Commander

Better Distributed Applications through CQRS, Event Sourcing, and Immutable Logs

Hi, I’m Bobby

I’m on the Technology Fellow’s team at 

I dislike accidental complexity

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

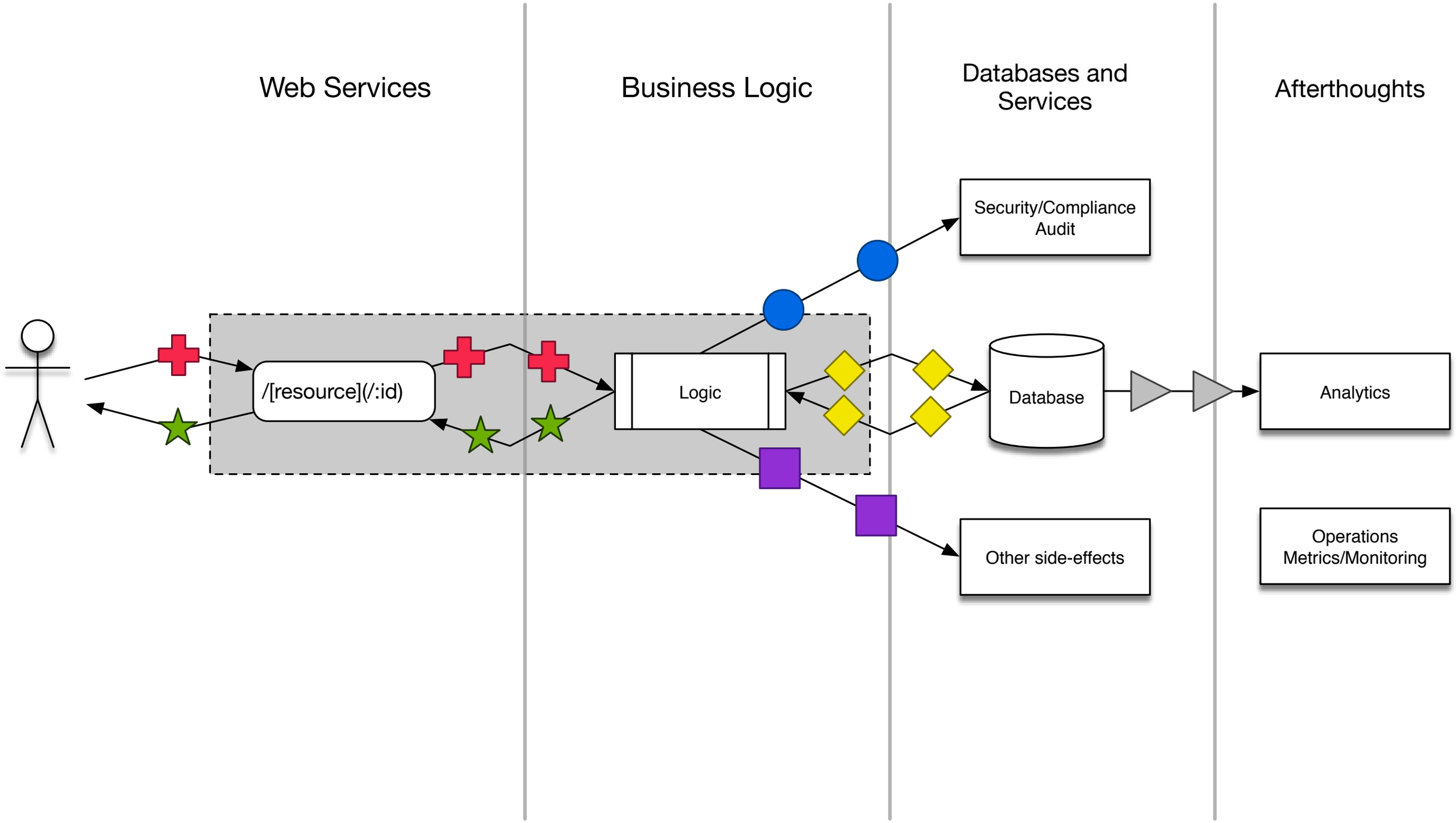
Problem Space

- Provide valuable informational and transactional services via Mobile and Web software
- To ***lots*** of customers, with excellent user experience
- Securely and in compliance with regulations
- With ability to easily enhance, experiment, monitor, maintain, and operate
- By many participants within a large organization

Big Ideas

- Immutability is central to information systems
- Data language of system >> Programming language of components
- Action and perception are not the same, and immutability facilitates their separation
- Businesses services are not databases, they're event stream reactors
- Cross-cutting concerns must be satisfied in the presence of Conway's Law

