PRISMTECH

Angelo Corsaro, PhD
Chief Technology Officer
angelo.corsaro@prismtech.com

Getting the Course Material

GETTING THE MATERIAL

Ensure **‘git’** is installed on your machine, then simply execute the command listed below:

```
$ git clone https://github.com/kydos/2016-DRIO-5010C.git
```

UPDATING THE MATERIAL

Once checked out the course material, you can keep up to date by issuing the following command:

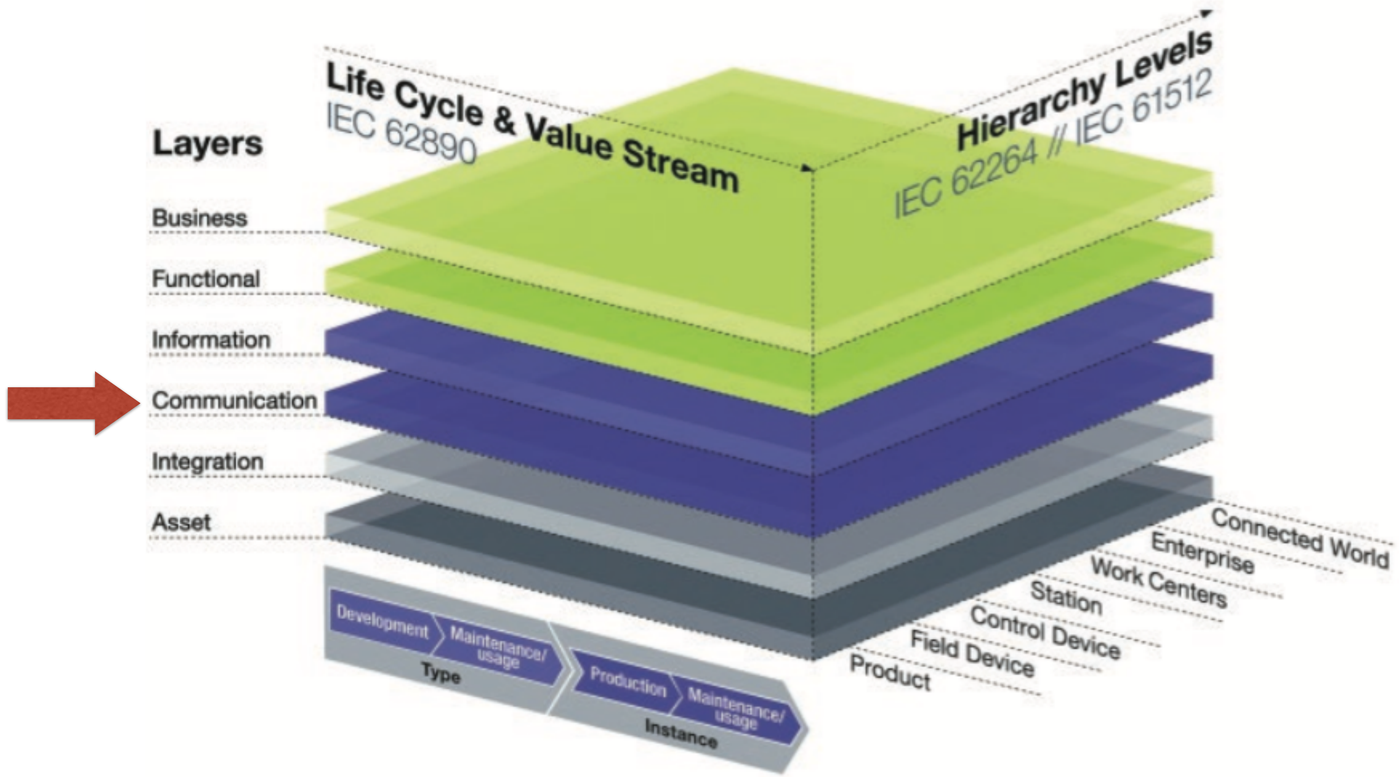
```
2016-DRIO-5010C $ git pull
```

Information Sharing in IIoT

COMMUNICATION LAYER

Standardisation of communication, using a uniform data format, in the direction of the Information Layer.

Provision of services for control of the Integration Layer.



INFORMATION LAYER

Run time environment for processing of events.

Persistence of the data which represent the models.

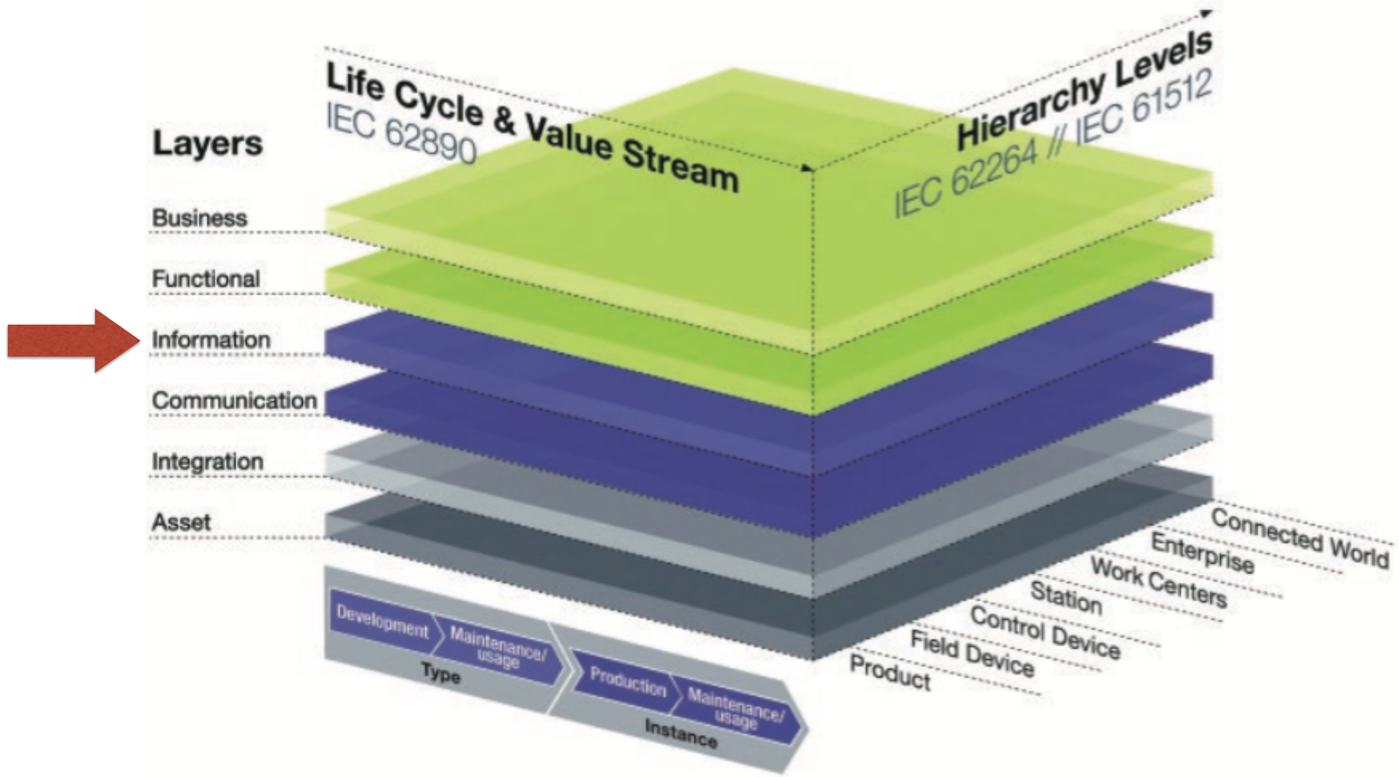
Ensuring data integrity.

Consistent integration of different data.

Obtaining new, higher quality data (data, information, knowledge).

Provision of structured data via service interfaces.

Receiving of events and their transformation to match the data which are available for the Functional Layer.

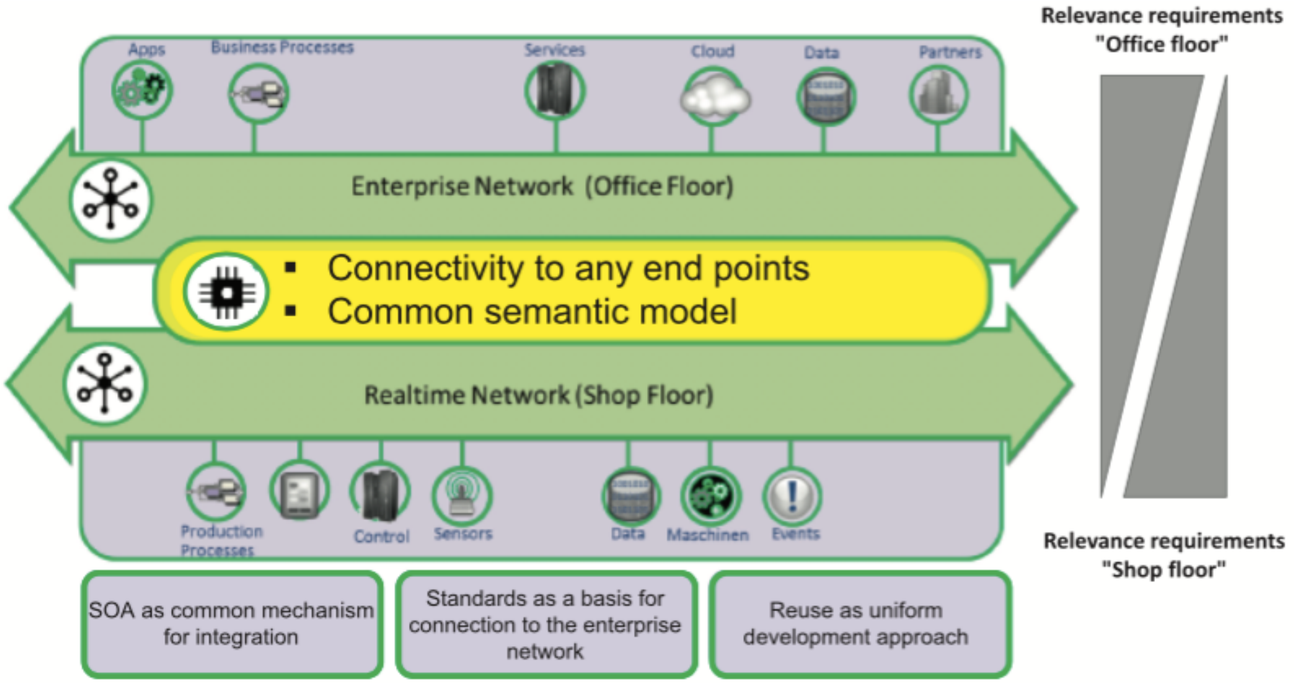


I4.0 COMPONENTS

To allow for **seamless integration** of the “Office Floor” and the “Shop Floor” I4.0 requires connectivity to any end points and a common semantic model.

Components must have certain common properties independently of the levels.

They are specified in the form of the I4.0 components.



I4.0 COMPONENT

The ability to virtualise physical entities and make information available is key to RAMI4.0 and captured as part of the I4.0 Component

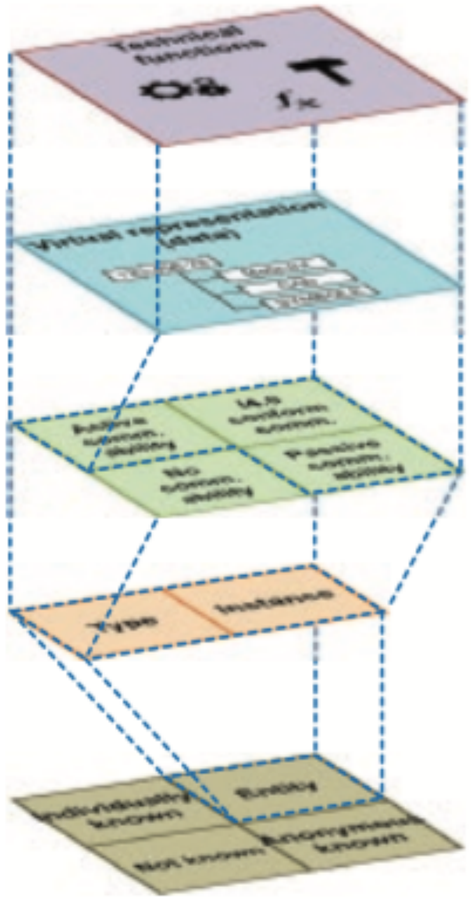


Image from: “Reference Architecture Model Industrie 4.0”

IIRA FUNCTIONAL DOMAINS

The IIRA decomposes an Industrial Internet System (IIS) in five **functional domains**: **Control**, **Operation**, **Information**, **Application** and **Business**

Data flows and **control flows** take place in and **between** these **functional domains**.

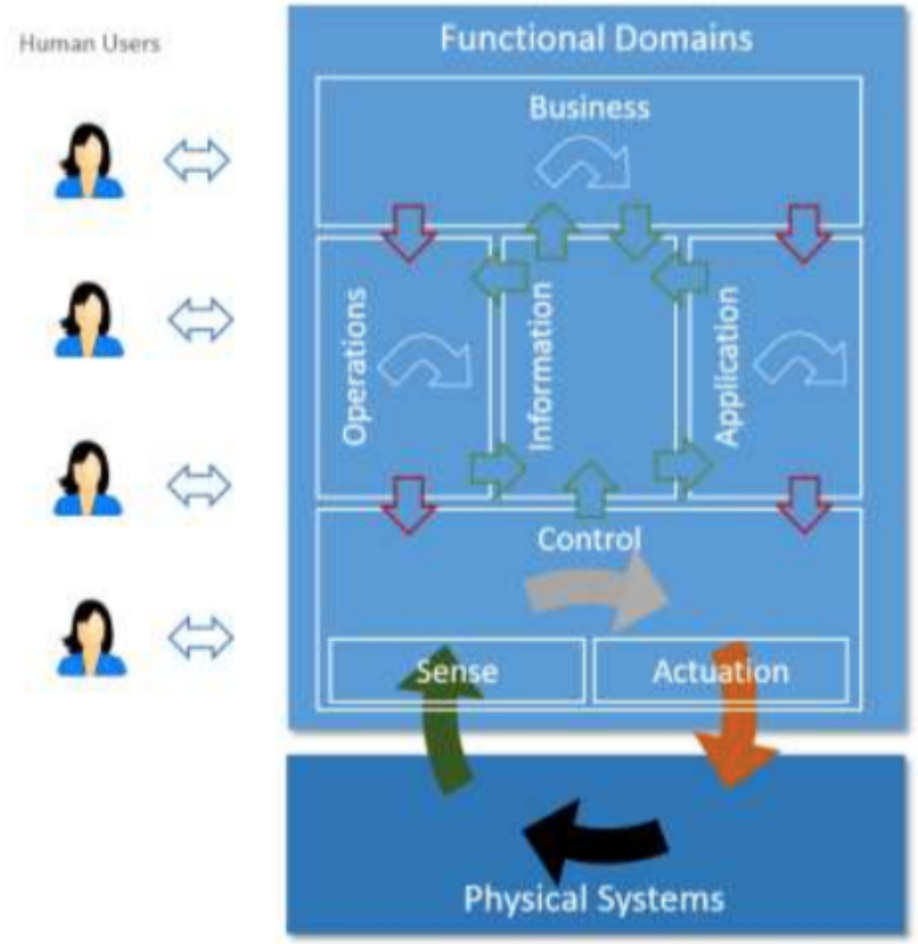


Image from: “Industrial Internet Consortium Reference Implementation v1.7”

PRISMTECH

The DDS Tutorial

Angelo Corsaro, PhD
Chief Technology Officer
angelo.corsaro@prismtech.com

DDS IN 131 CHARACTERS

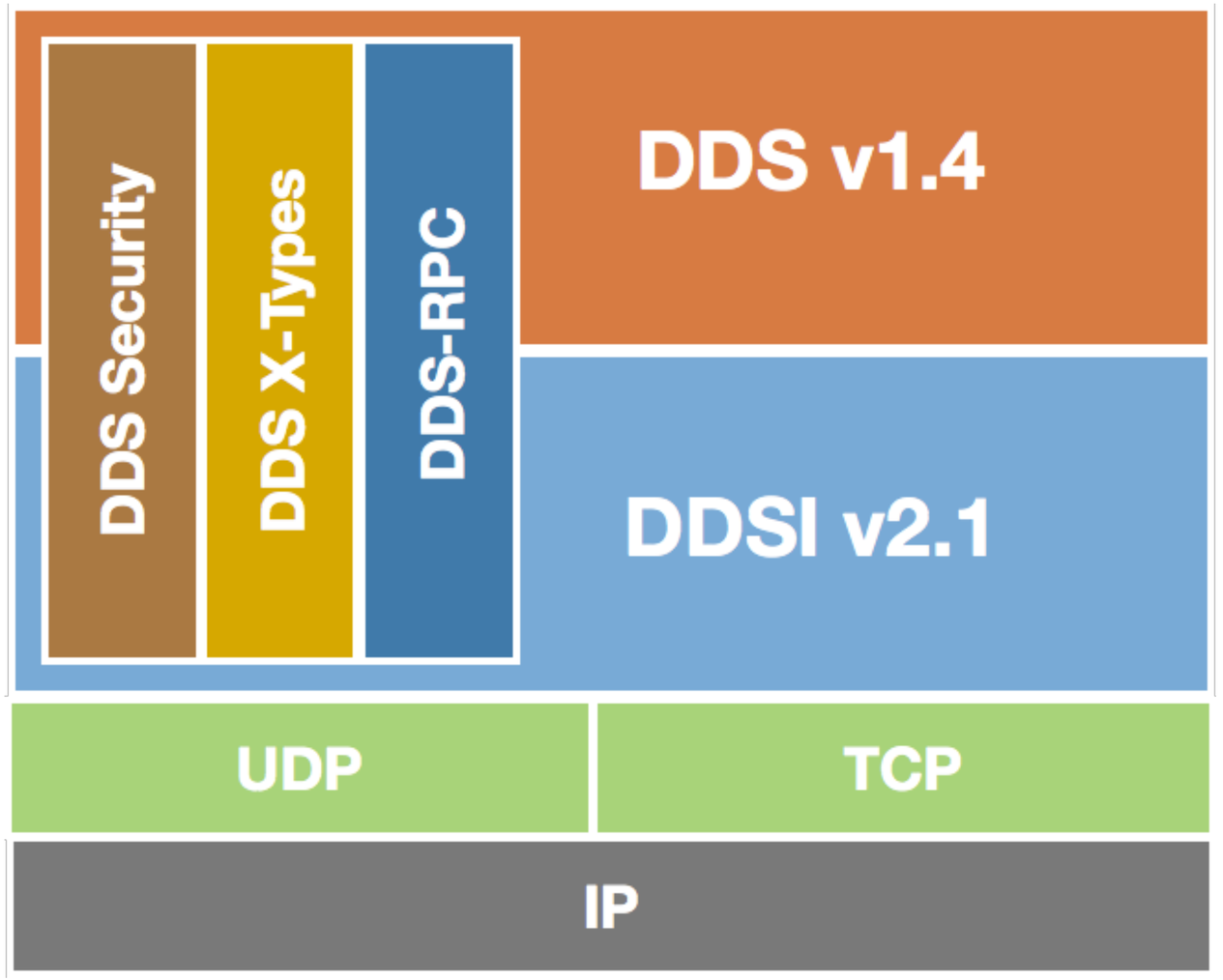
DDS is a standard technology for ubiquitous, interoperable, secure, platform independent, and real-time data sharing across network connected devices

