The use of in-memory computing on an epic journey of moving a legacy monolithic system to microservices architecture: A case study of Storage Area Network Management System

Aruna Sangli

11/13/2019

Agenda

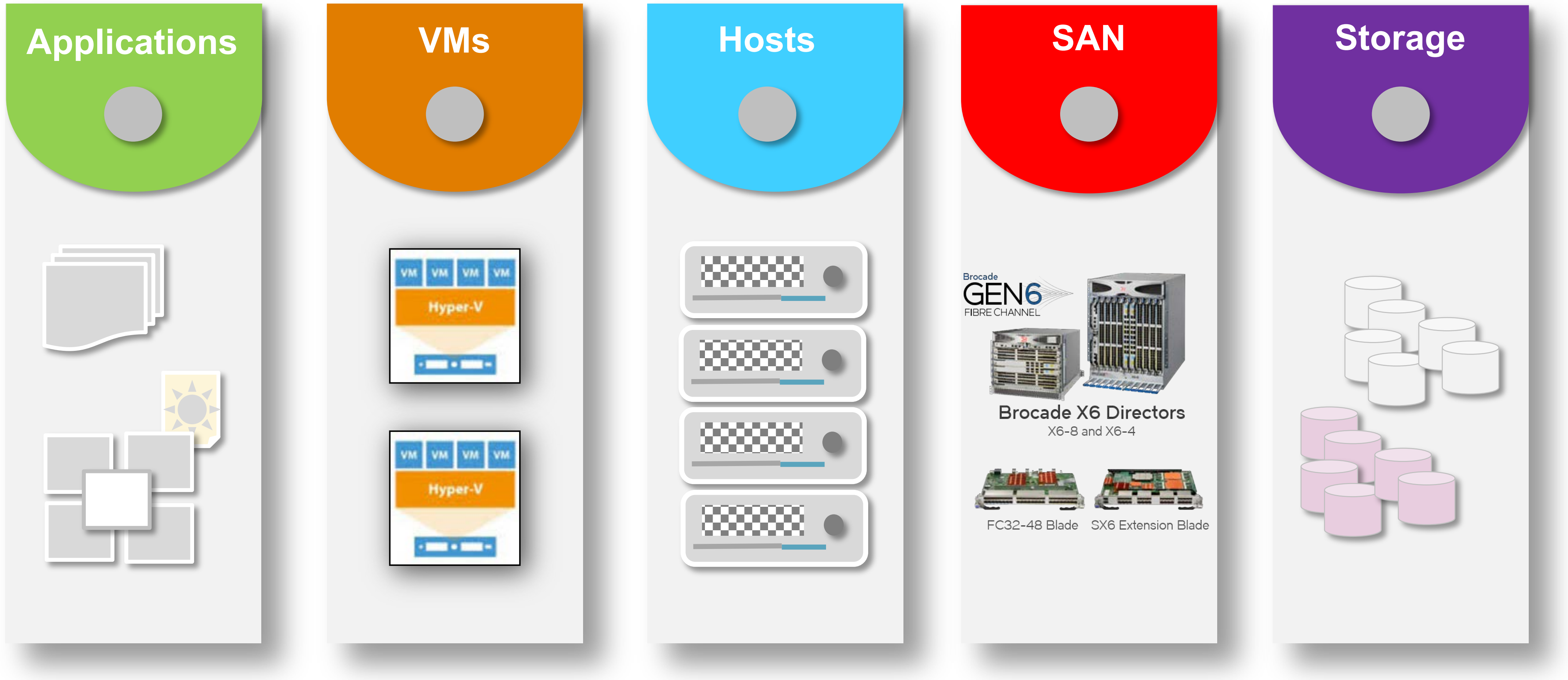
- SAN Management System Overview
- Background
- Use of In-Memory computing
  - Why
  - Apache Ignite integration
  - Use cases and patterns

SAN Management System Overview

Inside data center

Contd.

