Datastore

Osprey DCS
George McIntyre

- Snapshots of time-correlated experimental data for ML applications

Datastore Agenda

- What's the problem?
- Solution
  - Ingestion
  - Queries
- Datastore as part of an ML Data platform
- System Components
- Performance

What’s the problem

What if …

- We could gather experimental data as quickly as it is produced
- We could group experiment data across multiple devices into snapshots
- We could query this experiment data linking data elements by time
- We could export this data in any form needed for ML applications

- Save the world!

Datastore Requirements

High Volume, High Speed

all the data from all the experiments forever

very high ingestion speeds for real-time capture

Time-Series Database

special type of column database, that relates all data to an ever-present time field

Experiment Data

EPICS data compatible, data items identified by PV name

Stores scalars, scalar arrays (waveforms), or structures, with optional metadata, and attributes.

We have an Archiver! What gives?

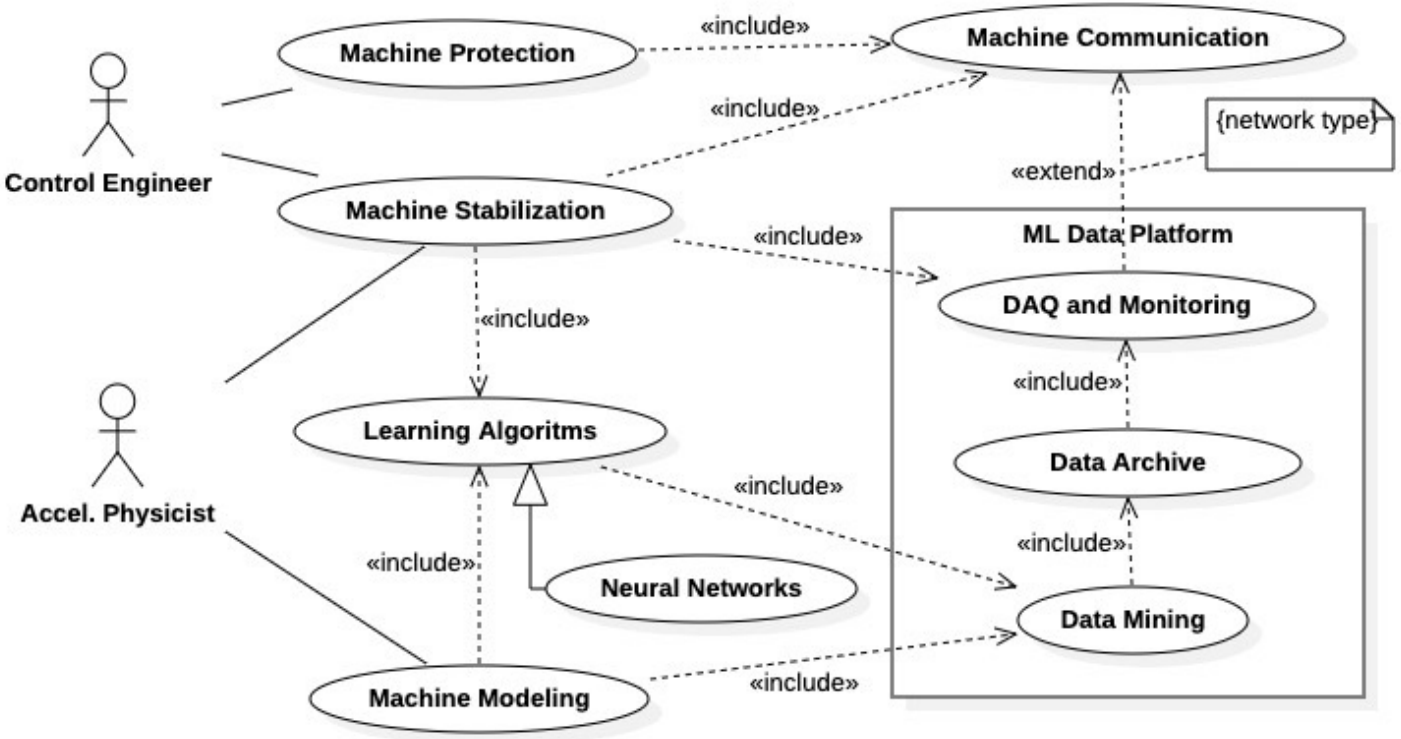
- Archiver
  - Massive volumes
    - Can capture all PVs
  - Distributed architecture
    - Clustered appliances
  - High speed retrieval
  - Multiple output formats

- Datastore = Archiver++
  - Snapshots
    - Identification of data snapshots
  - Timestamp precision
    - Nanosecond
  - Addition of arbitrary Attributes
  - Data Normalization
    - Always correlated to time, any data can be related to any other data
  - Query interface with Annotations
    - Massive multiparametric regression analysis and ML

Data Ingestion

- Data Ingestion Challenges
  - Need balanced cluster to provide throughput objectives
  - Data formats need to be flexible
  - Single master database limits ingestion speeds
- Successful Strategies
  - Multiple providers (like Appliances)
  - Flexible distributed architecture for both providers and database
  - Intelligent data partitioning
  - Binary wire format optimized for speed
- Unsuccessful Strategies
  - In-memory query results merging/caching inefficient
  - Semantics around data availability needs review

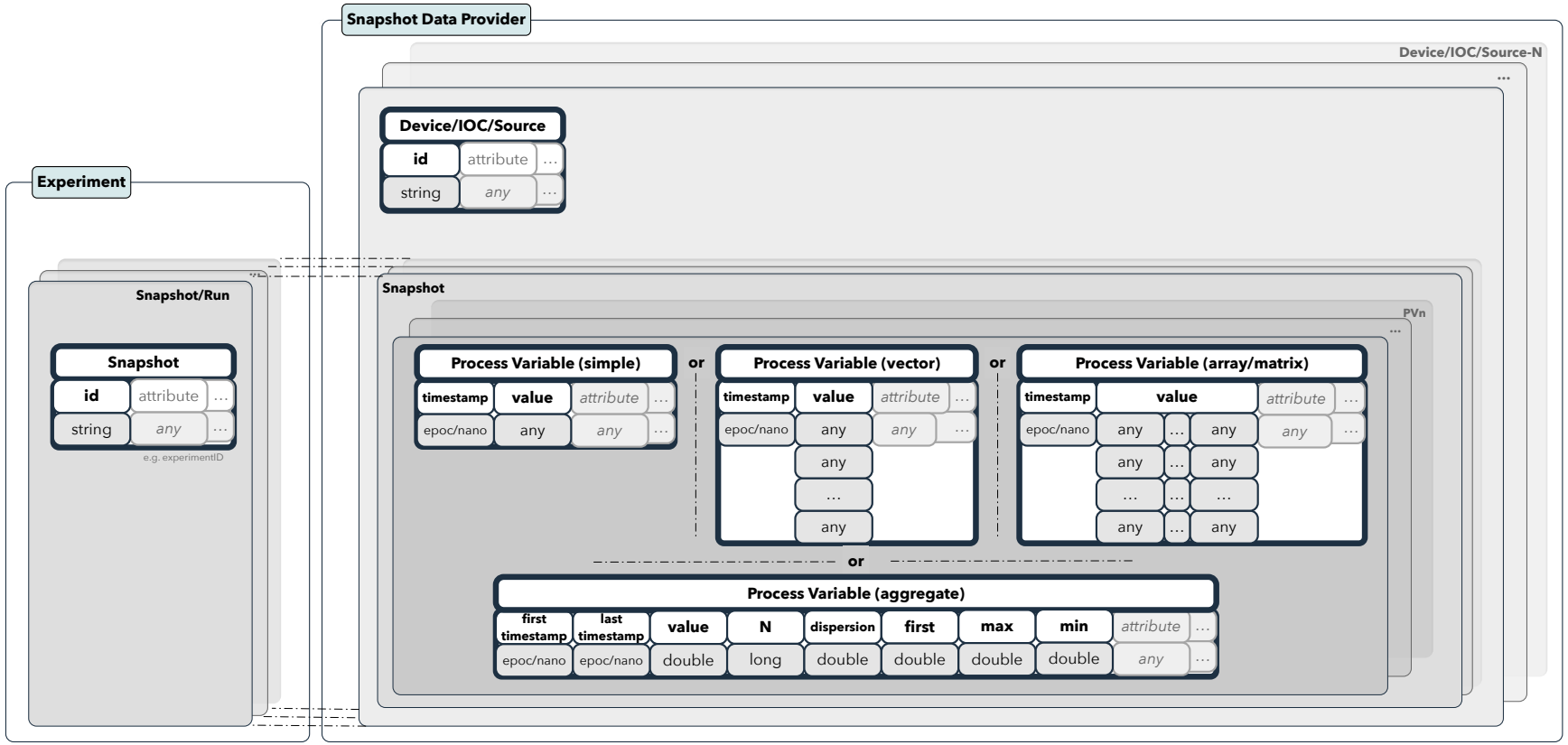
Datastore as part of an ML Data Platform



-- source Christopher Allen, 2022

Prototype Summary

Snapshots



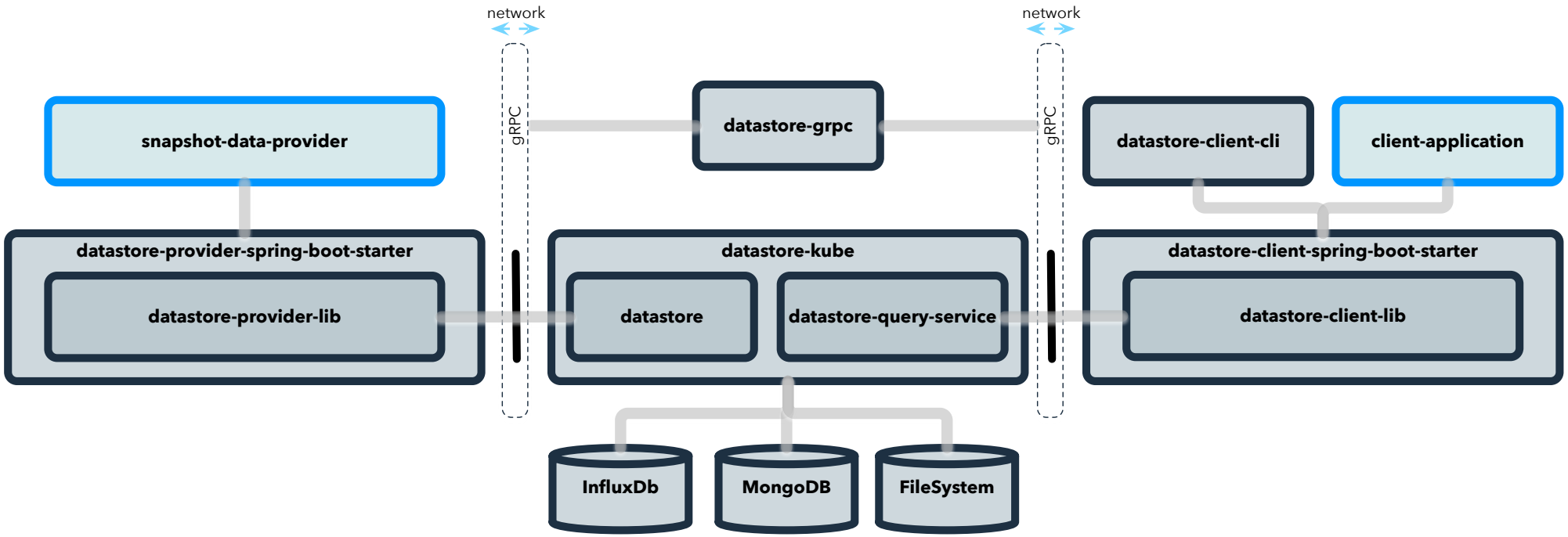
- Attribute examples: **run-id, experiment-id, exposure-time, operator-id, project**, etc