The DDS Standard

STANDARD

industrial internet® CONSORTIUM

DDS
The IoT Data Sharing Standard



RECOMMENDATIONS

MINISTRY OF DEFENCE

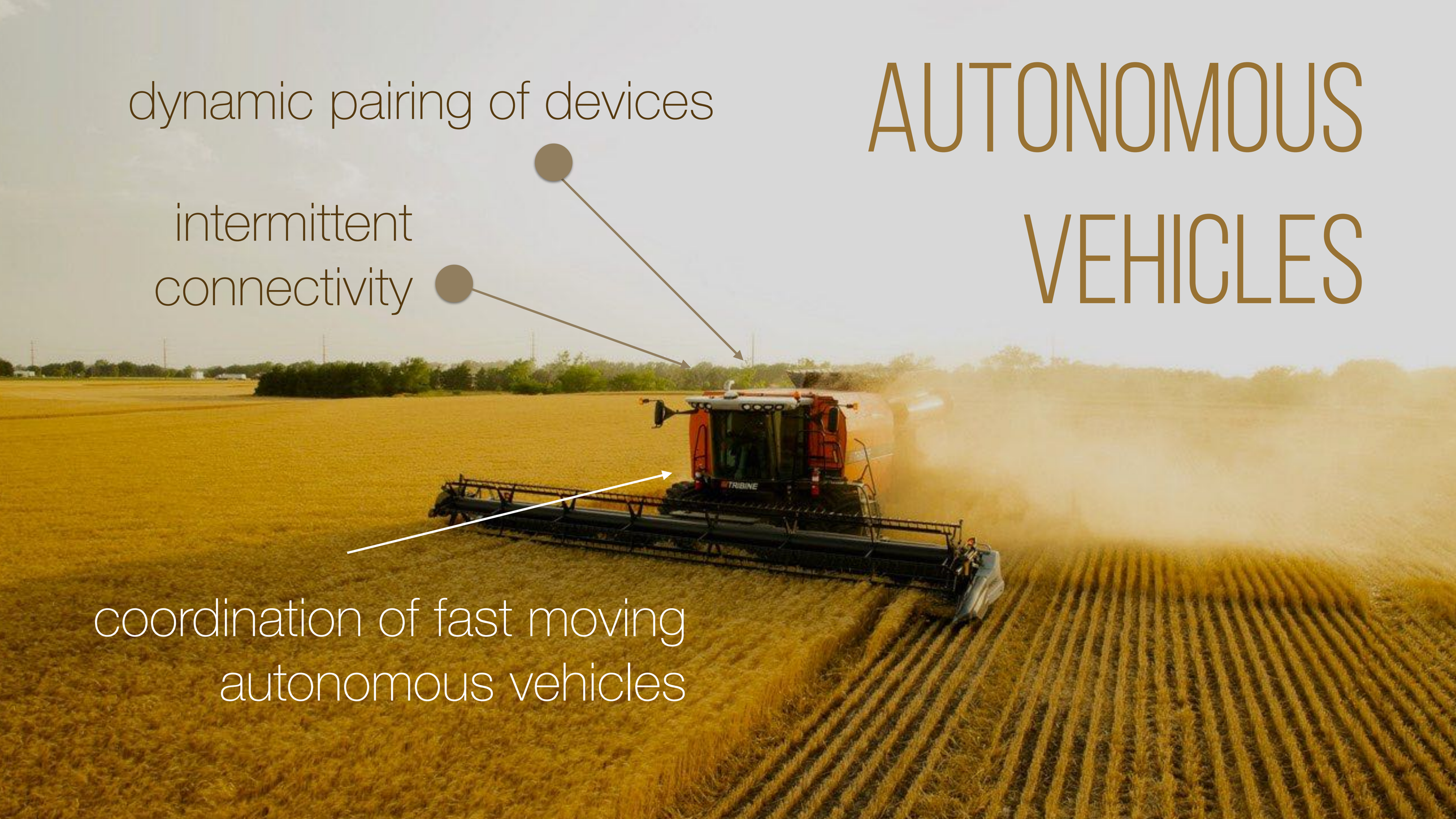
Commercial Adoption

AUTONOMOUS VEHICLES

dynamic pairing of devices

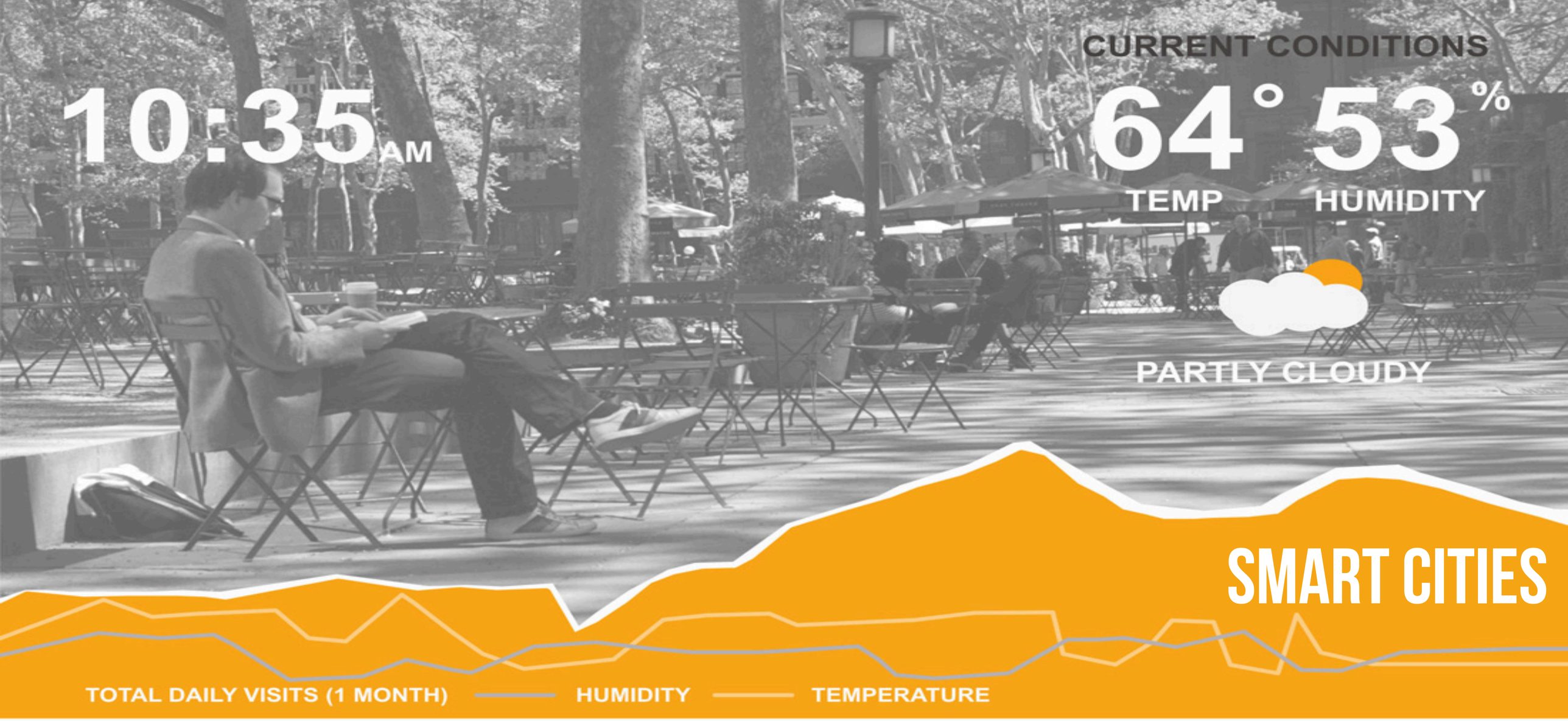
intermittent connectivity

coordination of fast moving autonomous vehicles



SMART GREEN HOUSES

Vortex used to virtualise sensor data and to distribute actions and insights



TOTAL VISITS TODAY

622 ↑

CURRENT CHAIR OCCUPANCY

65% ↑

OCCUPANCY @ TABLES

43% ↓

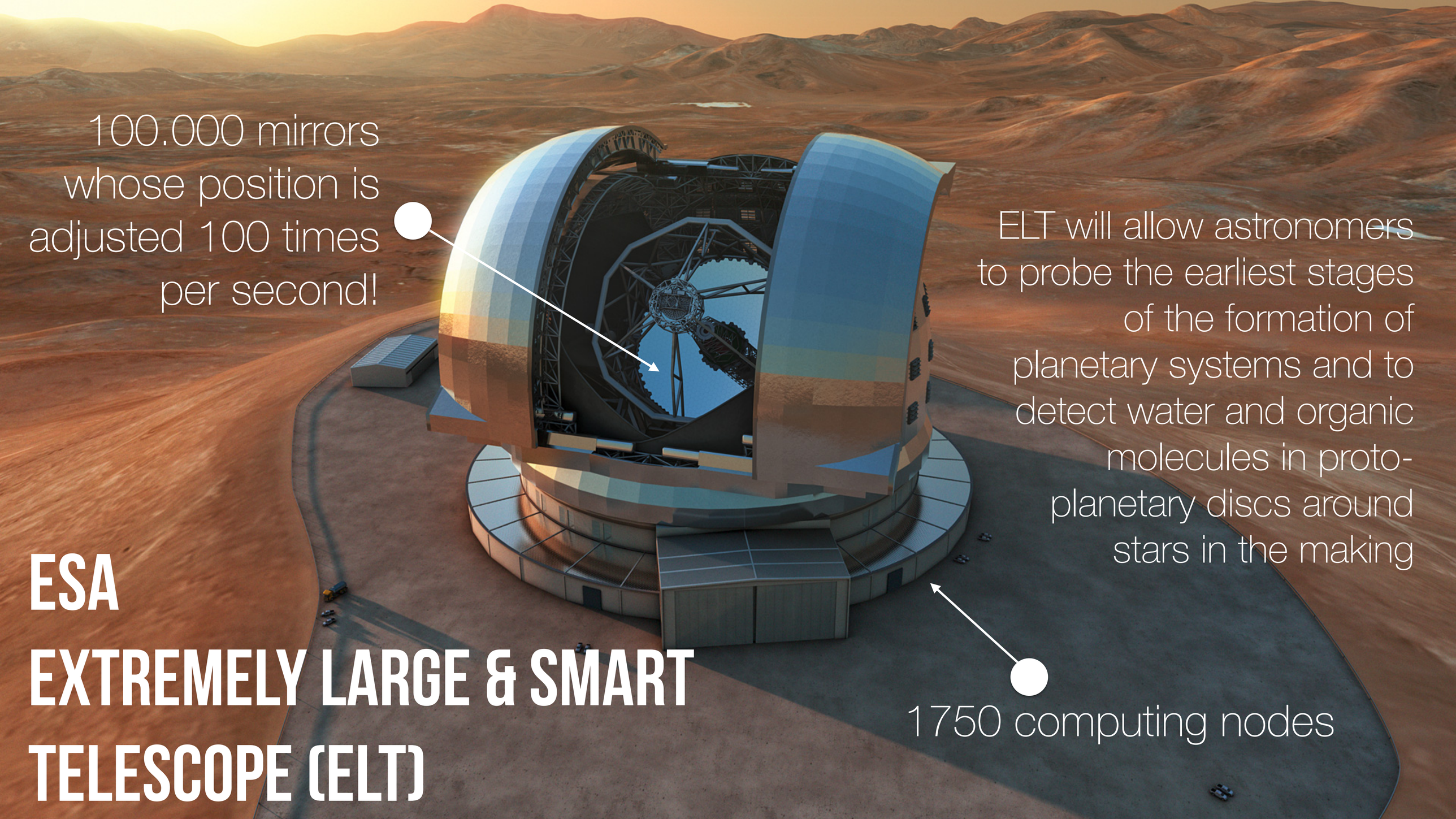
OCCUPANCY @ BENCHES

23% ↑


20ms deadline for phase
alignment data
SMART-GRID


Smart Factory
0.5 TB of data
produced per day


OIL RIG
30000 data points
only 1% of available data
used today

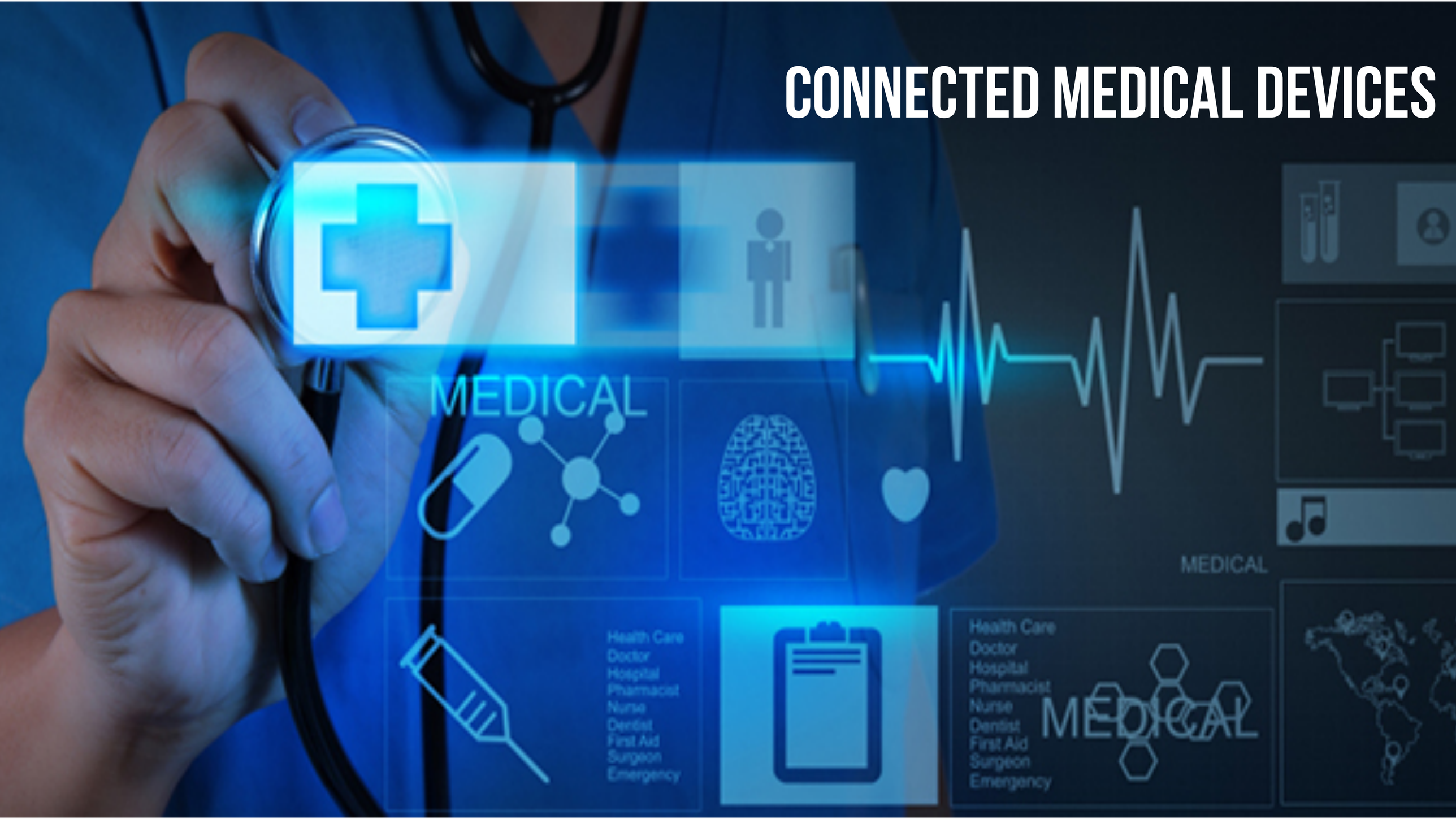

100.000 mirrors whose position is adjusted 100 times per second!
ELT will allow astronomers to probe the earliest stages of the formation of planetary systems and to detect water and organic molecules in proto-planetary discs around stars in the making
1750 computing nodes
ESA
EXTREMELY LARGE & SMART TELESCOPE (ELT)


80K+ data points with aggregate updates
rate of ~400K msgs/sec
LAUNCH SYSTEM

10 TB of data every
30m of flight

CONNECTED AIRCRAFTS

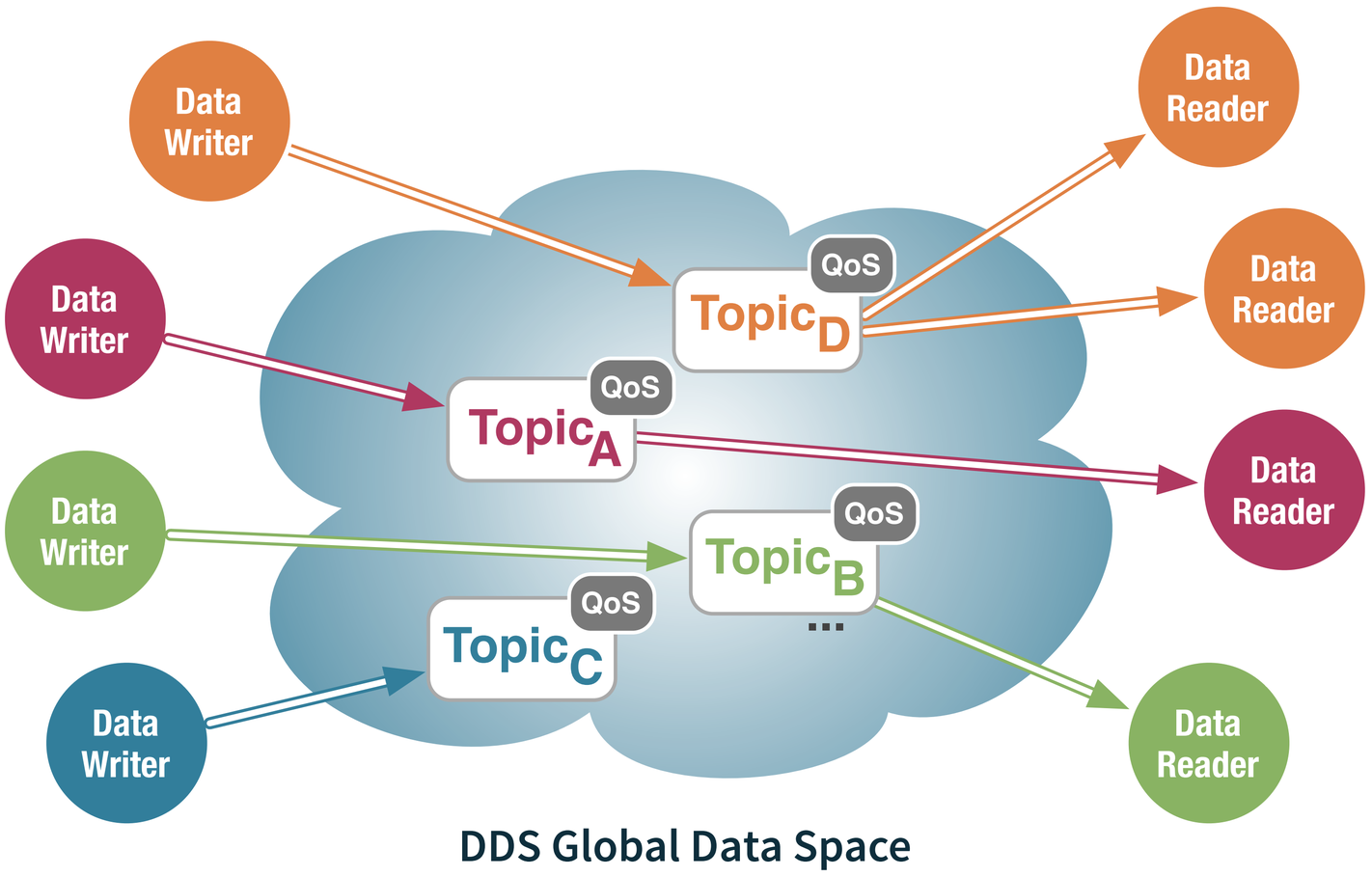
CONNECTED MEDICAL DEVICES



Grasping the Idea

VIRTUALISED DATA SPACE

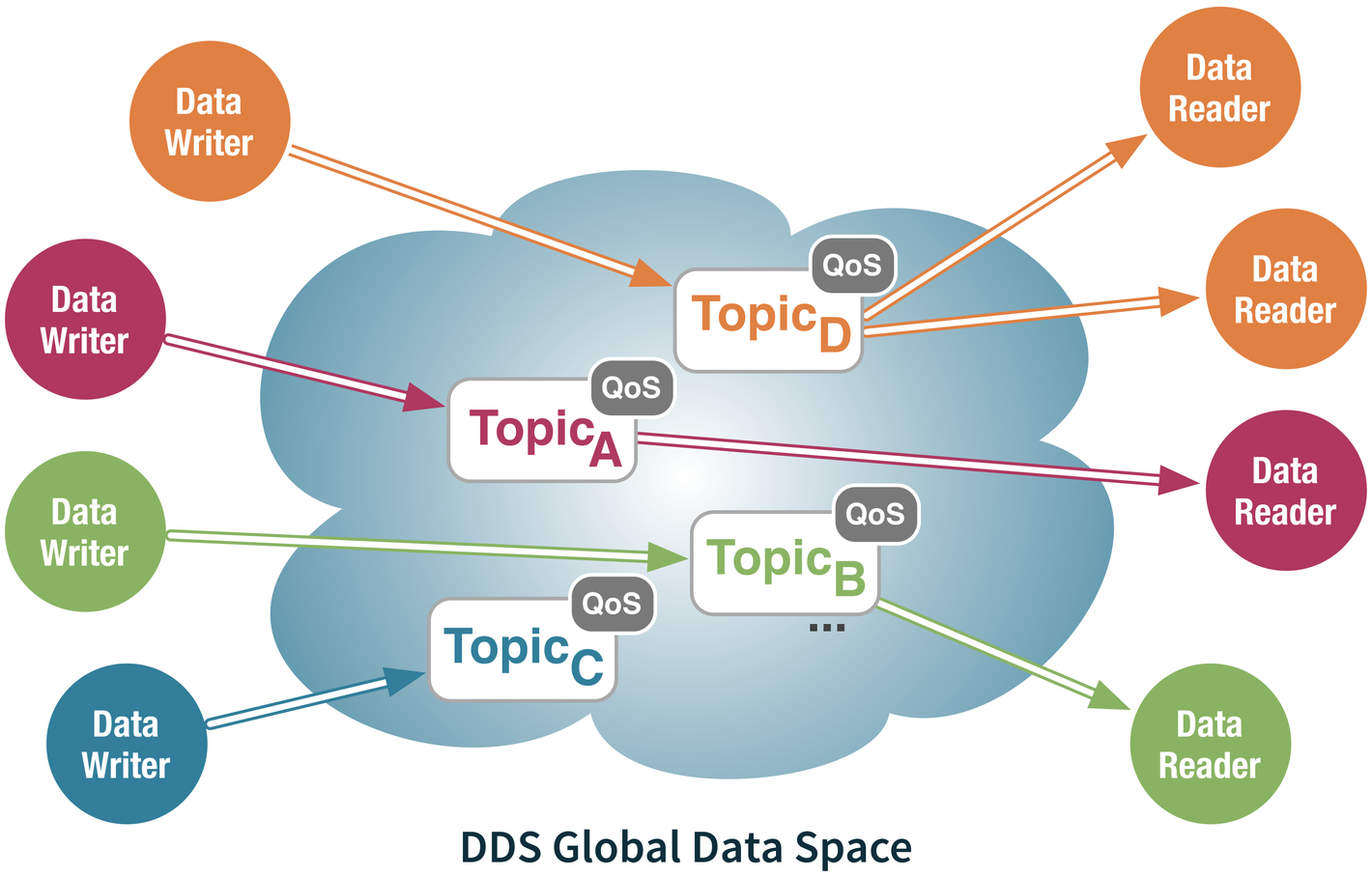
Applications can **autonomously** and **asynchronously read** and **write** data enjoying **spatial** and **temporal decoupling**



DDS Global Data Space

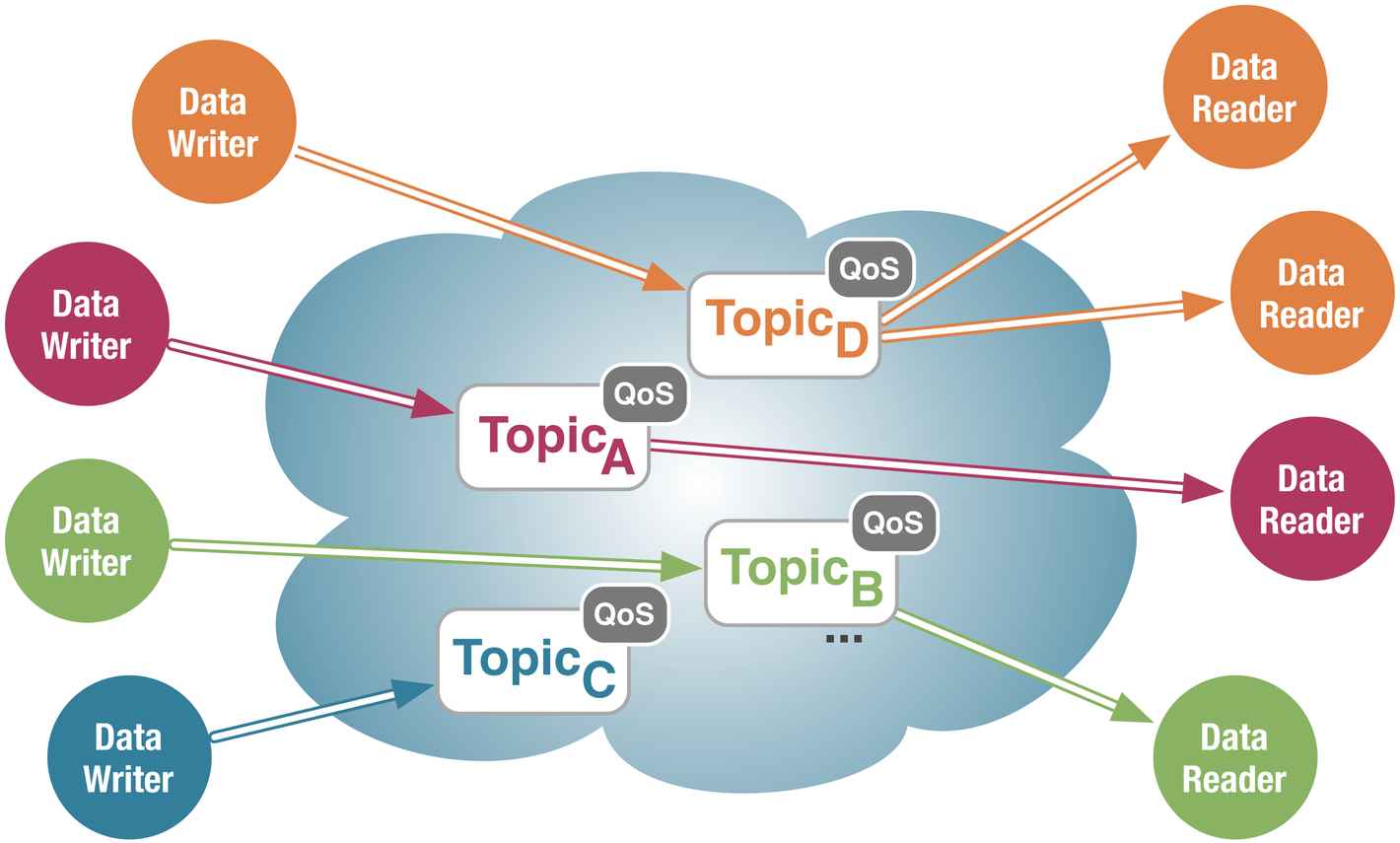
DYNAMIC DISCOVERY

Built-in dynamic discovery **isolates applications from network topology** and **connectivity** details



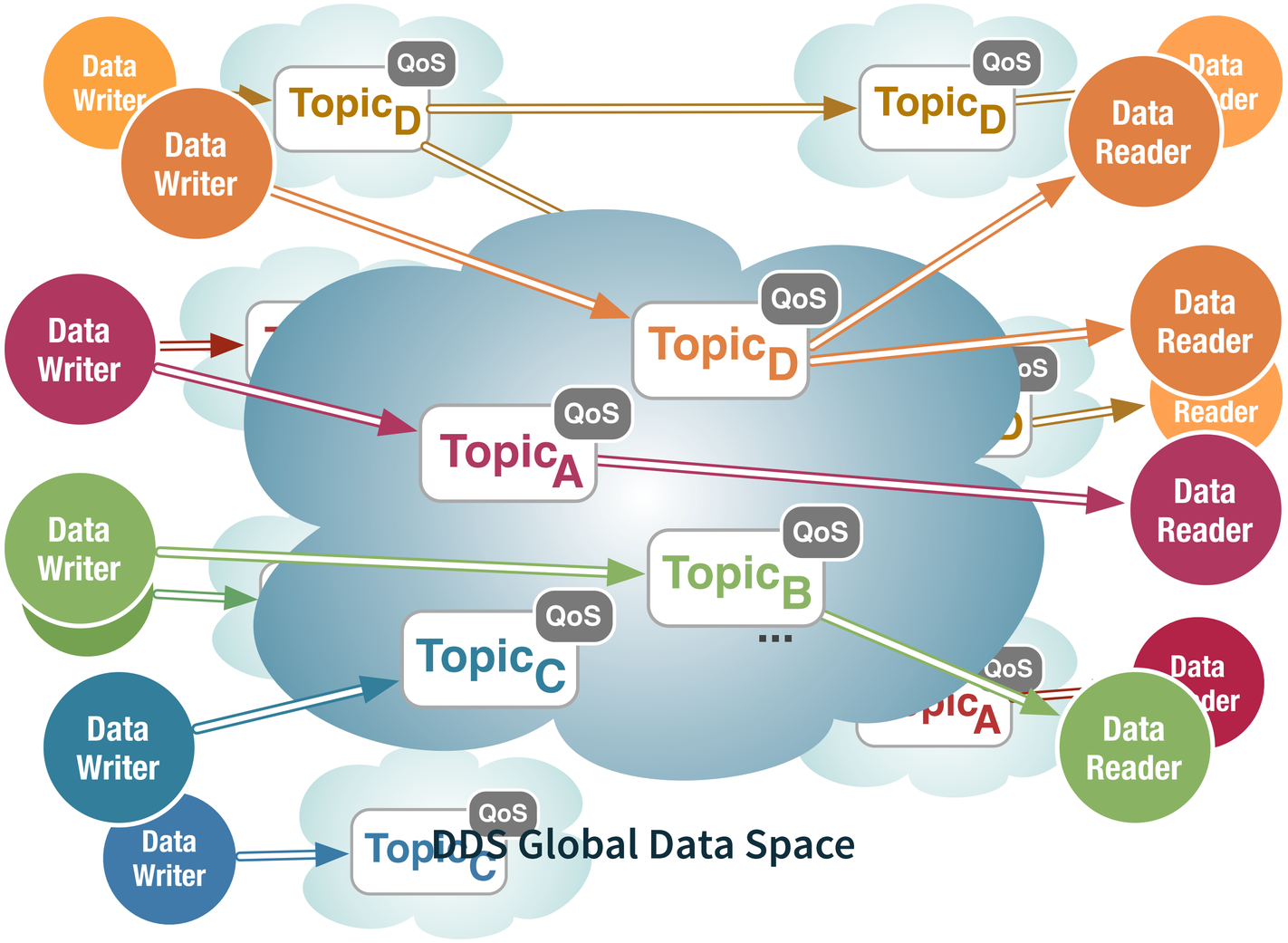
QOS ENABLED

QoS policies allow to express **temporal** and **availability constraints** for data



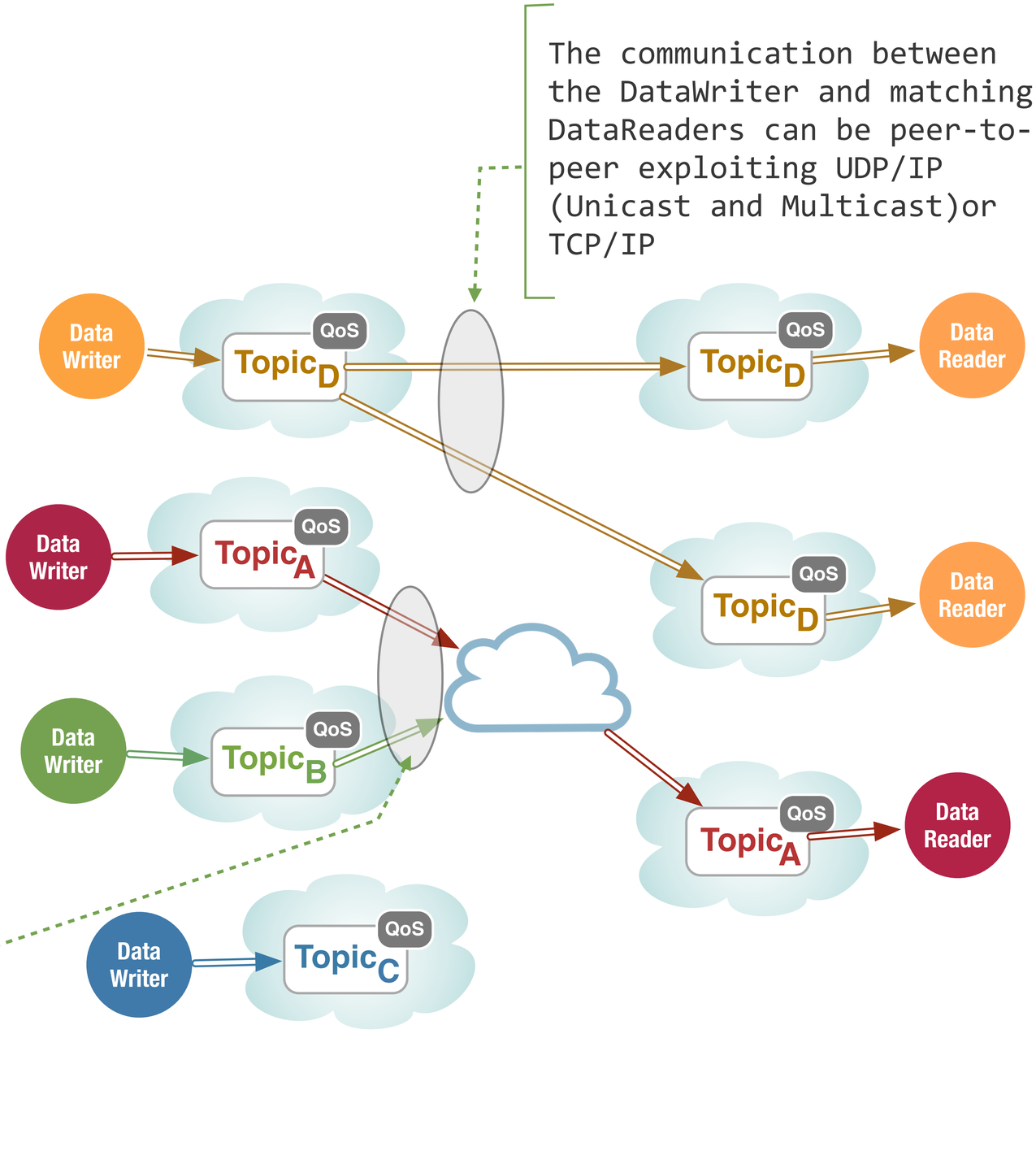
DECENTRALISED DATA-SPACE

No single point of failure or bottleneck



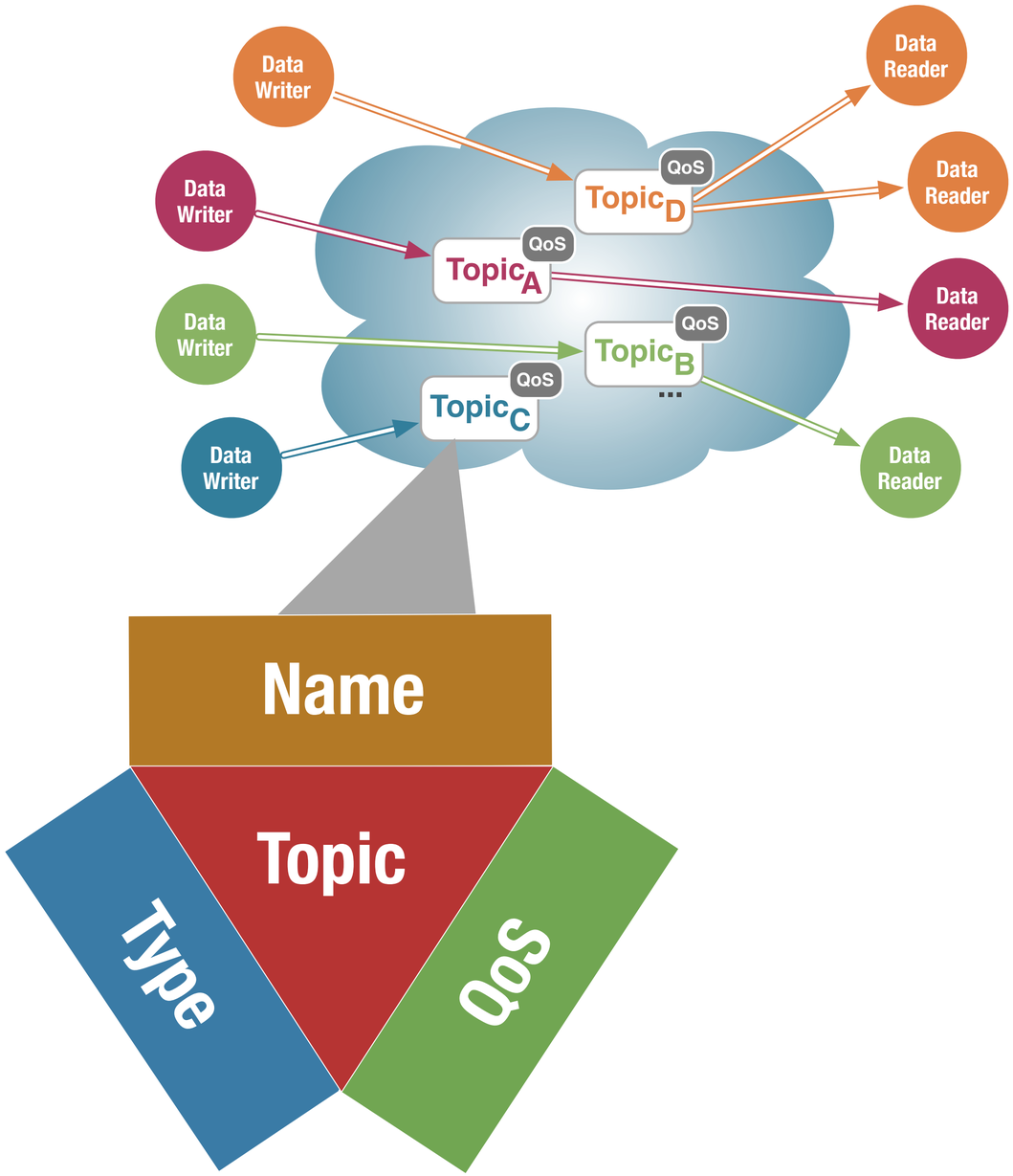
ADAPTIVE CONNECTIVITY

Connectivity is **dynamically adapted** to chose the most effective way of sharing data



TOPIC

A domain-wide information's class A **Topic** defined by means of a <name, type, qos>



