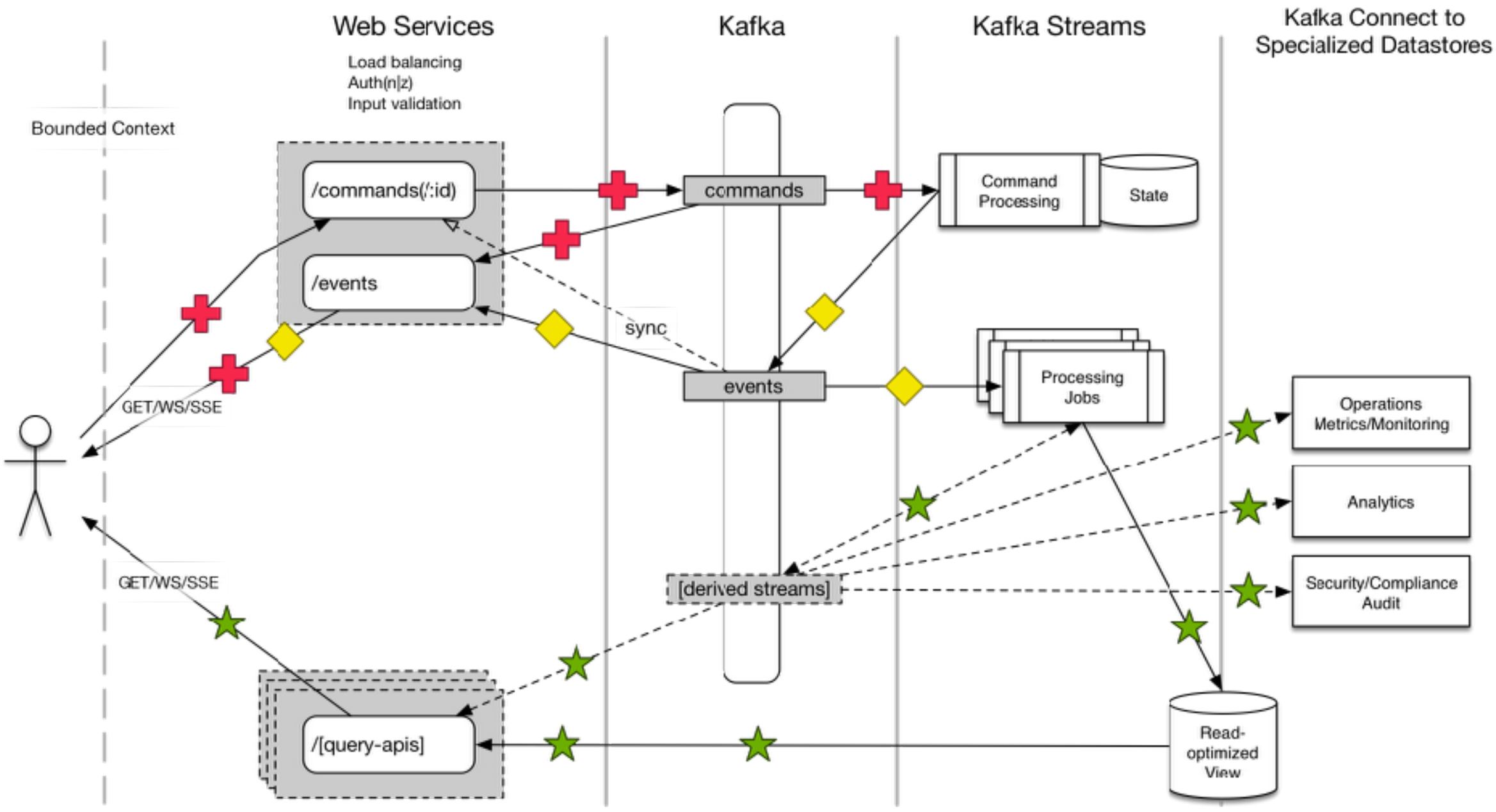
3. Aggregate state wherever and however it provides value

4. Manage outgoing reactions (“side-effects”) to state and events carefully

Capture changes at the edge to immutable event stream

- A (very) few authorized teams capture all raw observations (Events) and customer requests for action (Commands)
    - at the edge of bounded context
    - with minimal processing
    - immutably and durably
- Causally related events go on same log
- One (logical) writer per command/event stream
- No change gets into bounded context via any other means!