SAN Management Entities



Applications

VMs

Hosts

SAN

Storage

Brocade X6 Directors

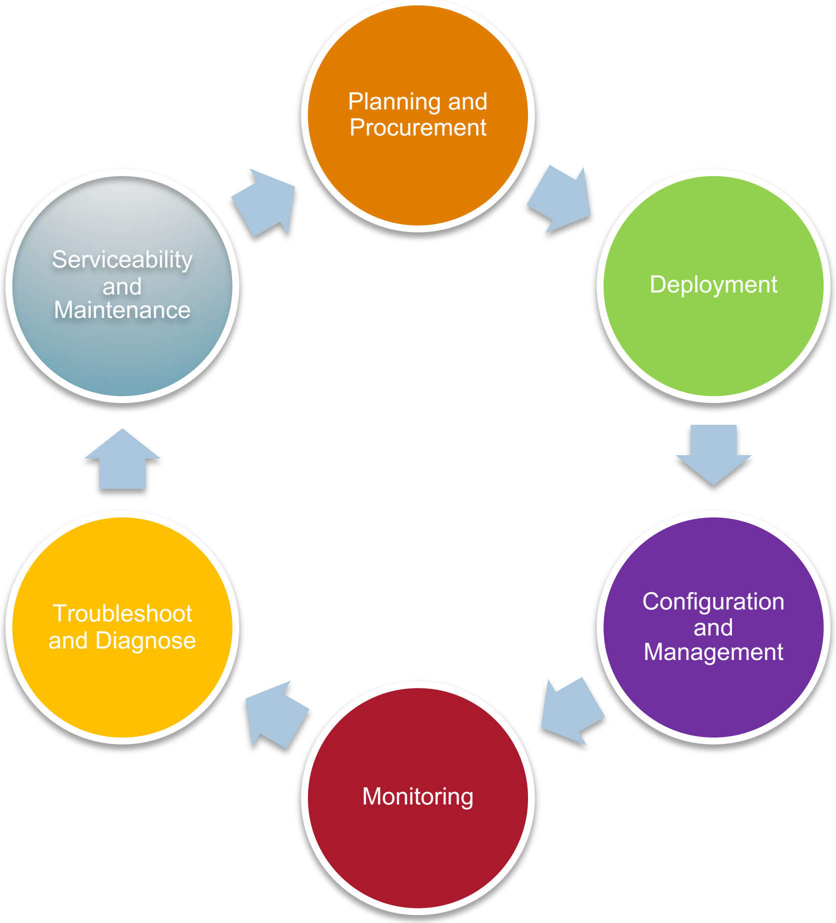
X6-8 and X6-4

FC32-48 Blade

SX6 Extension Blade

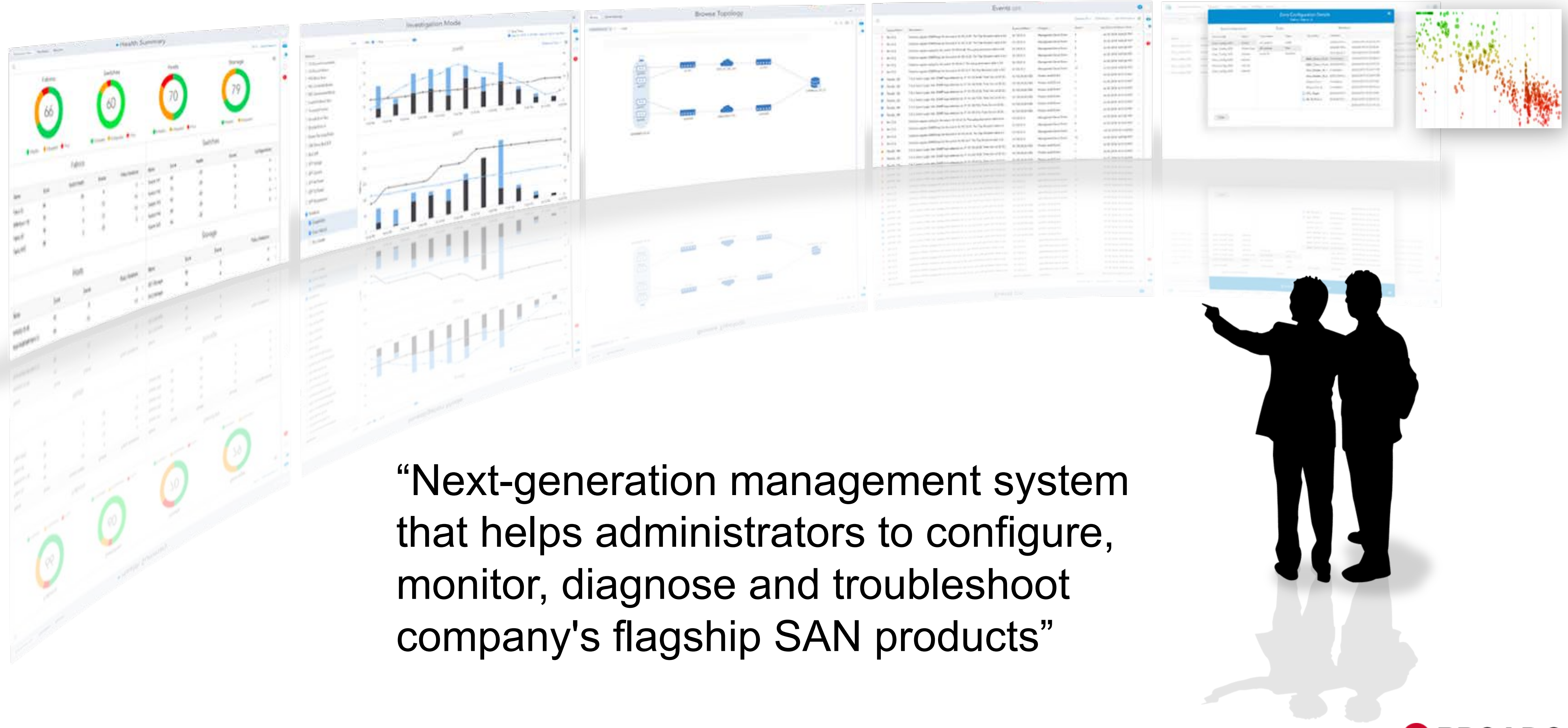
Contd.

Life cycle of deploying and managing SANs


Planning and Procurement
Deployment
Configuration and Management
Monitoring
Troubleshoot and Diagnose
Serviceability and Maintenance

Contd.

SANnav Management Portal



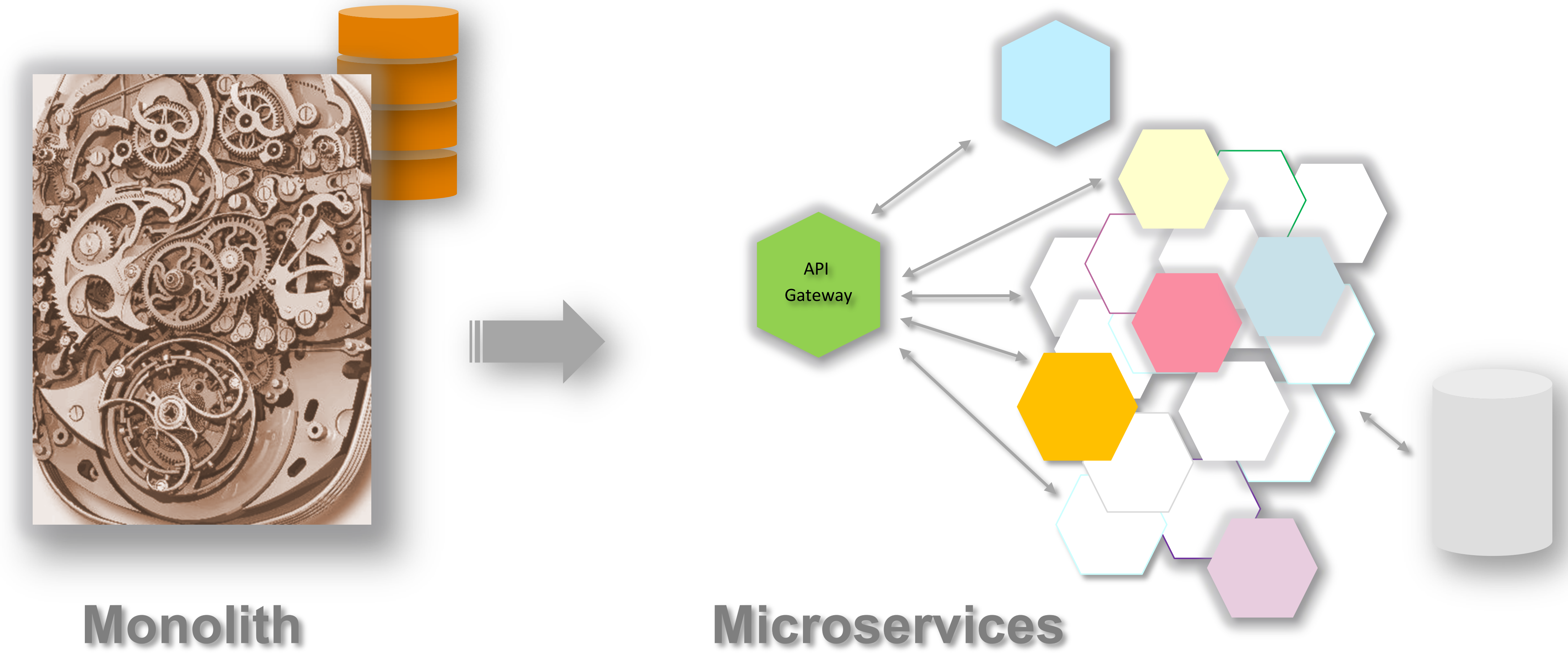
“Next-generation management system that helps administrators to configure, monitor, diagnose and troubleshoot company's flagship SAN products”

Background

- Over the years, we have built a Java and JBoss based network management system to configure and monitor the company's flagship SAN products.
- The product has continuously evolved with increasing complexity as we have added new features.
- There were challenges over time with this evolution: the product could not scale or we could add customer requested features without cost and maintenance issues.
- In order to stay ahead of the competition, re-architecting the product became a necessity.
- We started the journey of migrating to microservices architecture with flexibility, scalability and performance in mind.