PLATFORM INDEPENDENT

Programming language,
Operating System,
and HW architecture
Independent









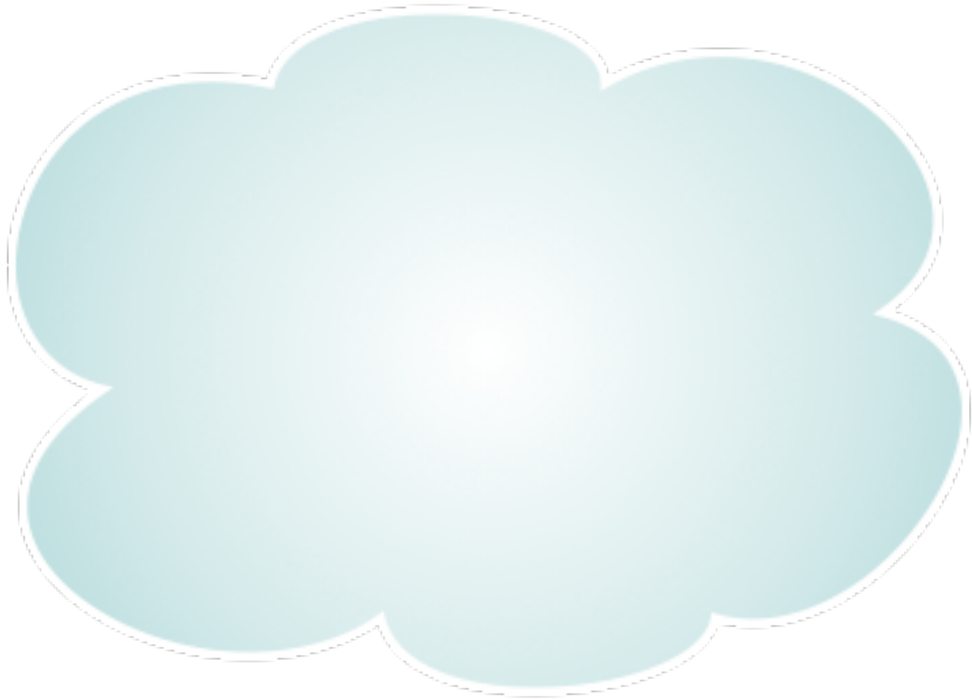
:: Decomposing DDS ::

Information Organisation

DOMAIN

DDS data lives within a domain

A domain is identified with a non negative integer, such as 1, 3, 31, with 0 representing the default domain

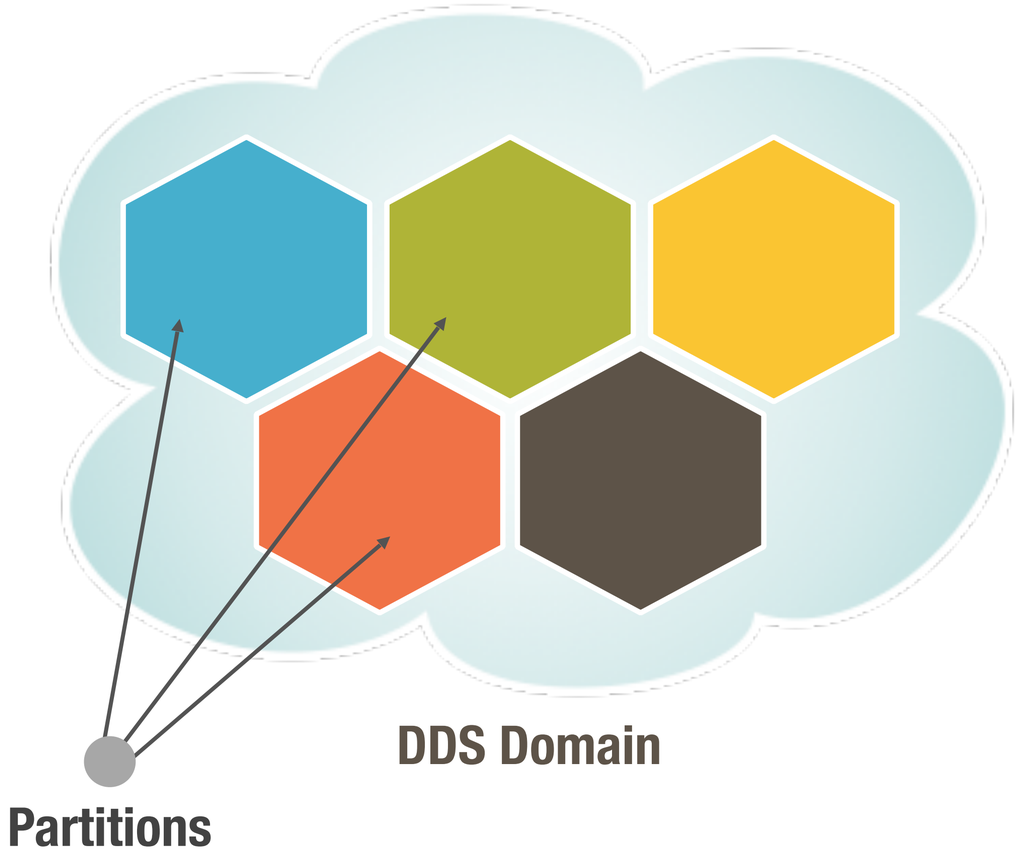


DDS Domain

PARTITIONS

Partitions are used to organise information within a domain.

Access to partitions is controlled through QoS Policies



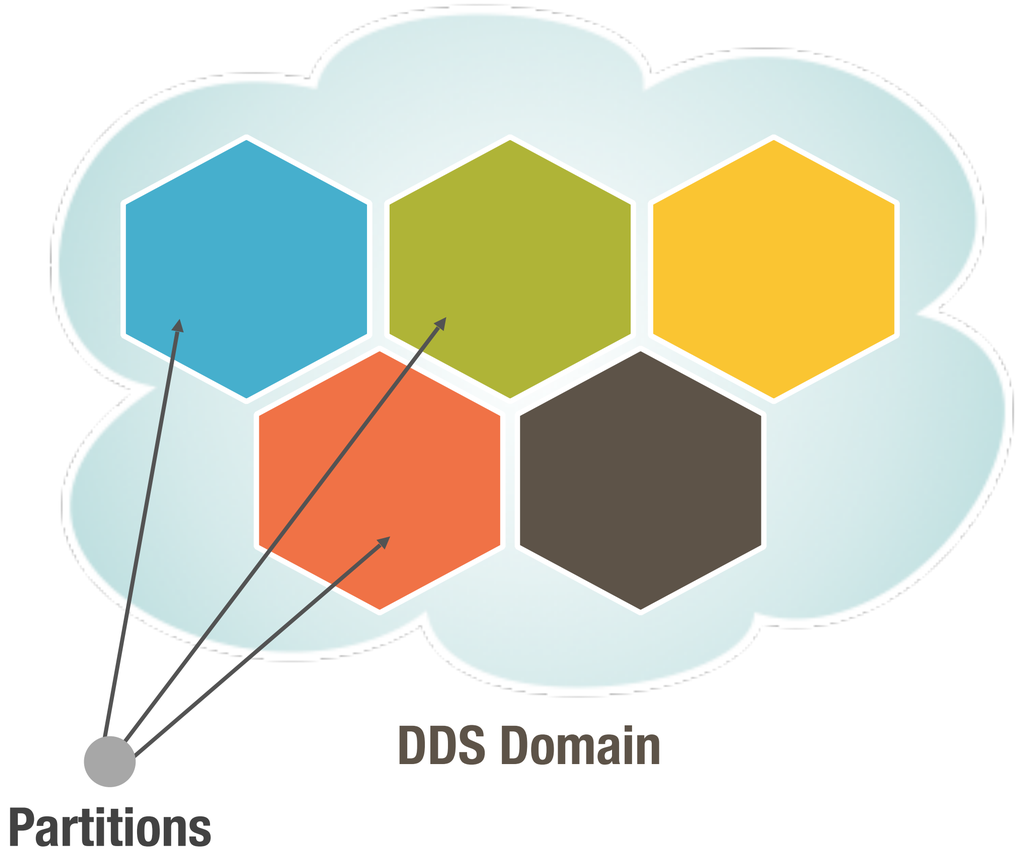
PARTITIONS

Partitions are defined as strings:

"system/log"

"system/telemetry"

"data/row-2/col-3"

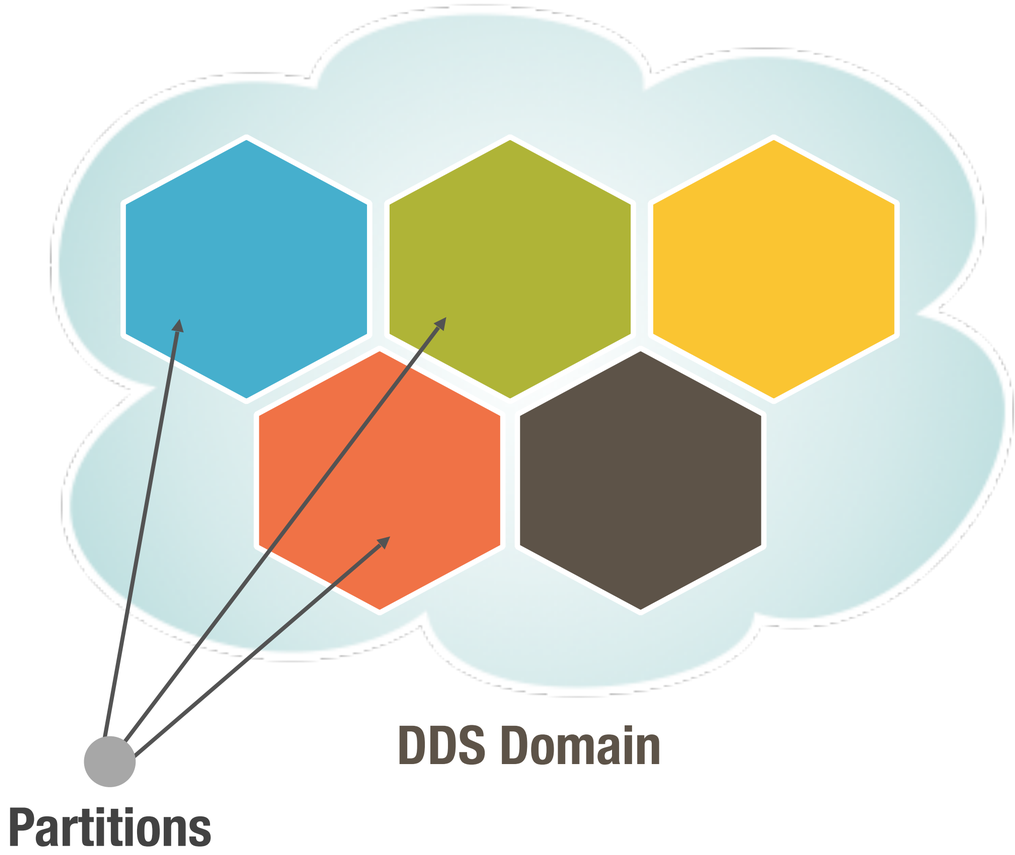


PARTITIONS

Partitions addressed by name or regular expressions:

"system/*"

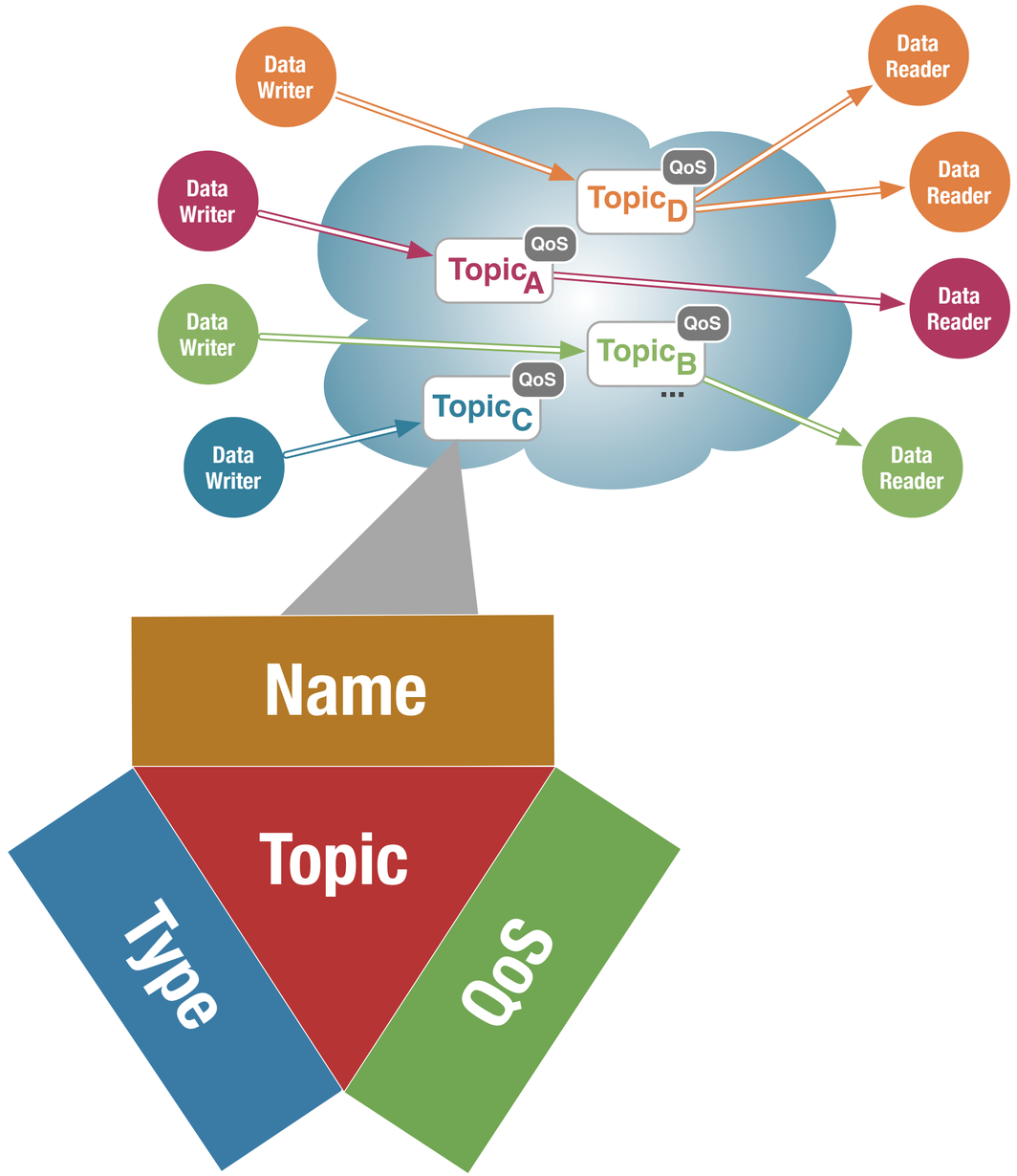
"data/row-*/col-3"



Topics

TOPIC

A domain-wide information's class A **Topic** defined by means of a <name, type, qos>



Language Independent Type Definition

TOPIC TYPE

Topic types can be expressed using different syntaxes, including IDL and ProtoBuf

IDL

```
struct CarDynamics {
   string  cid;
   long    x;   long  y;
   float   dx;  long  dy;
}
#pragma keylist CarDynamics cid
```

TOPIC TYPE

Topic types can be expressed using different syntaxes, including IDL and ProtoBuf

ProtoBuf

```
message CarDynamics {
    option (.omg.dds.type) =
      {name: "CarDynamics"};
    required string  cid = 0
      [(.omg.dds.member).key = true];
    required long    x = 1;
    required long    y = 2;
    required float   dx = 3;
    required long    dy = 4;
}
```

Language Specific Type Definition

TOPIC TYPE

Topic types can be expressed using different syntaxes, including IDL and ProtoBuf

CoffeeScript

```
class CarDynamics:
  constructor:
    (@cid, @x, @y, @dx, @dy) ->
```

TOPIC TYPE

Topic types can be expressed using different syntaxes, including IDL and ProtoBuf

C#

```
public struct CaDynamics {
        public string cid { get; set; }
        public int     x { get; set; }
        public int     y { get; set; }
        public int     dx { get; set; }
        public int     dy { get; set; }

        public CaDynamics (string cid,
                           int x, int y, int dx, int dy)
        {
            this.cid = cid;
            this.x = x;   this.y = y;
            this.dx = dx; this.dy = dy;
        }
    }
```

TOPIC TYPE

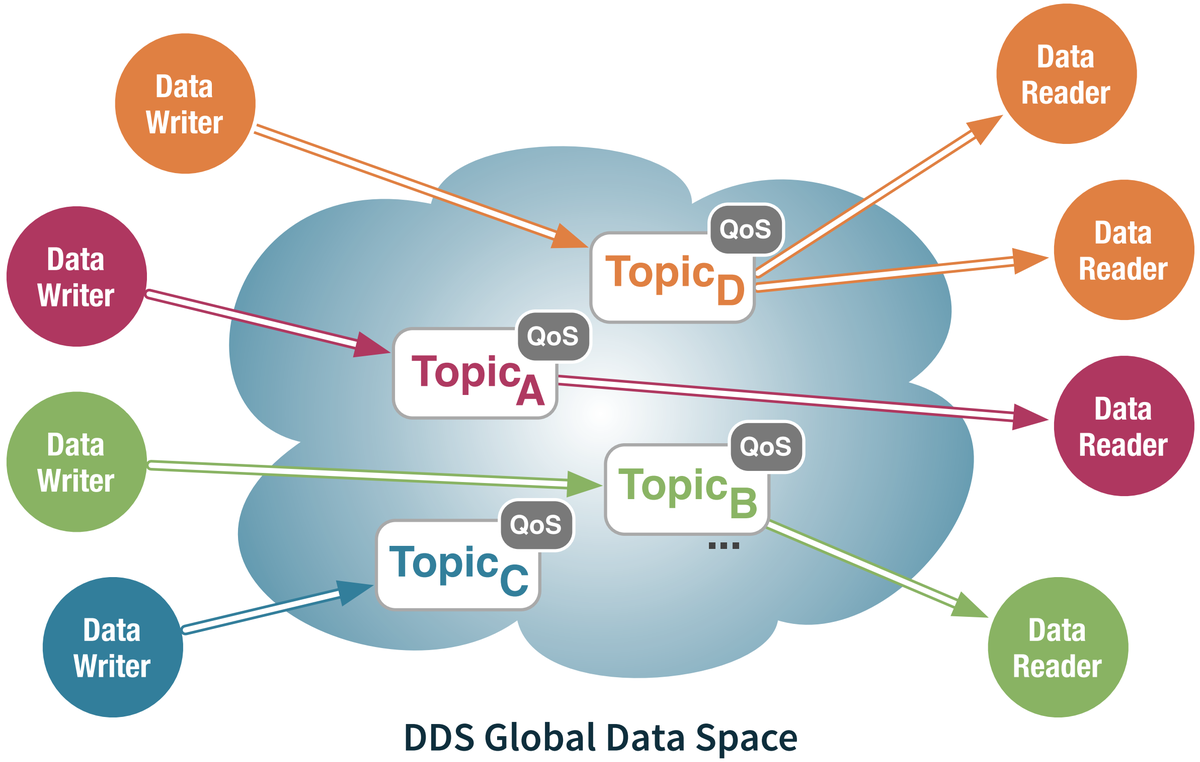
Topic types can be expressed using different syntaxes, including IDL and ProtoBuf

Java

```
@KeyList ( topicType = "CarDynamics",  keys = {"cid"})
public class CarDynamics {
    public String cid;
    public int x; public int dx;
    public int y; public int dy;

    public CarDynamics(String s, int a, int b,
                       int c,int d) {
        this.cid = s;
        this.x = a; this.dx = b;
        this.y = c; this.dy = d;
    }

    @Override
    public String toString() {
       …
    }

}
```

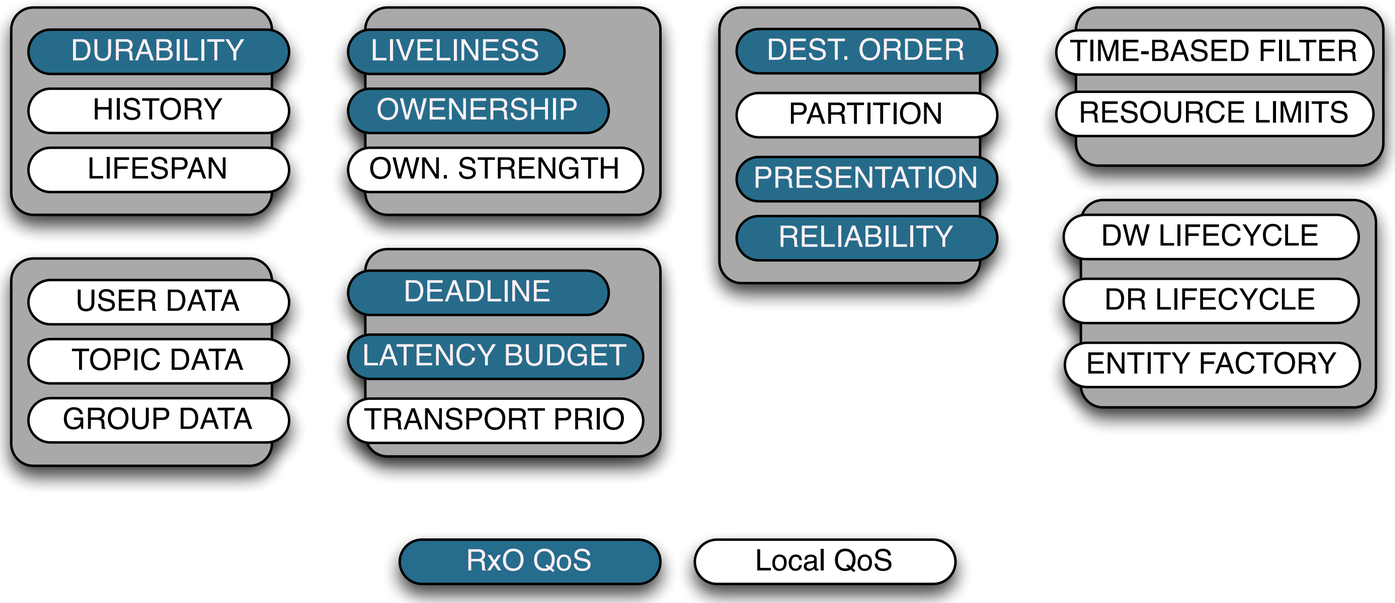
QOS - ENABLED

QoS policies allow to express temporal and availability constraints for data



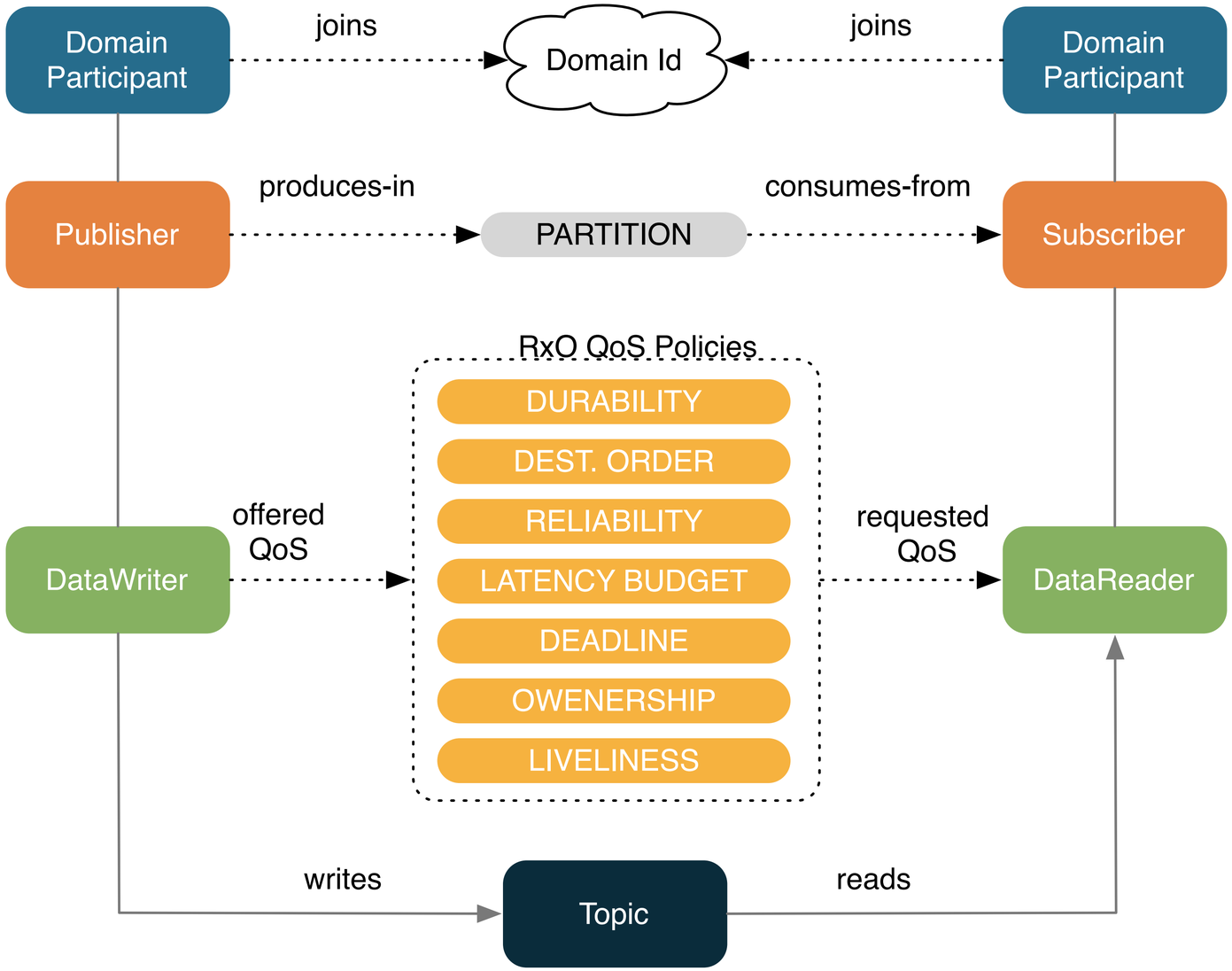
QOS

A collection of policies that control non-functional properties such as reliability, persistence, temporal constraints and priority



QOS

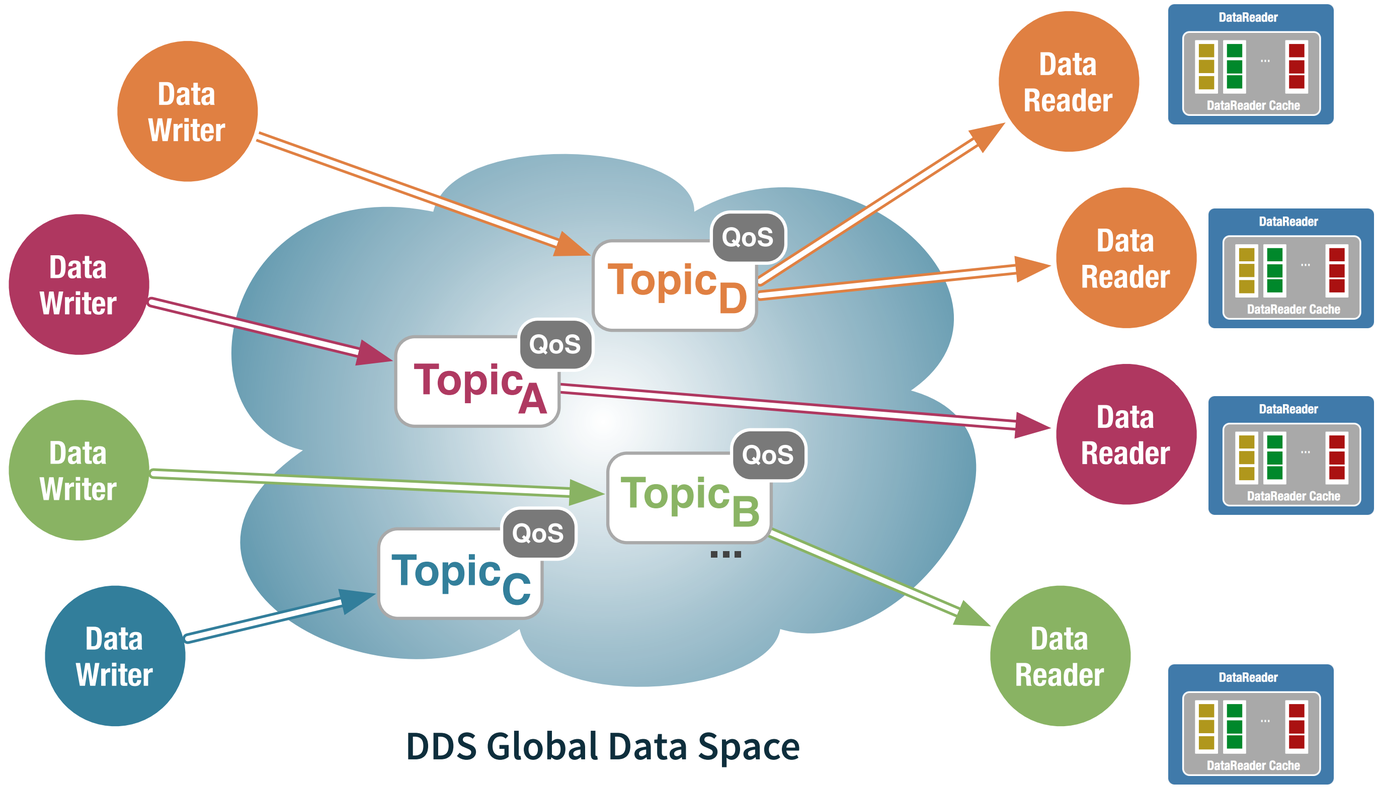
QoS Policies controlling end-to-end properties follow a Request vs. Offered