Capture changes at the edge to immutable event stream



Reactively calculate derived state from the event log

- A few authorized teams process Commands into Events (probably using aggregate state)

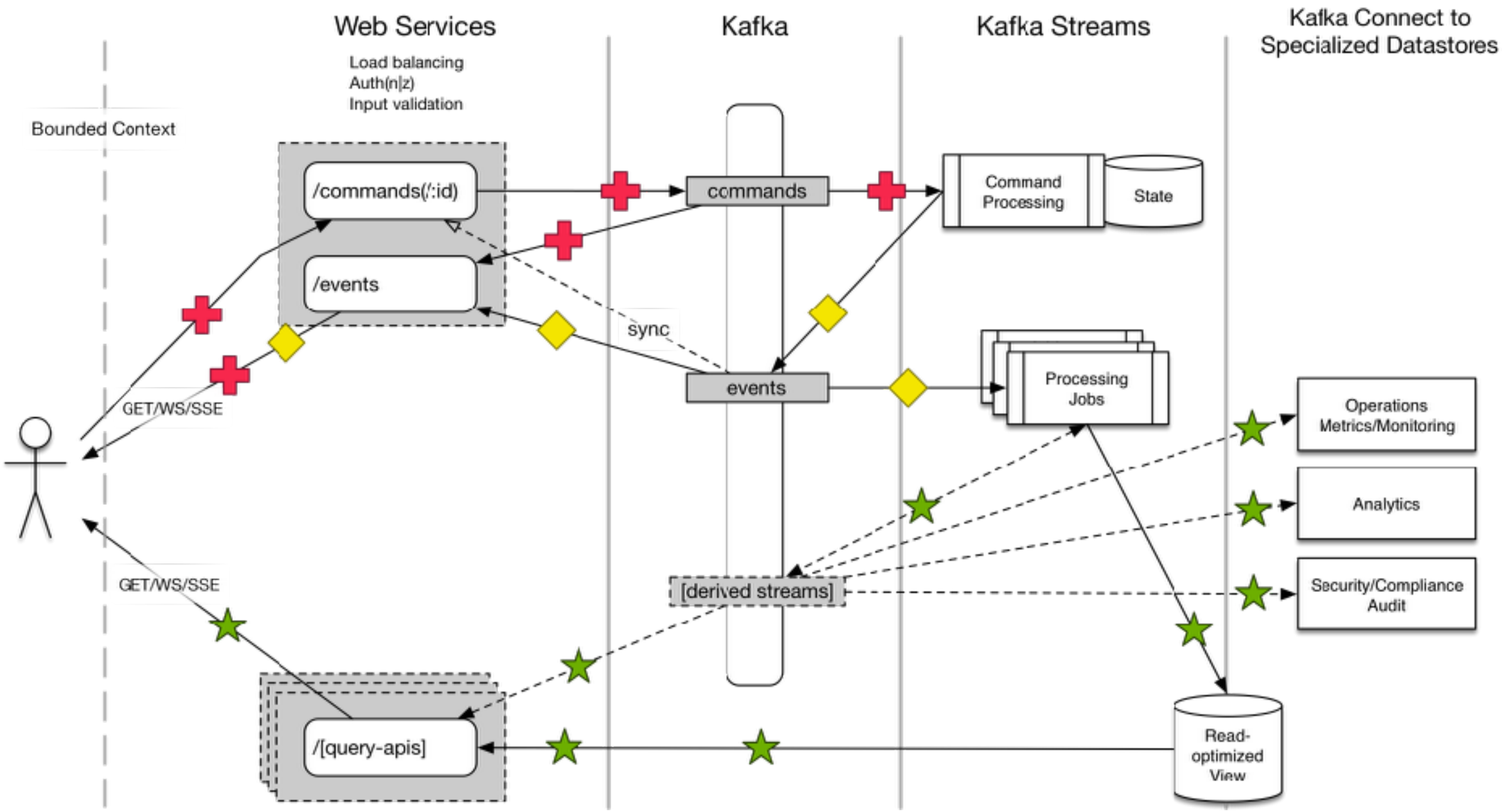
- A few (more) authorized teams calculate state streams of general interest derived from Events

    - Single-entity state changes

    - Regulated or audited calculations

- Could be recursive, i.e. certain state changes might trigger further events downstream