Problematic Architecture



Immutability is central to information systems

Event Sourcing Analogy

Image by Alan Light CC BY-SA 3.0

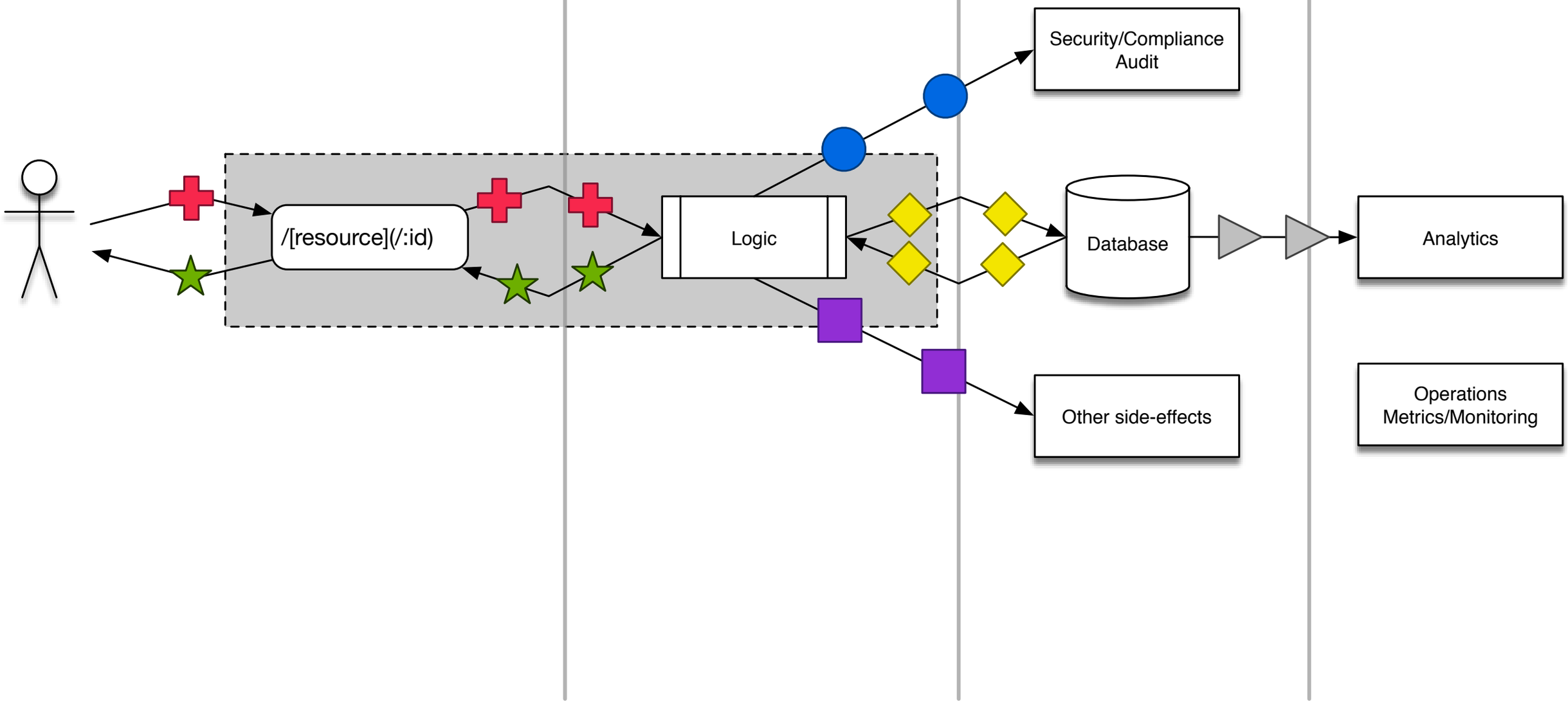
Data Loss by Design

Web Services

Business Logic

Databases and Services

Afterthoughts



The (data) language of the System

>>

The (runtime) language of each component

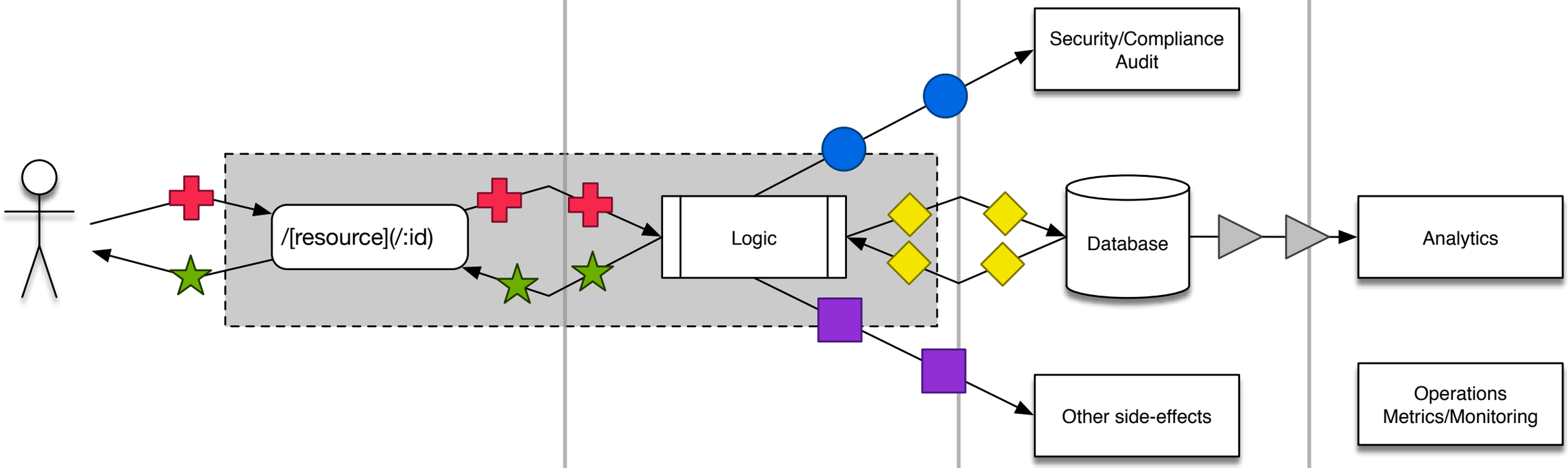
What is system language?

Web Services

Business Logic

Databases and Services

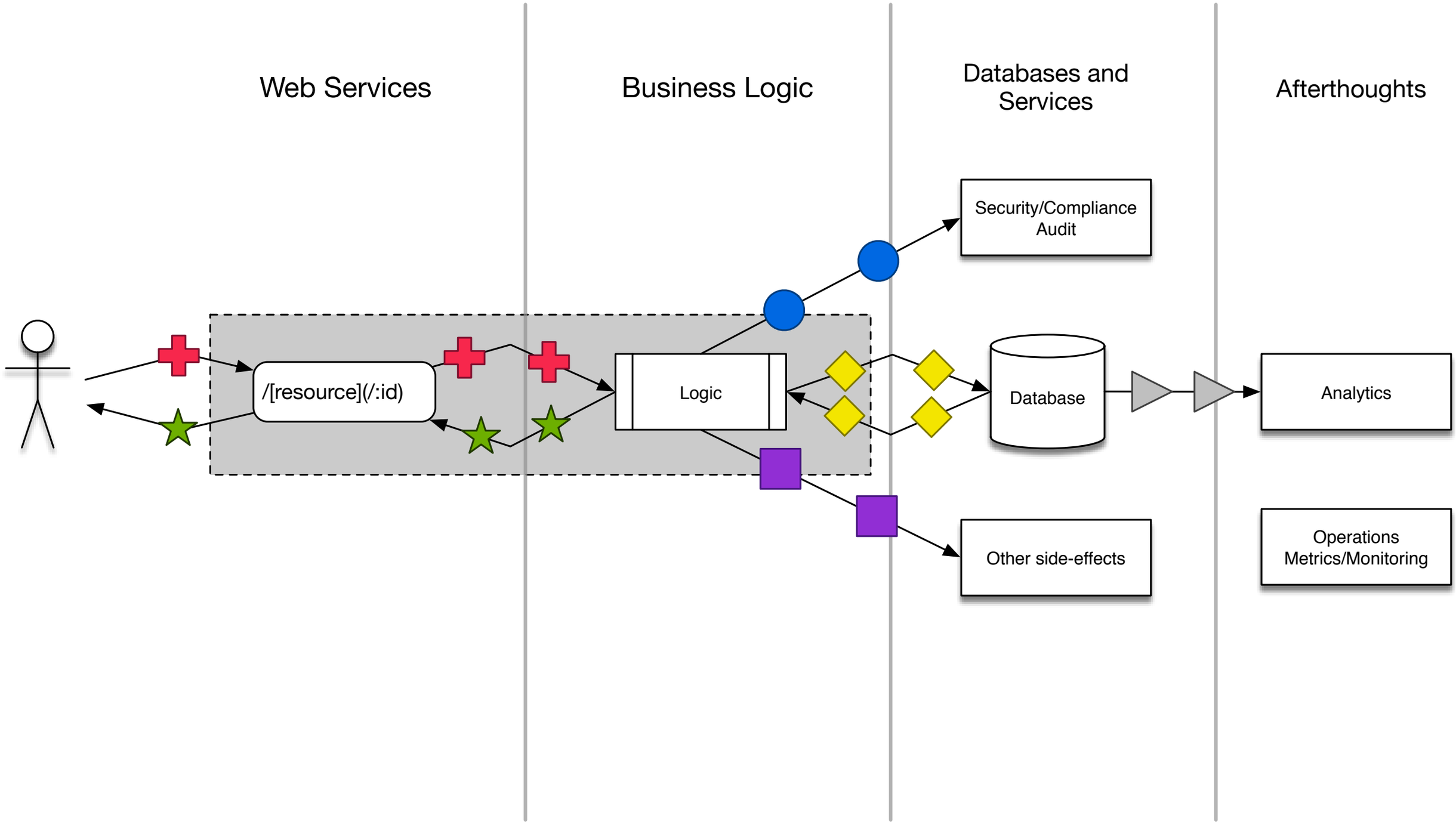
Afterthoughts



Action != Perception

Writes != Reads

Writes Tied to Reads



Business Services are ***not*** Databases

“We shape our tools and thereafter our tools shape us.”