Interacting with the Data Cache

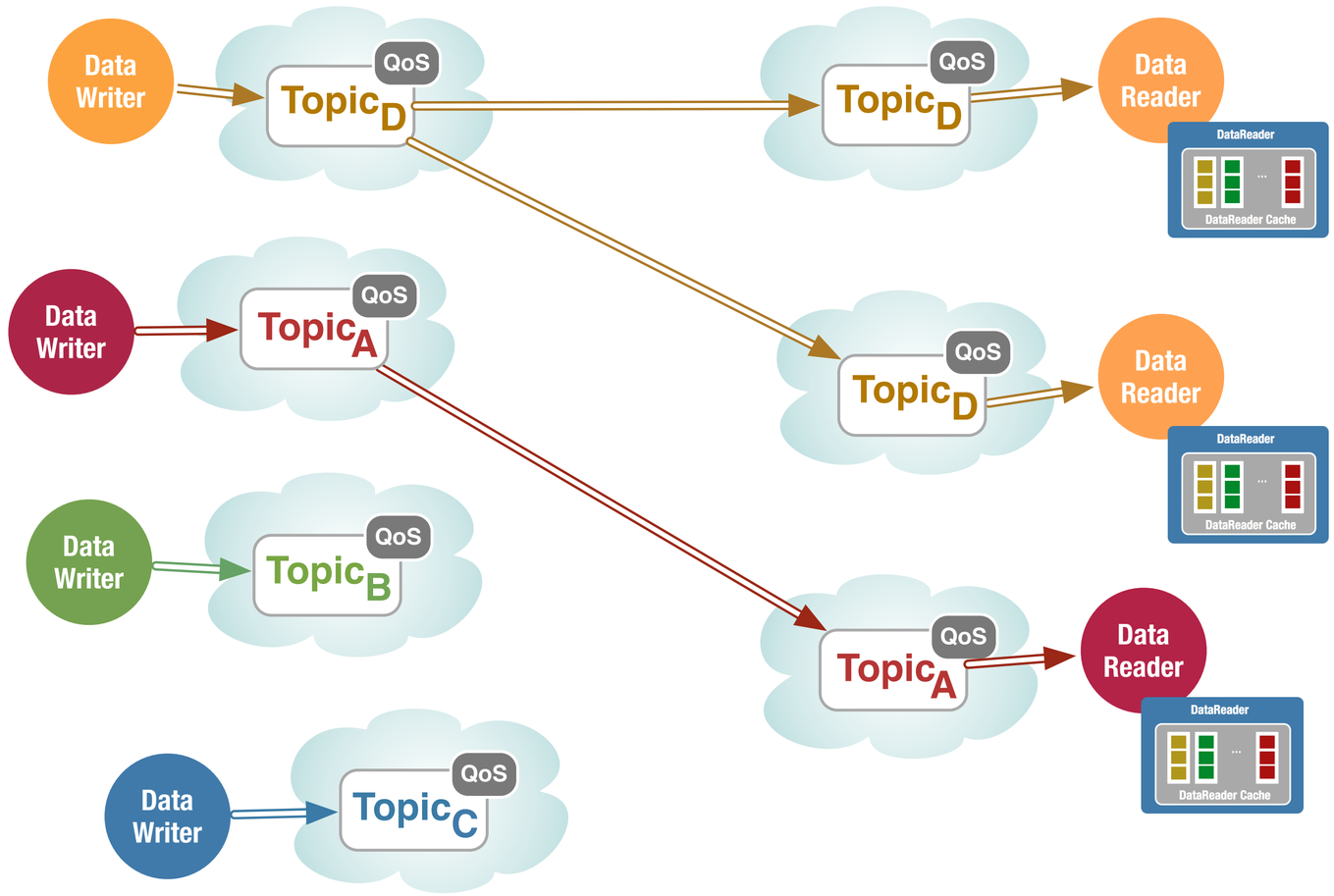
DATA CACHE

Each Data Reader is associated with a Cache



DATA CACHE

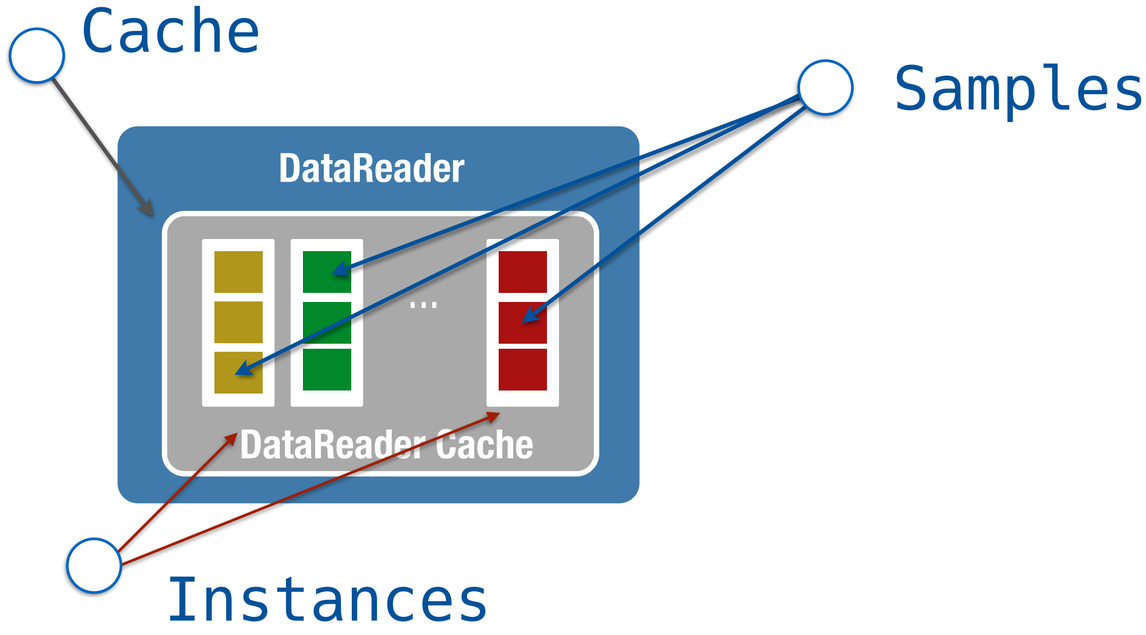
Each Data Reader is associated with a Cache



DATA CACHE

The Reader Cache stores the last $\mathbf{n} \in \boldsymbol{N}^{\infty}$ samples for each relevant instance

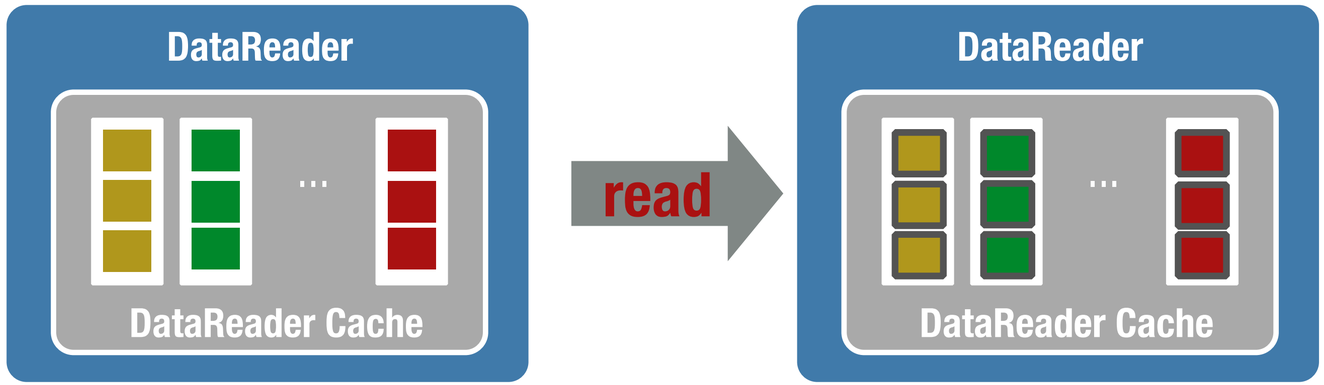
Where: $\boldsymbol{N}^{\infty} = \boldsymbol{N} \cup \{\infty\}$



READING DATA

The action of **reading** samples for a Reader Cache is non-destructive.

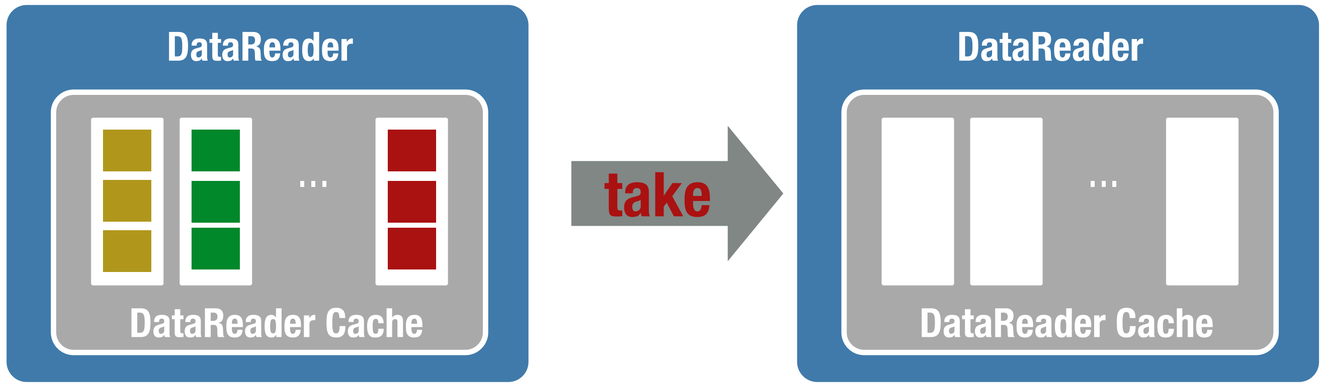
Samples are **not removed** from the cache



TAKING DATA

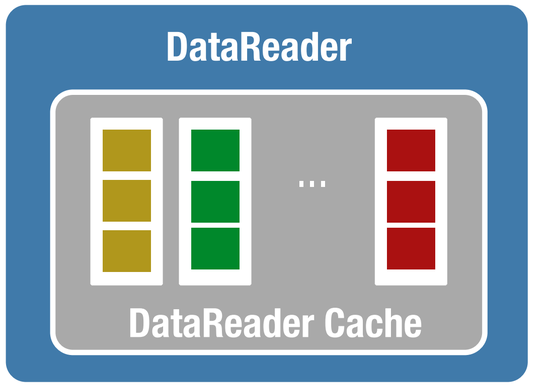
The action of **taking** samples for a Reader Cache is destructive.

Samples are **removed** from the cache



SAMPLE SELECTORS

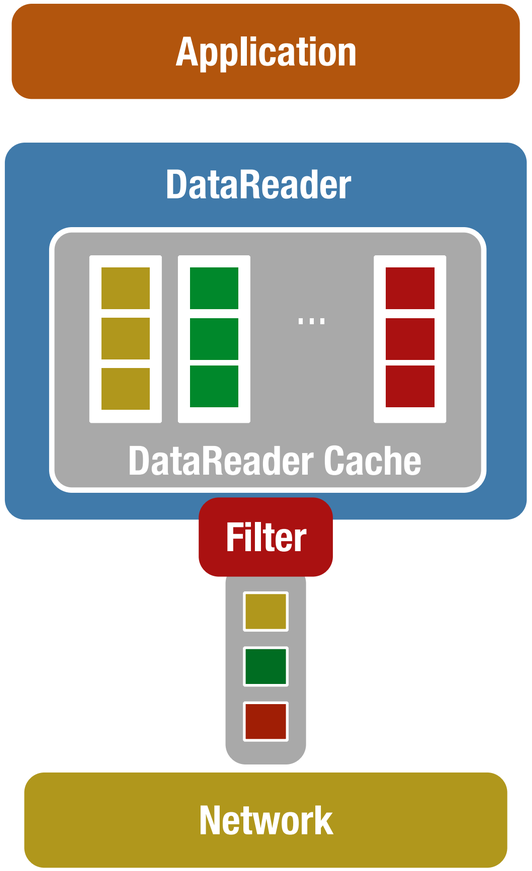
Samples can be selected
using **composable content**
and **status predicates**



CONTENT-FILTERING

Filters allow to control what gets into a DataReader cache

Filters are expressed as SQL where clauses or as Java/C/JavaScript predicates

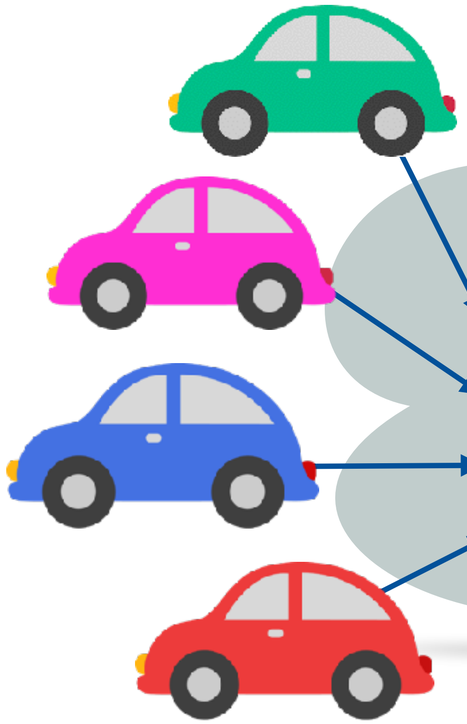


CONTENT FILTERS

Content Filters can be used to **project** on the local cache only the Topic data satisfying a given predicate

Type

```
struct CarDynamics {
    @key
    string  cid;
    long    x;    long  y;
    float   dx;   long  dy;
}
```



CarDynamics

cid	x	y	dx	dy
GR 33N GO	167	240	45	0
LO 00V IN	65	26	65	0
AN 637 OS	32	853	0	50
AB 123 CD	325	235	80	0

`"dx > 50 OR dy > 50"`

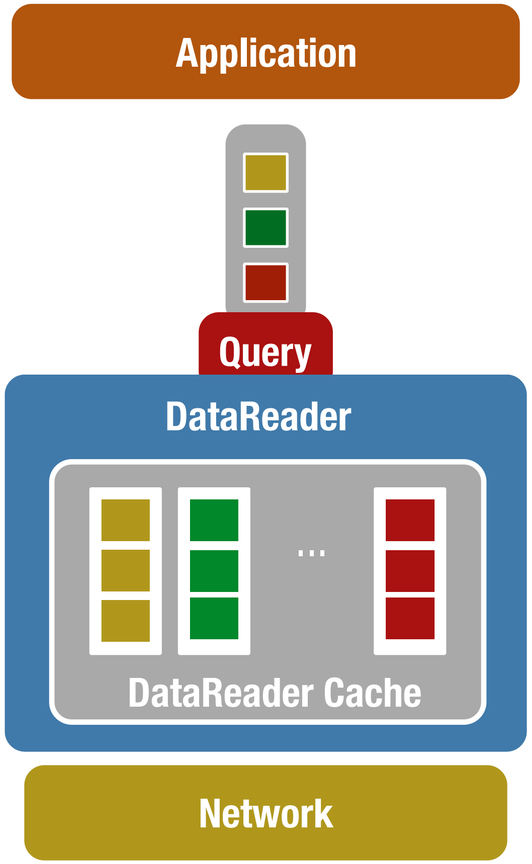
cid	x	y	dx	dy
LO 00V IN	65	26	65	0
AB 123 CD	325	235	80	0

Reader Cache

CONTENT-BASED SELECTION

Queries allow to control what gets out of a DataReader Cache

Queries are expressed as SQL where clauses or as Java/C/JavaScript predicates

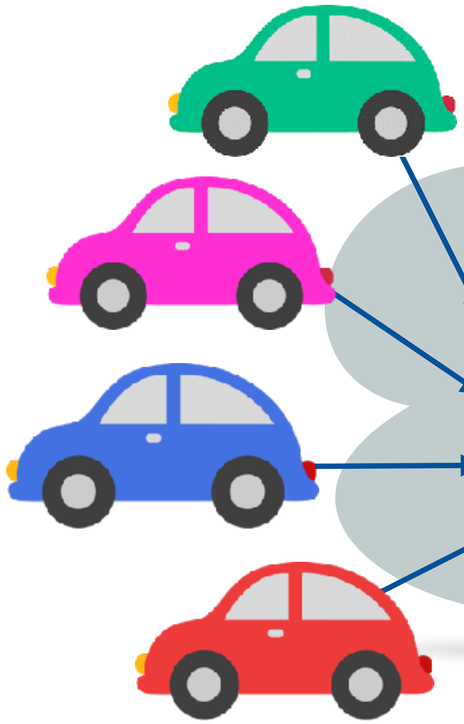


QUERIES

Queries can be used to select out of the local cache the data matching a given predicate

Type

```
struct CarDynamics {
    @key
    string  cid;
    long    x;    long  y;
    float   dx;   long  dy;
}
```



CarDynamics

cid	x	y	dx	dy
GR 33N GO	167	240	45	0
LO 00V IN	65	26	65	0
AN 637 OS	32	853	0	50
AB 123 CD	325	235	80	0

cid	x	y	dx	dy
LO 00V IN	65	26	65	0
AB 123 CD	325	235	80	0

“dx > 50 OR dy > 50” query

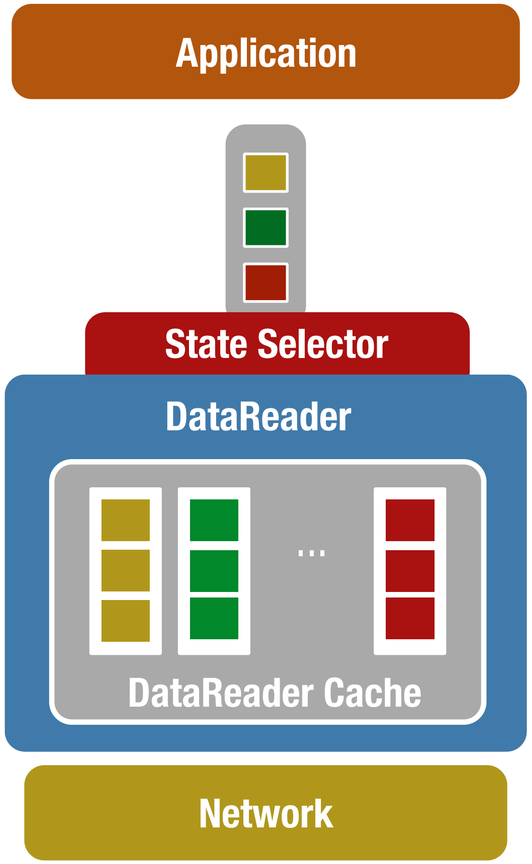
cid	x	y	dx	dy
GR 33N GO	167	240	45	0
LO 00V IN	65	26	65	0
AN 637 OS	32	853	0	50
AB 123 CD	325	235	80	0

Reader Cache

STATE-BASED SELECTION

State based selection allows to control what gets out of a DataReader Cache

State base selectors predicate on samples meta-information

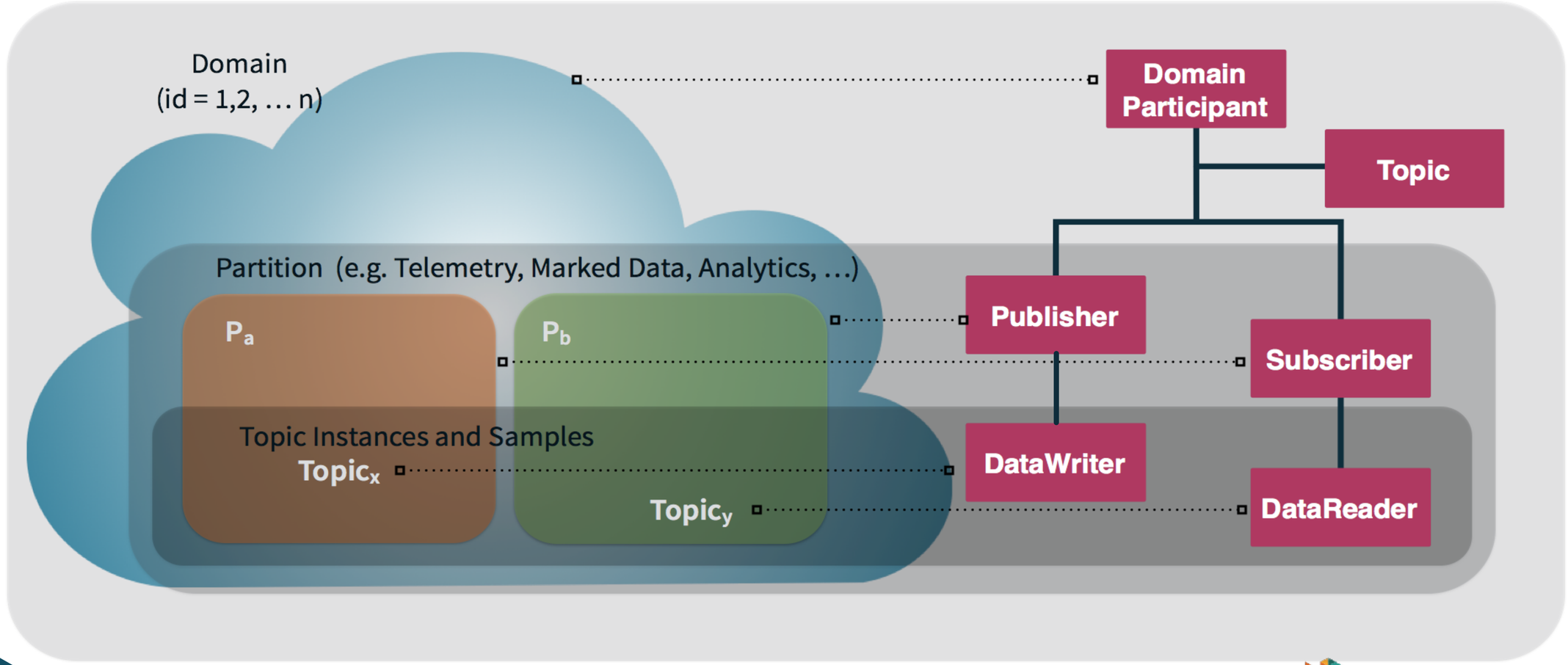


SELECTOR EXAMPLE

```
//   == ISO C++ DDS API ==

auto data =
   dr.select()
         .content(query)
         .state(data_state)
         .instance(handle)
      .read();
```

Your First DDS Application

Anatomy of a DDS Application



PRISMTECH

WRITING DATA IN C++

```
#include <dds.hpp>

int main(int, char**) {

  DomainParticipant dp(0);
  Topic<Meter> topic("SmartMeter");
  Publisher pub(dp);
  DataWriter<Meter> dw(pub, topic);

  while (!done) {
     auto value = readMeter()
     dw.write(value);
     std::this_thread::sleep_for(SAMPLING_PERIOD);
  }

  return 0;
}
```

```
enum UtilityKind {
    ELECTRICITY,
    GAS,
    WATER
};

struct Meter {
    string sn;
    UtilityKind utility;
    float reading;
    float error;
};
#pragma keylist Meter sn
```

READING DATA IN C++

```
#include <dds.hpp>

int main(int, char**) {

  DomainParticipant dp(0);
  Topic<Meter> topic("SmartMeter");
  Subscriber sub(dp);
  DataReader<Meter> dr(dp, topic);

  LambdaDataReaderListener<DataReader<Meter>> lst;
  lst.data_available = [](DataReader<Meter>& dr) {
       auto samples = data.read();
       std::for_each(samples.begin(), samples.end(), [](Sample<Meter>& sample) {
         std::cout << sample.data() << std::endl;
       }
   }
   dr.listener(lst);
   // Print incoming data up to when the user does a Ctrl-C
   std::this_thread::join();
   return 0;
}
```

```
enum UtilityKind {
    ELECTRICITY,
    GAS,
    WATER
};

struct Meter {
    string sn;
    UtilityKind utility;
    float reading;
    float error;
};
#pragma keylist Meter sn
```

WRITING DATA IN SCALA

```
import dds._
import dds.prelude._
import dds.config.DefaultEntities._

object SmartMeter {

  def main(args: Array[String]): Unit = {
    val topic = Topic[Meter](“SmartMeter”)
    val dw = DataWriter[Meter](topic)
    while (!done) {
      val meter = readMeter()
      dw.write(meter)
      Thread.sleep(SAMPLING_PERIOD)
    }
  }
}
```

```
enum UtilityKind {
    ELECTRICITY,
    GAS,
    WATER
};

struct Meter {
   string sn;
   UtilityKind utility;
   float reading;
   float error;
};
#pragma keylist Meter sn
```

READING DATA IN SCALA

```
enum UtilityKind {
    ELECTRICITY,
    GAS,
    WATER
};

struct Meter {
    string sn;
    UtilityKind utility;
    float reading;
    float error;
};
#pragma keylist Meter sn
```

```
import dds._
import dds.prelude._
import dds.config.DefaultEntities._


object SmartMeterLog {
  def main(args: Array[String]): Unit = {
    val topic = Topic[Meter](“SmartMeter”)
    val dr = DataReader[Meter](topic)
    dr listen {
      case DataAvailable(_) => dr.read.foreach(println)
    }
  }
}
```

WRITING DATA IN PYTHON

```
import dds
import time

if __name__ == '__main__':
    topic = dds.Topic("SmartMeter", "Meter")
    dw = dds.Writer(topic)

    while True:
        m = readMeter()
        dw.write(m)
        time.sleep(0.1)
```

```
enum UtilityKind {
    ELECTRICITY,
    GAS,
    WATER
};

struct Meter {
    string sn;
    UtilityKind utility;
    float reading;
    float error;
};
#pragma keylist Meter sn
```

READING DATA IN PYTHON

```
import dds
import sys

def readData(dr):
    samples = dds.range(dr.read())
    for s in samples:
        sys.stdout.write(str(s.getData()))

if __name__ == '__main__':
    t = dds.Topic("SmartMeter", "Meter")
    dr = dds.Reader(t)
    dr.onDataAvailable = readData
```

```
enum UtilityKind {
    ELECTRICITY,
    GAS,
    WATER
};

struct Meter {
    string sn;
    UtilityKind utility;
    float reading;
    float error;
};
#pragma keylist Meter sn
```

Application / DDS Interaction Models

Interaction Models

Polling

- The application proactively polls for data availability as well as special events, such as a deadline being missed, etc. Notice that all DDS API calls, exclusion made for *wait* operations, are non-blocking

Synchronous Notification

- The application synchronously waits for some conditions to be verified, e.g., data availability, instance lifecycle change, etc.

Asynchronous Notification

- The application registers the interest to be asynchronously notified when specific condition are satisfied, e.g. data available, a publication matched, etc.