Contd.

Legacy Monolith to Modernize Microservices Architecture



Contd.

Technology and Software Stack

- Microservices
- Docker and Docker Swarm
- Spring Boot and Spring Data
- Kafka
- PostgreSQL
- Apache Ignite
- Elastic Search

- Rest API
- Swagger
- Websockets
- React JS/Flux
- HighCharts
- D3

- Logstash
- Secured Syslog
- SNMP4j

Use of In-Memory Computing in SANnav Management Portal

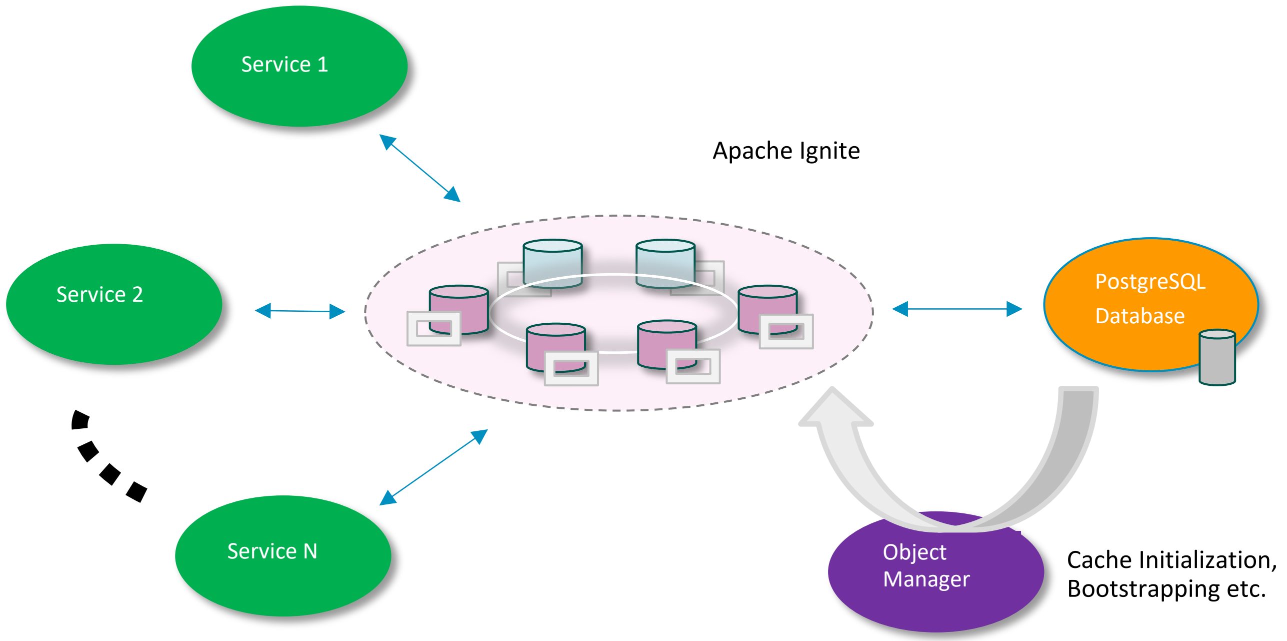
Note : Please note some of the ideas/concepts presented here are currently being used in the product or work in progress or will be incorporated in future release of the product.

BROADCOM®

Why In-Memory Computing?

- Currently PostgreSQL is being used and replacing is not an option. Alternative ways to improve the performance is needed.
- Data read/write speed needed to be increased across applications
- Efficient ways to support time series data store
- Support for high granular data
- Horizontal scalability
- Support new applications based on high granular and high volume data
- Need agility to address new business initiatives and customer requirements
- Seamless integration with underlying persistent store

Apache Ignite Integration – Initial Thought



- Goals
  - Seamless Ignite integration with existing PostgreSQL
- Behavior
  - Read-through
  - Write-through OR
  - Write-behind
- Options
  - MyBatis L2 Cache
  - Ignite automatic RDBMS Integration
  - Custom cache store implementation
  - Ignite as JDBC Store with MyBatis ORM
  - JDBC POJO Reflection based store