–John M. Culkin

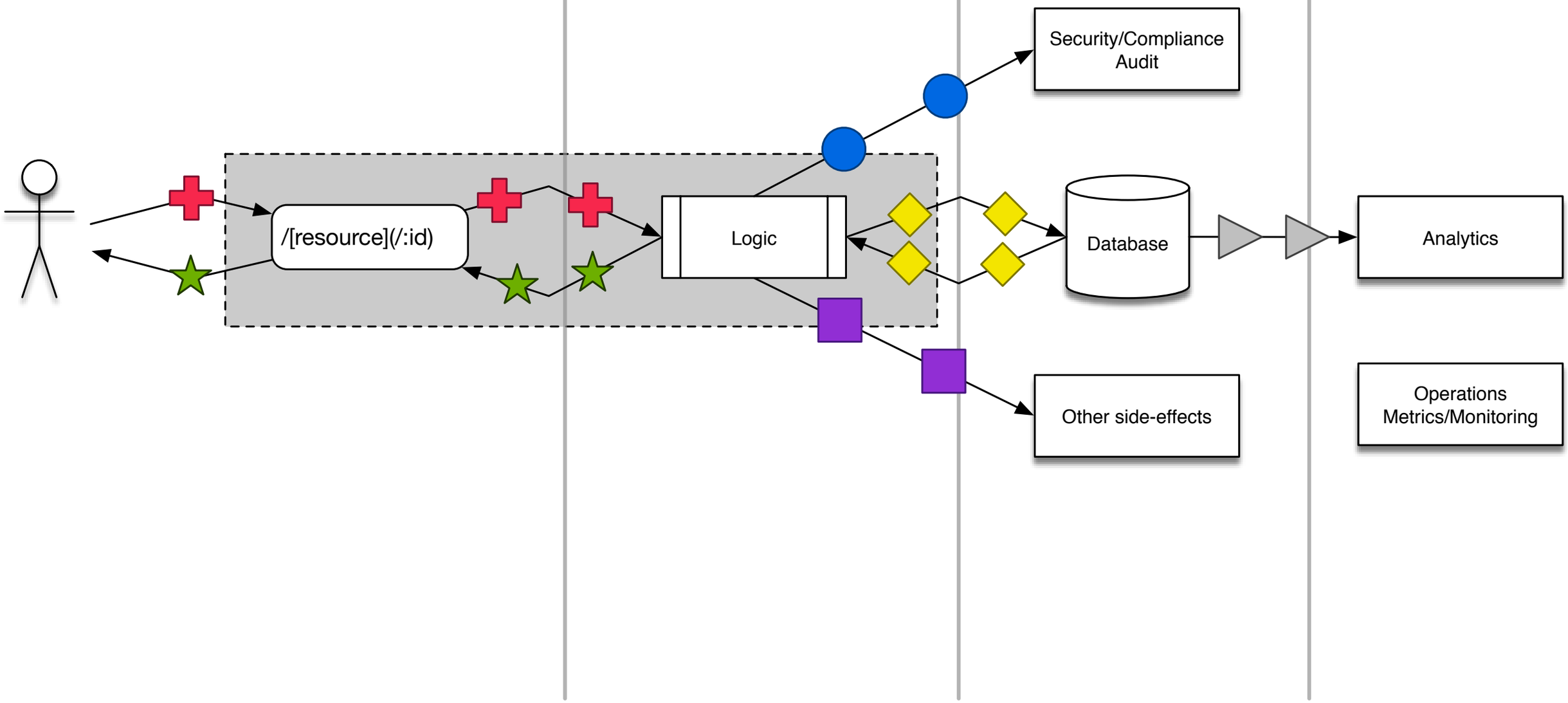
Database Leaking

Web Services

Business Logic

Databases and Services

Afterthoughts



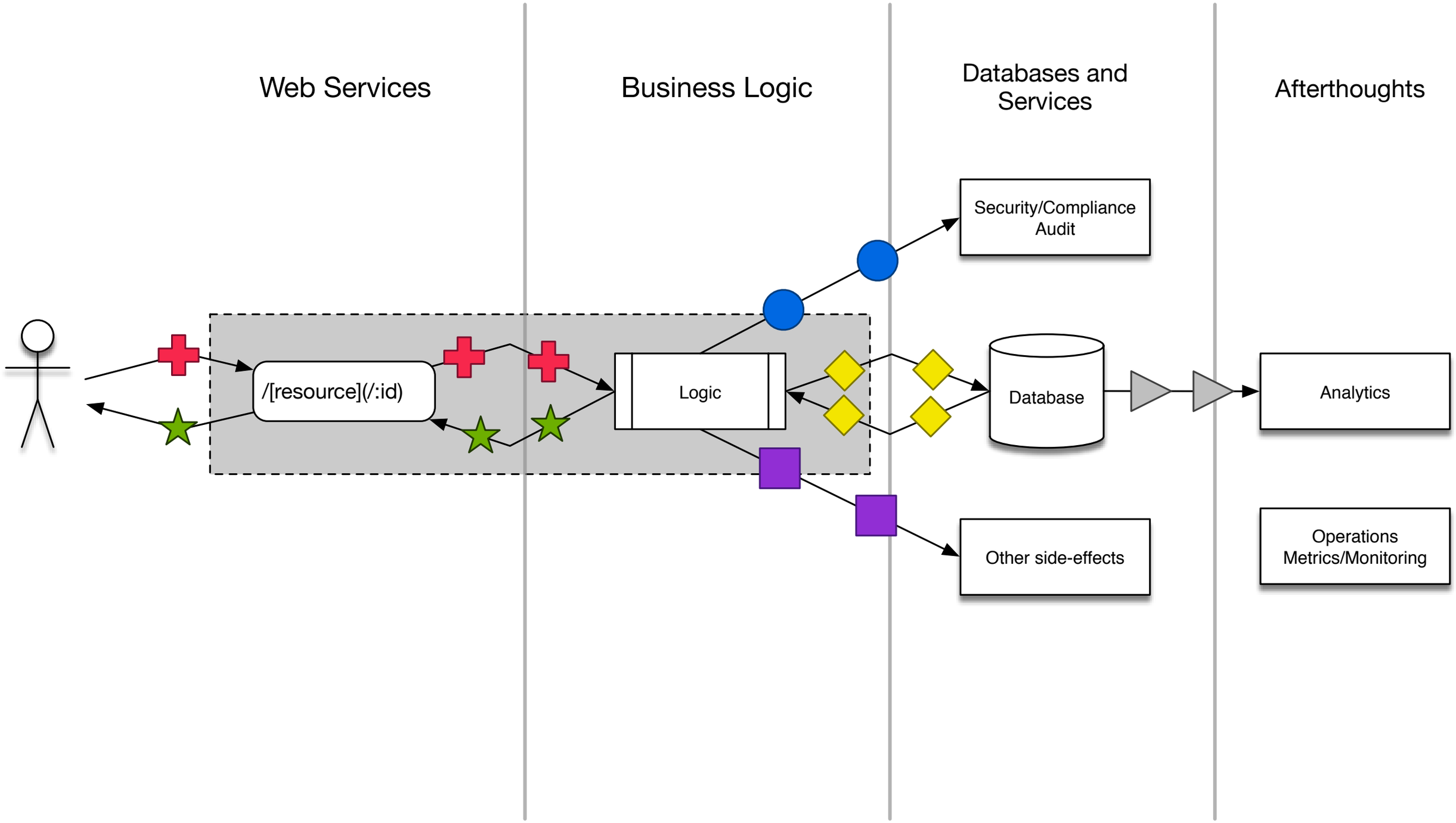
Conway’s Law

Yup, totally a thing

“organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations”

–Melvin Conway

Cross-cutting Concerns?