Synchronous Notifications

- DDS provides a mechanism known as **WaitSet** to synchronously wait for a condition
- Condition can predicate on:
  - communication statuses
  - data availability
  - data availability with specific content
  - user-triggered conditions

WaitSet

```
//   == Java 5 DDS API ==

// Create the waitset
WaitSet ws = runtime.createWaitSet();

Subscriber.DataState ds = sub.createDataState();

// Create the condition
Condition c = dr.createReadCondition(
                              ds.withAnyViewState()
                    .with(InstanceState.ALIVE)
                    .with(SampleState.NOT_READ));

// Attach the condition
ws.attachCondition(c);

// Wait for the condition
ws.wait();
```

Asynchronous Notifications

- DDS provides a mechanism known as **Listeners** for asynchronous notification of a given condition
- Listener interest can predicate on:
  - communication statuses
  - data availability

Listener Declaration

```
//   == ISO C++ DDS API ==

class ShapeListener : public dds::sub::NoOpDataReaderListener<ShapeType> {
public:
  ShapeListener() {}

  virtual void on_data_available(dds::sub::DataReader<ShapeType>& dr) {
    auto samples = dr.read();
    std::for_each(samples.begin(),
          samples.end(),
          [](const dds::sub::Sample<ShapeType>& sample) {
            if (sample.info().valid())  // Check if sample contains valid data
              std::cout << sample.data() << std::endl;
          });
  }

  virtual void on_liveliness_changed(dds::sub::DataReader<ShapeType>& the_reader,
                     const dds::core::status::LivelinessChangedStatus& status)
  {
    std::cout << ">> Liveliness Changed! " << std::endl;
  }
};
```

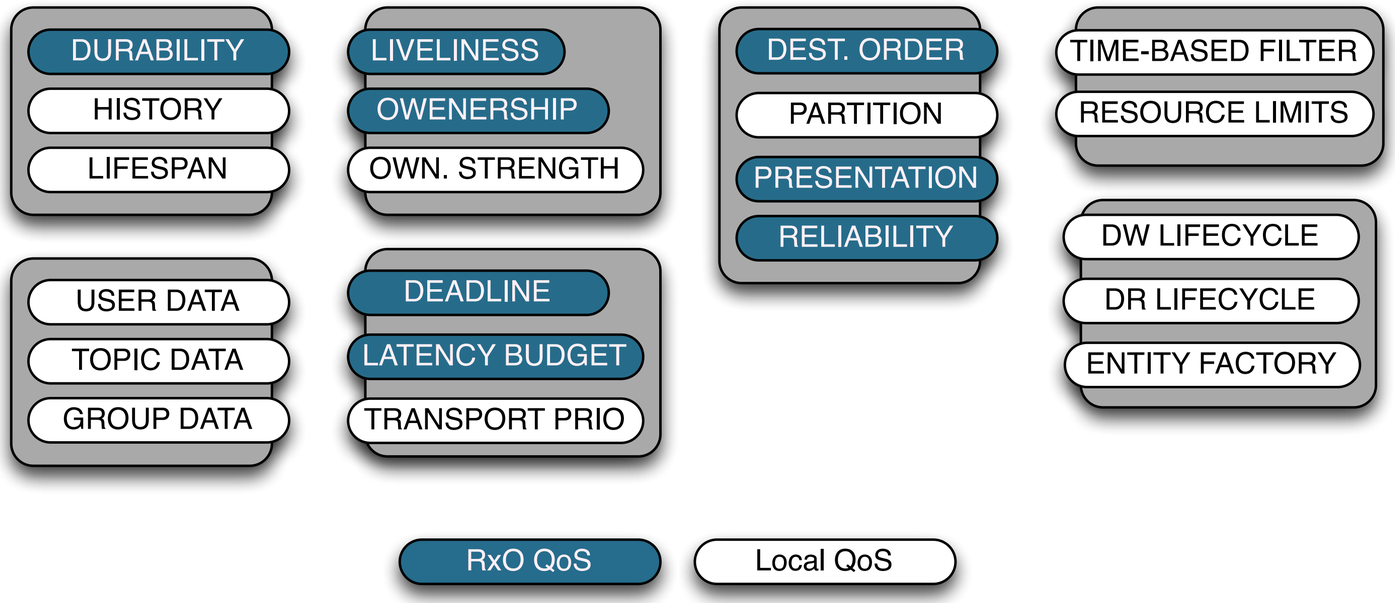
Listener Registration

```
//   == ISO C++ DDS API ==

auto l = new ShapeListener();

// Create a “nothing” status mask
StatusMask mask = StatusMask::none();

// Add the statuses we are interested in.
mask << StatusMask::data_available()
     << StatusMask::liveliness_changed()
     << StatusMask::liveliness_lost();

// Register the listener with the associated mask
dr.listener(l, mask);
```

:: Quality of Service ::

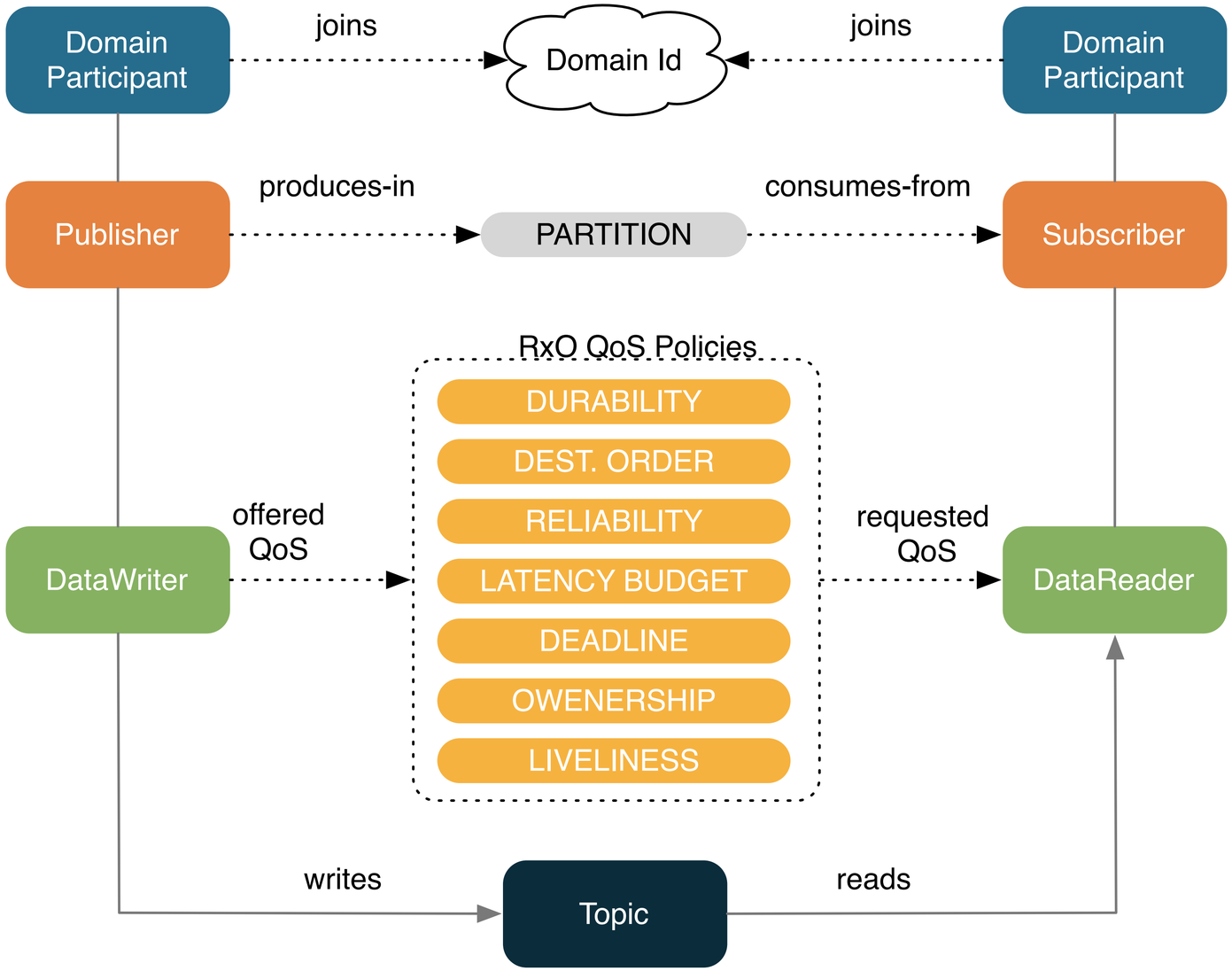
QOS

A collection of policies that control non-functional properties such as reliability, persistence, temporal constraints and priority



QOS

QoS Policies controlling end-to-end properties follow a Request vs. Offered



QoS DSL

- The ISO C++ and Java 5 APIs provide DSL for dealing with QoS Policies configuration
- The DSL uses language specific idioms, such as fluid interfaces, as well as specific features of the languages
- **Policies** as well as Entity QoS are **immutable** — this allows for better safety and object sharing
- Policies are treated as **algebraic data types** and the DSL provide constructors of each of the cases
- A QoS Provider can now be used to retrieve QoS settings from external sources, e.g. a file, an HTTP server, DDS durability

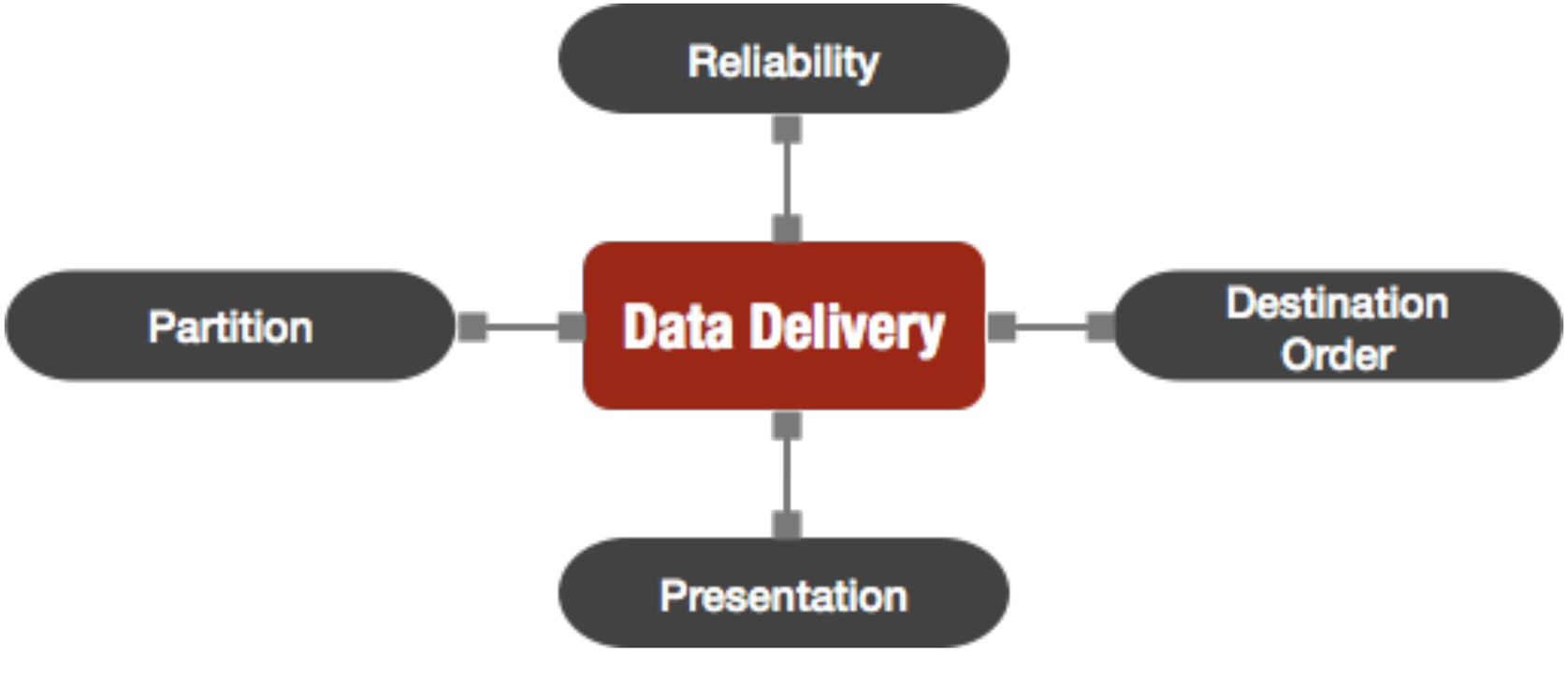
C++ QoS Policy DSL

```
//   == ISO C++ DDS API ==

DataWriterQos dwqos = pub.default_datawriter_qos()
  << History.KeepLast(10)
  << Durability.Transient();
```

Java 5 QoS Policy DSL

```
//   == Java 5 DDS API ==

final PolicyFactory pf = ...

DataWriterQos dwqos = pub.getDefaultDataWriterQos()
    .withPolicies (
        pf.History.withKeepLast(10),
        pf.Durability.withTransient(),
  );
```

Data Delivery



Reliability QoS Policy

The Reliability Policy controls the level of guarantee offered by the DDS in delivering data to subscribers

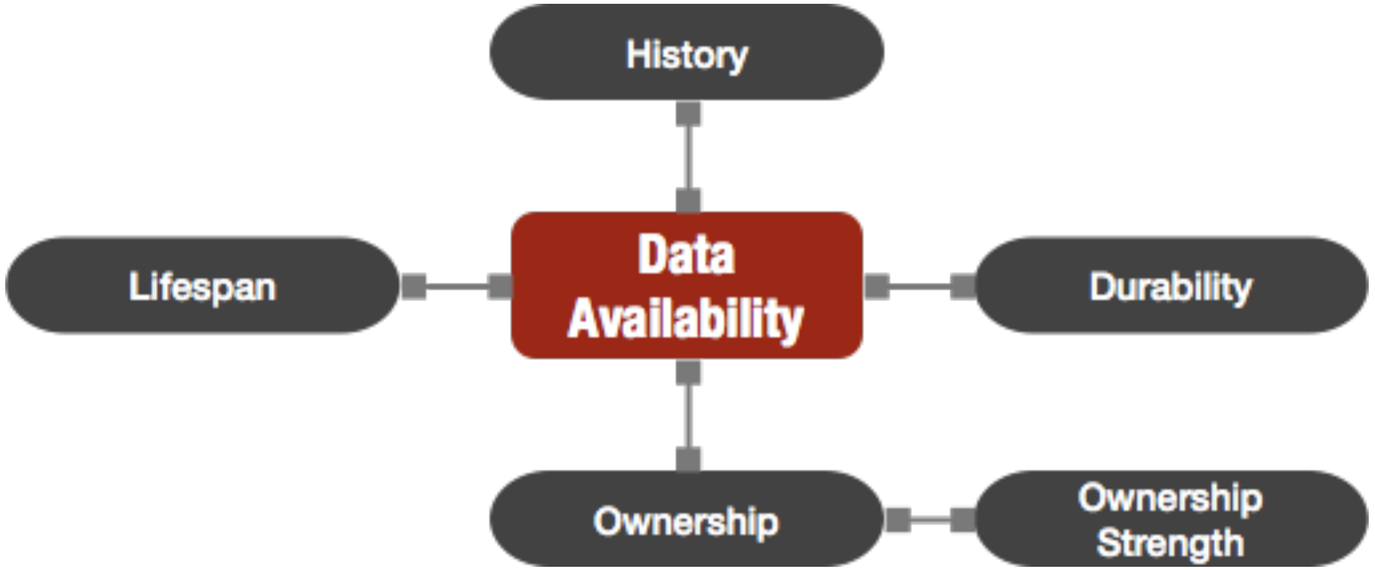
- **Reliable**. In steady-state, and with no data writer crashes, guarantees that all samples in the DataWriter history will eventually be delivered to all the DataReader

- **Best Effort**. Indicates that it is acceptable not to retry propagation of samples

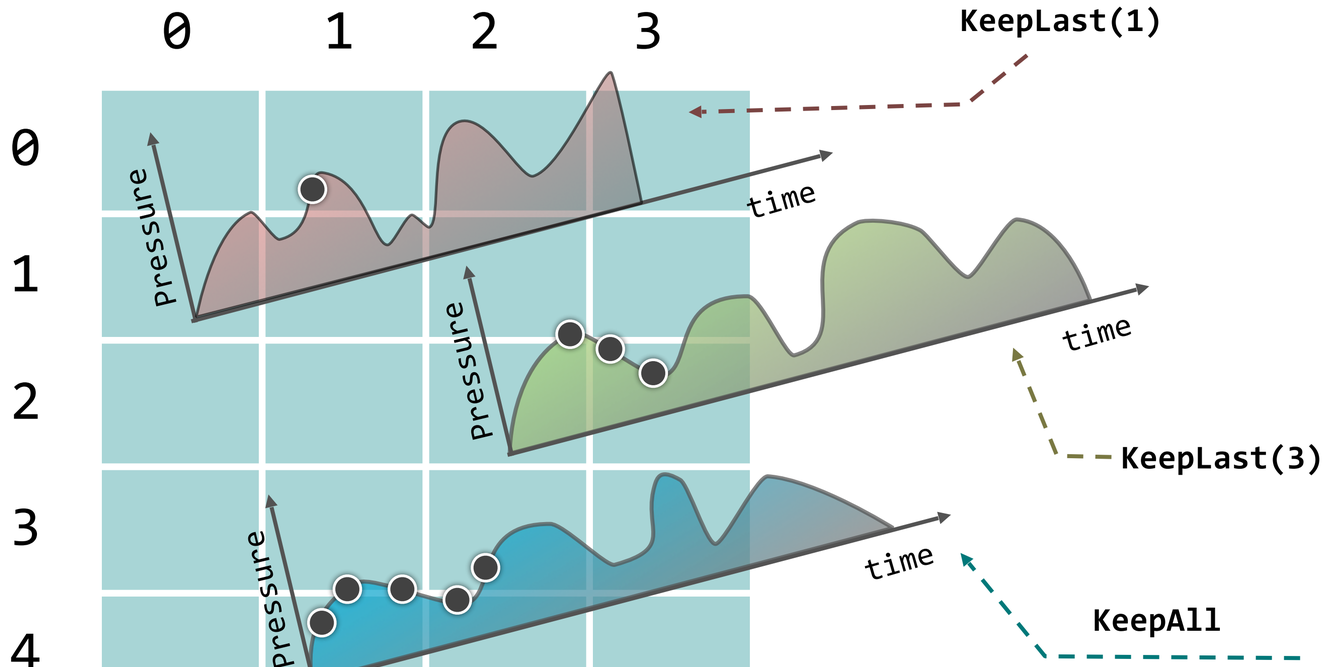
QoS Policy	Applicability	RxO	Modifiable
RELIABILITY	T, DR, DW	Y	N

Reliability Semantics

Data Availability



History QoS Policy



The **DataWriter** HISTORY QoS Policy controls the amount of data that can be made available to late joining DataReaders under TRANSIENT_LOCAL Durability

The **DataReader** HISTORY QoS Policy controls how many samples will be kept on the reader cache

- **Keep Last.** DDS will keep the most recent "depth" samples of each instance of data identified by its key

- **Keep All.** The DDS keep all the samples of each instance of data identified by its key -- up to reaching some configurable resource limits

QoS Policy	Applicability	RxO	Modifiable
HISTORY	T, DR, DW	N	N

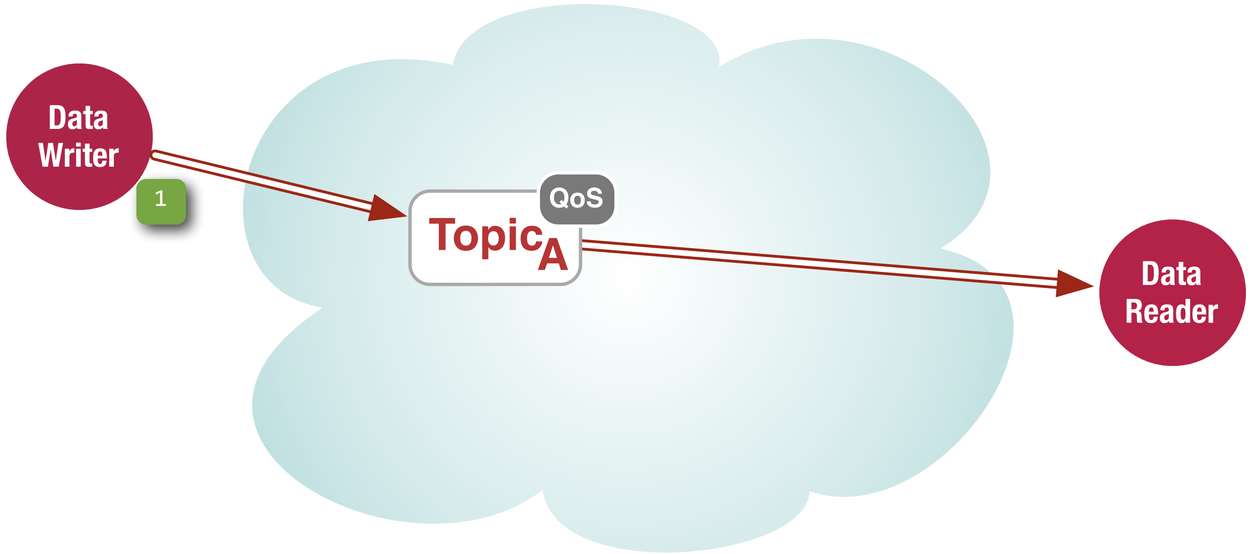
Durability QoS Policy

The DURABILITY QoS controls the data availability w.r.t. late joiners, specifically the DDS provides the following variants:

- **Volatile**. No need to keep data instances for late joining data readers
- **Transient Local**. Data instance availability for late joining data reader is tied to the data writer availability
- **Transient**. Data instance availability outlives the data writer
- **Persistent**. Data instance availability outlives system restarts

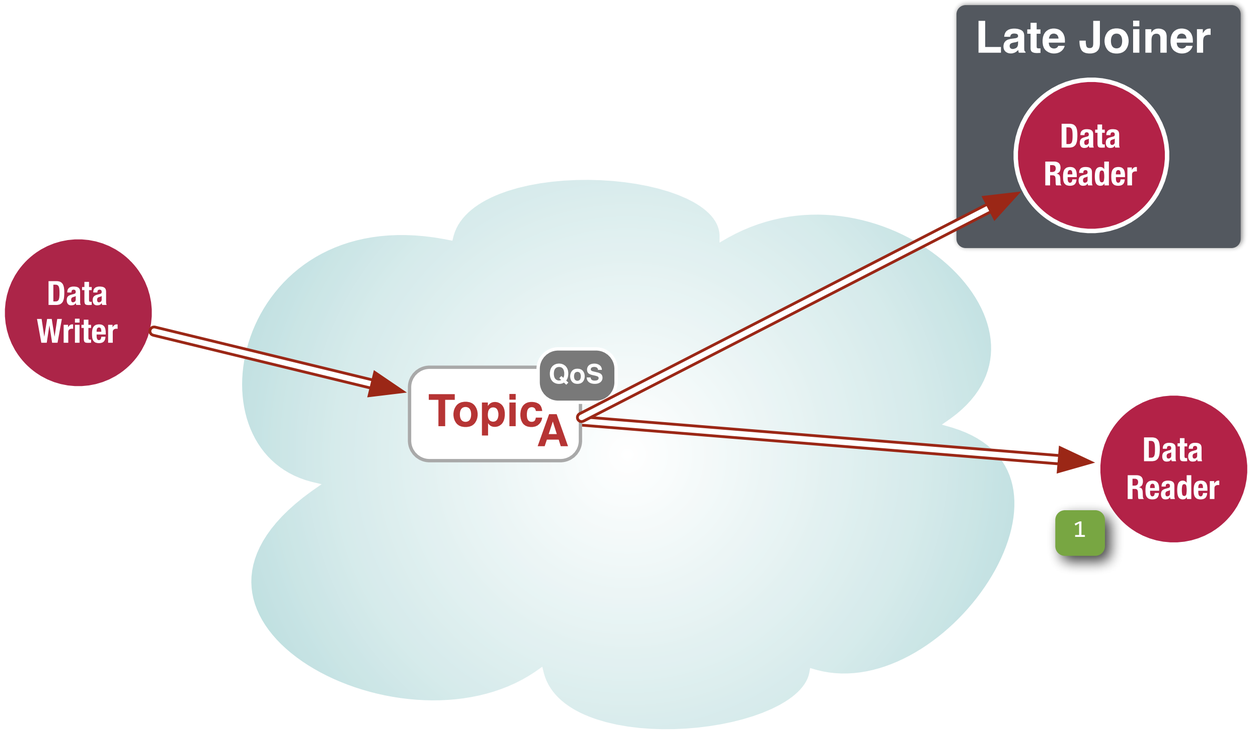
QoS Policy	Applicability	RxO	Modifiable
DURABILITY	T, DR, DW	Y	N

Volatile Durability



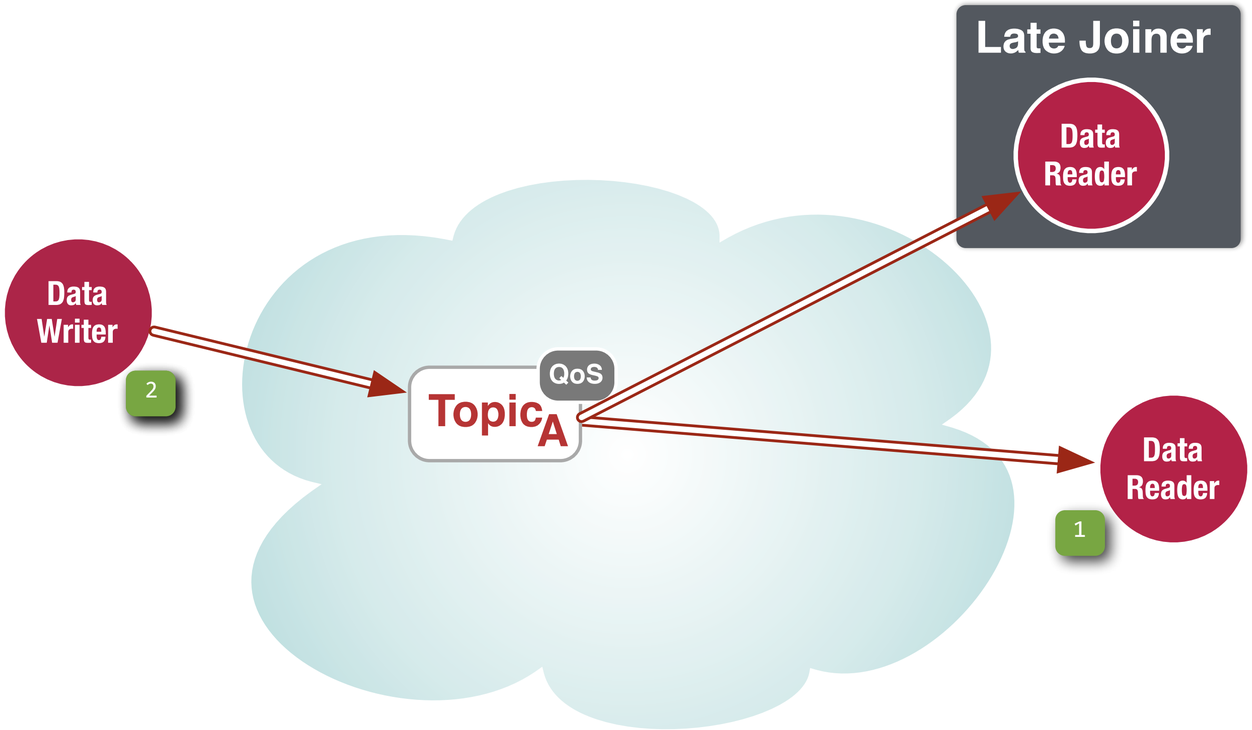
- **No Time Decoupling**
- Readers get only data produced after they joined the Global Data Space

Volatile Durability



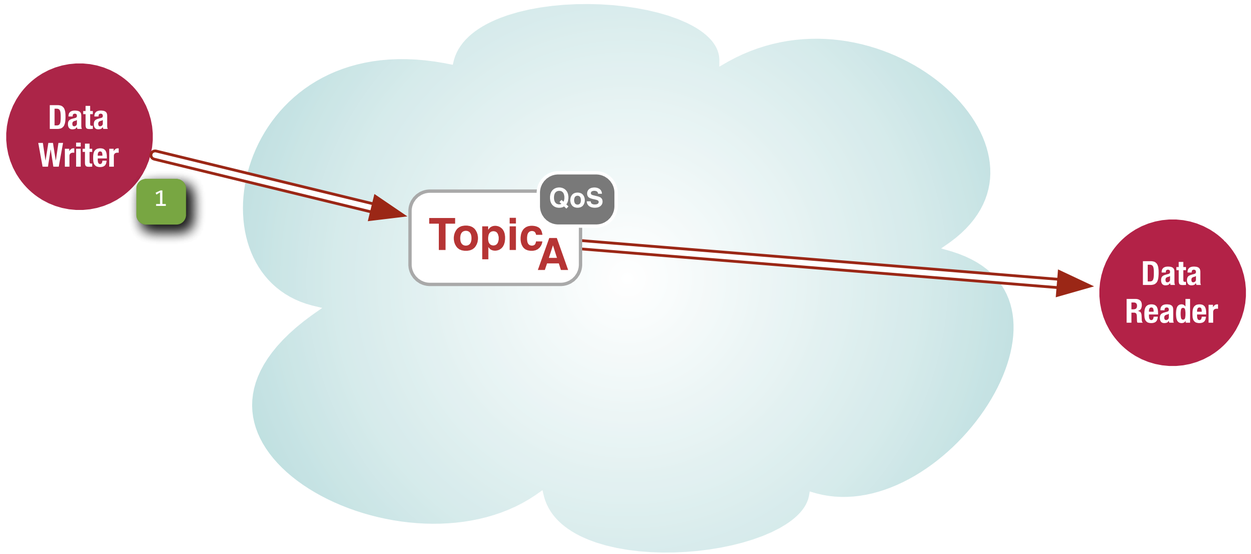
- **No Time Decoupling**
- Readers get only data produced after they joined the Global Data Space

Volatile Durability



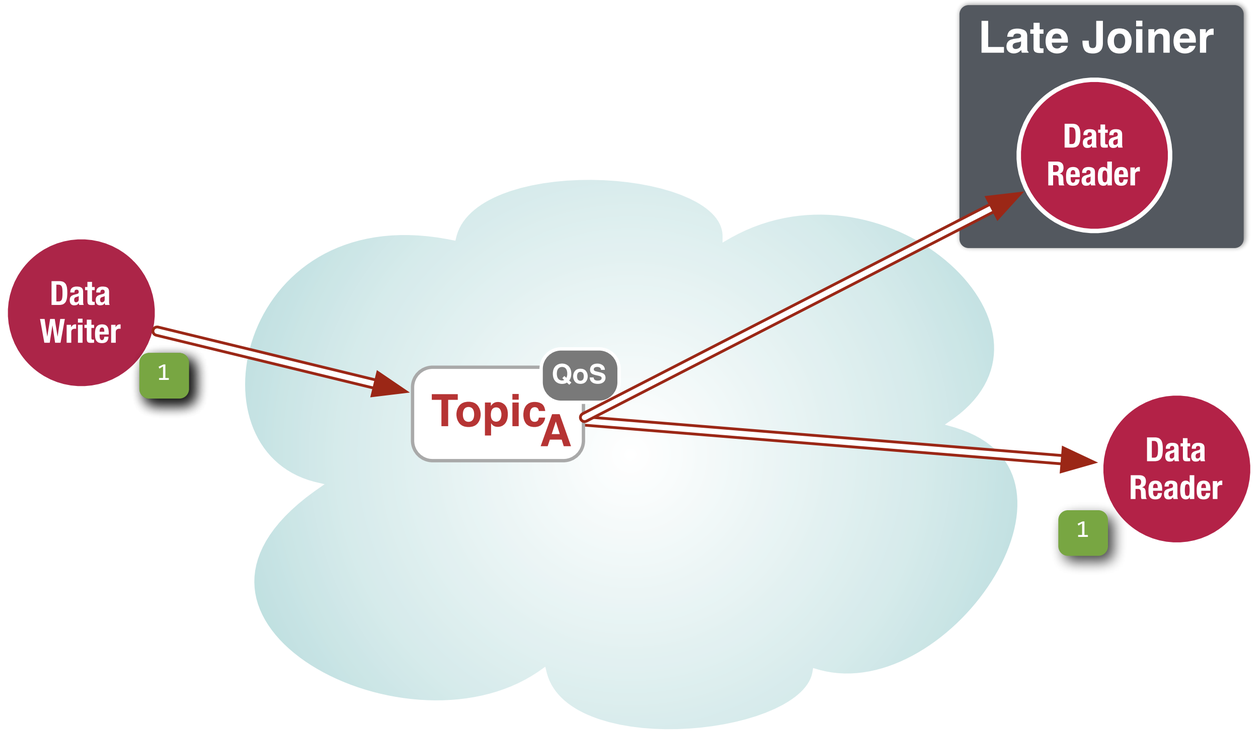
- **No Time Decoupling**
- Readers get only data produced after they joined the Global Data Space