Datastore Ingestion

- **gRPC ingestion API**
  - **High speed - gRPC multi-platform ingestion API programmatically controlled**
  - Partitioned ingestion when in clustered mode
- **JAVA/CPP ingestion library**
  - **Java wrapper around gRPC ingestion API for simple implementation**
  - C++ wrapper around gRPC ingestion API for simple implementation
- **Datastore EPICS Channel Monitor**
  - Configure with EPICS channels to monitor (PV Access subscription)
  - Map to Datastore PV names and fields

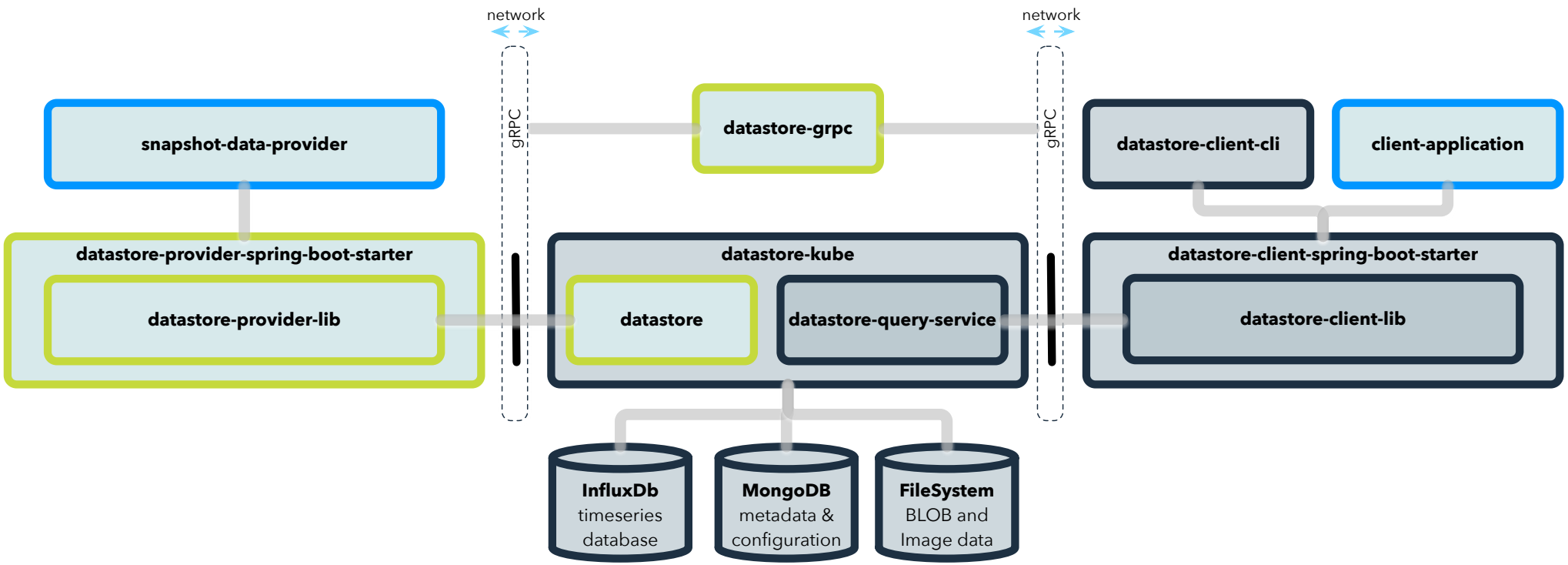
- Datastore Query Language - DQL
- gRPC query API for transporting DQL and delivering results
  - gRPC multi-platform query API programmatically controlled
  - Partitioned queries with results merging when in clustered mode
- Query library
  - Java wrapper around gRPC query API for simple implementation
  - C++ wrapper around gRPC query API for simple implementation
- Export binary responses as:
  - JSON
  - YAML
  - HDF5

Datastore Queries



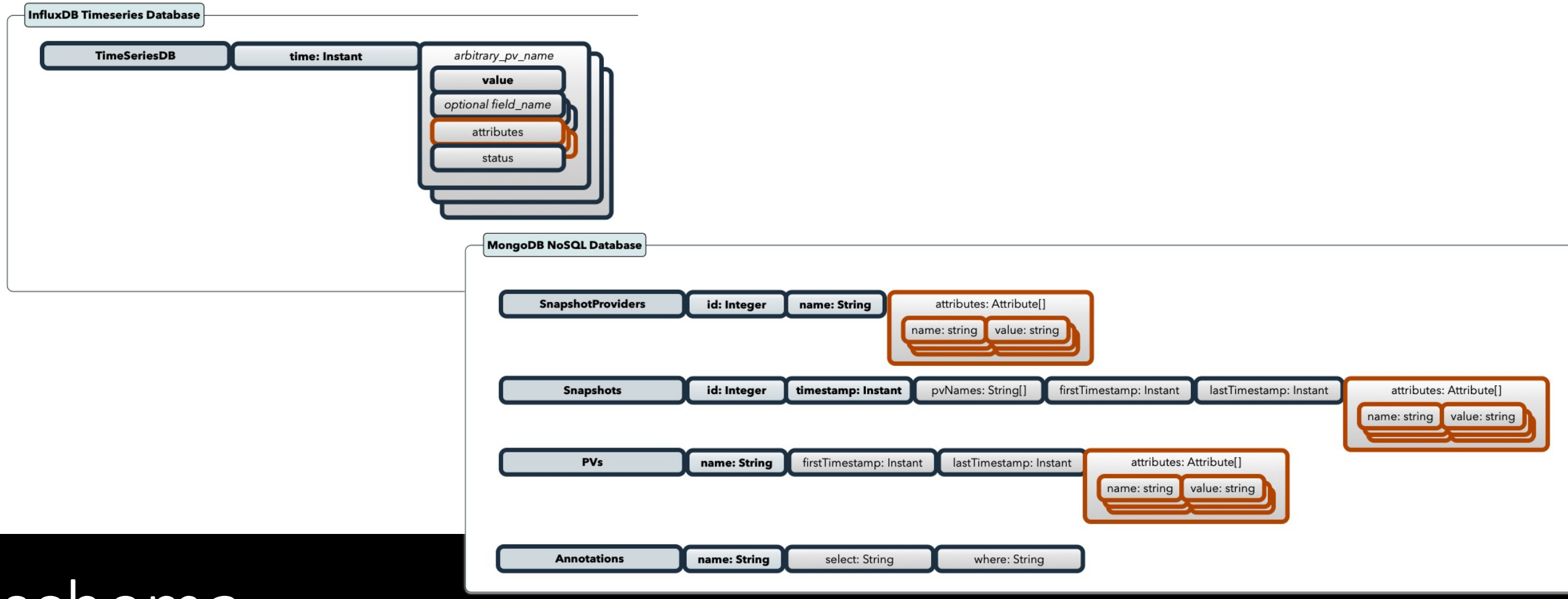
- The Datastore prototype allows **snapshot data providers to stream data to a timeseries database**
- Data can be **simple scalars** to **complex structured data** and support for **Normative Types** is included right out of the box
- The data is **correlated in real time** with all other data sent from any other source
- Client applications can **query this correlated data seamlessly**
- Data can be exported as **CSV, HDF5** or other formats

System Components



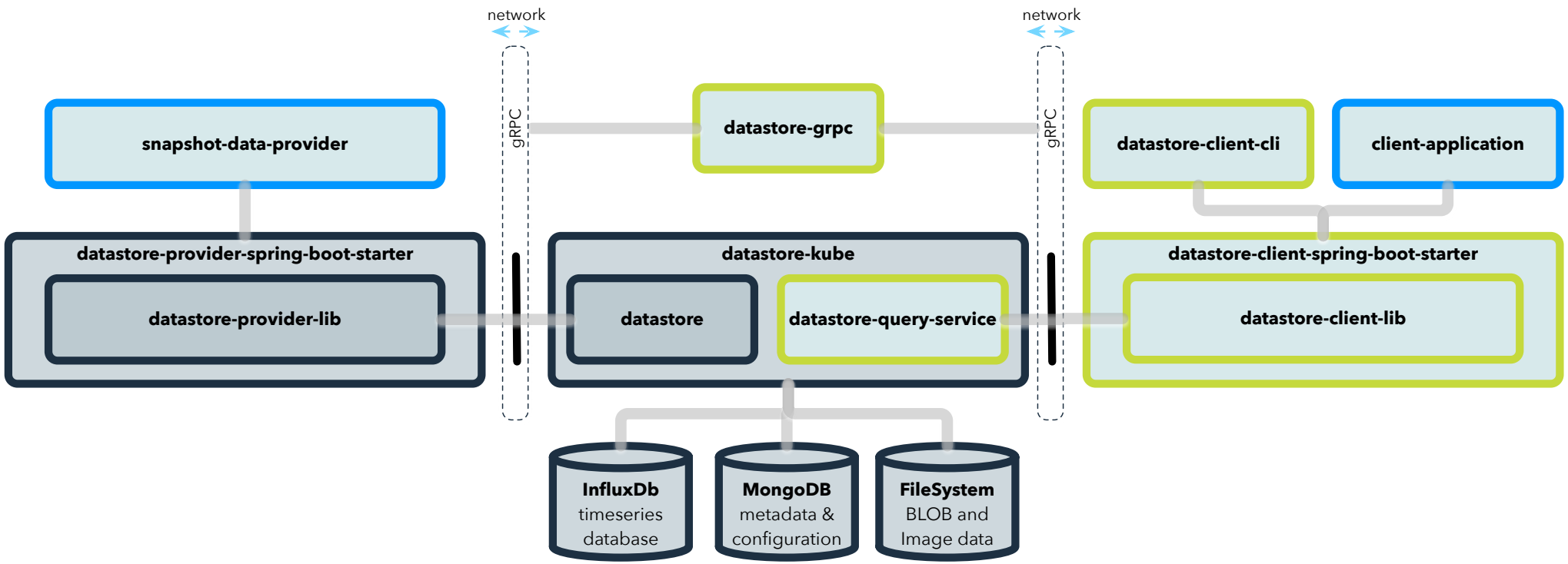
high speed storage of Snapshot Data

- leveraging high speed binary **gRPC** transport
- snapshot-data-providers can **stream data** into the datastore in **real-time** or **bulk**
- **libraries** facilitate implementation
- **Normative Type support** directly in provider-libraries
- under the hood data is stored in **InfluxDB** a high performance scalable timeseries database
- metadata and configuration information is stored in **MongoDB** and merged into results