We can do better!

Commander

- A better architecture for APIs and services using REST + Immutable Event Log + Reactive Event Stream Processing

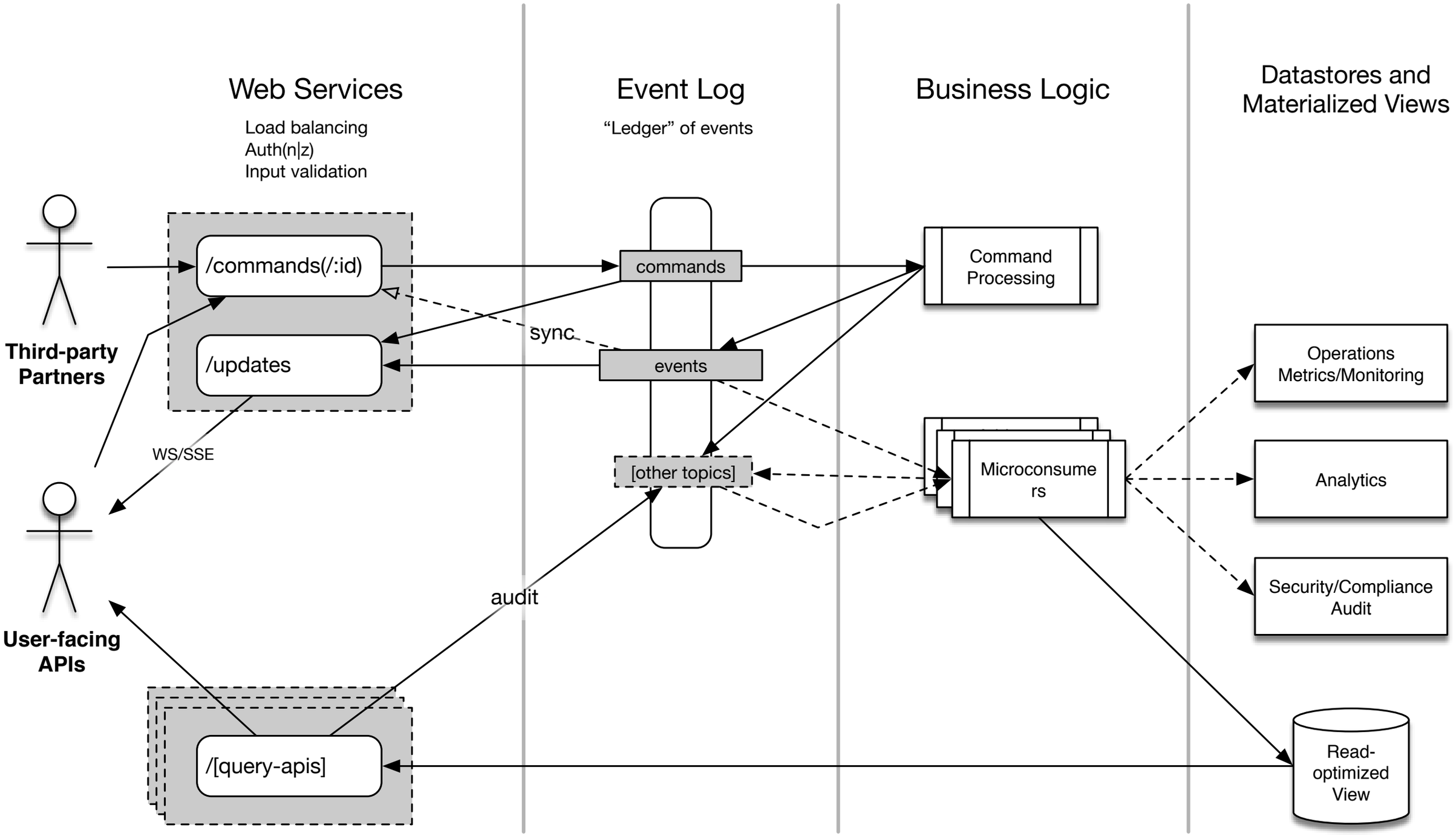
- The write-handling component of that architecture, my implementation is in Clojure

Commander Architecture

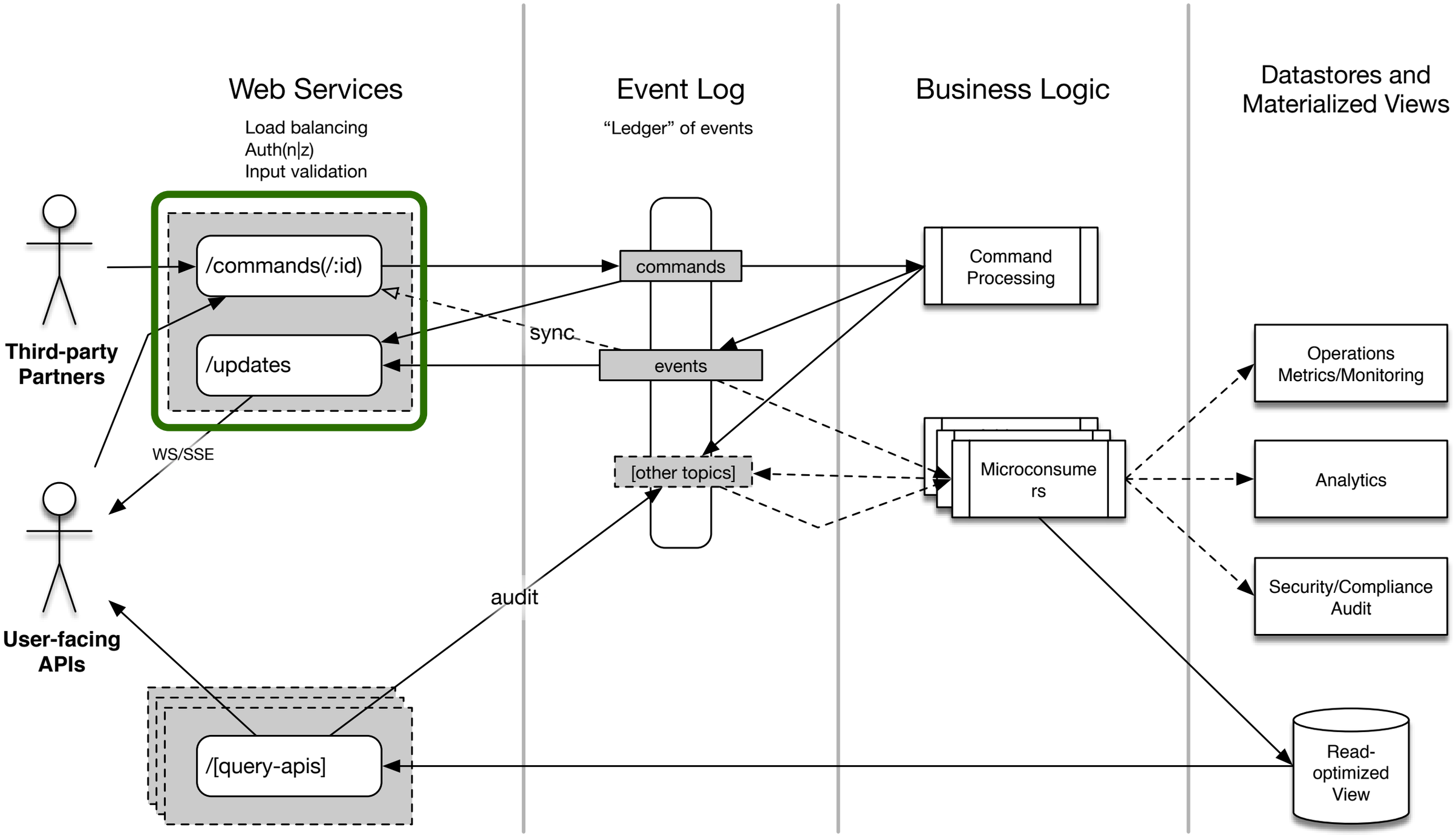
- Several categories of microservices with structured interactions among them