Transient Local Durability



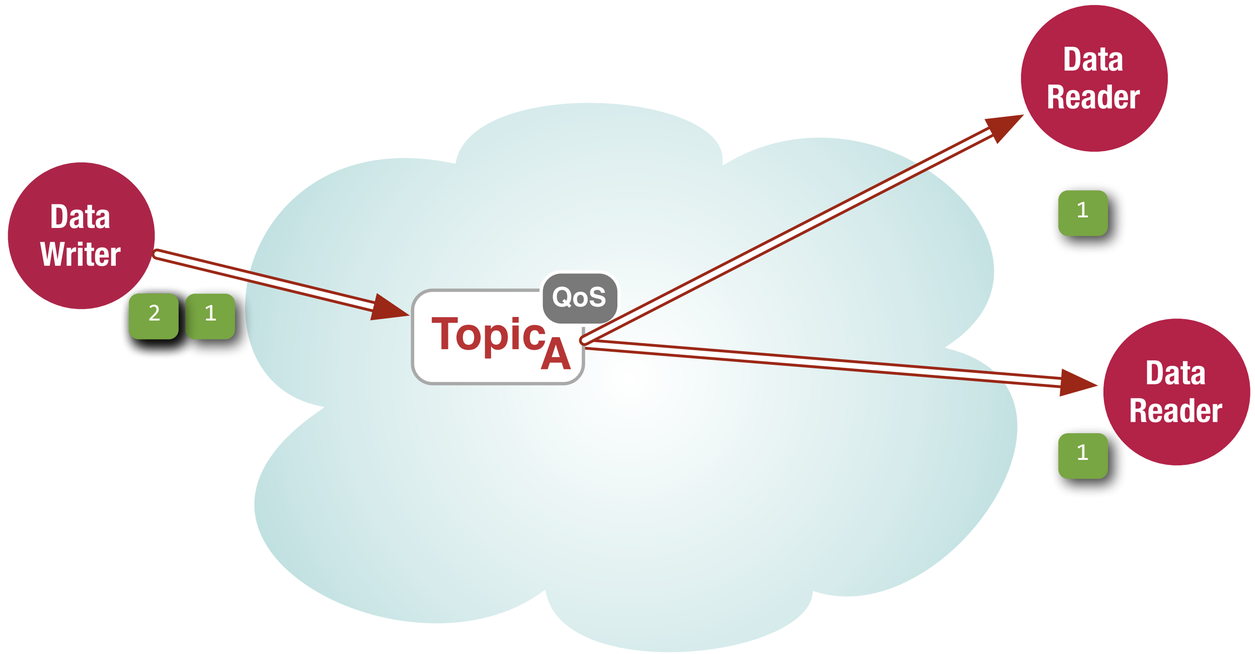
- **Some Time Decoupling**
- Data availability is tied to the availability of the data writer and the history settings

Transient Local Durability



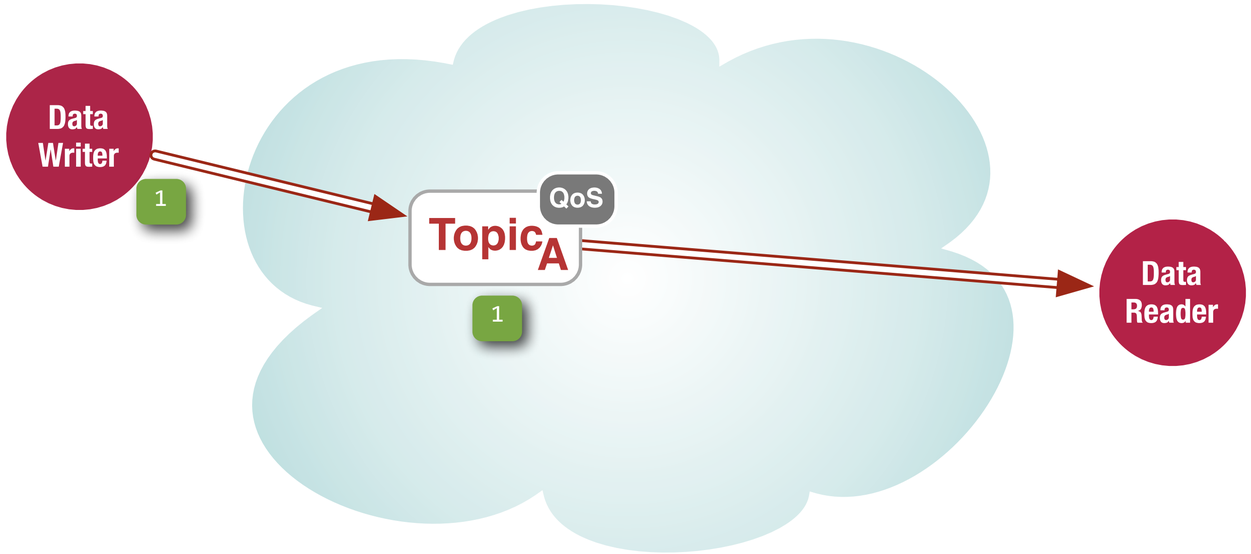
- **Some Time Decoupling**
- Data availability is tied to the availability of the data writer and the history settings

Transient-Local Durability



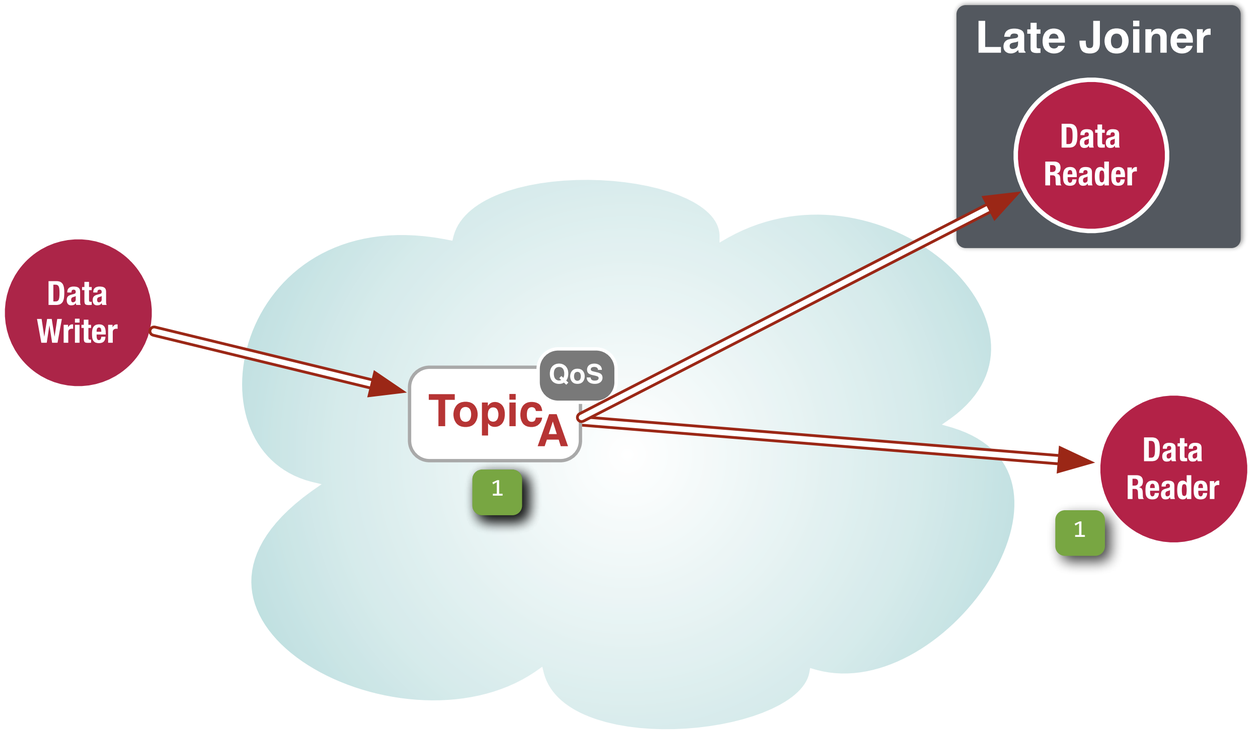
- **Some Time Decoupling**
- Data availability is tied to the availability of the data writer and the history settings

Transient Durability



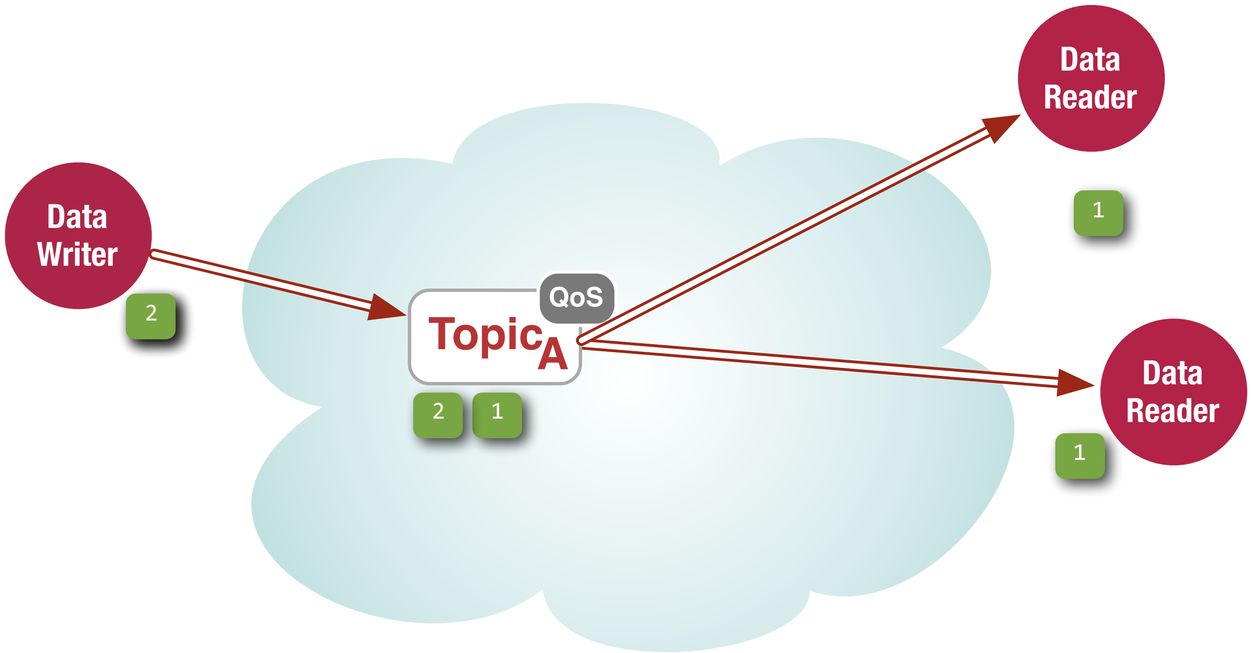
- **Time Decoupling**
- Data availability is tied to the availability of the durability service

Transient Durability



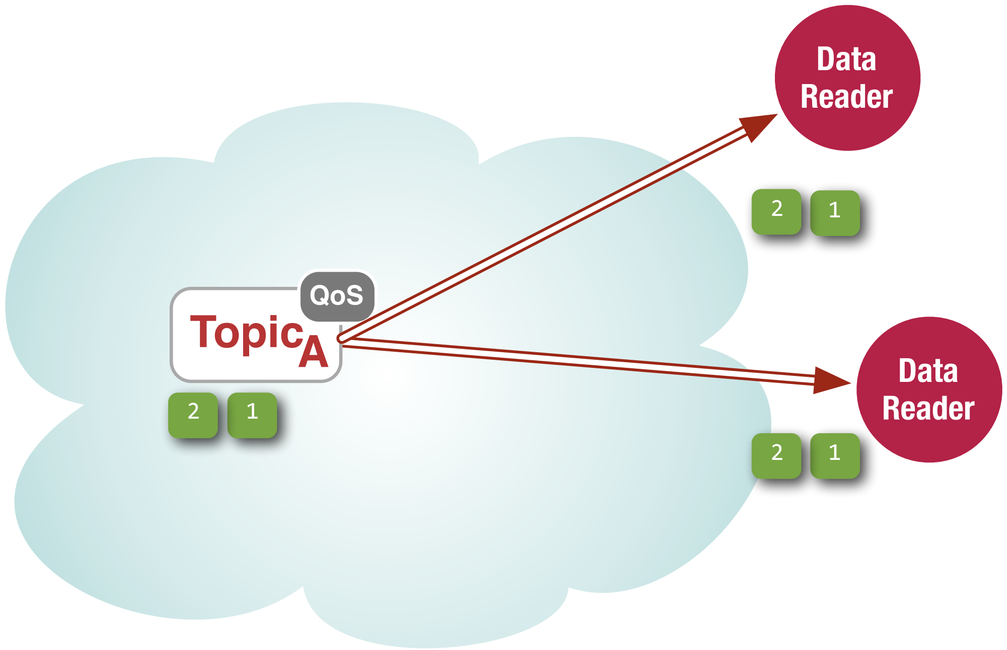
- **Time Decoupling**
- Data availability is tied to the availability of the durability service

Transient Durability



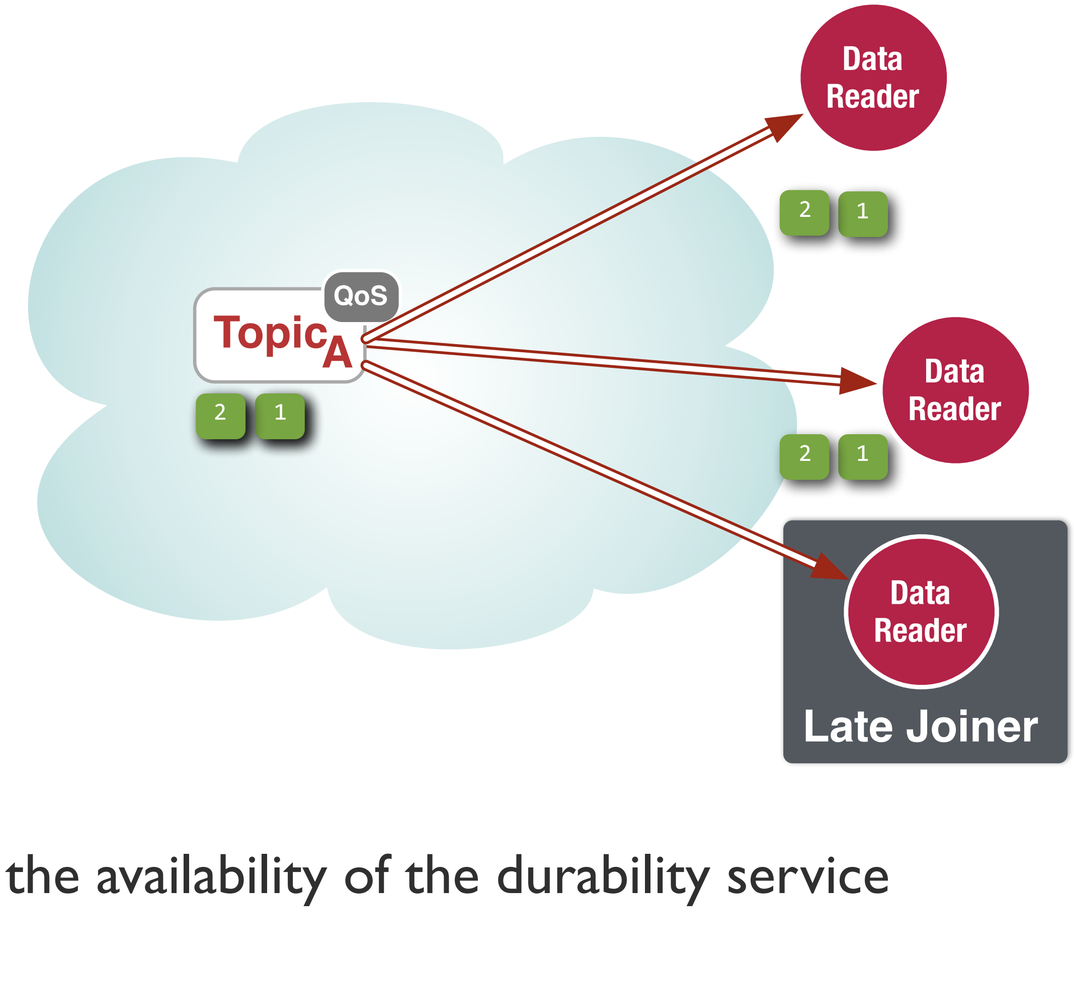
- **Time Decoupling**
- Data availability is tied to the availability of the durability service

Transient Durability



- **Time Decoupling**
- Data availability is tied to the availability of the durability service

Transient Durability



- **Time Decoupling**
- Data availability is tied to the availability of the durability service

Idioms

Soft State

- In distributed systems you often need to model **soft-state** -- a state that is **periodically updated**
- Examples are the reading of a sensor (e.g. Temperature Sensor), the position of a vehicle, etc.
- The **QoS** combination to model **Soft-State** is the following:

```
Reliability       =>  BestEffort
Durability        =>  Volatile
History           =>  KeepLast(n) [with n = 1 in most of the cases]
Deadline          =>  updatePeriod
LatencyBudget     =>  updatePeriod/3 [rule of thumb]
DestinationOrder  =>  SourceTimestamp [if multiple writers per instance]
```

Hard State

- In distributed systems you often need to model **hard-state** -- a state that is **sporadically updated** and that often has **temporal persistence** requirements
- Examples are system configuration, a price estimate, etc.
- The **QoS** combination to model **Hard-State** is the following:

```
Reliability       =>  Reliable
Durability        =>  Transient | Persistent
History           =>  KeepLast(n) [with n = 1 in most of the cases]
DestinationOrder  =>  SourceTimestamp [if multiple writers per instance]
```

Events

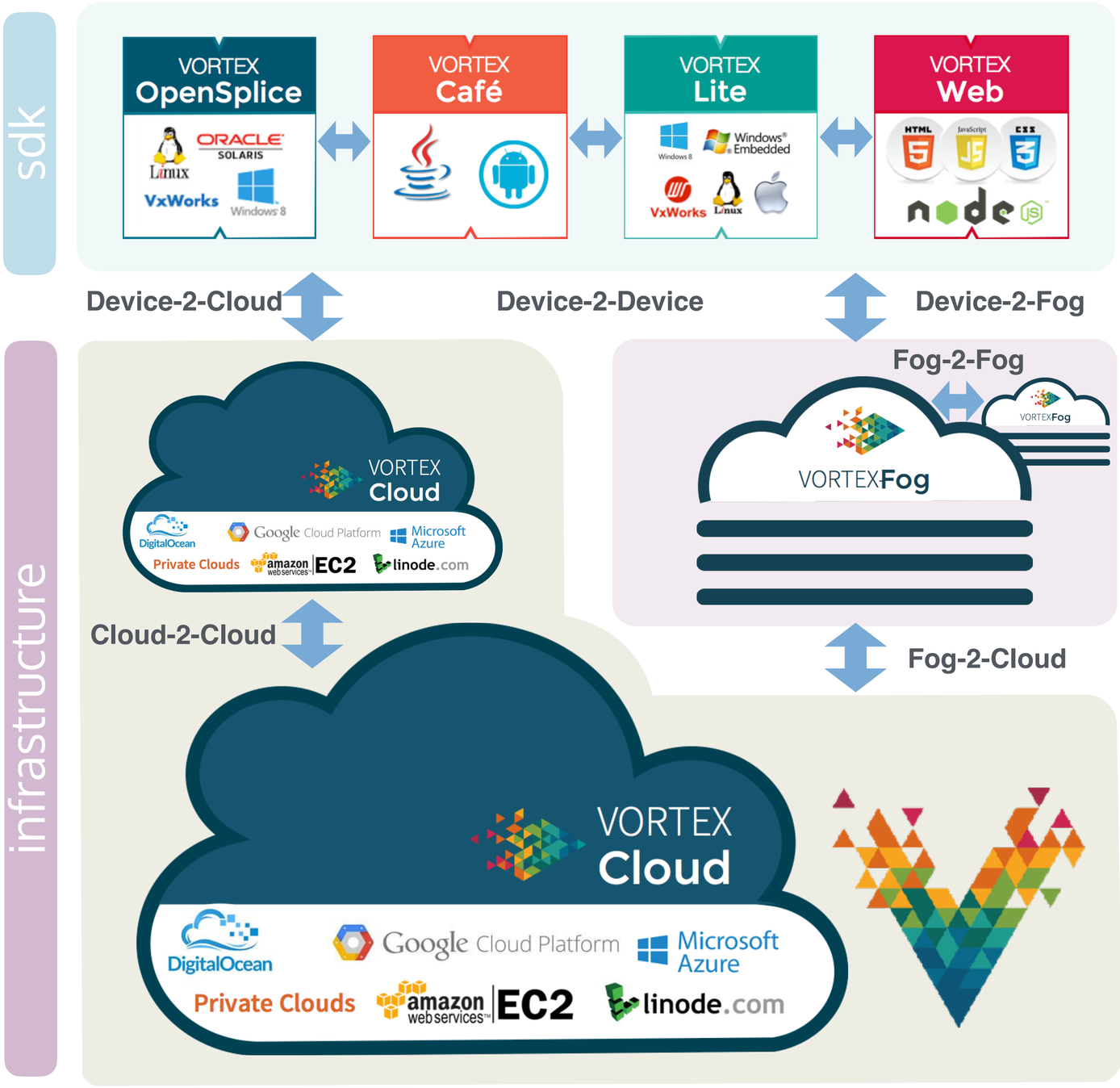
- In distributed systems you often need to model **events** -- the **occurrence** of something noteworthy for our system
- Examples are a collision alert, the temperature beyond a given threshold, etc.
- The **QoS** combination to model **Events** is the following:

```
Reliability       =>  Reliable
Durability        =>  any          [depends on system requirements]
History           =>  KeepAll [on both DataWriter and DataReader!]
DestinationOrder  =>  SourceTimestamp
```