schema

- The timeseries database has a very **simple schema**
  - **`time`**: nanosecond timestamp
  - *PV*
    `.field` fields of arbitrary type on any PV
    `:prop` properties including alarm status, metadata, and attributes.
- **Heterogenous** data coexists in the same datastore and is automatically correlated with **nanosecond precision**
- The NoSQL database is used to **store metadata** on snapshot providers, snapshots, and PVs
- The NoSQL database also **stores annotations** that can be added to queries so that they can be saved and re-used



powerful query language and capabilities

- `SELECT <column-expression> [INTO ANNOTATION <annotation-name>] WHERE <time-expression> [AND <filter-expression>] [FROM ANNOTATION <annotation-name>]`
- Search for data across **any PV and time range** in the same query
- Find records where beam was lost and save as BEAM_LOSS **annotation** to be reused later
- List providers, snapshots, and PVs by powerful queries to get **information about data that is stored in the Datastore**
- Export data to **HDF5 and CSV** from the client library and CLI

Datastore Query Language - DQL

Symbol		Expression
and-filter-expression	:=	*<filter-expression>* "*AND*" *<filter-expression>*
annotation-name	:=	*<string-literal>*
binary-filter-expression	:=	*<and-filter-expression>* | *<or-filter-expression>*
binary-time-expression	:=	*<unary-time-expression>* "*AND*" *<unary-time-expression>*
column-expression	:=	*<pv-specification>* | *<pv-specification>* "," *<column-expression>*
filter-expression	:=	*<pv-specification>* *<operator>* *<literal>* | *<binary-filter-expression>*
instant-time-expression	:=	"*time*" "=" *<time-literal>*
literal	:=	*<string-literal>* | *FLOATING_POINT_LITERAL* | *INTEGER_LITERAL*
operator	:=	"<" | "<=" | "=" | "!=" | ">" | ">="

Symbol		Expression
or-filter-expression	:=	*<filter-expression>* "*OR*" *<filter-expression>*
pv-literal	:=	*<pvName>* "." *<field>*
pv-regex	:=	"`" *REGEX* "`"
pv-specification	:=	*<pv-literal>* | *<pv-regex>*
string-literal	:=	"'" *STRING* "'"
time-expression	:=	*<instant-time-expression>* | *<unary-time-expression>* | *<binary-time-expression>*
time-literal	:=	*<string-literal>*
time-expression-operator	:=	"<" | "<=" | ">" | ">="
unary-time-expression	:=	"*time*" *<time-expression-operator>* *<time-literal>*

SELECT <column-expression> [**INTO ANNOTATION** <annotation-name>] **WHERE** <time-expression> [**AND** <filter-expression>] [**FROM ANNOTATION** <annotation-name>

Example Queries

```
SELECT `*.*` WHERE time >= '-5m' AND mpexPv01.value > 10.0
```

- This will return all pvs and all fields from the last 5 minutes where mpexPv01 is above 10

```
SELECT `pv01.*` WHERE time >= '-30m' AND pv01.value > 10.0
```

- This will return all fields and values of pv01 that are above 10, from the last 30 minutes

Example Results

time	mpexPv01.value	mpexPv02.value	mpexPv03.value	...	mpexPv64.value	mpexPvImage.dimensions	mpexPvImage.value
2022-08-03 T16:17:21.269Z	0.27076009879565344	0.07234648703261869	0.5283747996399034	...	0.00536410111983138	[{"binning":1,"size":3,"offset":0,"reverse":false,"fullSize":3},{"binning":1,"size":640,"offset":0,"reverse":false,"fullSize":640},{"binning":1,"size":240,"offset":0,"reverse":false,"fullSize":240}]	dat/c8fc512c-3227-4e9e-84b5-c6432825a8c

- Results always have a **timestamp**
- PVs always have at least a **value** field
- Data that matches the query, regardless of which provider it comes from will be returned by queries
- Images can be returned by reference or as data arrays

Datastore Java Libraries

Install with Maven

```
<dependency>
    <groupId>com.ospreydcs.datastore</groupId>
    <artifactId>lib</artifactId>
    <version>1.0.0</version>
</dependency>
```

```
<dependency>
    <groupId>com.ospreydcs.datastore-client</groupId>
    <artifactId>lib</artifactId>
    <version>1.0.0</version>
</dependency>
```

Java Programming Interface

DatastoreServiceClient

- `DatastoreServiceClient client = DatastoreService.getClient(String host, int port, int timeoutSeconds, boolean usePlaintext, boolean keepAliveWithoutCalls[, boolean gzip=false])`
  - the static method, is called to give you a client object for communication with the Datastore
- `Dataset dataset = client.findAll(String query)`
  - is called to get the correlated data. Blocks until all data is retrieved. Behind the scenes calls findPages()
- `Dataset dataset = client.listData(String query)`
  - is called to get the correlated data. Call datastore-service synchronously on the listSnapshotData() endpoint
- `DataPager dataPager = client.findPages(PaginatedQuery query[, Consumer<Dataset> pageProcessor])`
  - is called to get the correlated data in pages. The first requested page of data is returned synchronously, while the remaining pages are streamed to returned DataPager in the background. If a pageProcessor is provided, it is called back as pages become available.
- `List<DatastorePVMetadata> pvMetadataList = client.findAllPVs([String queryRegex])`
  - to retrieve PV metadata. You can filter the list with the optional regex
- `List<DatastoreAnnotationMetadata> annotatationMetadataList = client.findAllAnnotations([String queryRegex])`
  - to retrieve annotation metadata. You can filter the list with the optional regex

Java Programming Interface

Dataset

- `interface Map<String, List<Object>>`
  - keyset() - field names - two special fields are always present: time and labels containing Instant and String respectively
  - values() - the list of values. Each value is an arbitrary type of scalar Double, Map, List, byte[], etc.
- `List<String> fieldNames = dataset.asHeadings()`
  - The column name for each column in the dataset. Usually the pvName followed by a dot then the field name.
- `Integer rows = dataset.length()`
  - The number of rows in the datasets
- `List<List<Object>> rows = dataset.asRows()`
  - provides the data as a list of rows of values as in relational database results
- `JSONObject jsonObject = dataset.asJson()`
  - converts the Dataset to a JSONObject
- `void dataset.asHDF5(Path outputFile, [boolean withStatus])`
  - creates an HDF5file in the given outputFile. By default, just the values are returned, but if withStatus is returned then an extra field in each value contains the status. The status field is null if the status is normal.
- `Integer pageNumber = dataset.getPageNumber()`
  - returns the page number of this dataset
- `Dataset newDataset = dataset.filteredBy(Predicate<Entry<String, List<Object>>> predicate)`
  - to create a new database based on this

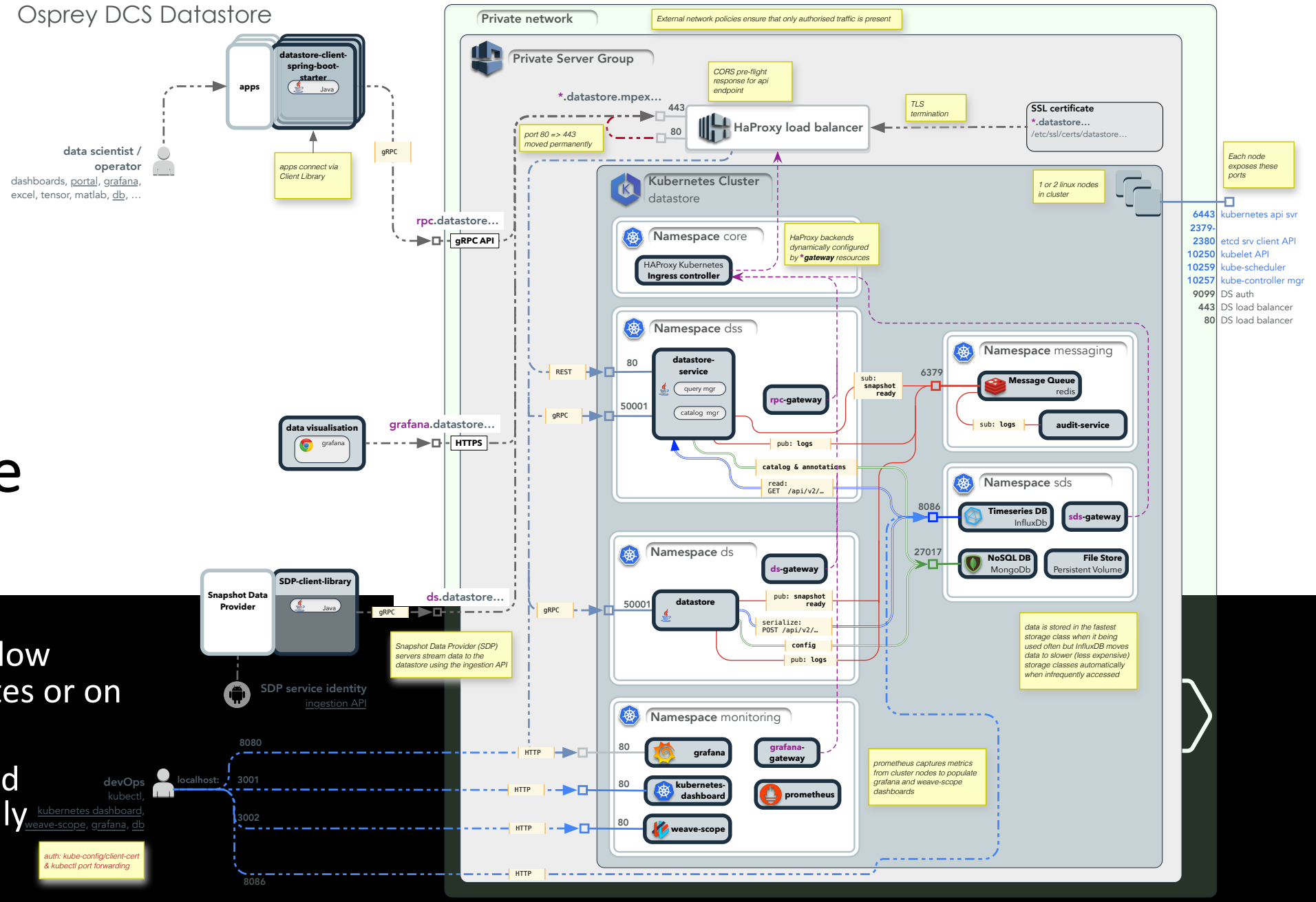
Java Programming Interface

DataPager

- `Integer totalPages = dataPager.getTotalPages()`
  - returns total number of pages in results set
- `Integer page = dataPager.getCurrentPage()`
  - returns the one-based page number of the current page in the DataPager. If all data is retrieved in one page then this is always 1.
- `Integer totalRows = dataPager.getSize()`
  - returns total number of time-correlated items (rows) in result set
- `Dataset dataset = dataPager.get(Integer page)`
  - returns dataset for the given page of data. If the requested page is not yet fetched then the method blocks until the requested page is ready. Current page is set to the given page plus 1
- `void dataPager.callWhenReady(Consumer<Dataset> callback, Integer page)`
  - calls given callback function with Dataset when the requested page's dataset is availabl
- `void dataPager.nextWhenReady(Consumer<Dataset> callback)`
  - calls given callback function with Dataset when the current page's dataset is available
- `Boolean dataPager.hasNext()`
  - returns true if a page of results (dataset) corresponding to the DataPager's current page exists in the results set
- `Boolean dataPager.isReady([Integer page])`
  - returns true if asynchronous loading has completed for the current page, or the optionally specified page
- `void dataPager.close()`
  - is called to close the data pager and release the cached data. If the program is being closed then this operation is not required as resources will be released automatically