- REST + CQRS + Event Sourcing + Reactive Event Stream Processing

Commander Architecture

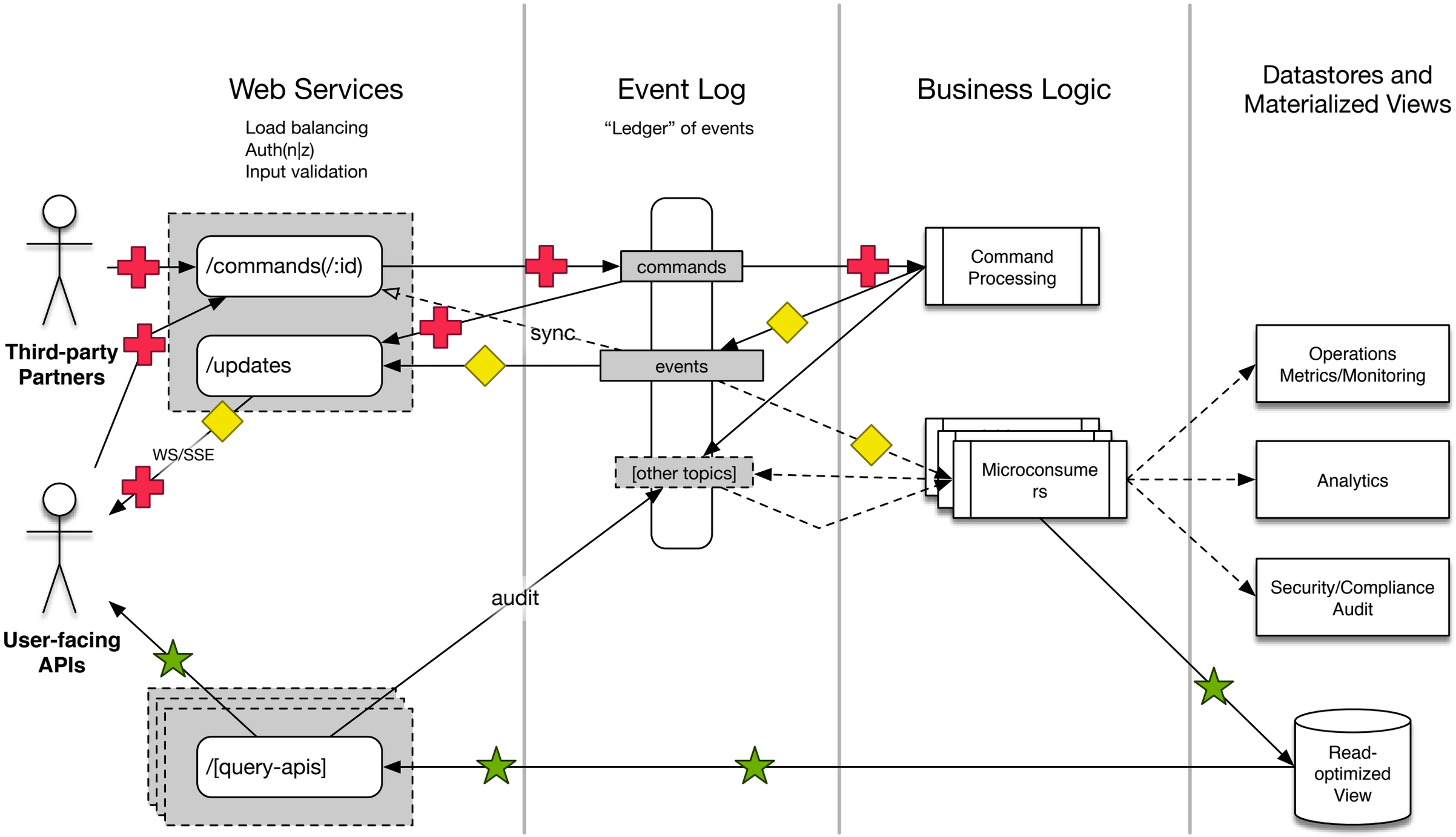


Commander Component



Embrace Immutability

Immutable Data Log



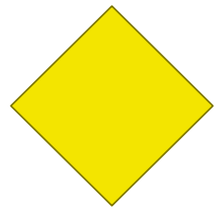
Express actions in domain language

(not in database language)

A Command 


```
{"id": "33bb75db-6e13-48ee-8a54-b3976d3d065b",
 "action": "transfer-money",
 "data": {"from_account": "12345",
          "to_account": "54321",
          "amount": 10000}
 "timestamp": "2016-05-20T14:33:28.902-00:00"}
```