Challenges

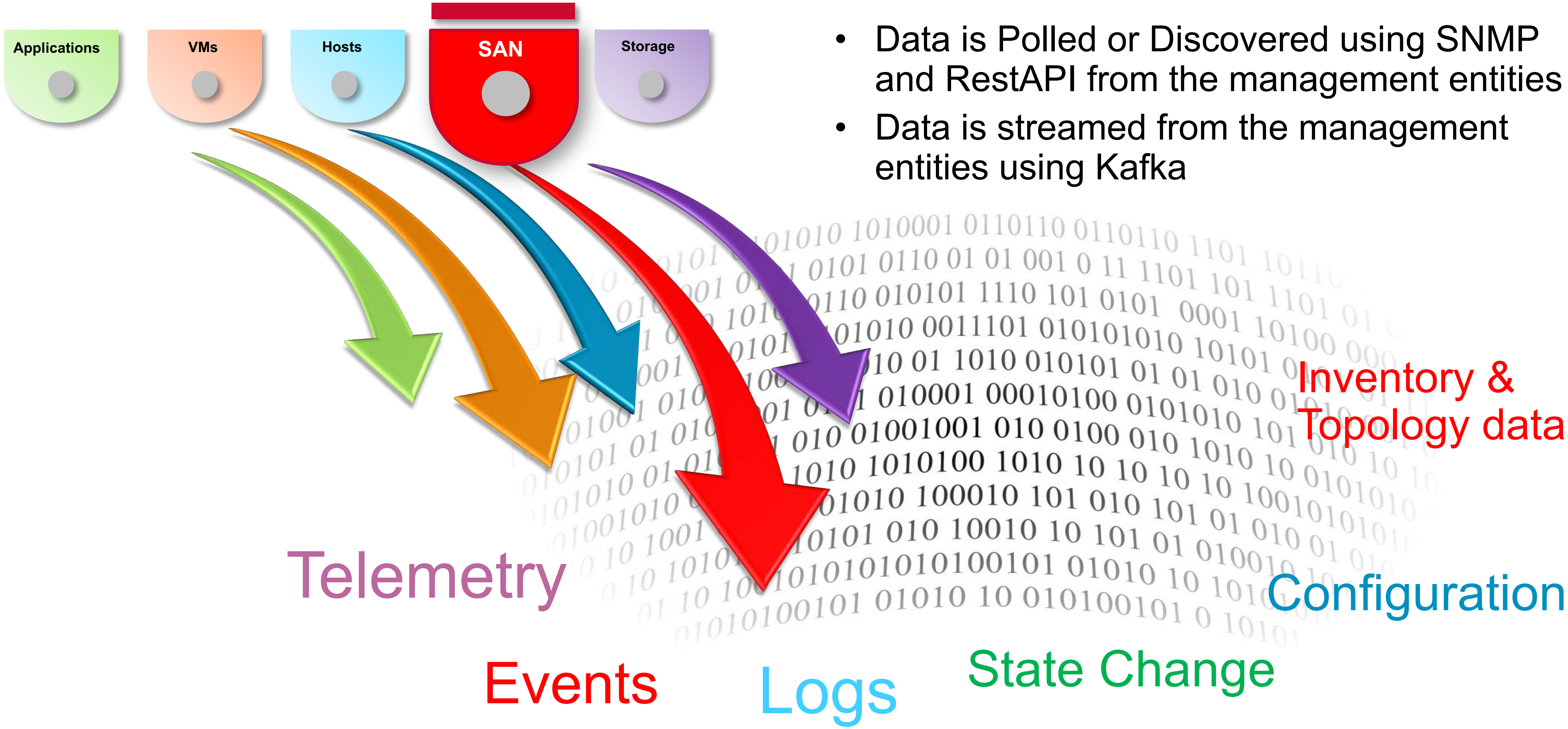
- Existing legacy Data model
  - With hundreds of tables and views
- Closely knit persistent model with long transactions
- Schema changes and Maintenance
- Tight coupling of application layer, cache and database
- Tools and Integration software not matured enough
- Write-behind is not an option
- Services write data to concrete database tables and use views to access the data; ignite doesn’t support views natively
- Having in-memory data does not guarantee efficient read/write/computation

Our Approach

- One solution doesn’t fit all
- Holistically look into Data source, type and scope
- Consider data domain, its volume and access pattern
- Store ‘Materialized views’ in ignite for transactional acid data
- Have a different store and access strategy for time series data etc.
- Use of matured tools and methodology

Data Source & Type

- Data is Polled or Discovered using SNMP and RestAPI from the management entities
- Data is streamed from the management entities using Kafka



Applications

VMs

Hosts

SAN

Storage

Inventory & Topology data

Telemetry

Configuration

Events

Logs

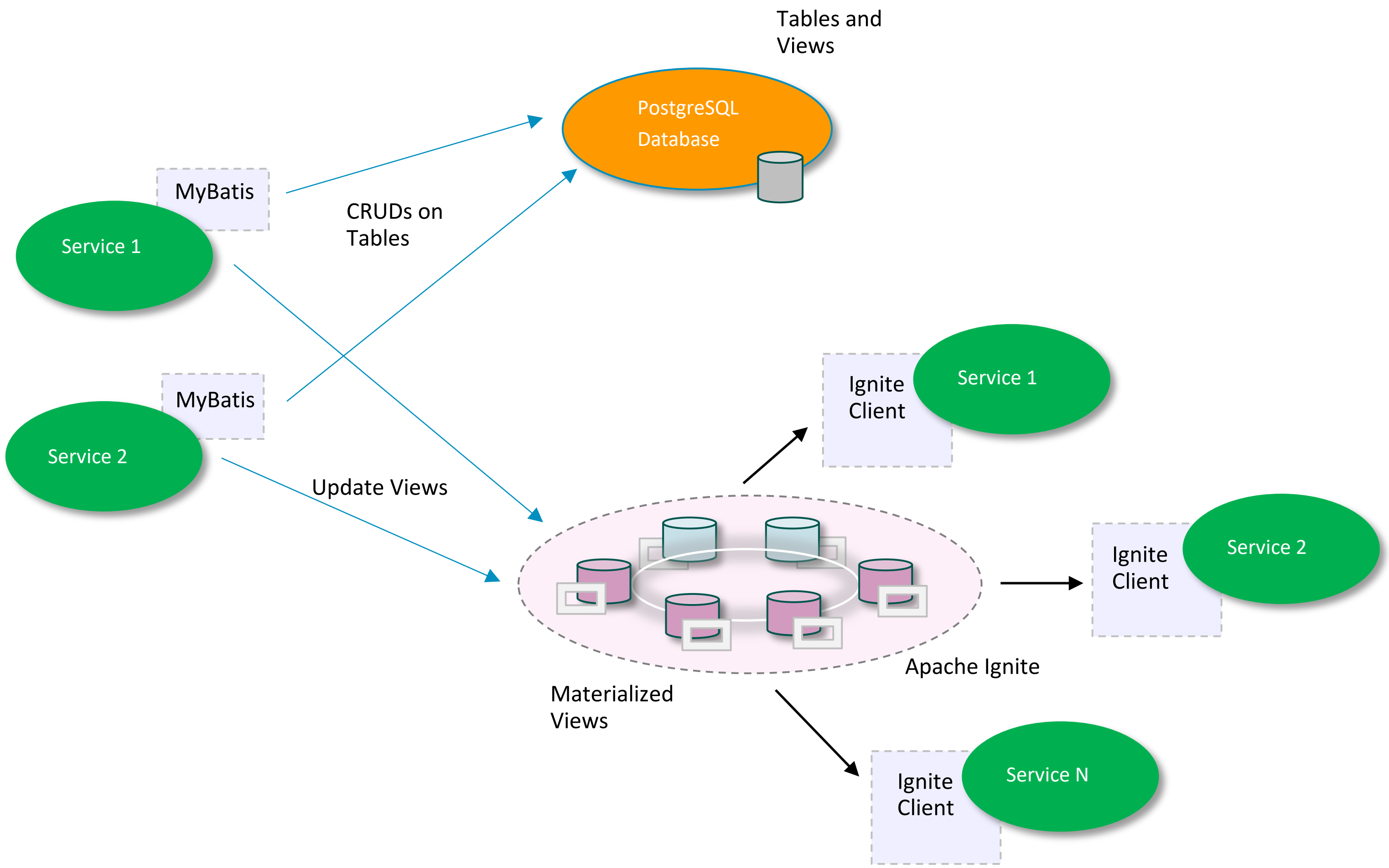
State Change

Ignite Fabric - Design Patterns

1. Maintaining Materialized view in ignite cache for transactional acid data
2. Caching time series telemetry data – ingestion, cleaning and storing
3. Caching time series telemetry data – high granular, low volume & longer duration
4. Caching time series telemetry data – high granular, high volume & shorter duration
5. Caching events and logs streaming data
6. Executing compute job on the Ignite node using ignite queue framework
7. Spring Data to access repository (Ignite Cache)