deploy simply using Kubernetes or directly on bare metal

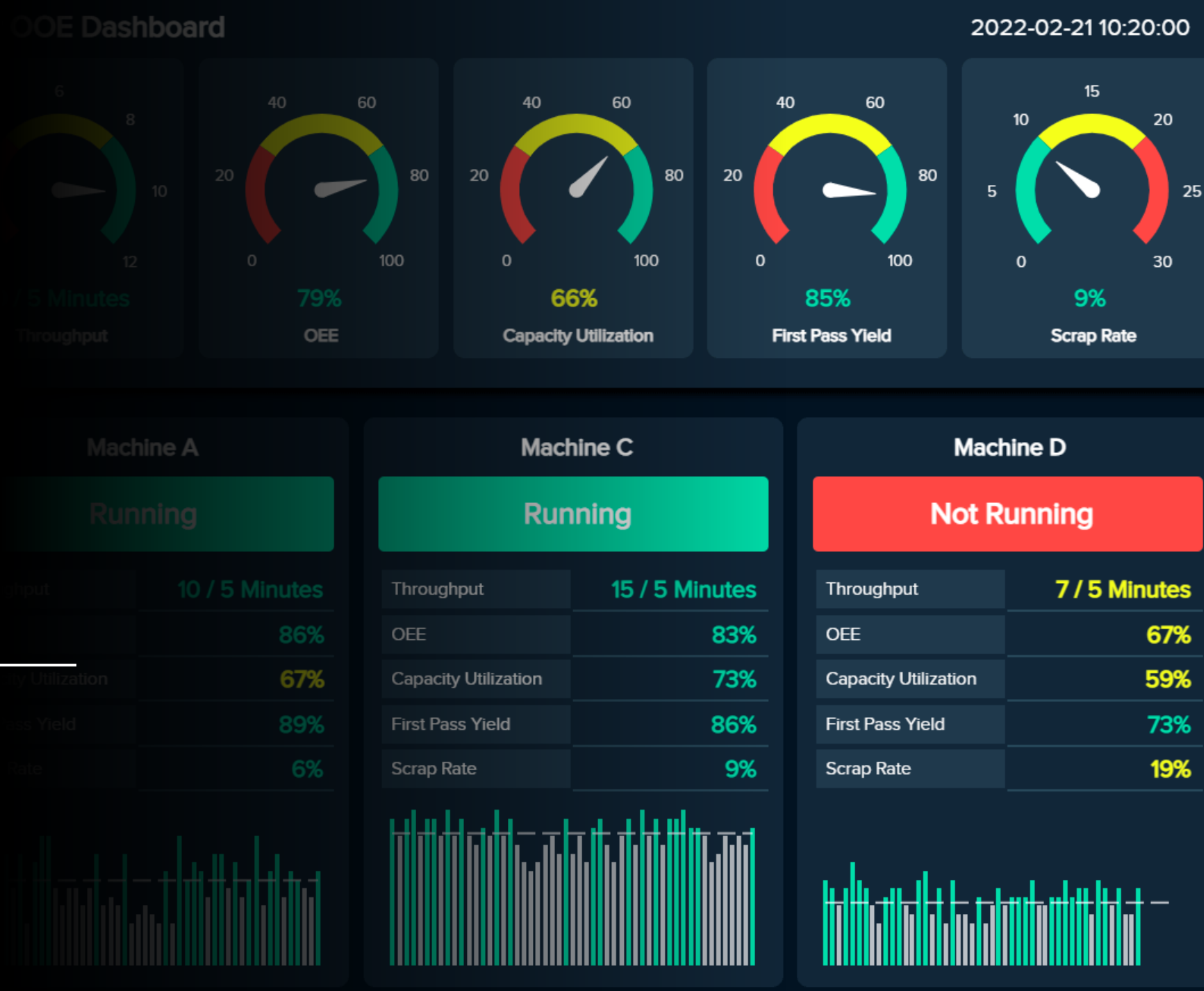
- Modular components allow deployment in Kubernetes or on bare-metal
- Lends itself to distributed deployments and is highly scalable



Example code

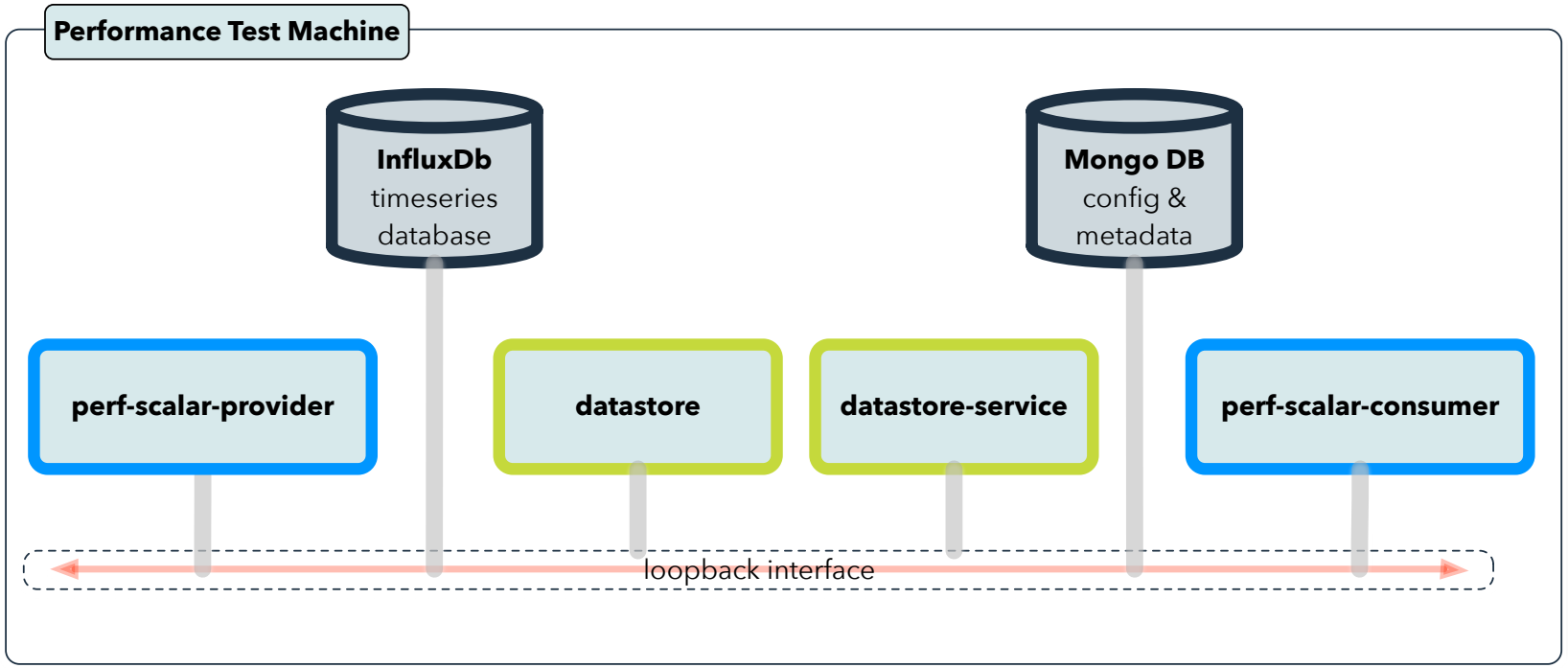
```
class DatastoreServiceClientExample {

    // Service running on port 50002 on localhost.

    private static String host = "localhost";

    private static Integer port = 50002;


    // Use plain text for messages, keep the connection alive

    private static Boolean usePlaintext = true;

    private static Boolean keepAliveWithoutCalls = true;

    private static Integer timeoutSeconds = 30;


    public void example1() {

        // Get a datastore service client connection

        DatastoreServiceClient client = DatastoreServiceClient.getClient(host, port,
timeoutSeconds, usePlaintext, keepAliveWithoutCalls);


        // Get all snapshot data received in the last 5 minutes as json

        Dataset dataset = client.findAll(Query.of("SELECT `*.*` WHERE time >= '-5m'"));

        JSONObject jsonData = dataset.asJson();


        // Get another page of snapshot data

        var perPage = 1000;

        Dataset pagedDataset = client.findPages(Query.of("SELECT `*.*` WHERE time >= '2022-07-
29T15:46:17.126Z' AND time < '2022-07-29T15:49:18.938Z'", perPage));


        // Get a value.  first value in `pvName`: mpexPv01, `pvField`: value (note that if a
value field is the only field

        // in a structure then the structure is replaced by the value field directly, so in
this case the vector

        // is a List of Doubles not a List of Maps to Doubles.

        Double value = pagedDataset.get("mpexPv01").get(0);


        // Get metadata for all PVs with Image in the name

        PVMetadata pvMetadata = client.findAllPVs(".*Image");


        // Get metadata for all annotations with FAIL in the name

        AnnotationMetadata annotationMetadata = client.findAllAnnotations("FAIL");

    }

}
```