Vortex Technology Stack

Device implementations optimised for OT, IT and consumer platforms

Native support for Cloud and Fog Computing Architectures



Platforms Support

Available across IT, Consumer and OT platforms

iOS

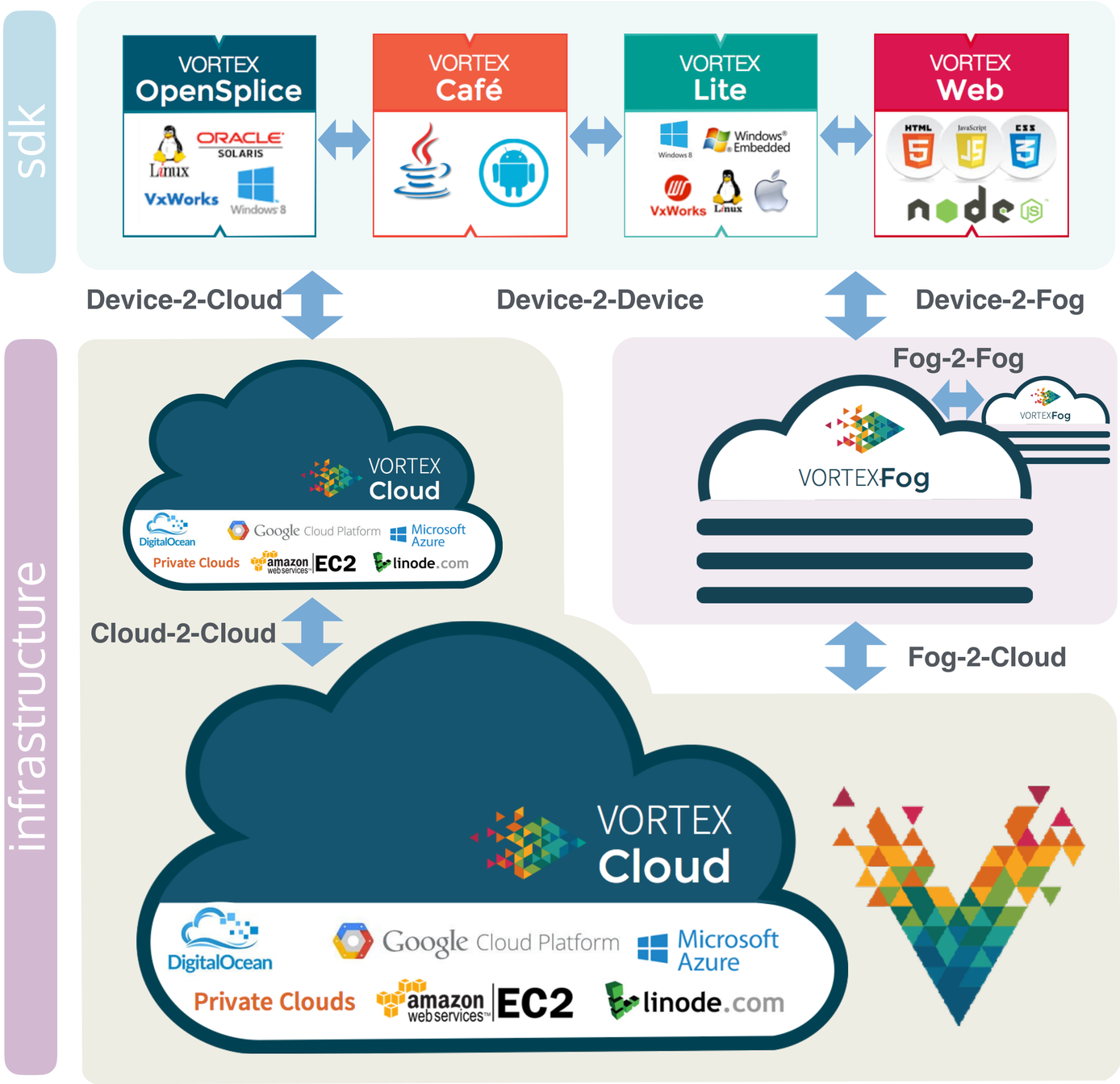
HTML

QNX

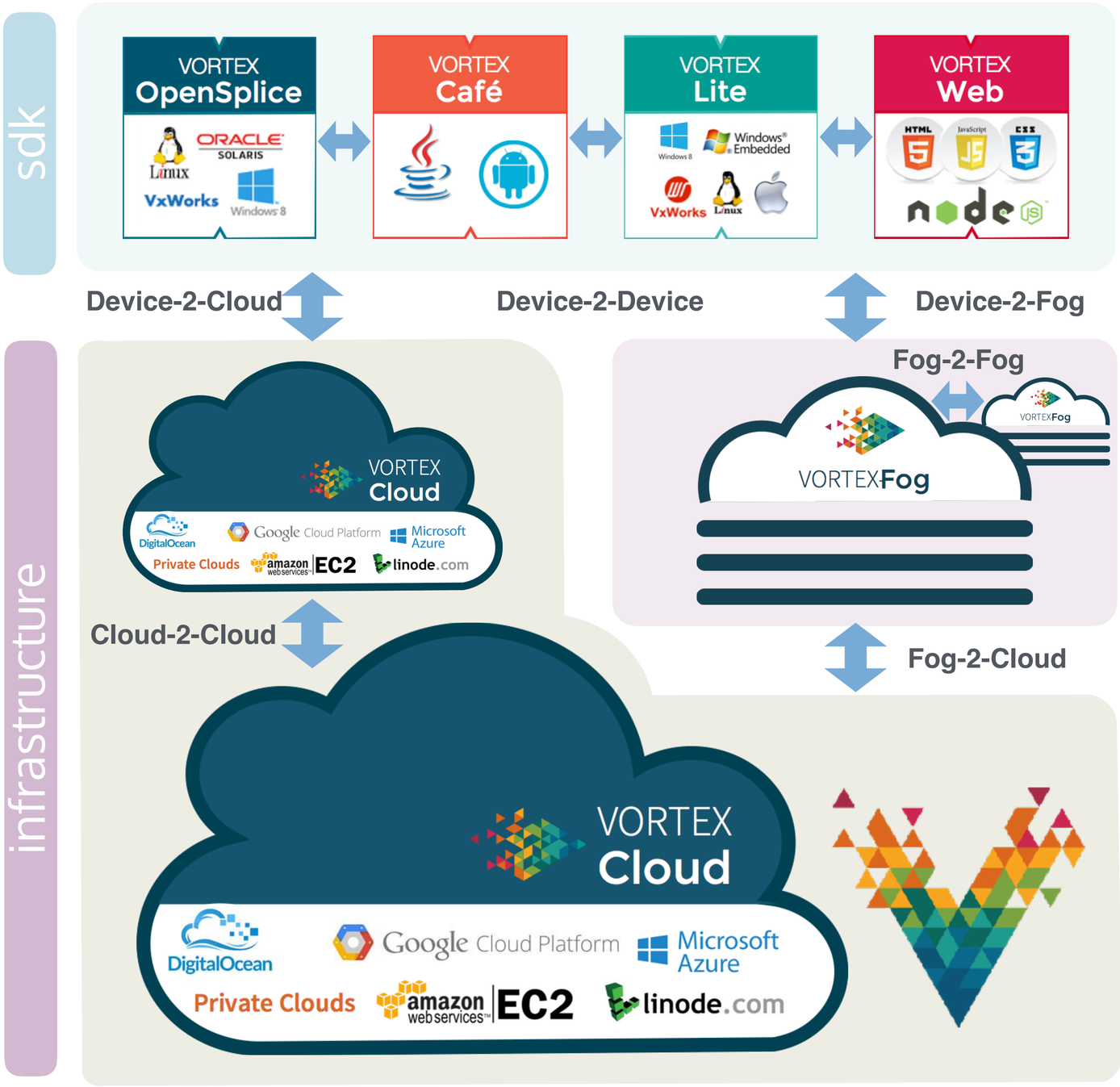
Green Hills SOFTWARE

VxWorks®

WIND RIVER



Polyglot and Interoperable across Programming Languages



Fully Independent of the Cloud Infrastructure

amazon web services EC2

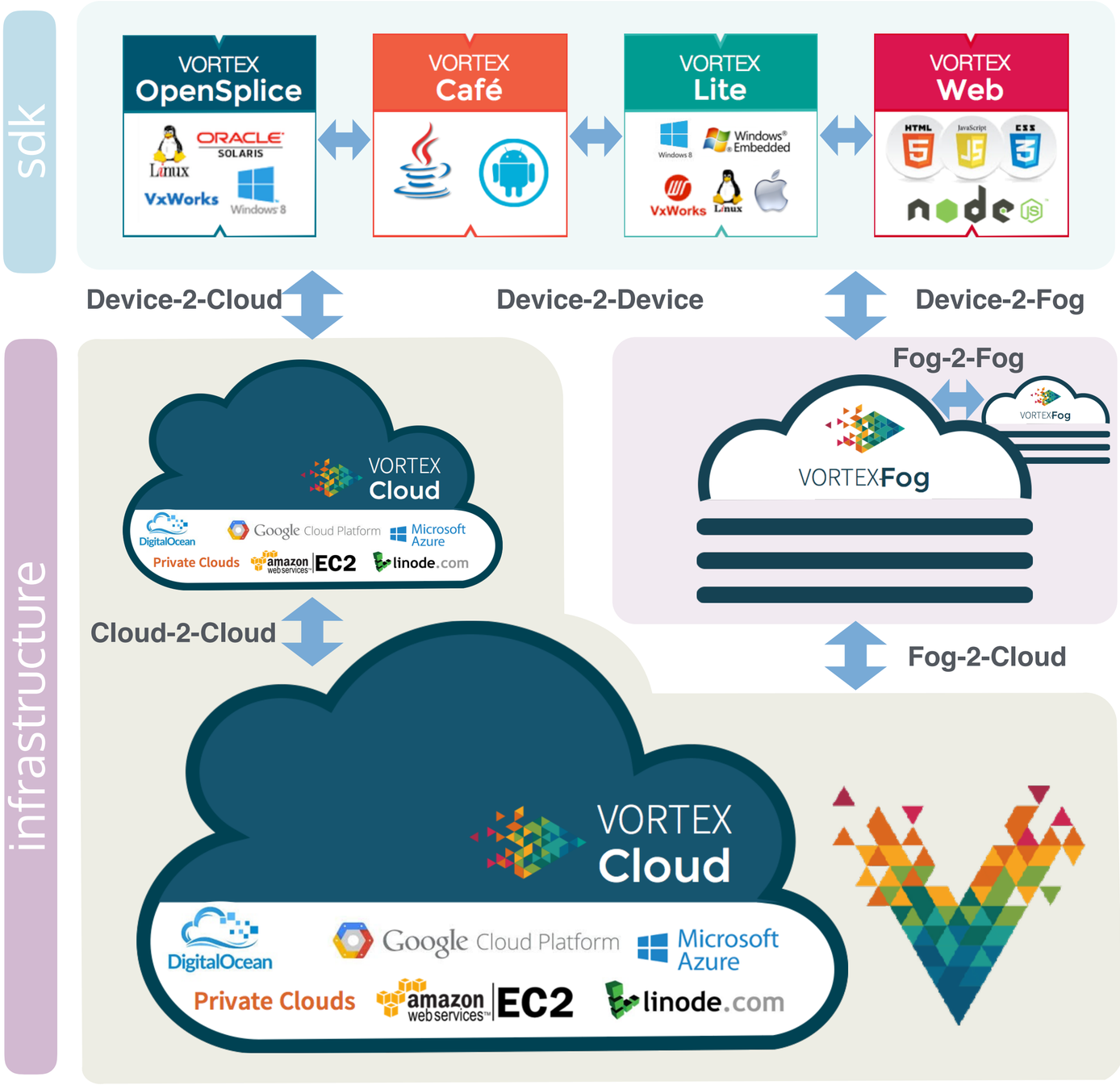
Microsoft Azure

linode

DigitalOcean

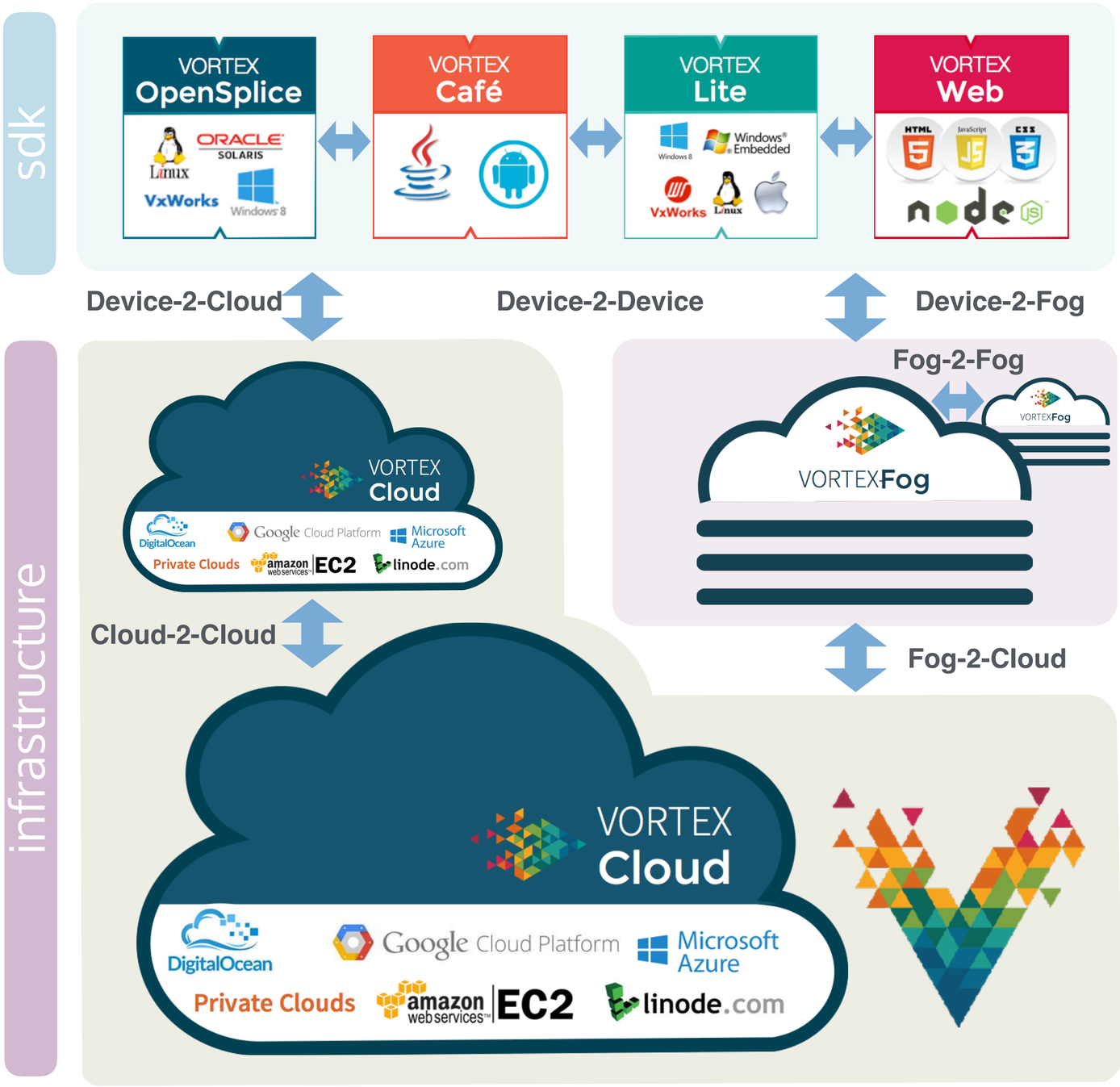
Private Clouds

Google Cloud Platform

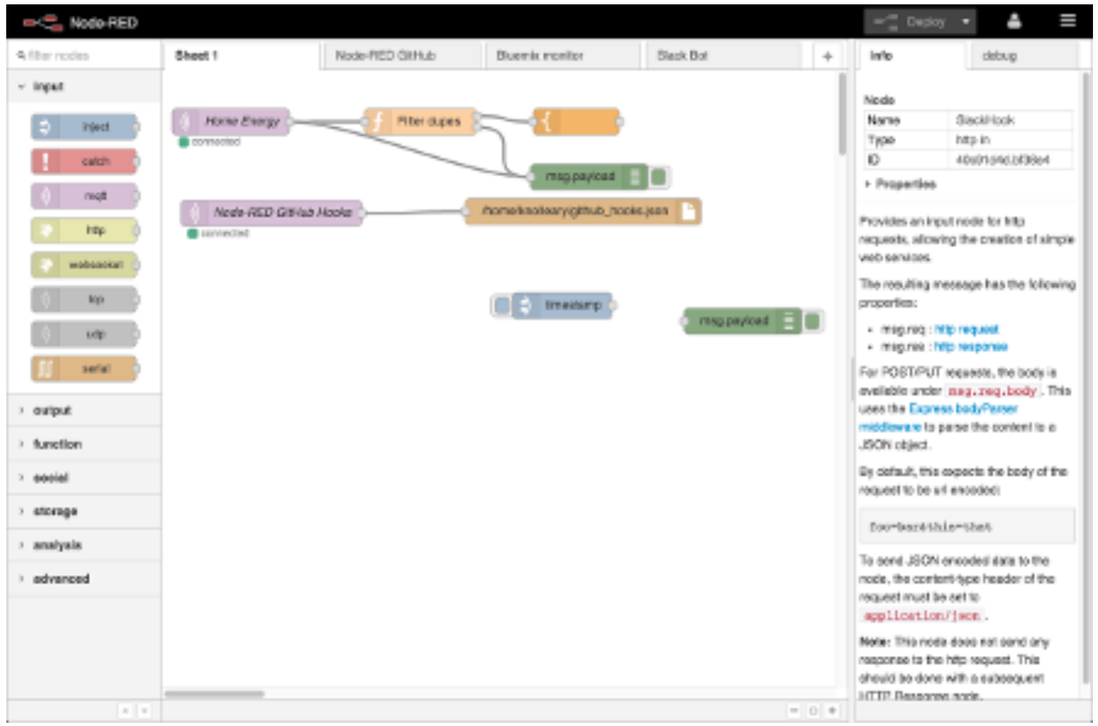


Integration

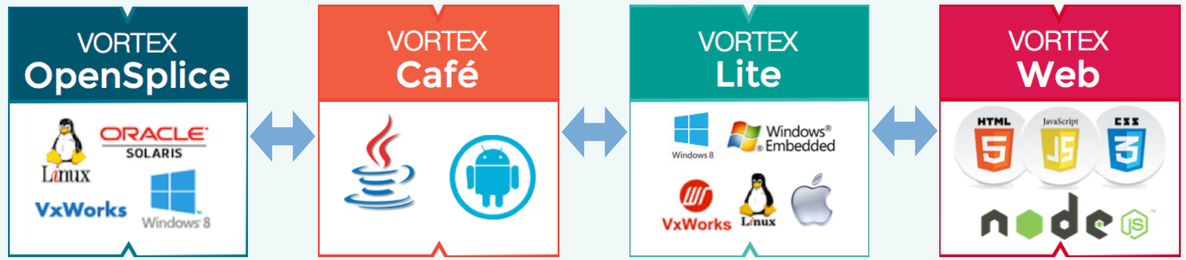
Native Integration with the hottest real-time analytics platforms and CEP



Integration with the popular **Node-RED** framework



sdk



Device-2-Cloud

Device-2-Device

Device-2-Fog

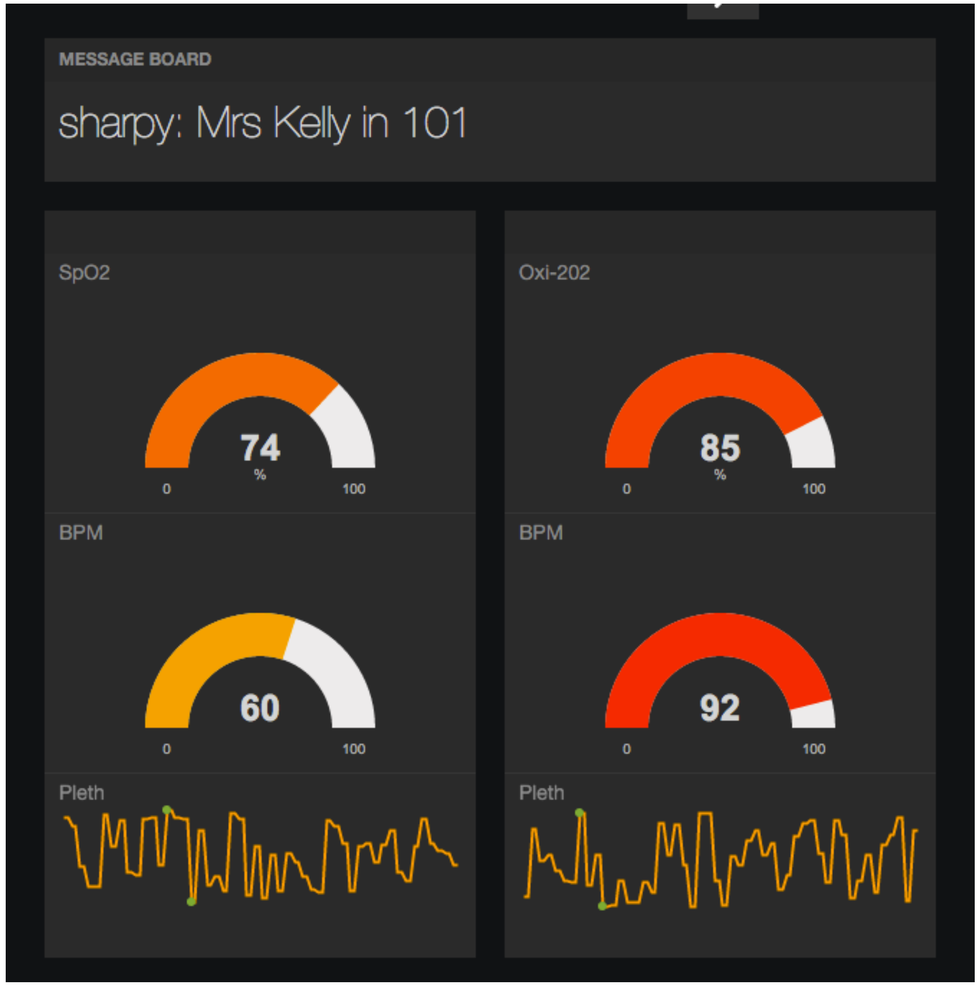
infrastructure

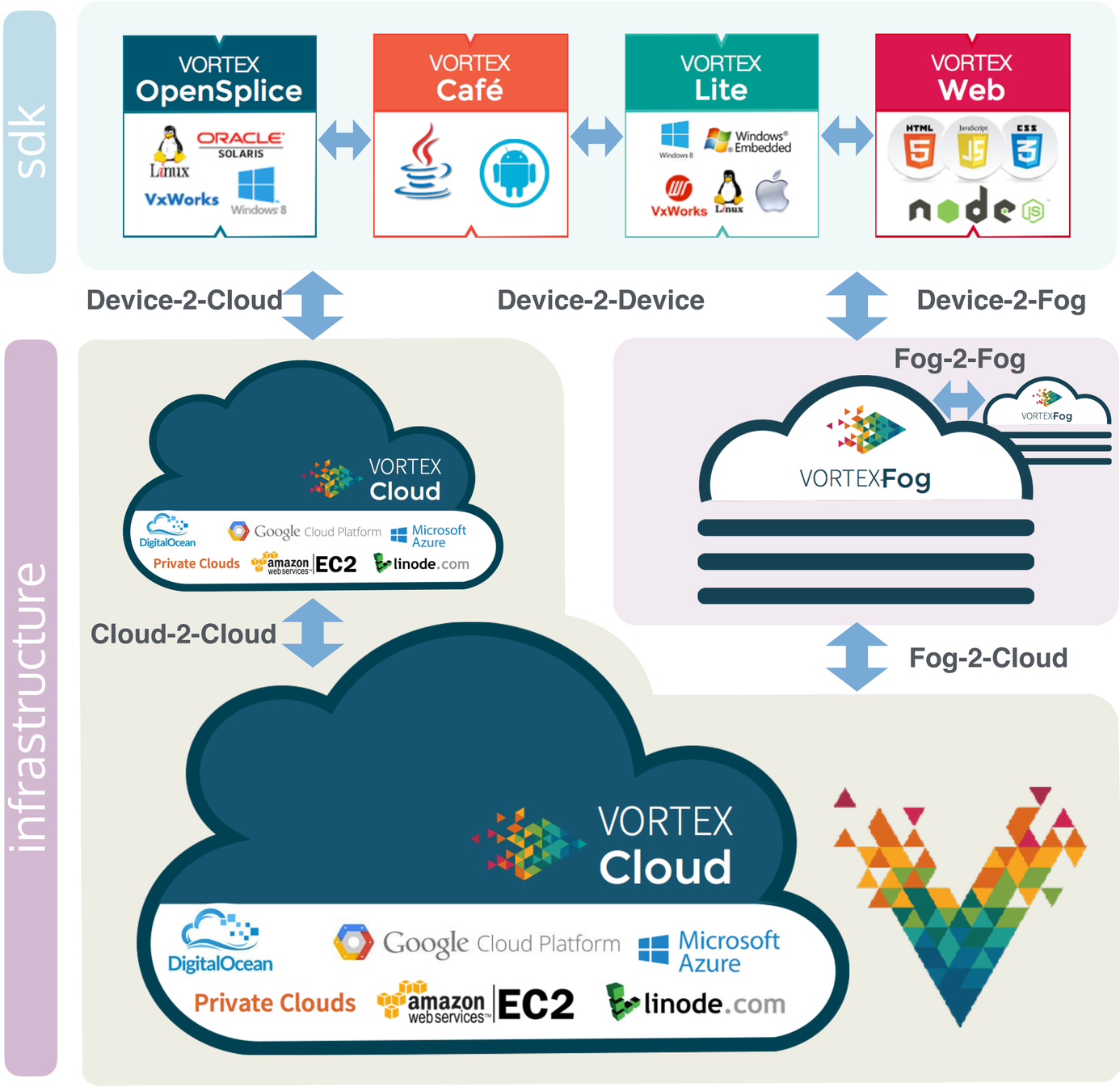
Fog-2-Fog

Cloud-2-Cloud

Fog-2-Cloud

Integration with mainstream Dashboard Technologies





Vortex Deployments

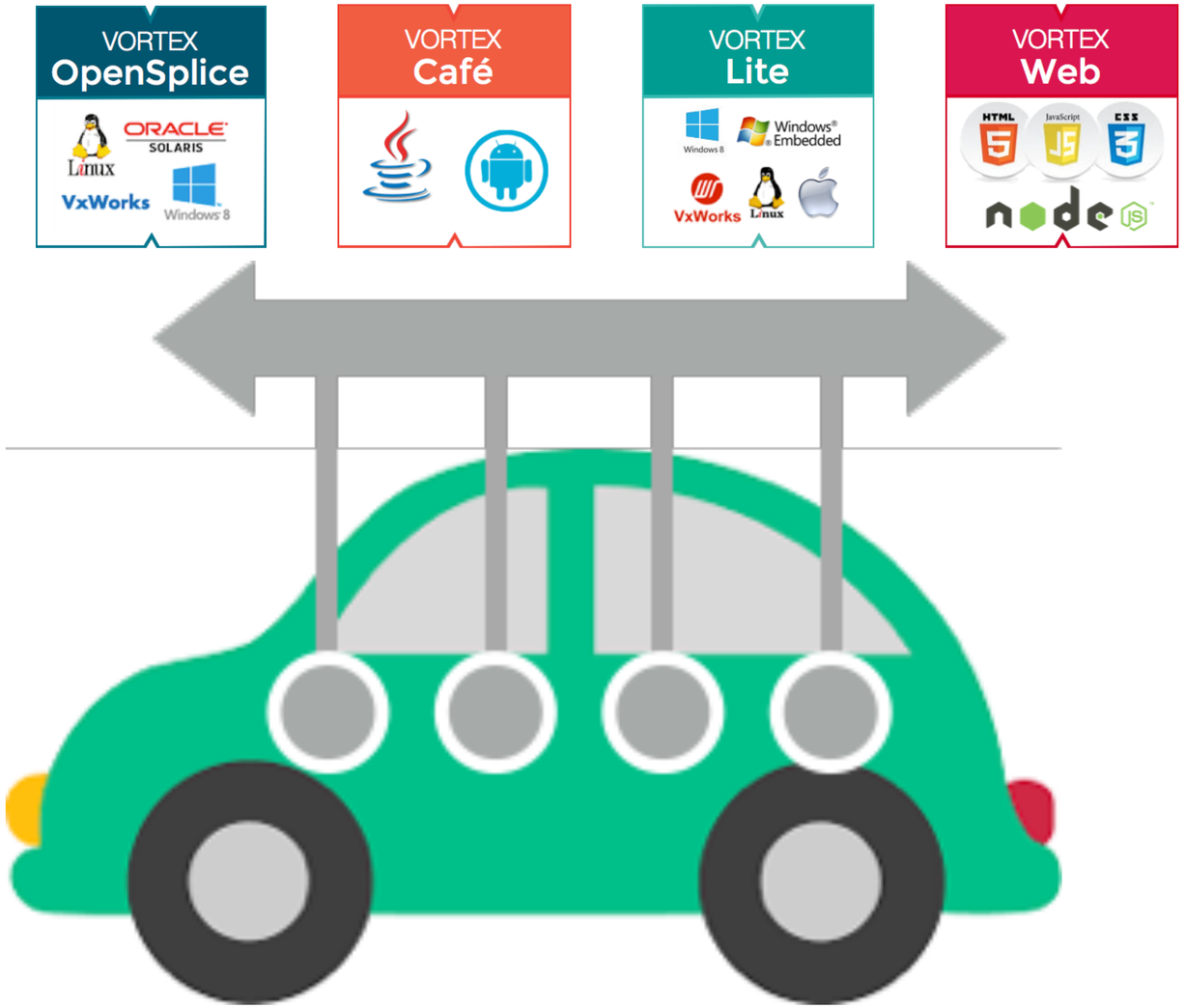
CONNECTED CARS

Powering Infotainment, and driver assistance

Vortex device, such as Lite, Café, and Web are used to share data between different kinds of applications within a car

Café and Web are typically used Android / HTML5 based infotainment

Lite in typically used in ECU, sensors and onboard analytics

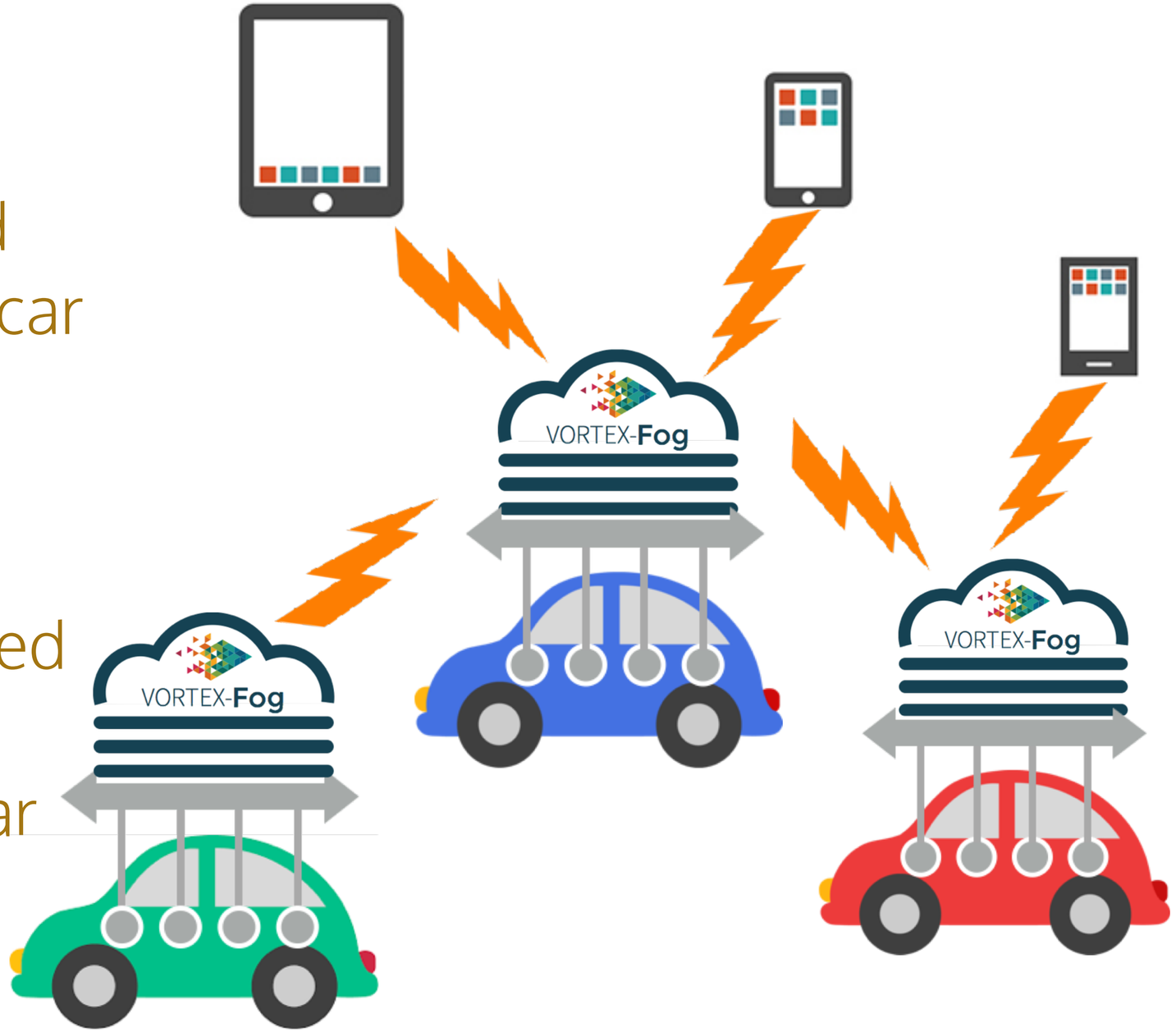


Vortex Fog is used to transparently (for in car apps) decouple and control the data sharing within and across the car

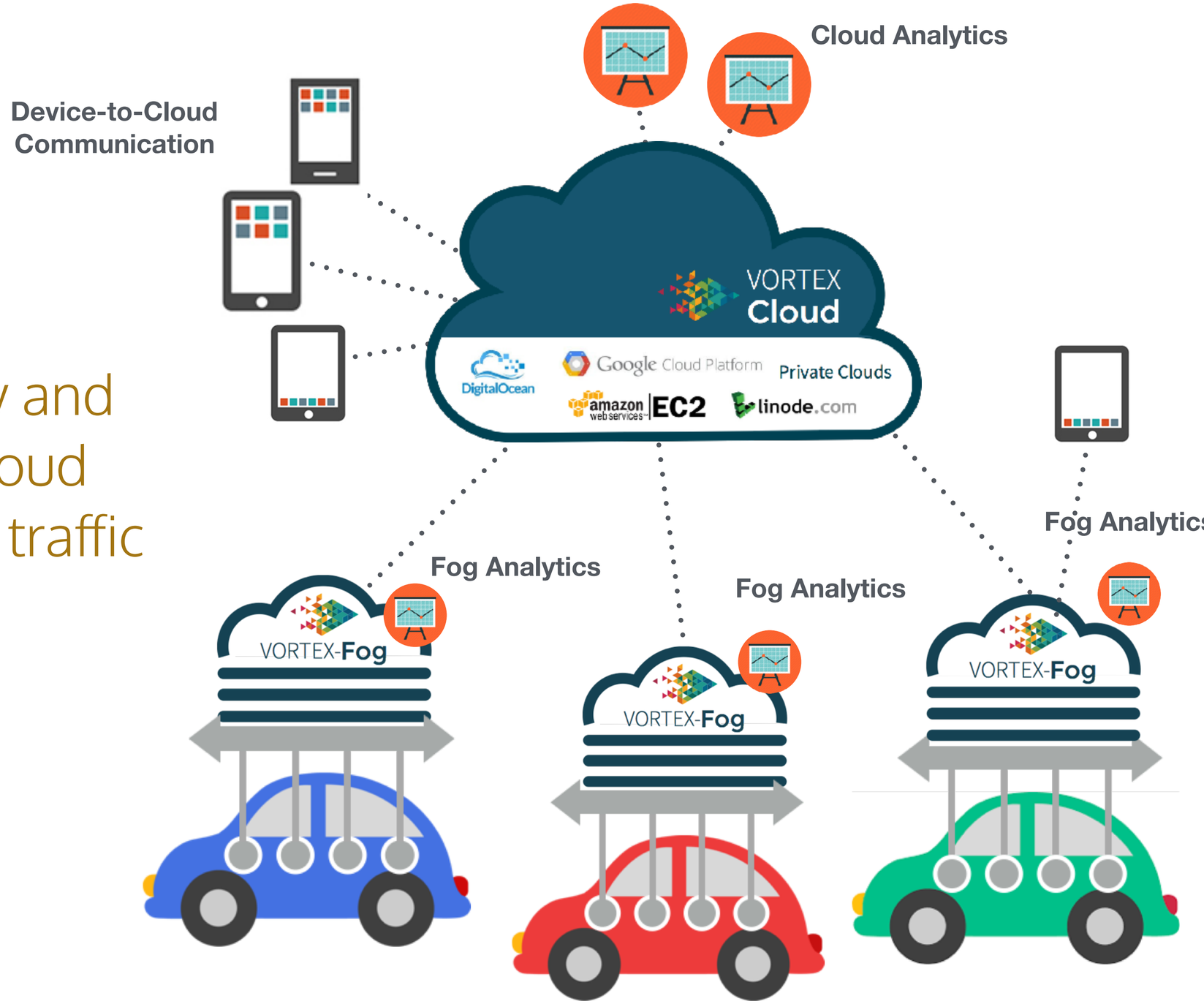
Vortex Fog also helps defining security boundaries and policies