1. Materialized View in Cache

View update by applications



- Application updates the database
- Commit transaction
- Get view and update the cache

- Building & maintaining the two models required doing "dual-writes“ making the application logic complex, error-prone & difficult to maintain

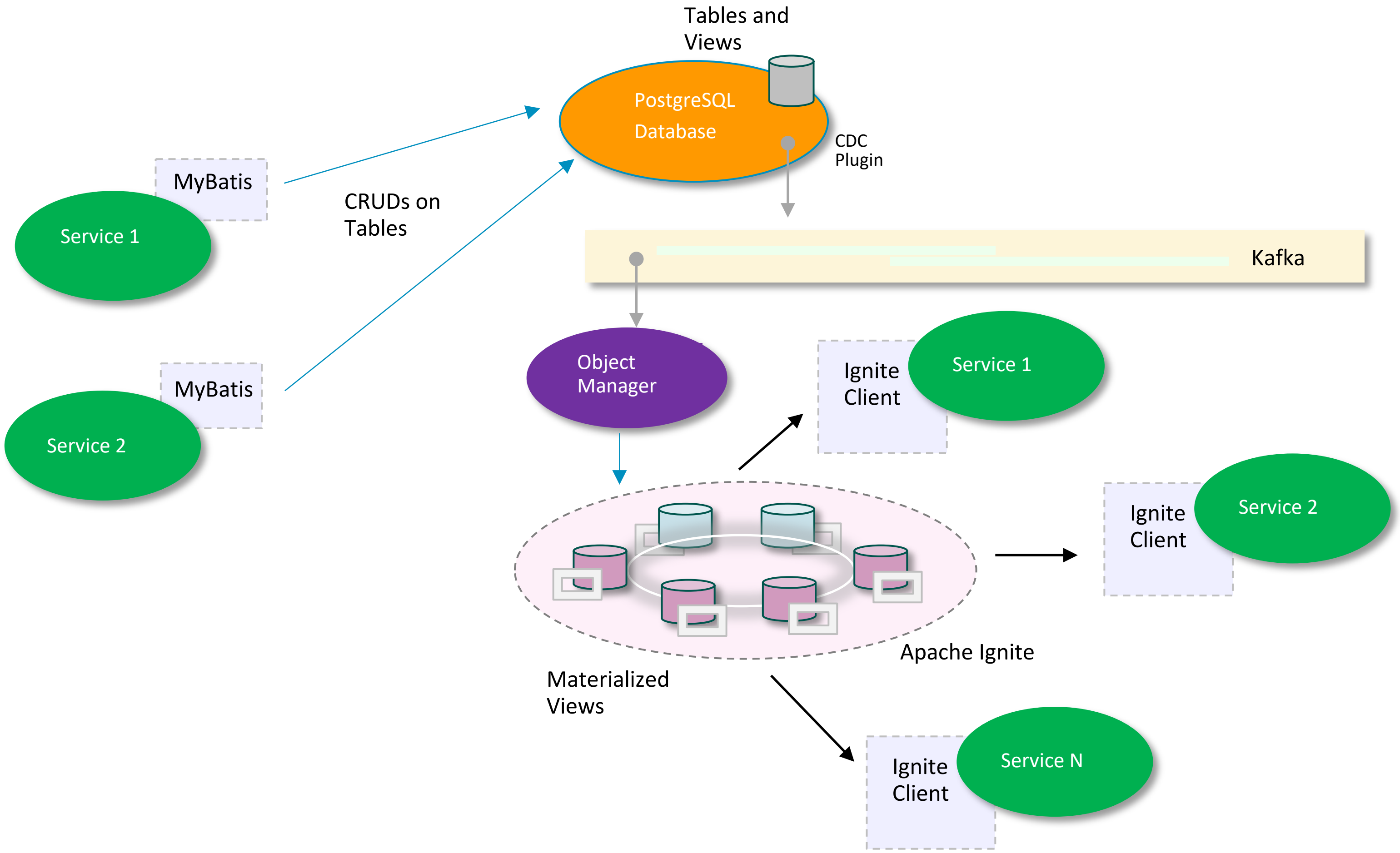
An alternative approach

CDC based Ignite Data Fabric Integration

- Transparently build & maintain Ignite materialized views of the data persisted in PostgreSQL.
- PostgreSQL uses Write-Ahead Logging (WAL) as a standard method for ensuring data integrity
- PostgreSQL's logical decoding feature and transaction isolation semantics provides
  - extraction & processing of the change log records committed
  - consistent materialized view if processed in the order in which they are streamed
- Streaming data from PostgreSQL to Kafka
  - Using Debezium, an open source change data capture (CDC) platform

Contd.