Performance



Configuration

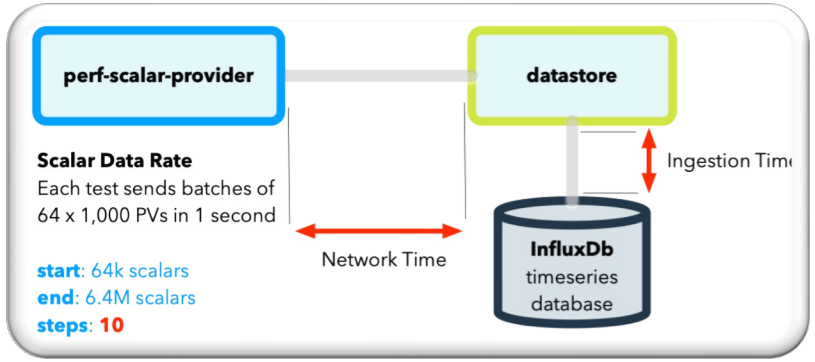
- 10 Core
- 32GB heap space:
  - Datastore
  - Datastore Service
  - Perf-Scalar Provider
  - Perf-Scalar-Consumer
- Transmitted over loopback network interface
- Scalar data with 0.1% Random Errors



Three Test Scenarios, 23 Tests

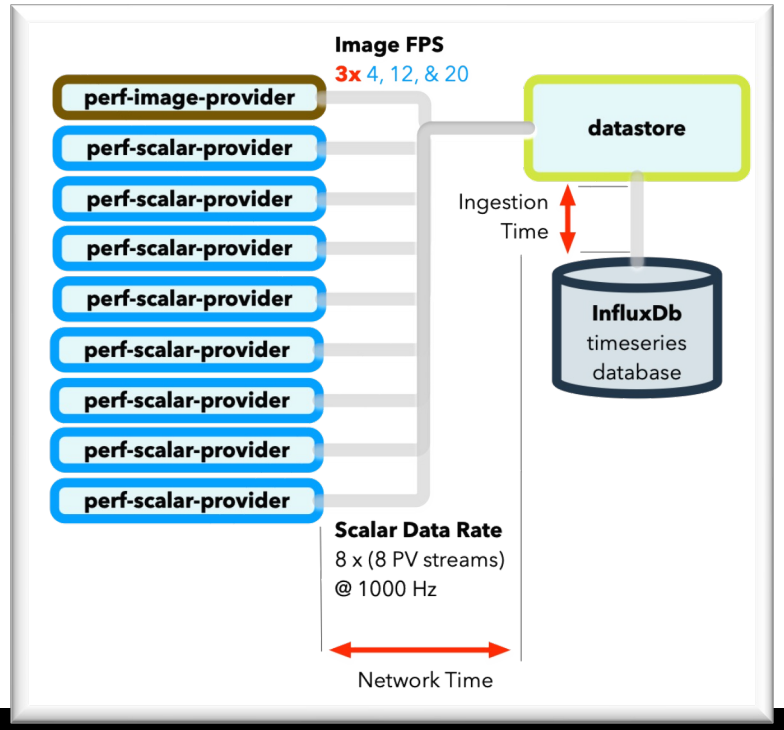
1. Gradual Loading

Simulation of large data loads ramping up from low to high



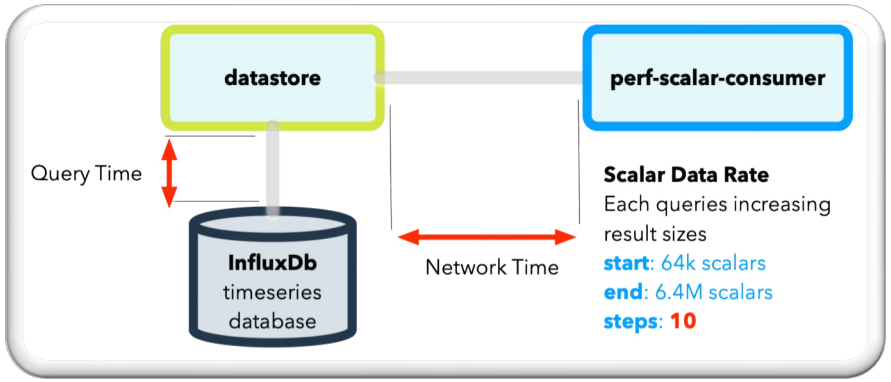
2. Concurrency

Simulation of multiple concurrent data sources streaming data into the Datastore



3. Query

Query Performance ramping up from small to large query results



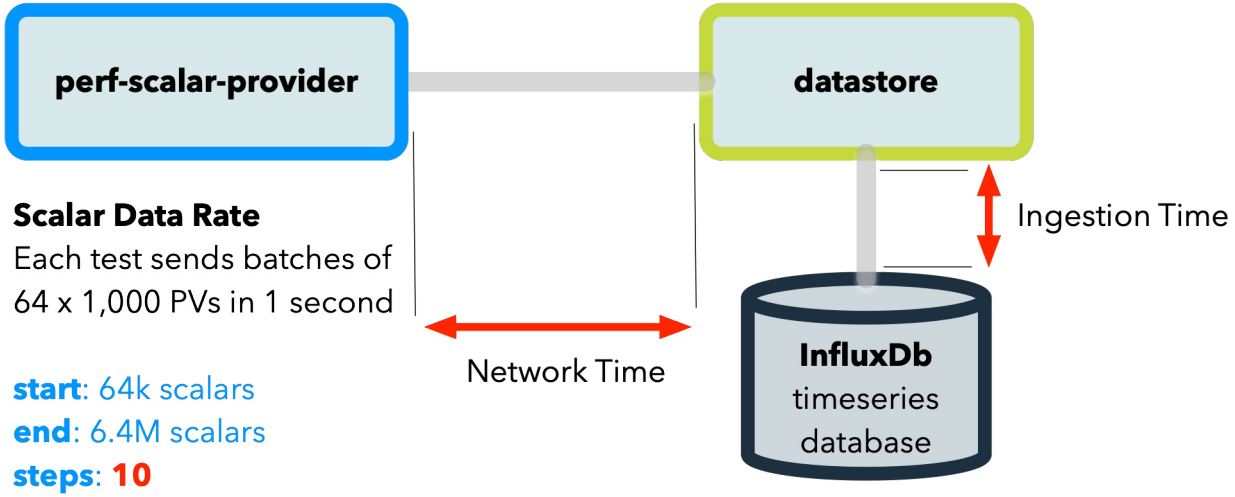
Results Summary

Without Data partitioning or clustered DB

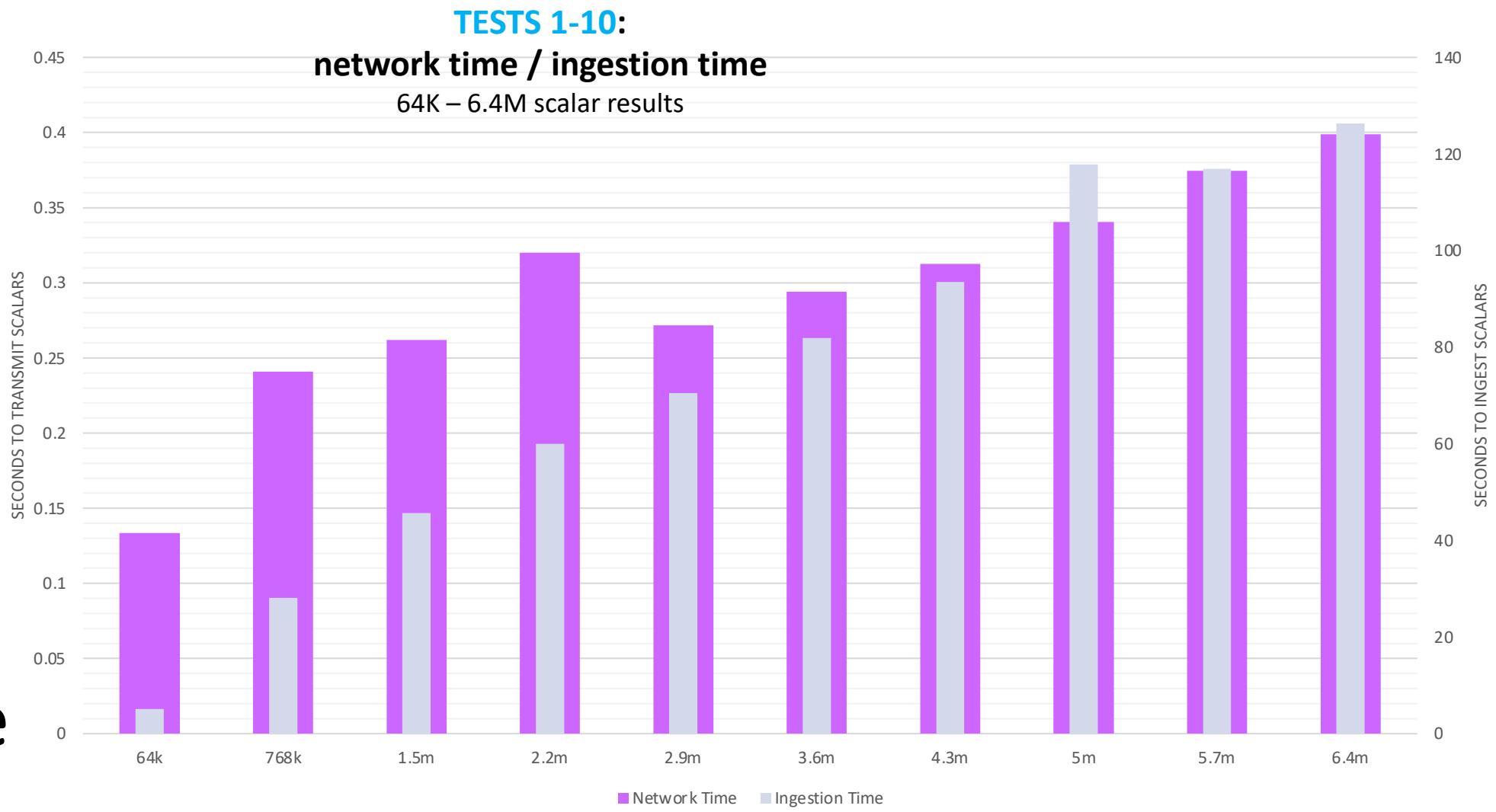
- **With given Configuration, Datastore performance is summarized as follows**
- **Datastore**
  - **Up to 16.0M Scalars can be received per second (1Gbps card saturated)**
  - **Up to 50.6K Scalars can be stored per second (10Mbps write speed)**
  - **Up to 37 FPS video feed can be transmitted live (with concurrent scalar data test 177Mbps +scalars)**
  - **Up to 23 FPS video feed can be ingested live (with concurrent scalar data test 110Mbps +scalars)**
- **Concurrent access**
  - **Overall performance increases with concurrency**
  - **No concurrency limit detected at 9 concurrent streams**
- **Datastore Service Scalar Query**
  - **7.1M Scalar results can be delivered per second (1Gbps card saturated)**
  - **113K Scalar results can be retrieved from the database per second (16Mbps write speed)**
  - **SOME Queries cause VERY BAD performance**