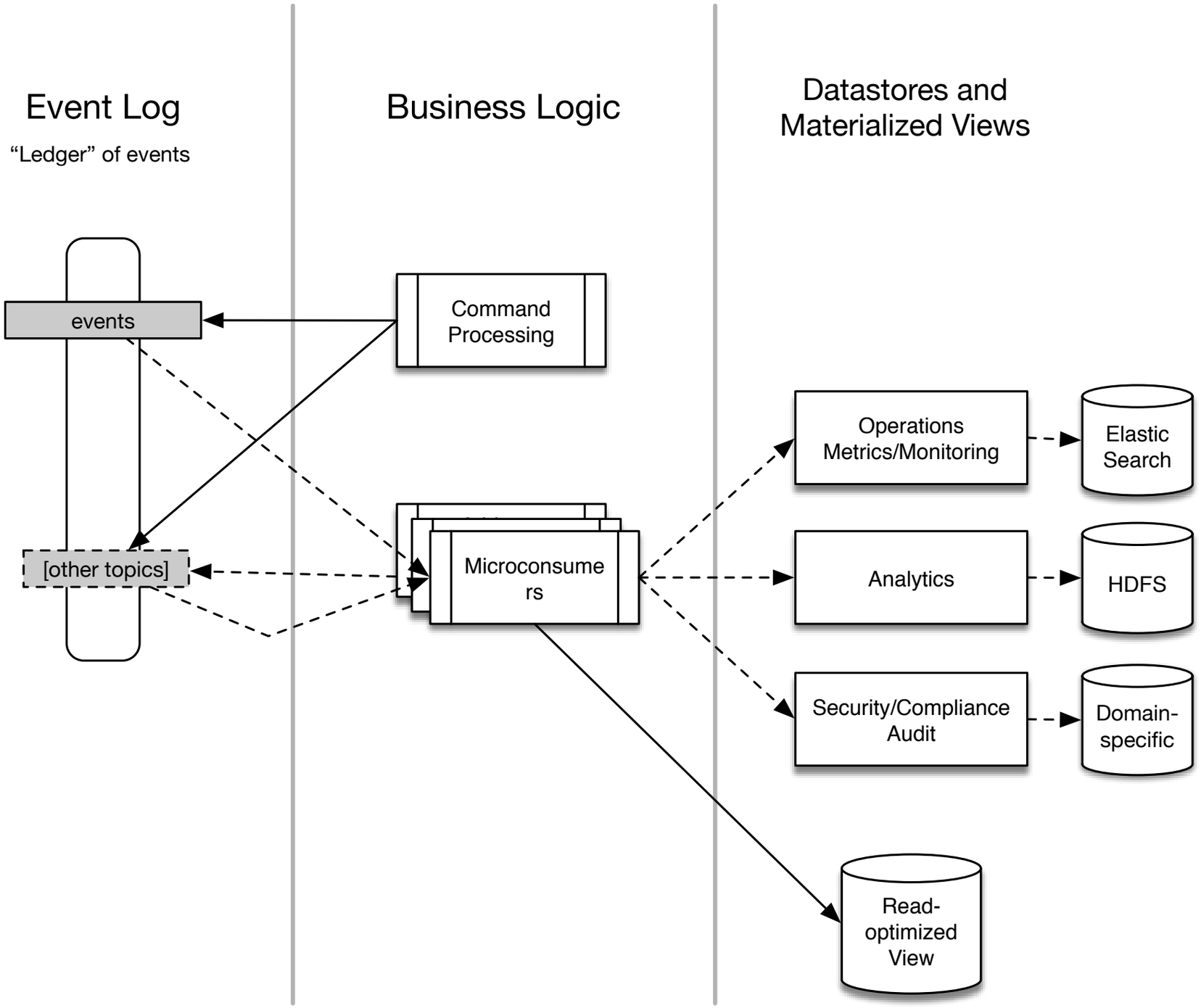
An Event



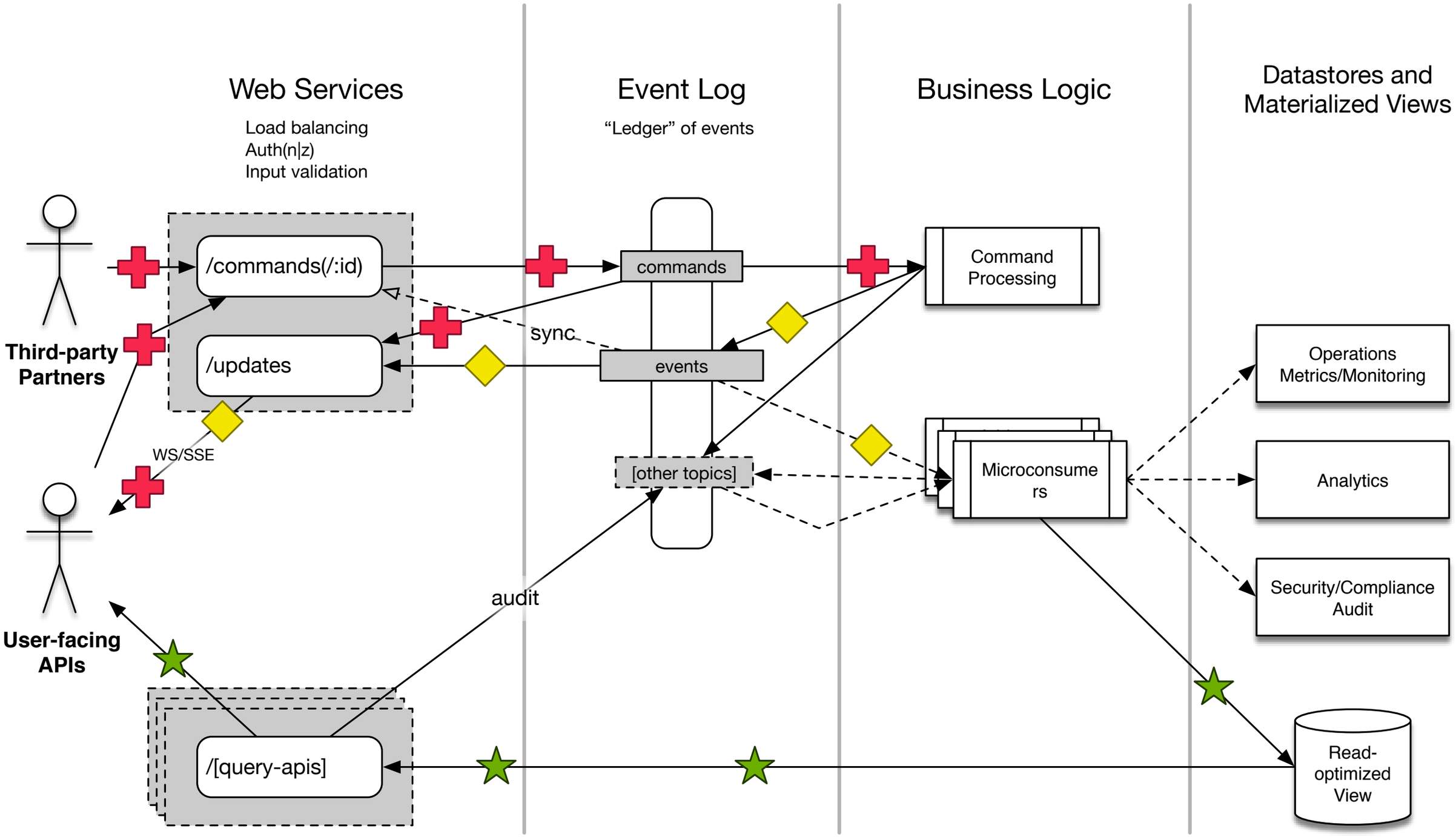

```
{"id": "d435ed18-4ff7-4cae-a21b-3adb7b06fe58",
 "parent": "33bb75db-6e13-48ee-8a54-b3976d3d065b",
"action": "money-transferred",
"data":    {"id": "a6b903f6-0b9c-4c5b-95fa-afd4cc3bf938",
            "from_account": "12345",
            "to_account": "54321",
            "amount": 10000},
"timestamp": "2016-05-20T14:33:28.904-00:00"}
```