View update by CDC mechanism

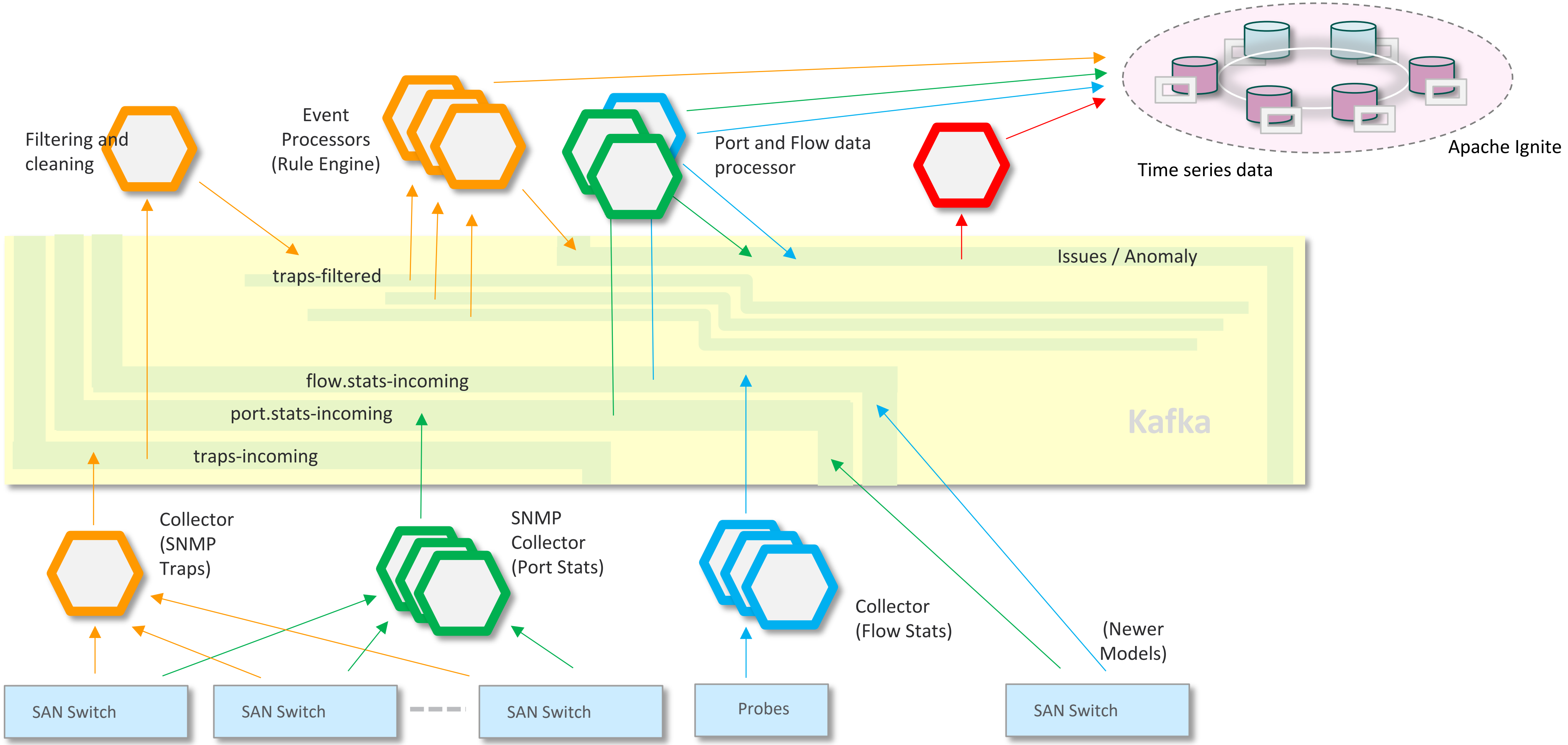


- Application updates the database
- Commit transaction
- Change Data Capture publishes the message
- OM upon consuming the message updates the cache with data just changed

- Using CDC allows to decouple data ingestion from building & updating consumption models asynchronously

2. Caching time series telemetry data

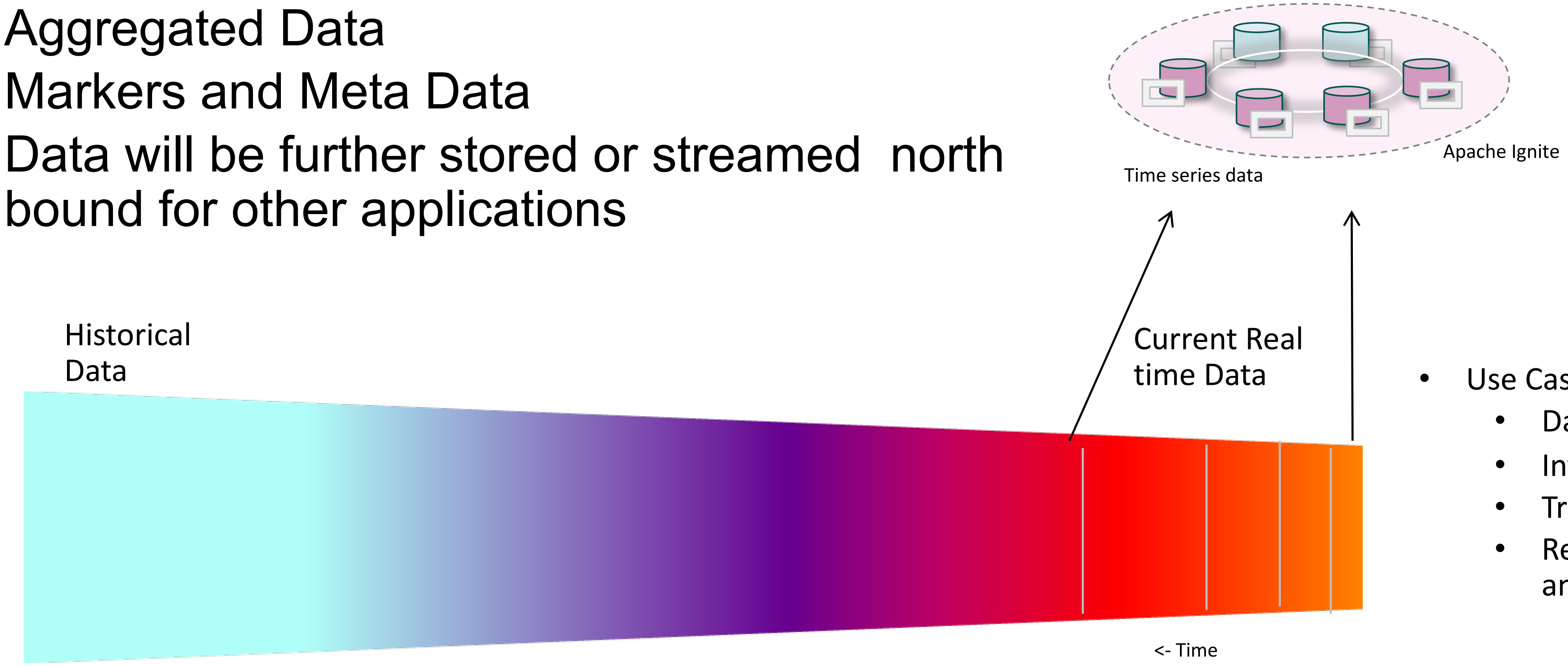
Ingestion, cleaning and storing



Contd.

What is stored in Ignite Cache and Use Cases

- Current Real-time Data
- Aggregated Data
- Markers and Meta Data
- Data will be further stored or streamed north bound for other applications



- Use Cases
  - Dashboards
  - Investigation
  - Troubleshooting
  - Real time analytics

Contd.

Problems, Challenges and How

- Should be able to ingest, process and store high granular and high volume data efficiently
- After further processing and aggregation in the cache save data in persistent storage (PostgreSQL or Elastic search) for historical analysis and report purposes
  - Aggregation, correlation and basic analytics
  - Analyzing the large volume of short lived samples for detecting patterns. i.e. congestions etc...
- Providing insight and instance access to data raw or computed based on hot, warm or cold data
  - Build and keep metadata in ignite cache to provide access to time series data quickly to the applications
  - Provision to compute and perform analytics inside the ignite cache instead of shipping truck load of data to the applications to further process

Contd.