1. Ramp-up data ingestion rate

- TEST 1: 64K SCALARS
- TEST 2: 768K SCALARS
- TEST 3: 1.472M SCALARS
- TEST 4: 2.176M SCALARS
- TEST 5: 2.88M SCALARS
- TEST 6: 3.584M SCALARS
- TEST 7: 4.288 SCALARS
- TEST 8: 4.992M SCALARS
- TEST 9: 5.696M SCALARS
- TEST 10: 6.4M SCALARS

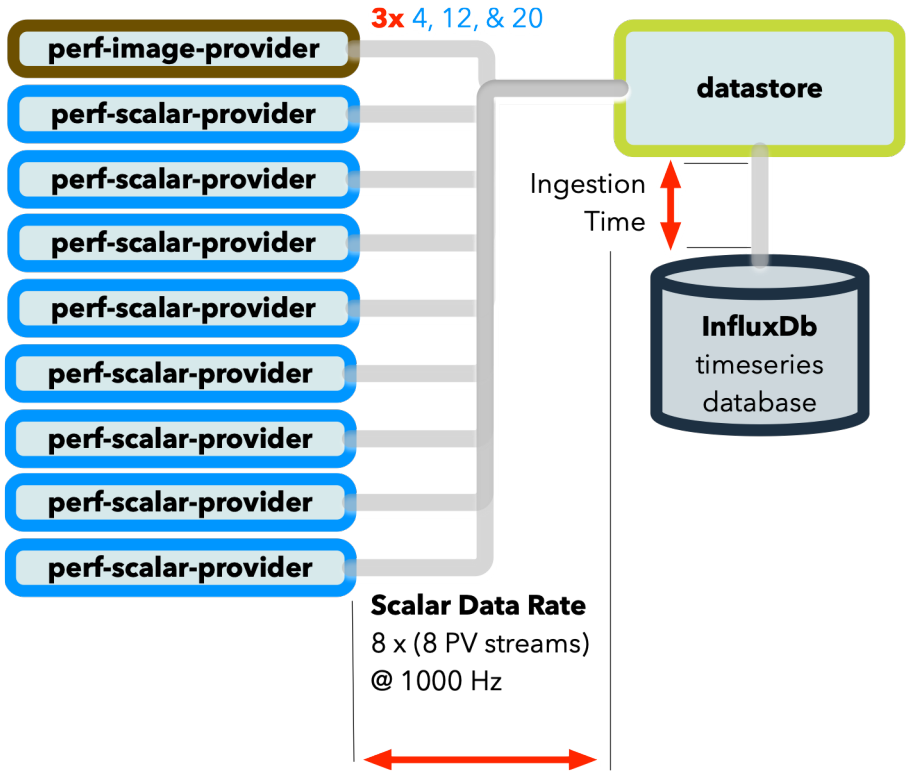


Ramp up Ingestion Performance

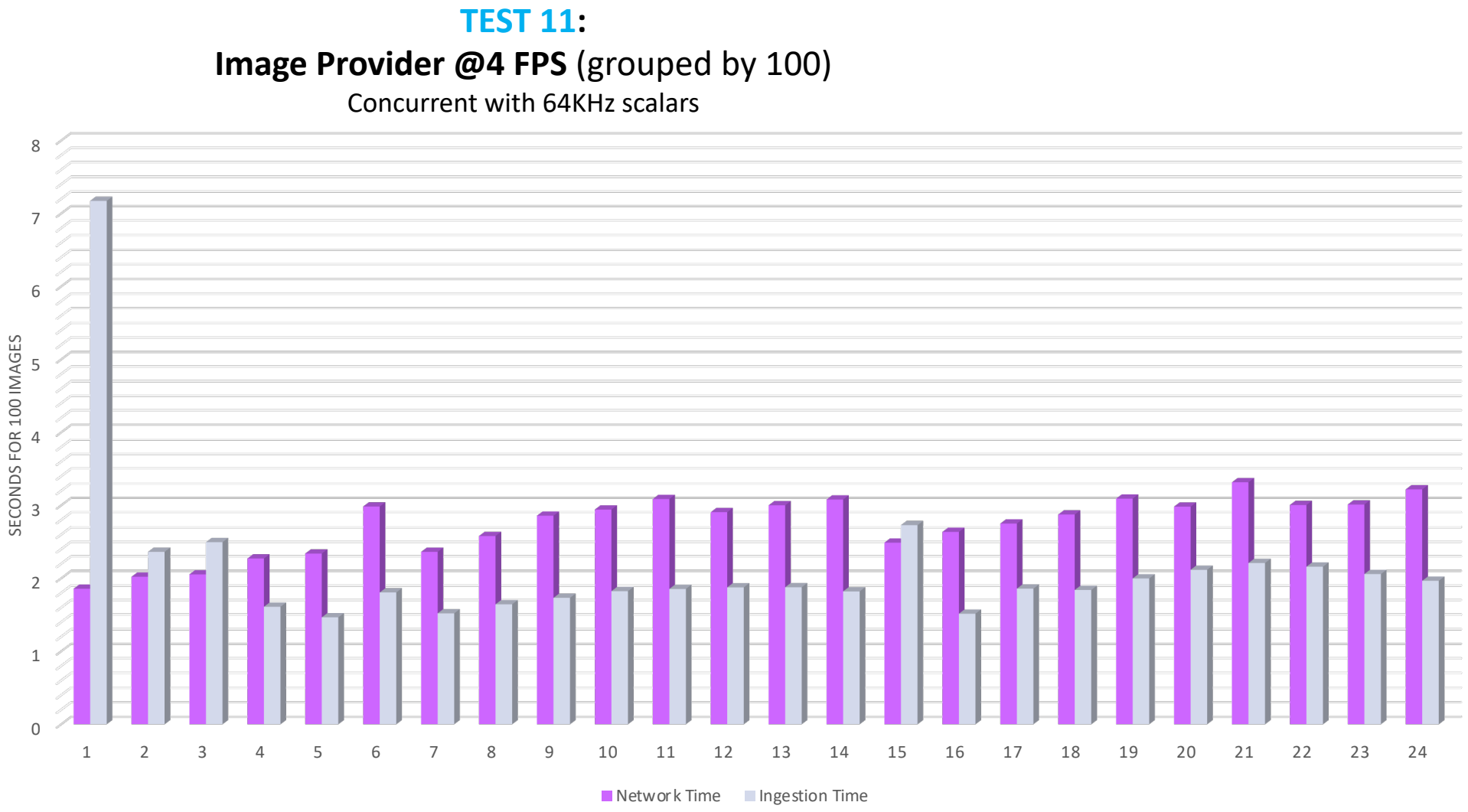


2. Ramp-up volumes during concurrent ingestion

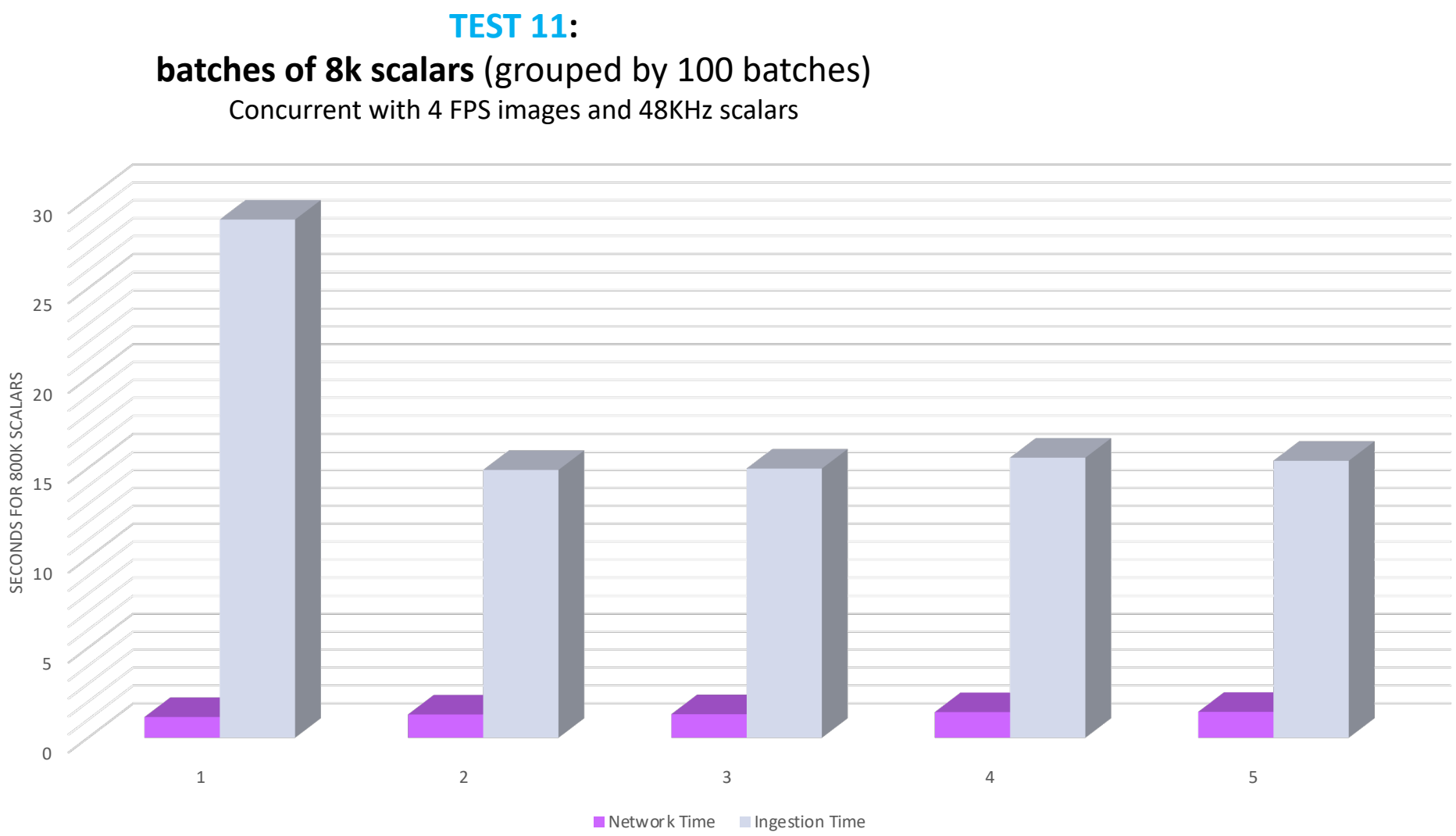
- SCALAR PROVIDERS
  - COUNT: 8
    - PVS: 8 EACH
    - SCALAR: DOUBLE
    - SPEED: 1KHZ
- IMAGE PROVIDER
  - SIZE: 640X240X3
  - RAMP-UP IMAGE PROVIDER RATE
    - TEST 11: 4 FPS
    - TEST 12: 12 FPS
    - TEST 13: 20 FPS