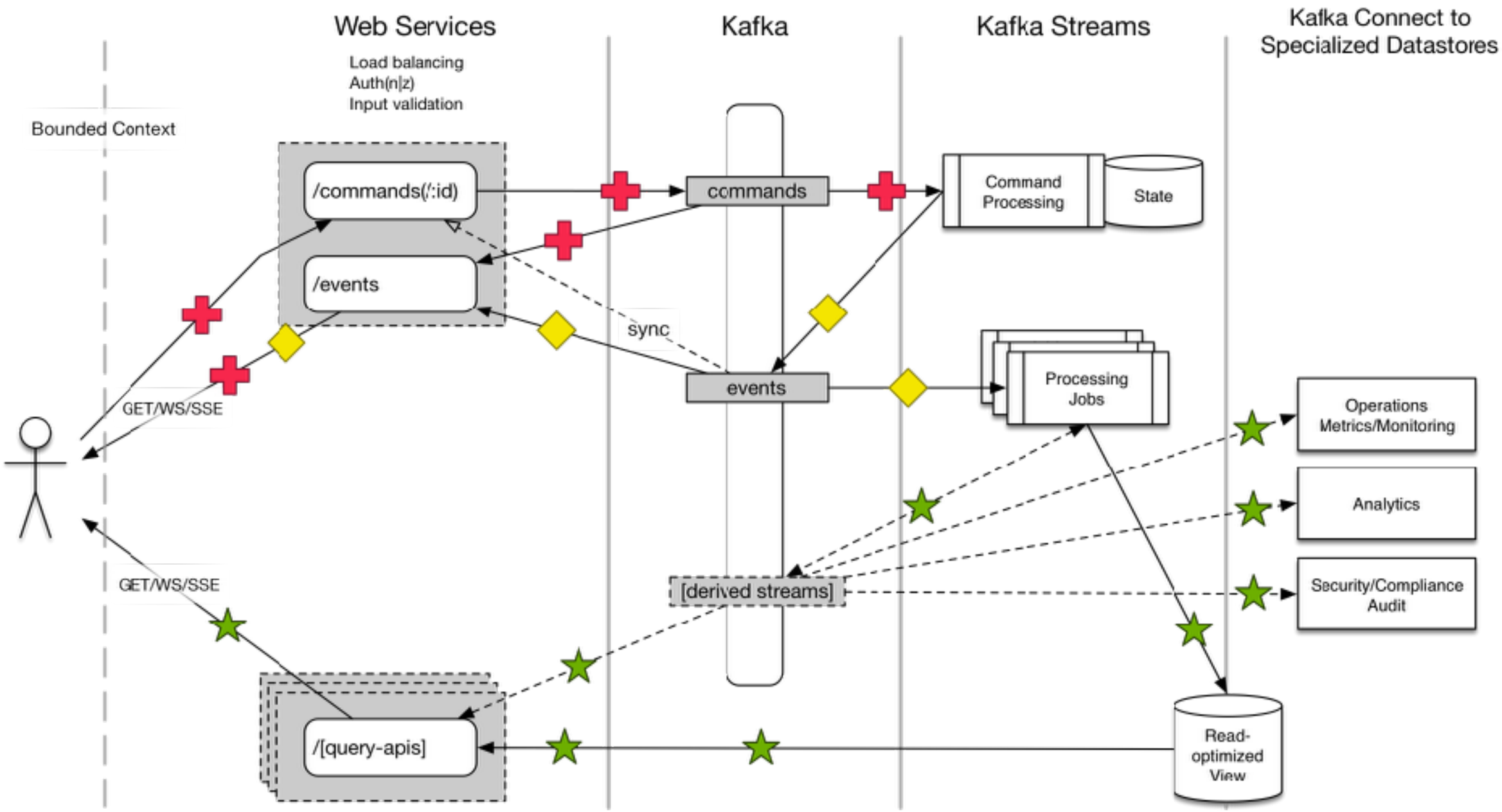
Reactively calculate derived state from the event log



Aggregate state wherever and however it provides value

- Many teams across orgs aggregate streams into views for their respective use-cases
    - Into whatever data store makes sense for required data access pattern
        - Possibly emitting views back onto streams (a la Kafka Streams' KTable)
- Facilitates stateful computations: joins, windowed aggregates, command processing
- Everyone traces lineage all the way back to edge Command/ Event

Aggregate state wherever and however it provides value



Manage “side-effects” carefully

- A few authorized teams react to Events by reaching outside of bounded context to cause “side-effects”
    - A “side-effect” in this case is any action not associated with reading or writing the streams or aggregates, e.g.
        - Sending email, text message
        - Making a call to an outside web service
        - Calling a service to write a command/event to a stream you don’t own
- Log results of attempt to stream to facilitate retry, Saga/rollback/ compensating action

Tools

- Kafka
- Kafka Streams
- Kafka Connect
- Various data stores
- Serverless integrations (e.g. OpenWhisk Kafka package)

Example

Questions?

References

- Sagas: https://www.cs.cornell.edu/andru/cs711/2002fa/reading/sagas.pdf
- Ben Stopford on Events/Microservices: https://www.confluent.io/blog/build-services-backbone-events/
- Rich Hickey:
  - Clojure's approach to state: https://clojure.org/about/state
  - The Value of Values: https://www.infoq.com/presentations/Value-Values
  - The Language of the System: https://www.youtube.com/watch?v=ROor6_NGIWU