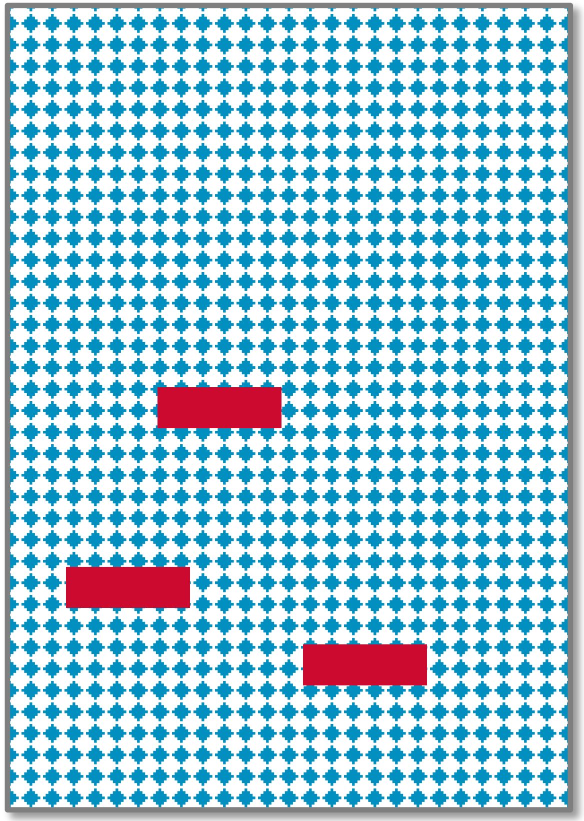
Problems, Challenges and How

- Having in-memory data is not guaranteed to have efficient computation
  - The in memory data with data pre organized for efficiency allows us to perform faster in memory computations for many of our use cases.
- Various microservices have different expectations with respect to read performance and write performance.
  - With the combination of ignite in memory capability and data structure geared for efficient queries we are able to achieve frequent writes for high volume data and dense data computation with efficient queries.
  - For example, the write heavy service receives half a million samples every minute

3. Caching time series telemetry data

High granular, low volume & longer duration (transient caching)

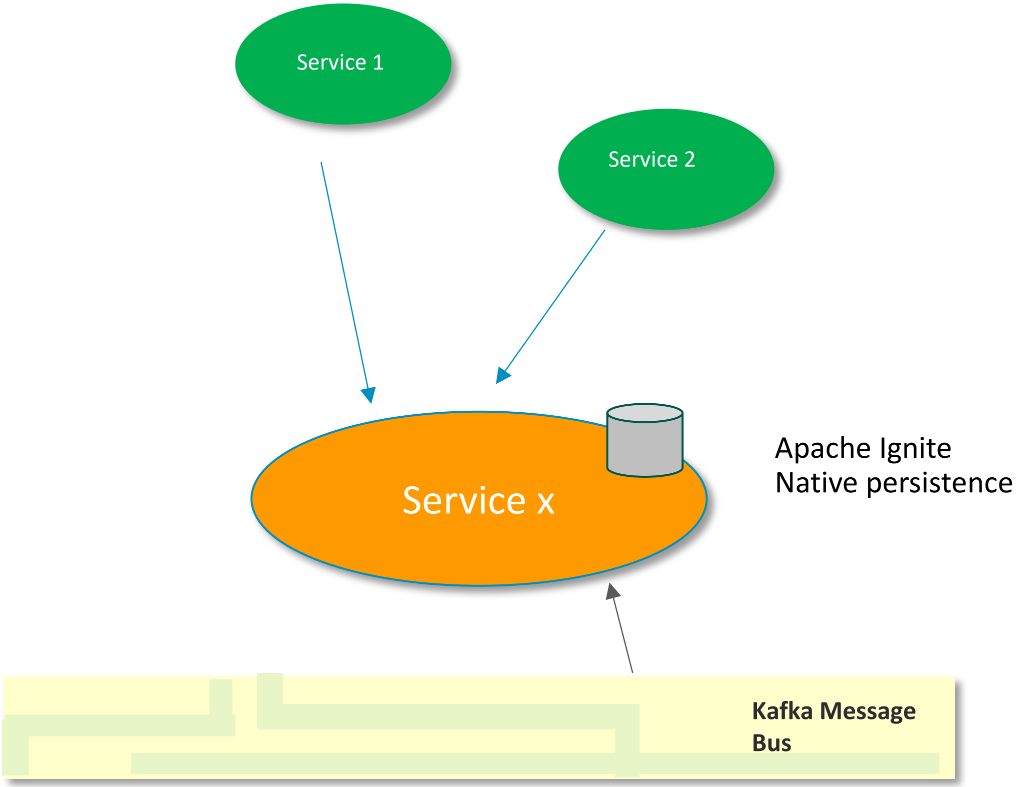
- On demand access of high granular time series data
- Efficiently get data given a time interval and list of network objects
- Problem:
  - Low ROI to store data as individual rows in a database table or to store in the memory
- Solution
  - Store in database as blobs
  - Load into ignite cache on demand



4. Caching time series telemetry data

High granular, high volume & shorter duration

- Ignite native persistence at microservice level
  - Used only by a service
  - High granular data access only on demand
- Persist very high granular data for shorter duration