Vortex Fog efficiently and securely deals with car to car communication

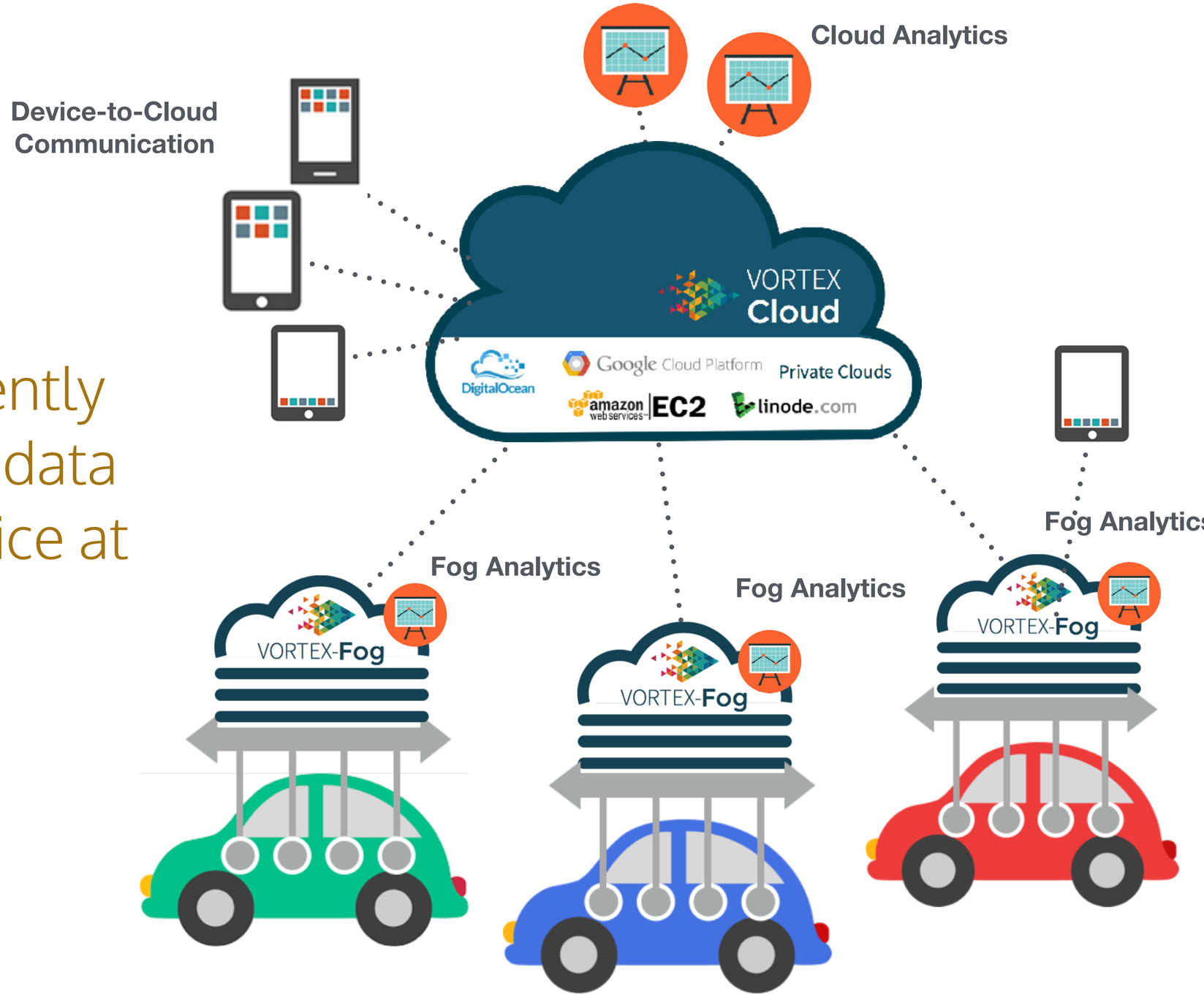
Vortex Fog allows to decouple the transport used for in-car communication and that used for car-to-car communication



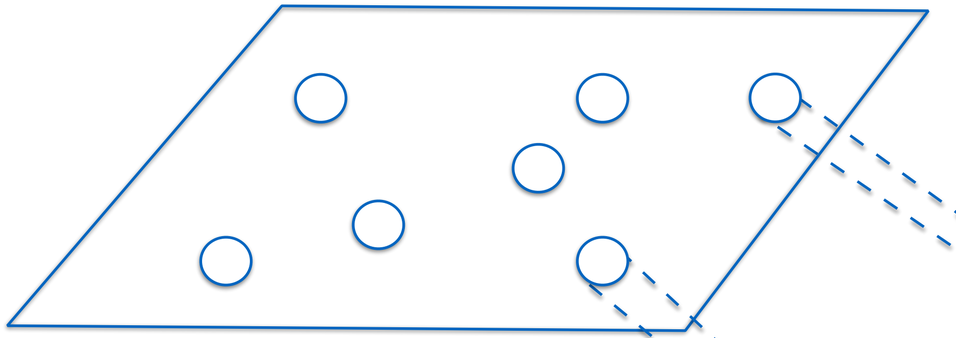
Vortex Fog efficiently and securely deals with cloud connectivity adapting traffic flows and protocols



Vortex Cloud efficiently and securely makes data available to any device at an Internet Scale



Vortex virtualises data…



Data Federation (instances)

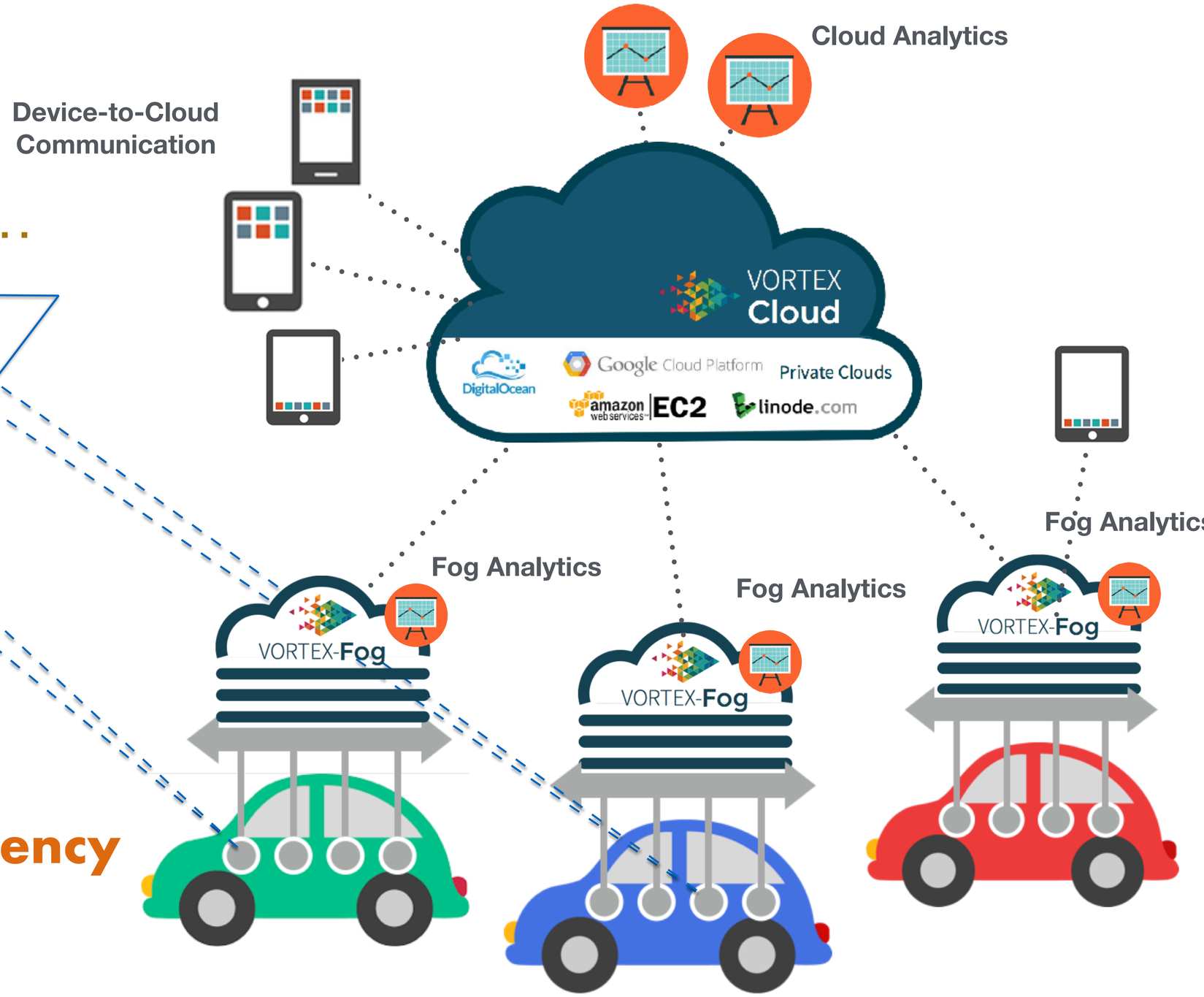
Query

Data Delivery

Location Transparency

Technology Abstraction

History

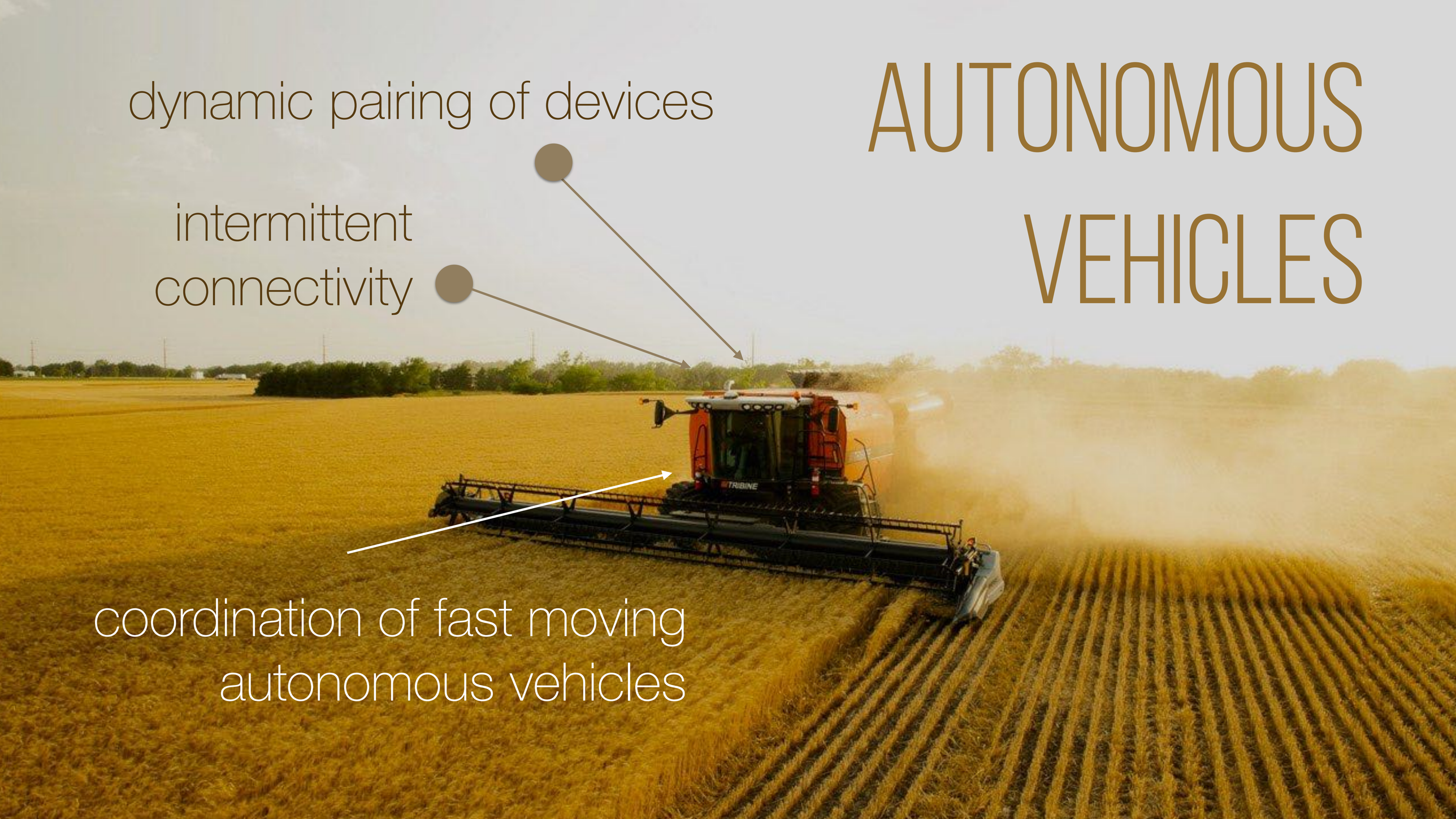


AUTONOMOUS VEHICLES

dynamic pairing of devices

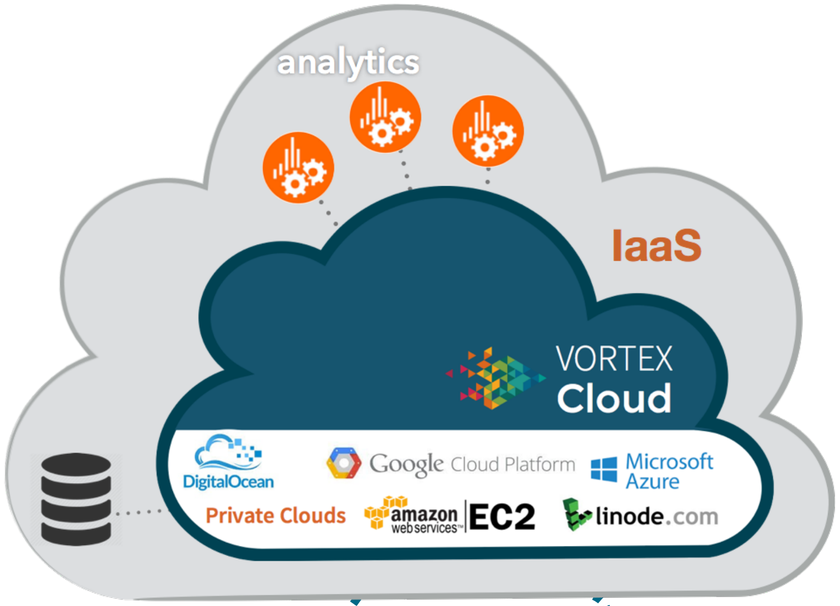
intermittent connectivity

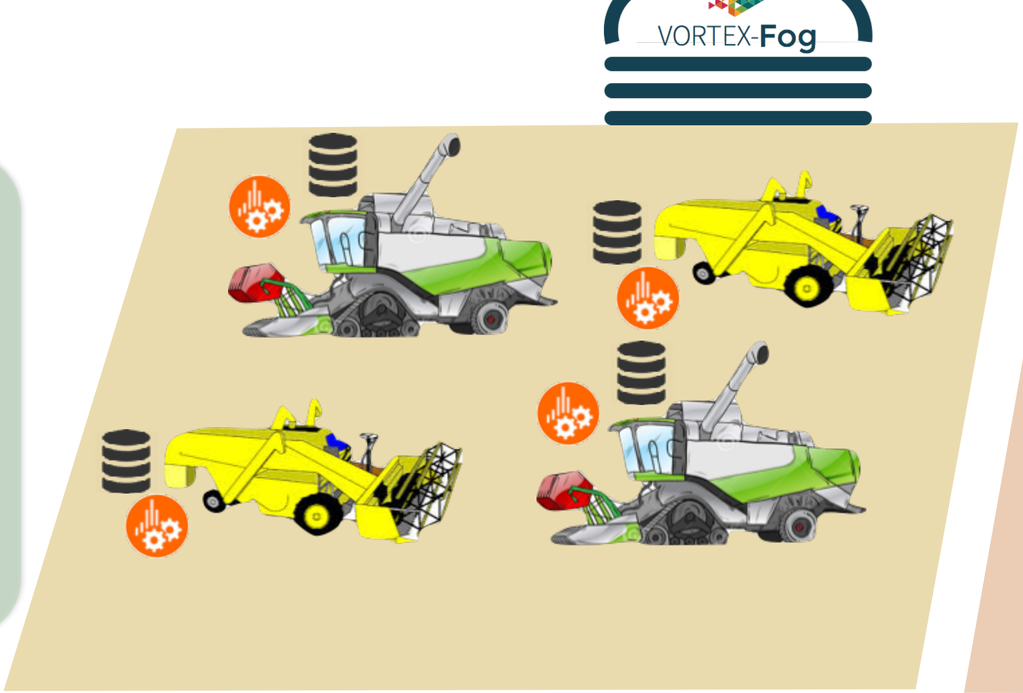
coordination of fast moving autonomous vehicles



Architecture

Some telemetry data is pushed to Vortex Cloud to enable preventive maintenance other kinds of business intelligence



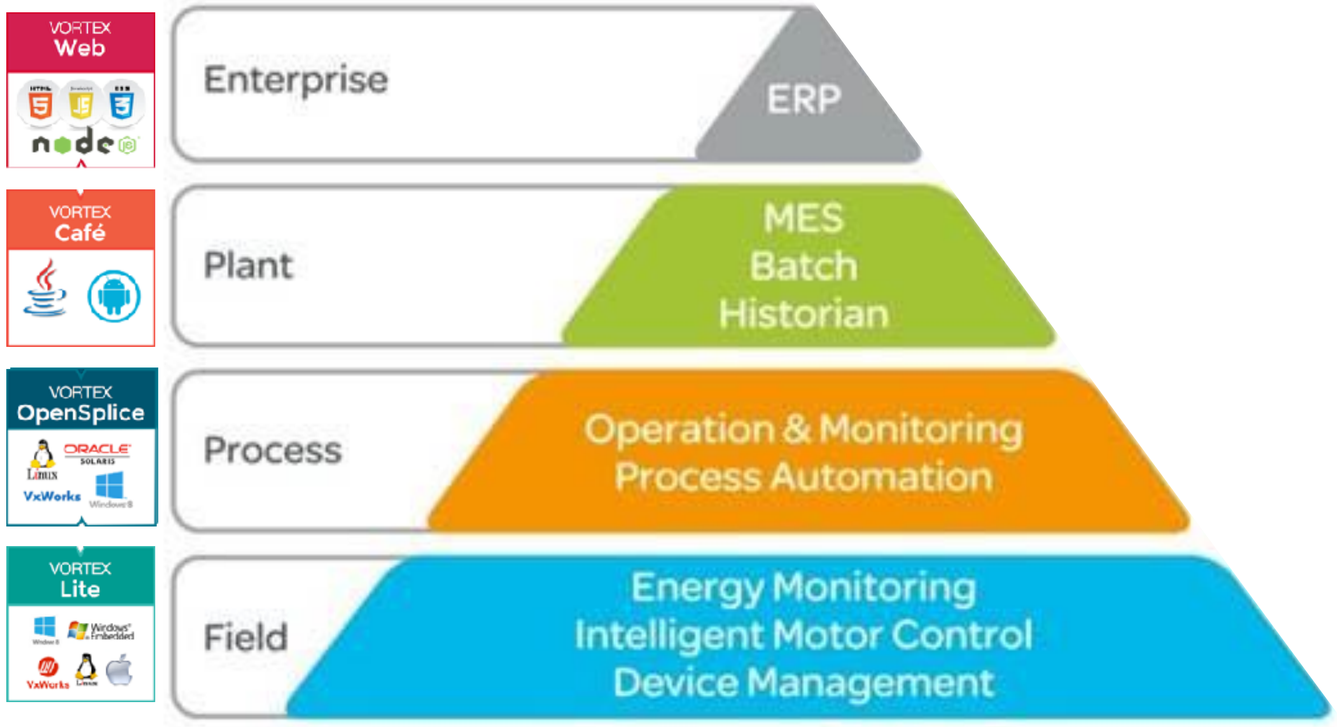


Harvesters, combiners and other kinds of machinery communicate peer-to-peer to (1) exchange position to avoid crashing into each other, (2) agree on the division of labor to optimally harvest the field


Smart Factory
0.5 TB of data
produced per day

Power Generation Platform

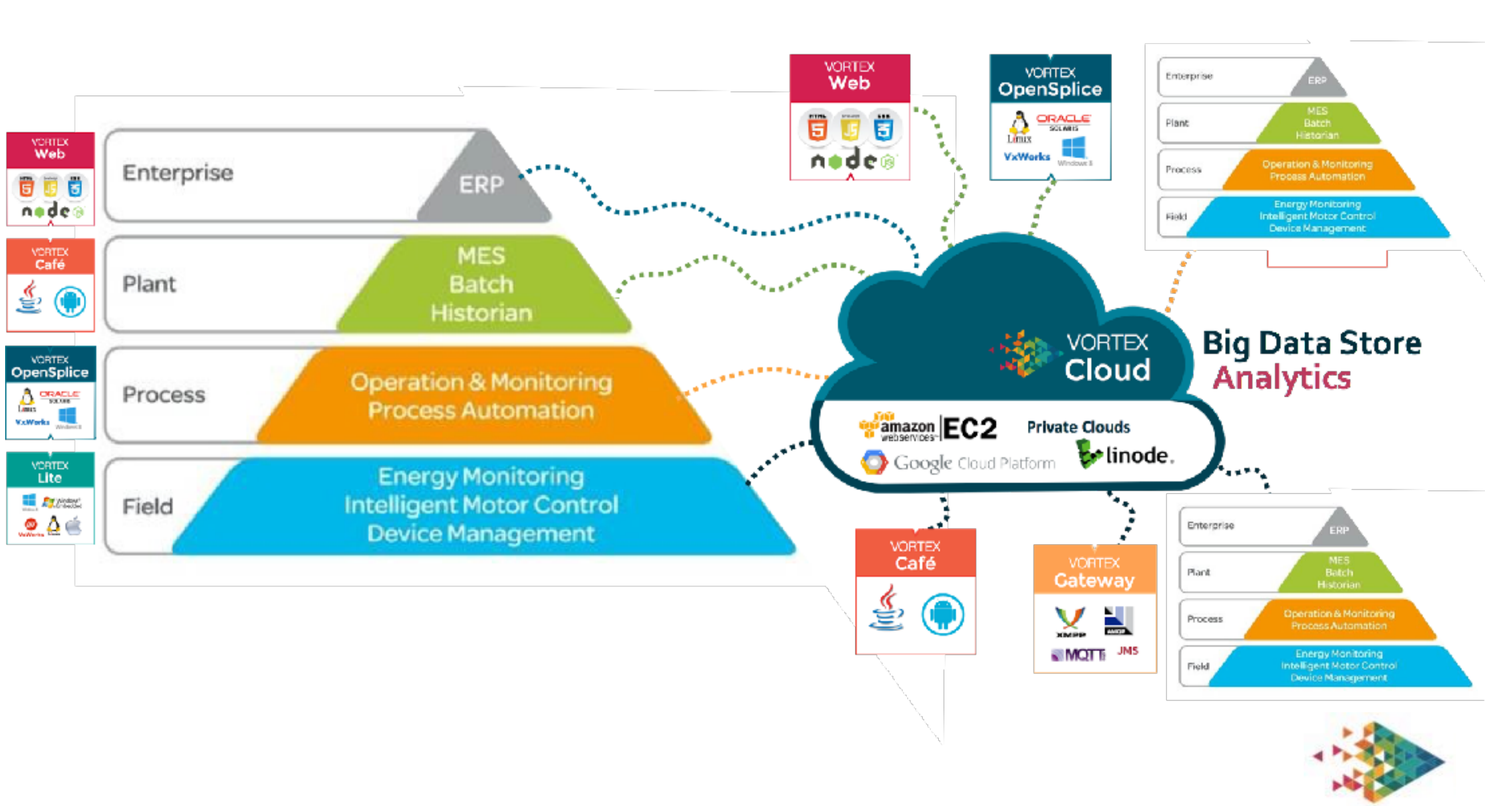
- VORTEX provides a single technology for addressing OT as well as the IT requirements
- VORTEX users are able to seamlessly integrate applications across Field, Process, Plant and Enterprise level eliminating the IT/OT integration challenges and promoting agile and extensible architectures



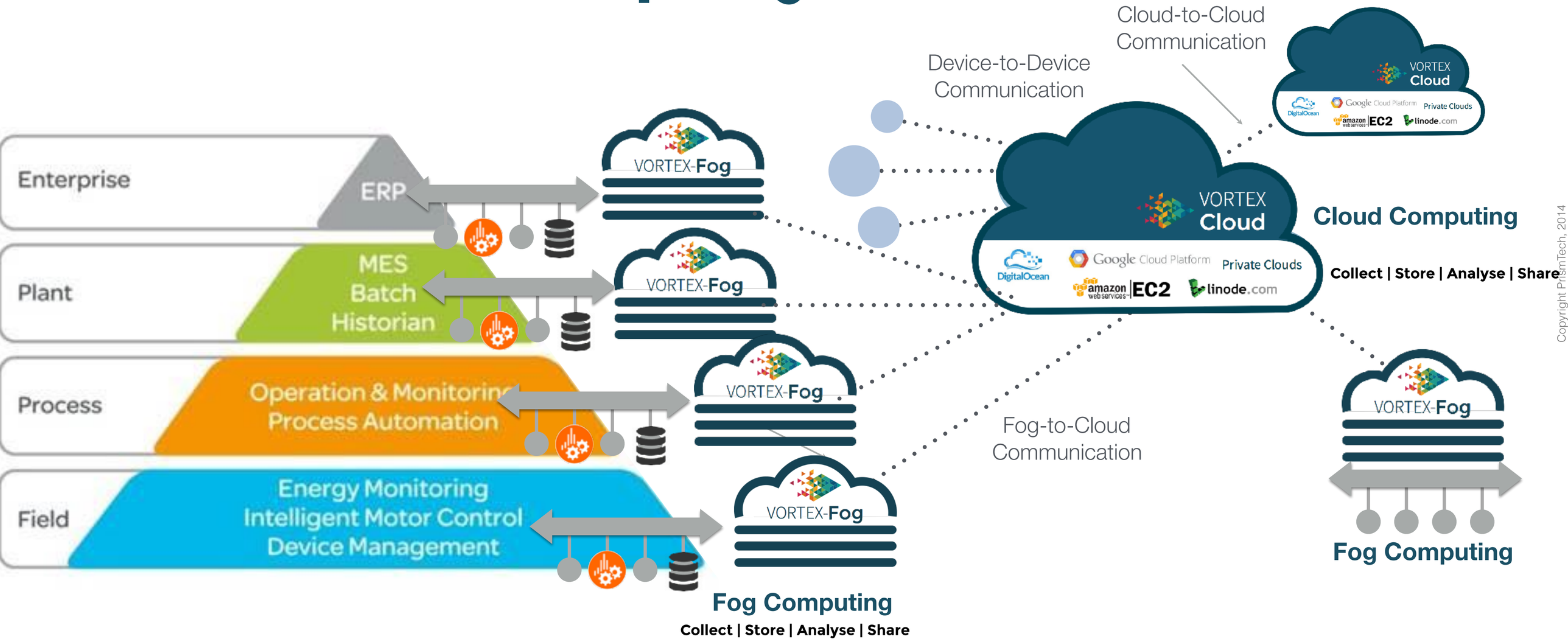
Power Generation Platform

- Vortex Real-Time Cloud Messaging Technology is used achieve horizontal integration

- Additionally, VORTEX customers exploit support for Fog and Cloud computing to ensures that the systems are minimally impacted by changes in connectivity



Deployment



Your First Vortex App

WRITING DATA IN PYTHON

```
import dds
import time

if __name__ == '__main__':
    topic = dds.Topic("SmartMeter", "Meter")
    dw = dds.Writer(topic)

    while True:
        m = readMeter()
        dw.write(m)
        time.sleep(0.1)
```

```
enum UtilityKind {
    ELECTRICITY,
    GAS,
    WATER
};

struct Meter {
    string sn;
    UtilityKind utility;
    float reading;
    float error;
};
#pragma keylist Meter sn
```

READING DATA IN PYTHON

```
import dds
import sys

def readData(dr):
    samples = dds.range(dr.read())
    for s in samples:
        sys.stdout.write(str(s.getData()))

if __name__ == '__main__':
    t = dds.Topic("SmartMeter", "Meter")
    dr = dds.Reader(t)
    dr.onDataAvailable = readData
```

```
enum UtilityKind {
    ELECTRICITY,
    GAS,
    WATER
};

struct Meter {
    string sn;
    UtilityKind utility;
    float reading;
    float error;
};
#pragma keylist Meter sn
```

IN SUMMARY

Vortex provides a uniform abstraction supporting Cloud, Fog and Mist computing architectures

Vortex is a proven technology powering some of the most challenging IIoT systems