Separate Action from Perception

1 Log => n Data Views

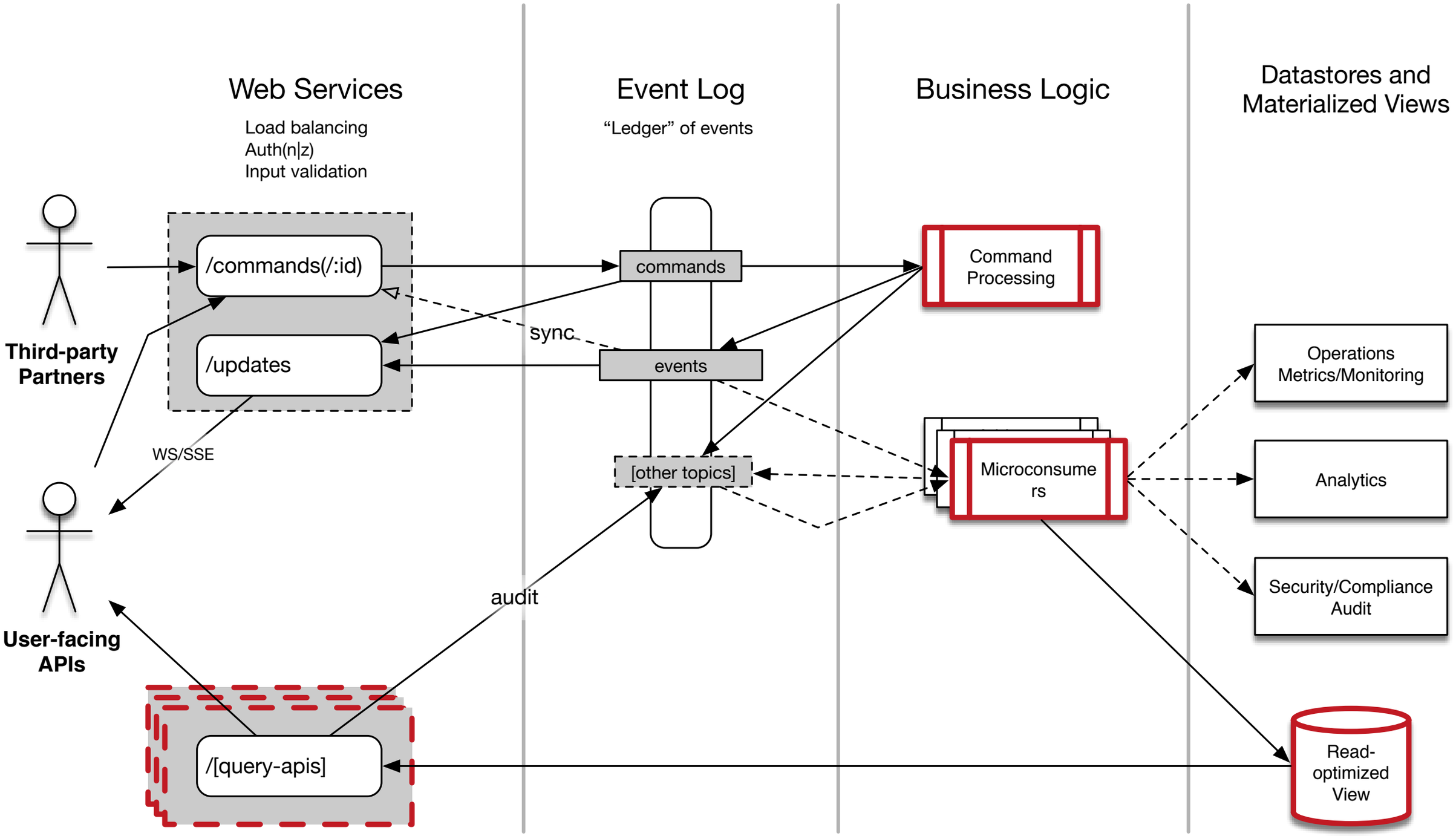


Commander Architecture

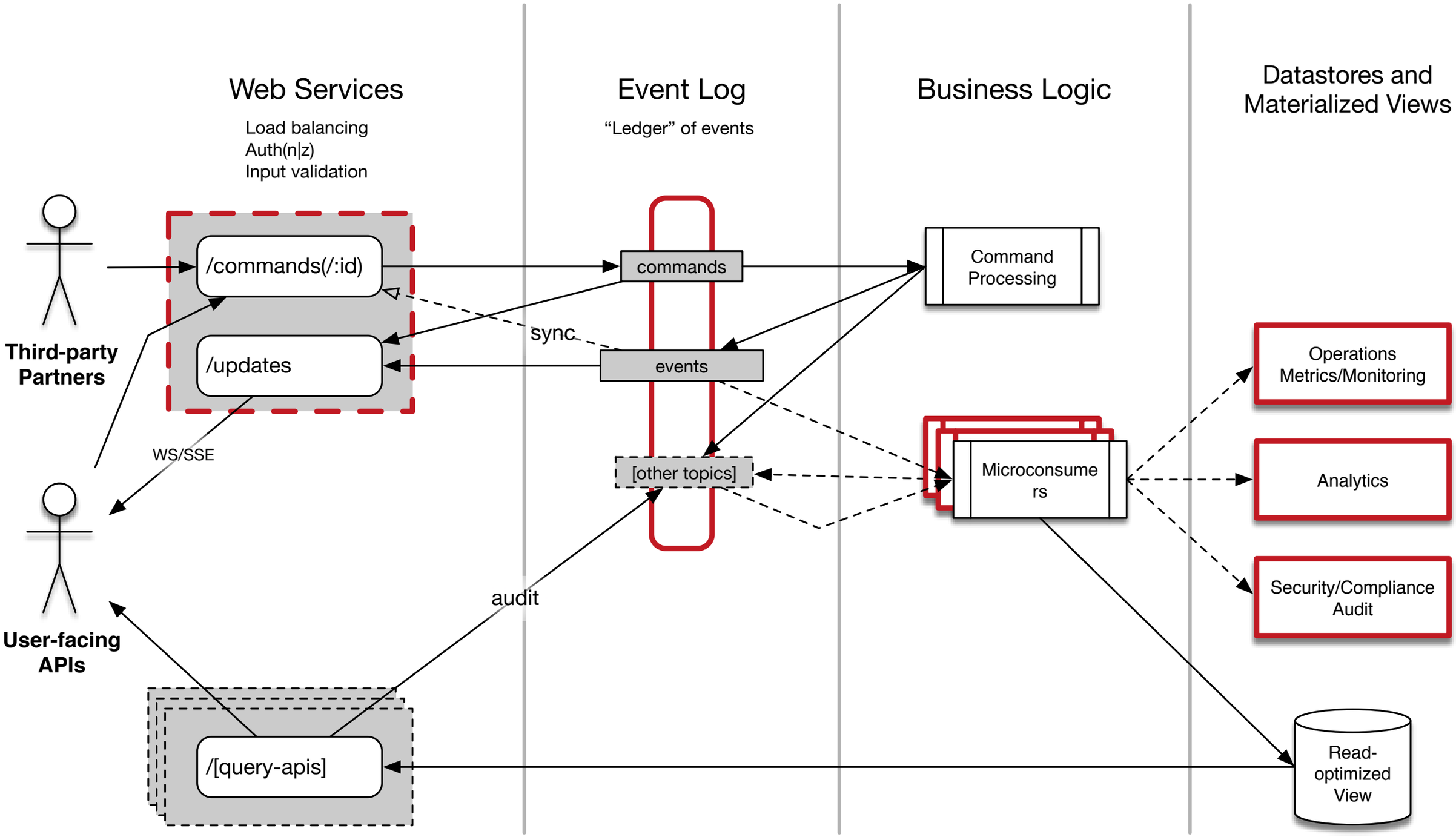


Exploit Conway’s Law

Primary Team Provides



Enterprise Provides



Why Commander Component?

- Single writer to commands topic
- Ensuring schema conformance
- Indexing all Commands and Events for reads and server-push
- Provides optional illusion of synchrony to clients

How to implement Reactive Services?

Kafka Streams!

What is Kafka?