Concurrent streams



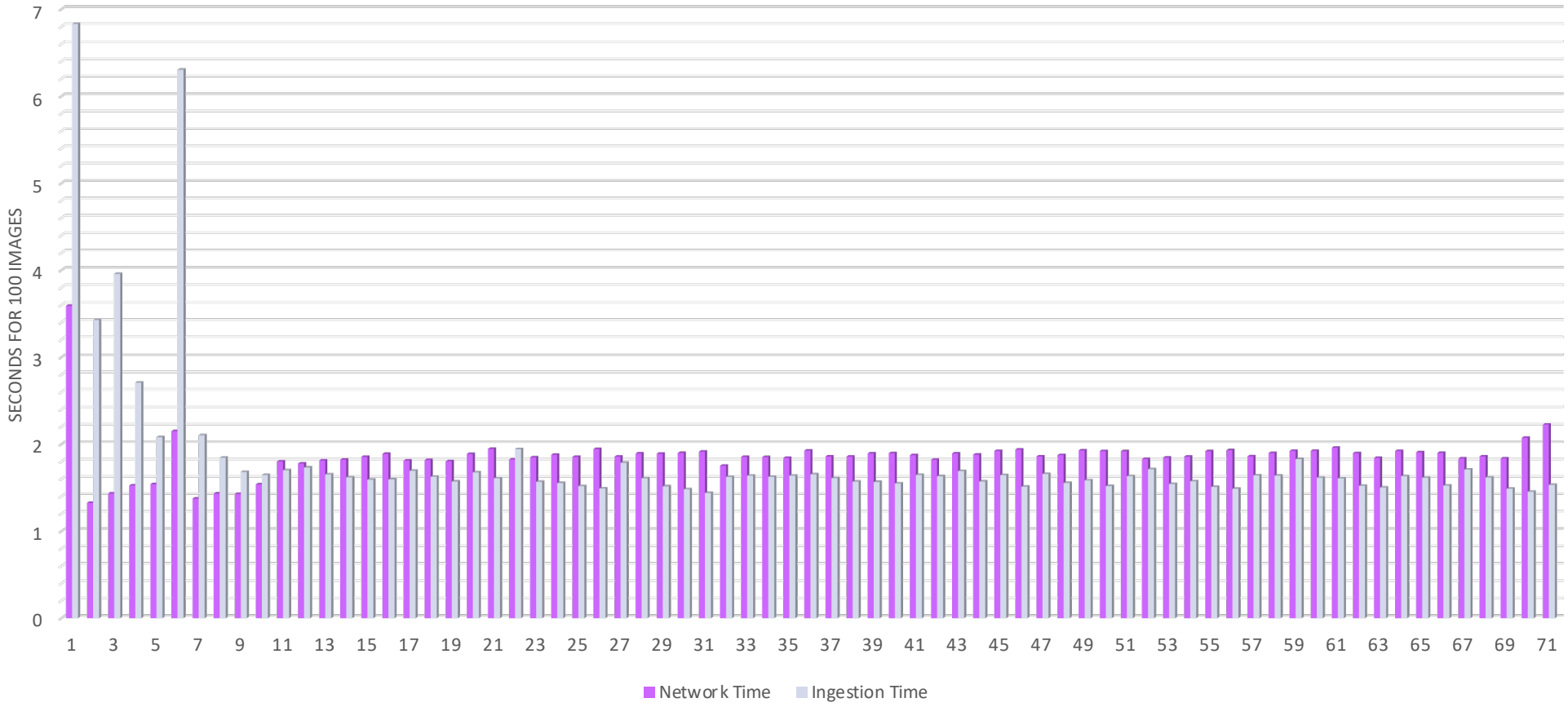
Concurrent streams



TEST 12:

Image Provider @ 12 FPS (grouped by 100)

Concurrent with 64KHz scalars

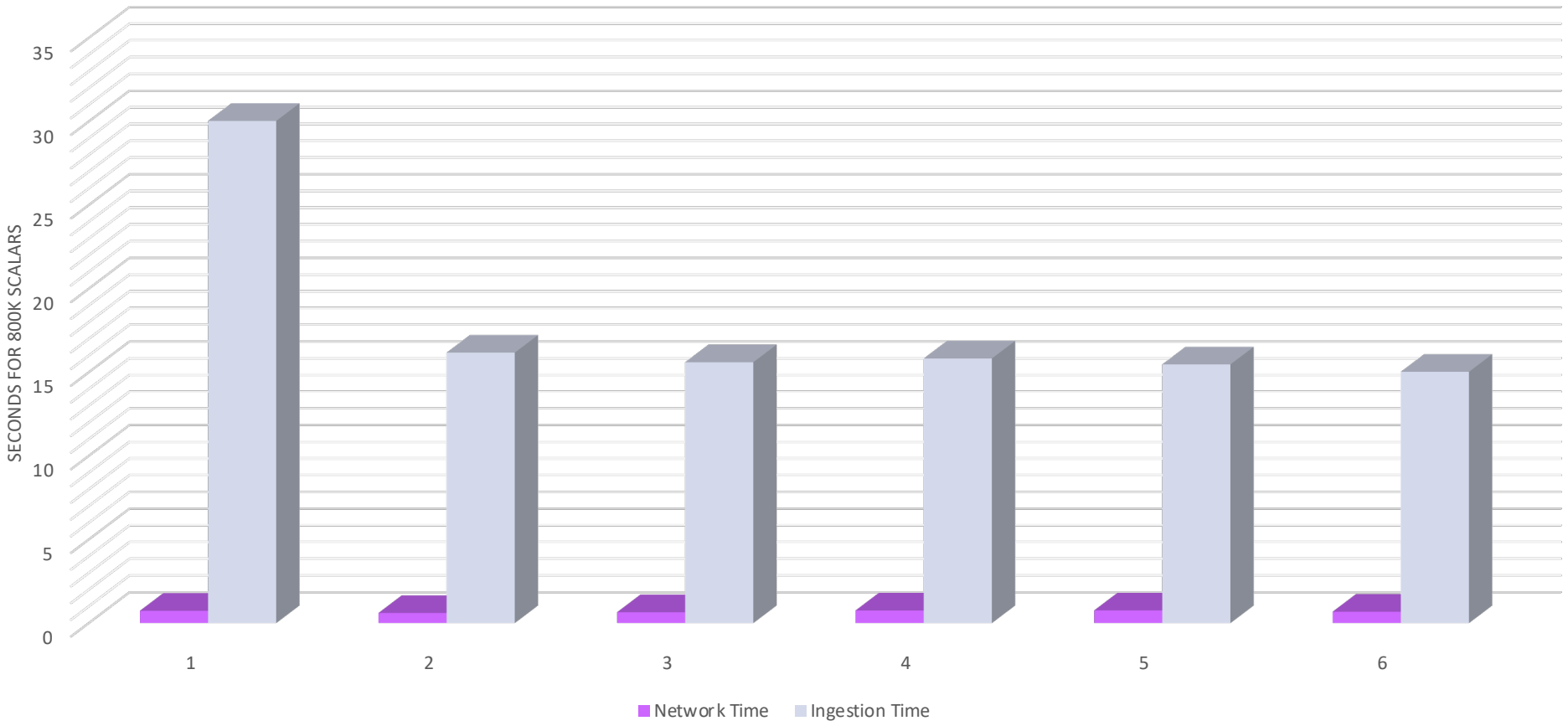


Concurrent streams

TEST 12:

8k scalar batches (grouped by 100 batches)

Concurrent with 12 FPS images and 48KHz scalars

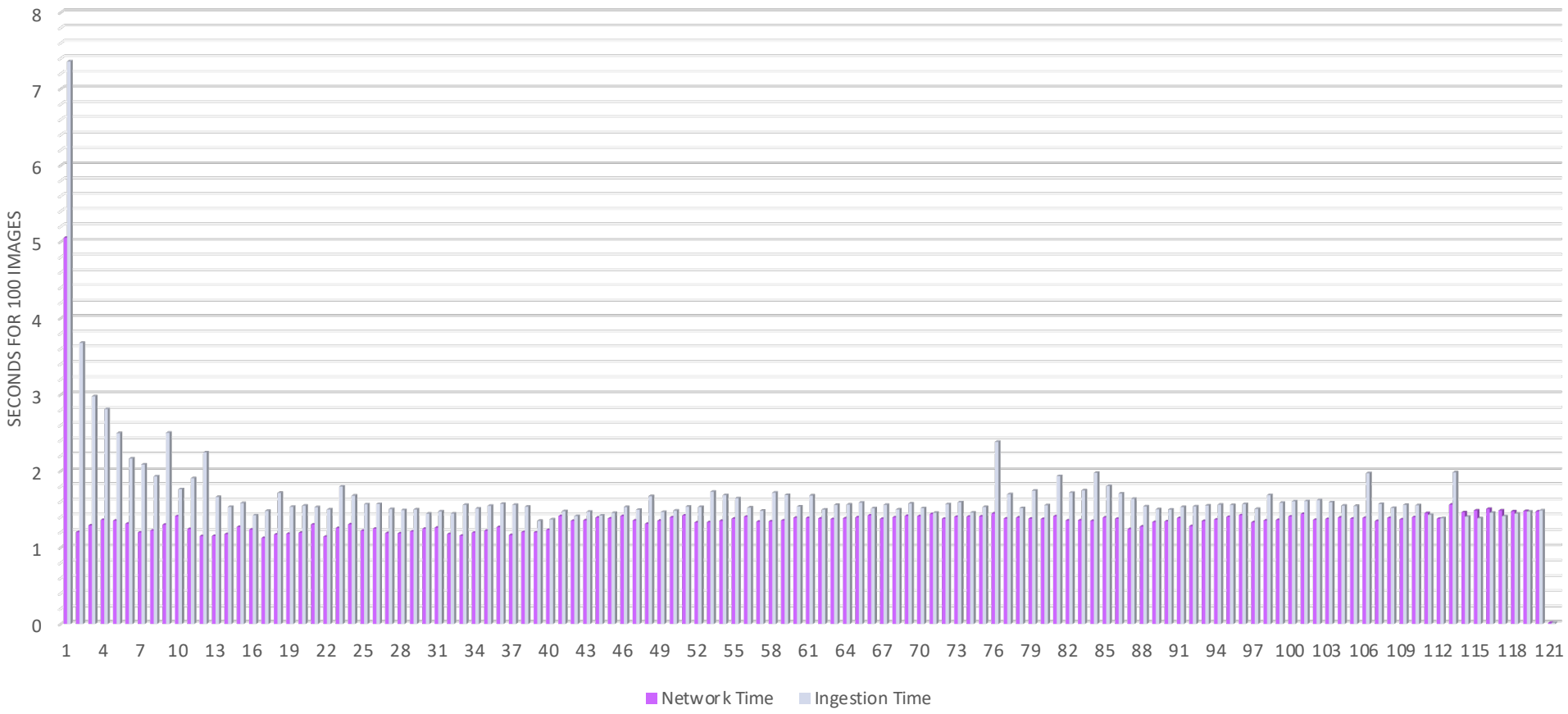


Concurrent streams

TEST 13:

Image Provider @20 FPS (grouped by 100)

Concurrent with 64KHz scalars

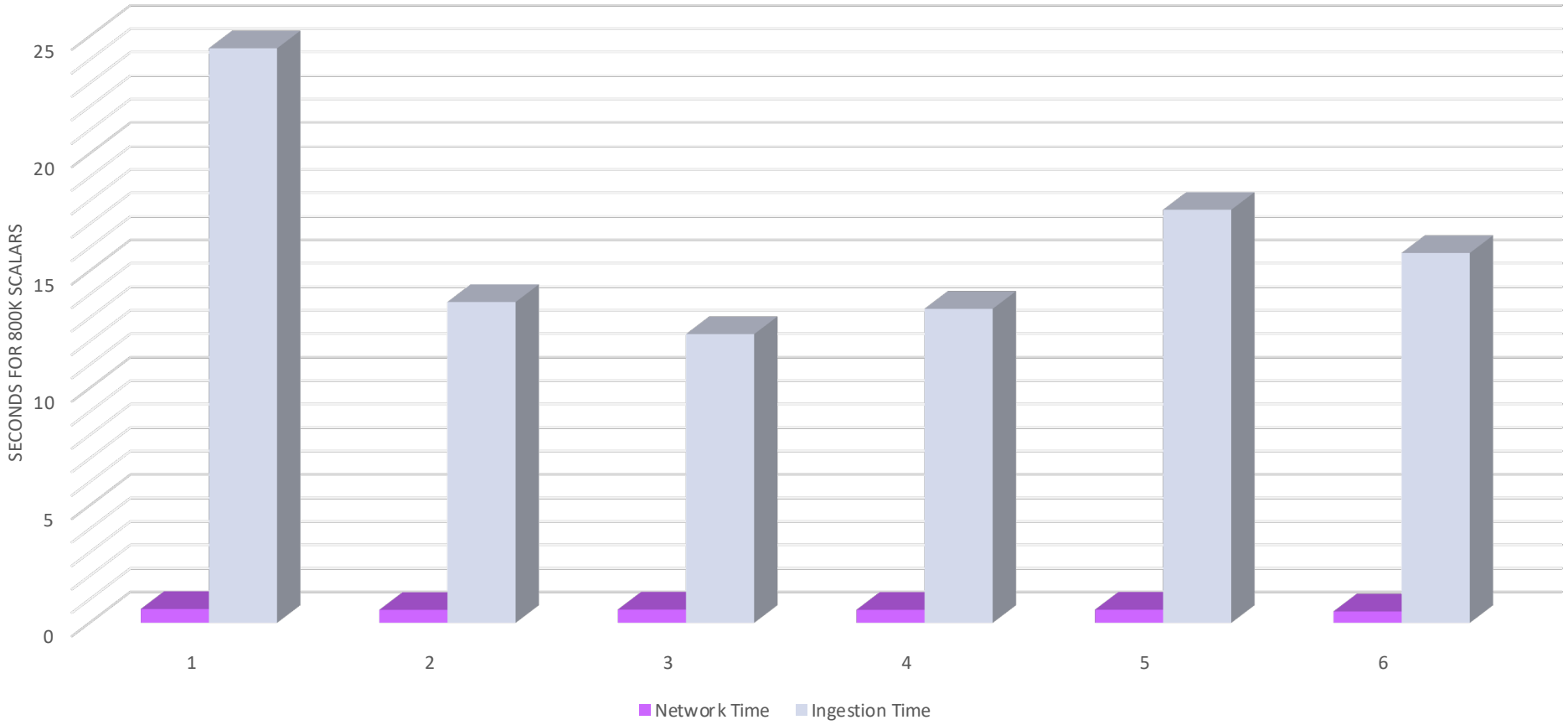


Concurrent streams

TEST 13:

8k scalar batches (grouped by 100 batches)

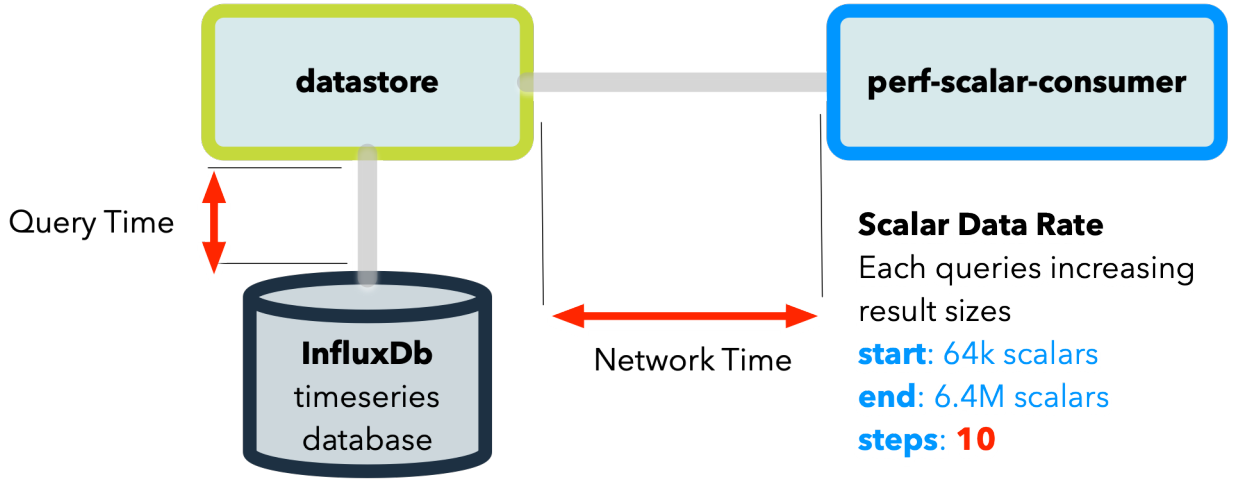
Concurrent with 20 FPS images and 48KHz scalars



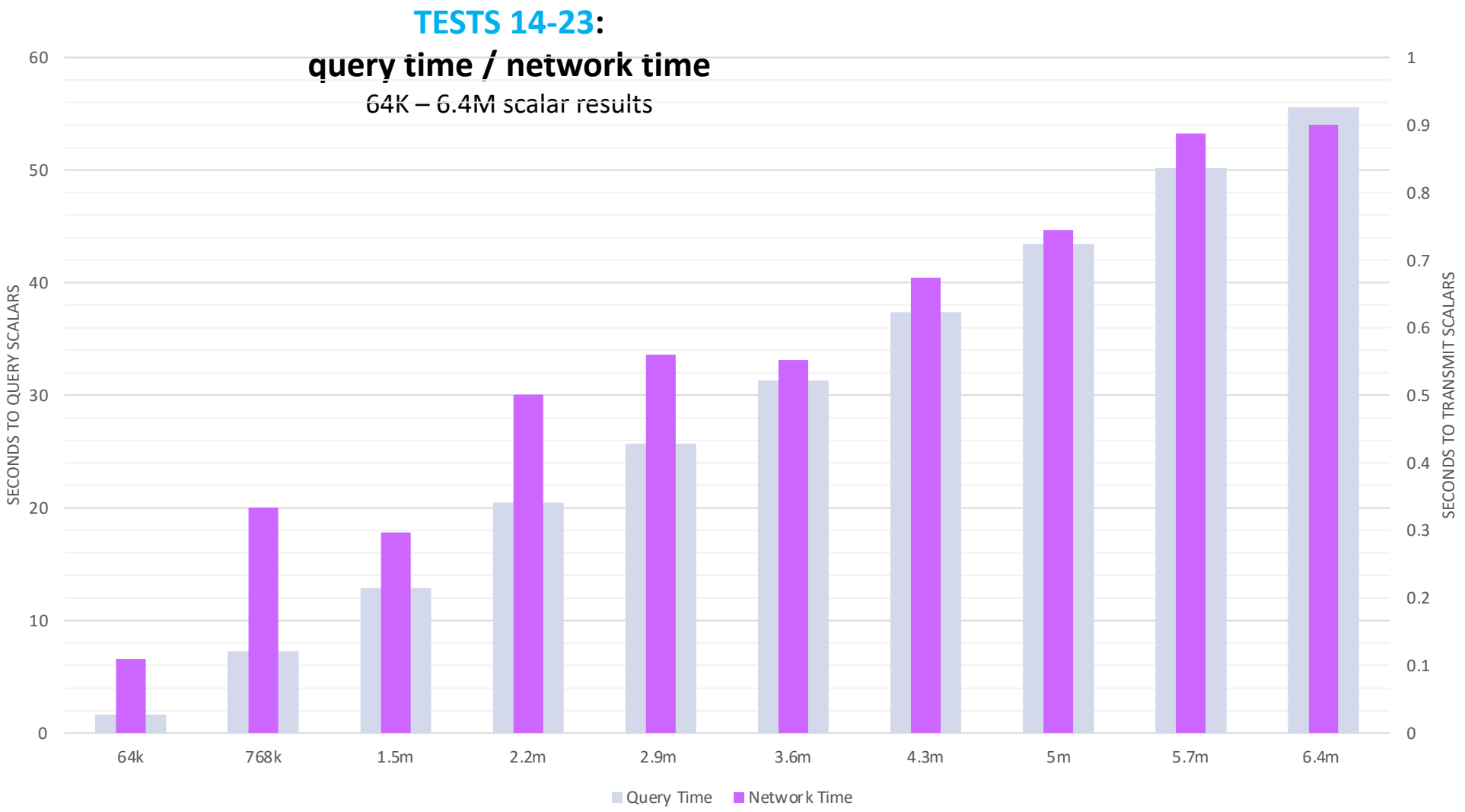
Concurrent streams

3. Ramp Up Query Size

- TEST 14: 64K SCALARS
- TEST 15: 768K SCALARS
- TEST 16: 1.472M SCALARS
- TEST 17: 2.176M SCALARS
- TEST 18: 2.88M SCALARS
- TEST 19: 3.584M SCALARS
- TEST 20: 4.288 SCALARS
- TEST 21: 4.992M SCALARS
- TEST 22: 5.696M SCALARS
- TEST 23: 6.4M SCALARS



Ramp up Query Performance



Datastore
from
Osprey DCS

- Thank You

George McIntyre
george@level-n.com

Contributors
- **Craig McChesney**
- **Christopher Allen**
- **Bob Dalesio**