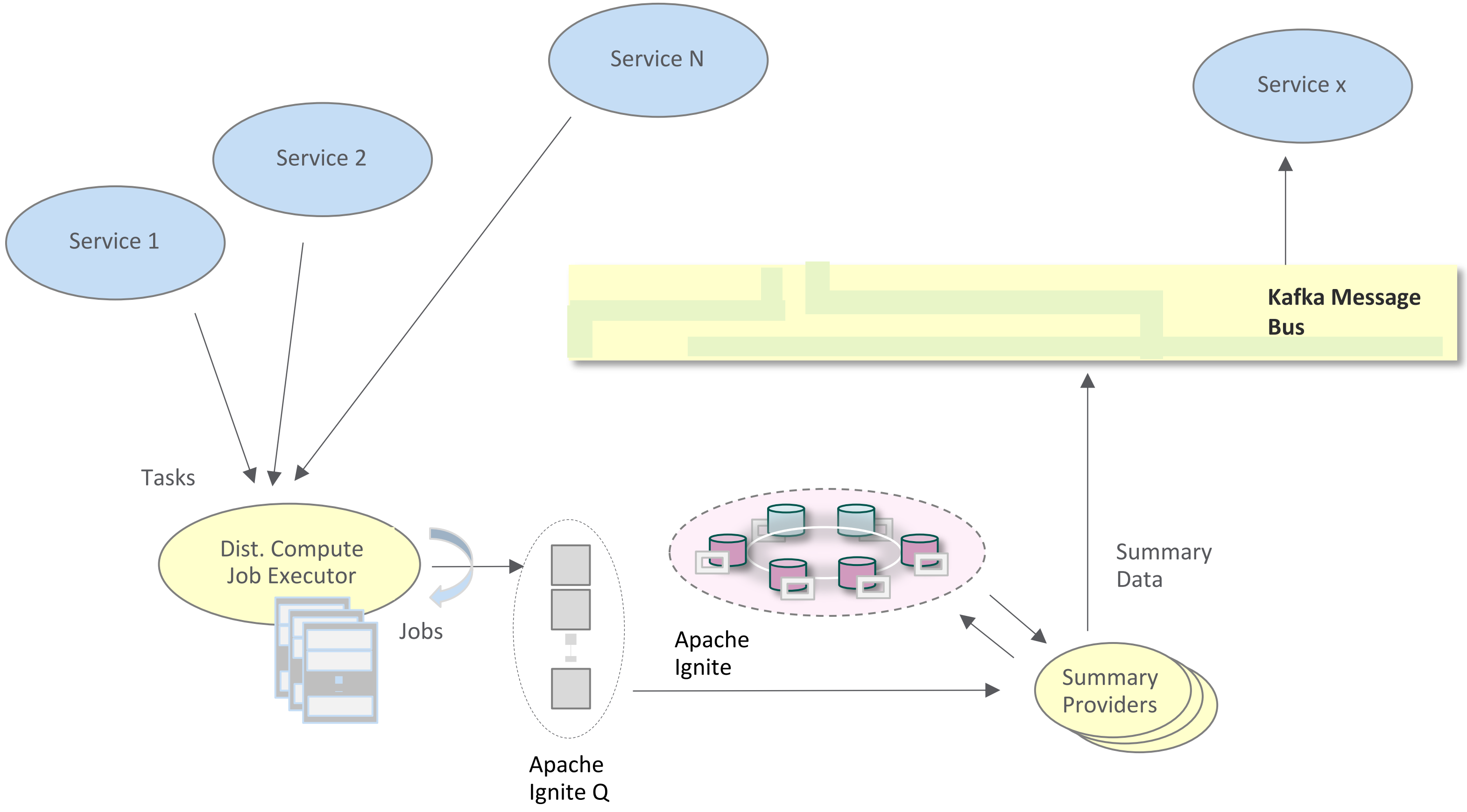
5. Caching events and logs streaming data

(Need to capture how Ignite helps here)

- Handle massive amounts of streamed events
- Discrete and burst events preprocessing
- Event Filters
- Event Rule Engine
- Aggregation and correlation
  - identify the most meaningful events and patterns from multiple data sources, analyze their impacts, and act on them in real time

6. Distributed Compute Job Executor Framework

Using ignite queue and cache



7. Spring Data

- Why Spring Data
  - Consistent, uniform model for data access (Repository abstraction)
  - Avoids error prone boilerplate code through the use of declarative model
  - Flexibility to utilize the full capabilities of underlying data store when needed, while still keeping the store access abstracted from the application
  - Promotes code reuse

Contd.

- How Ignite's Spring Data integration abstracts & simplifies ignite data fabric access & usage
  - Ignite's Spring integration ensures transparency (Distributed System Transparency) allowing Ignite to be simply injected & used in the application code
  - Ignite's Spring Data implementation brings the full power of Spring Data to Ignite's Data & SQL Grid by enabling seamless access to the data fabric abstracting away the underlying mechanics & details of the connection & access.
  - Together they enable weaving an Ignite Data Fabric in our micro services architecture

Q & A

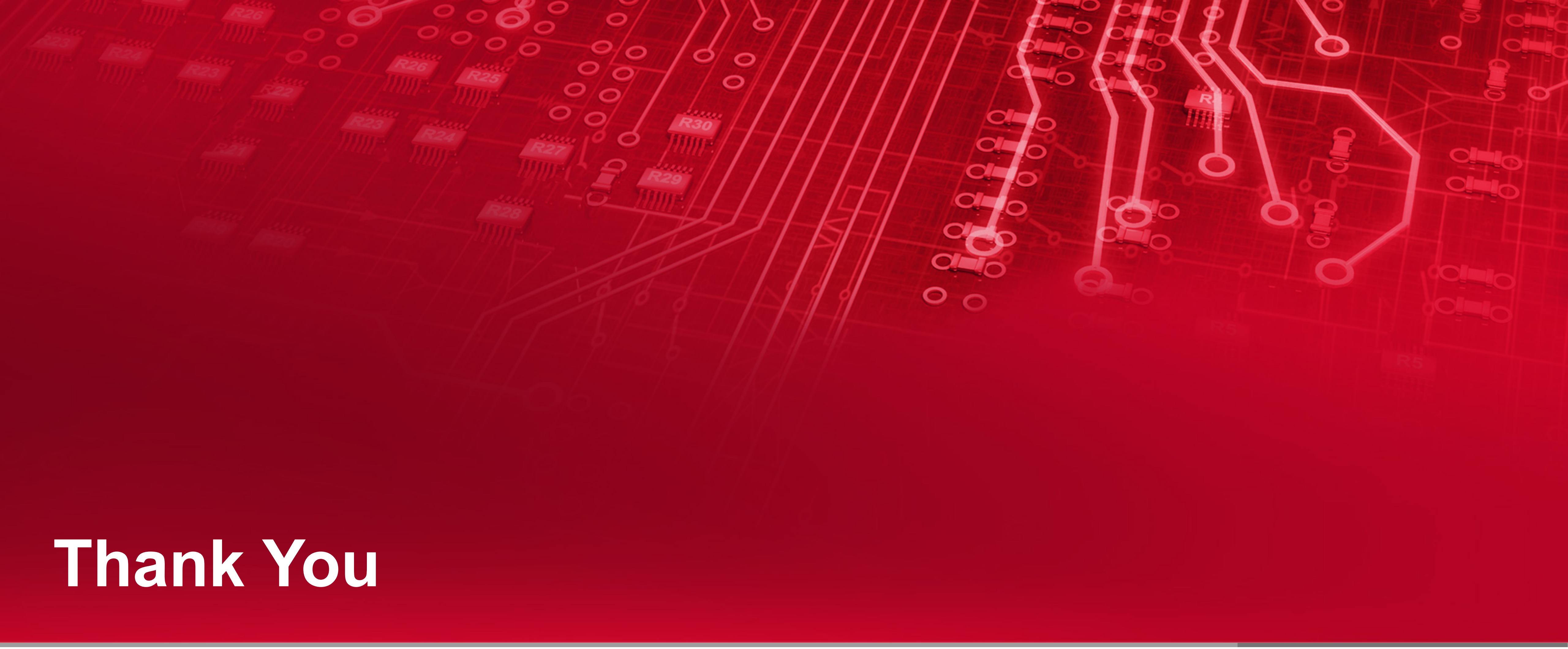
BROADCOM®

Contact

Aruna Sangli

Email : aruna.sangli@broadcom.com

Mobile:



Thank You

BROADCOM®

connecting everything®