- Apache Kafka is publish-subscribe messaging rethought as a distributed commit log

- But it's not really about messaging, that's just the interface

- Logs > Messages for my domain

- **It provides distributed, immutable logs!**

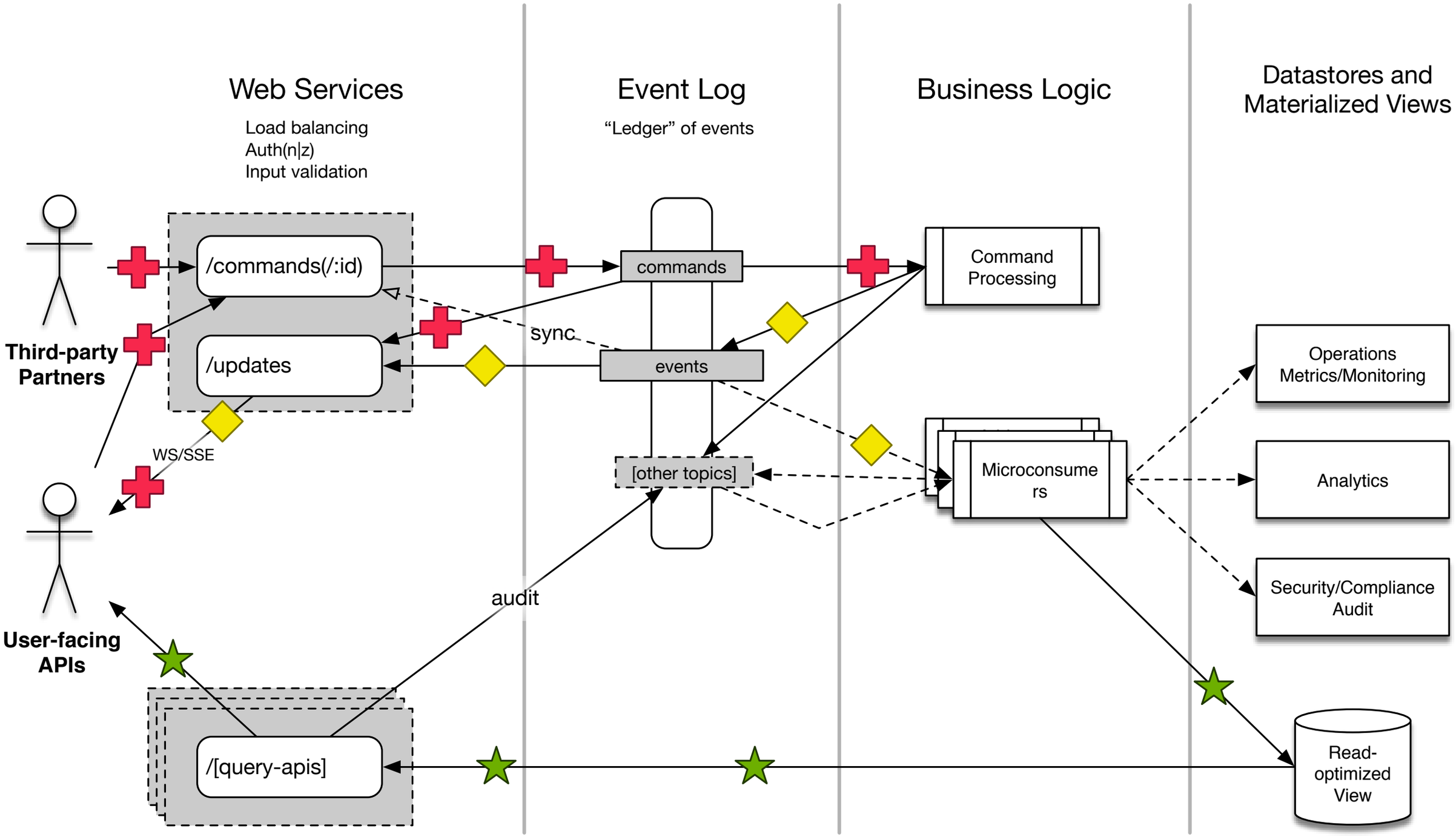
What is Kafka Streams?

- A Java library for building streaming applications on top of Kafka, lives in your application
- Low-level API for building topologies of processors, streams, and tables
- High-level DSL for common patterns like filter, map, aggregations, joins, stateful and stateless processing
- Nice operational characteristics (low latency, elastic, fault-tolerant)

How to use Kafka Streams within Commander

- Implement Command Processor
- Implement Event consumers and producers
- Provide local state management as backend for APIs

Commander Architecture



```
KStreamBuilder builder = new KStreamBuilder();

KStream<UUID, Map> commands = builder.stream(commandsTopic);
KStream<UUID, Map> customerEvents = commands
    .filter((id, command) -> command.get("action")
                                    .equals("create-customer"))
    .map((id, command) -> {
            Map userEvent = new HashMap(command);
            userEvent.put("action", "customer-created");
            userEvent.put("parent", id);
            Map userValue = (Map) userEvent.get("data");
            userValue.put("id", UUID.randomUUID());
            return new KeyValue<>(UUID.randomUUID(), userEvent);
        }).through(eventsTopic);

KStream<UUID, Map> customers = customerEvents
    .map((id, event) -> {
            Map customer = (Map) event.get("data");
            UUID customerId = (UUID) customer.get("id");
            return new KeyValue<UUID, Map>(customerId, customer);
        });

customers.through(customersTopic);

StateStoreSupplier store = Stores.create("Customers")
    .persistent()
    .build();
builder.addStateStore(store);

customers.process(customerStore, "Customers");

this.kafkaStreams = new KafkaStreams(builder, kafkaStreamsConfig);
this.kafkaStreams.start();
```

```
public class CustomerStore implements Processor<UUID, Map> {
    private KeyValueStore<UUID, Map> store;

    public List<Customer> getCustomers() {
        List<Customer> customers = new ArrayList<>();
        KeyValueIterator<UUID, Map> iterator = store.all();
        while (iterator.hasNext()) {
            KeyValue<UUID, Map> entry = iterator.next();
            customers.add(new Customer(entry.value));
        }
        iterator.close();
        return customers;
    }

    public Customer getCustomer(UUID id) {
        return new Customer(store.get(id));
    }

    @Override
    public void init(ProcessorContext processorContext) {
        this.store = (KeyValueStore<UUID, Map>) context.getStateStore("Customers");
    }

    @Override
    public void process(UUID uuid, Map map) {
        store.put(uuid, map);
    }

    @Override
    public void punctuate(long l) {}
    @Override
    public void close() {}
}
```

Demo

Summary

- Capture customer intent and business events as immutable data ***in domain language***

- From these action streams, services implement their own functionality in this common lingua franca

- ***building many independent data views***

- **reactively**

- **without temporal or organizational coordination**

Giant Shoulders

- **Immutability:** Rich Hickey, Stu Halloway
- **CQRS:** Udi Dahan, Martin Fowler, Chris Richardson
- **Kafka Event Stream Reactors:** Neha Narkhede, Jay Kreps, Martin Kleppmann
- **Organization and Management:** Mel Conway, Eliyahu Goldratt, Gene Kim, Michael Nygard

References

- **https://tinyurl.com/capital-one-cmdr**
- http://www.datomic.com/
- http://blog.cognitect.com/?tag=NewNormal+Series
- http://www.confluent.io/blog/event-sourcing-cqrs-stream-processing-apache-kafka-whats-connection/
- https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying
- https://www.infoq.com/presentations/Value-Values