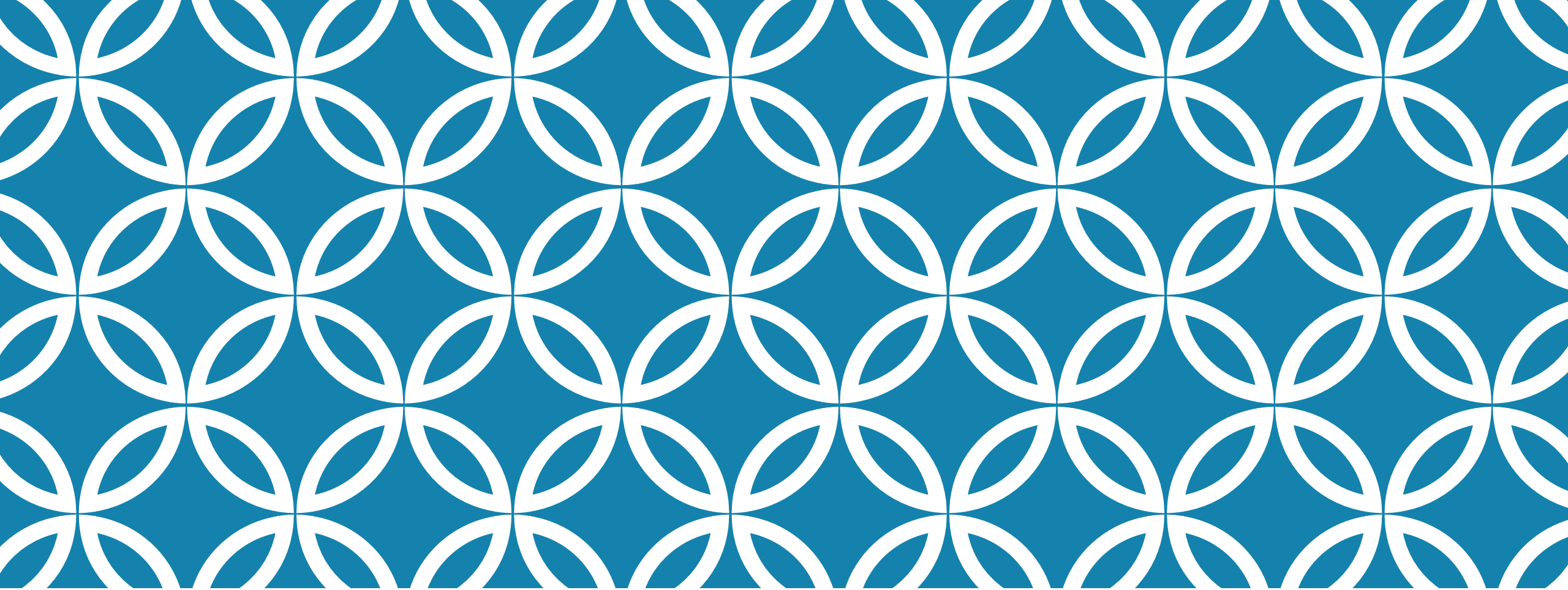


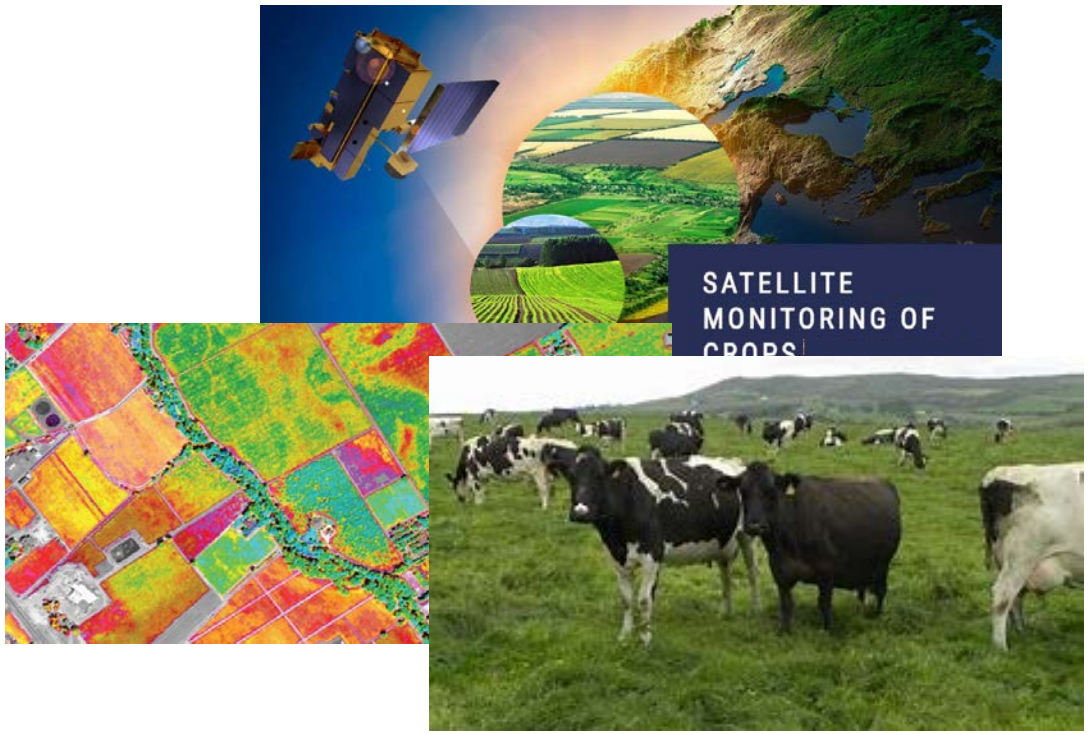
CASCADE: FULL DETAILS

Ken Birman

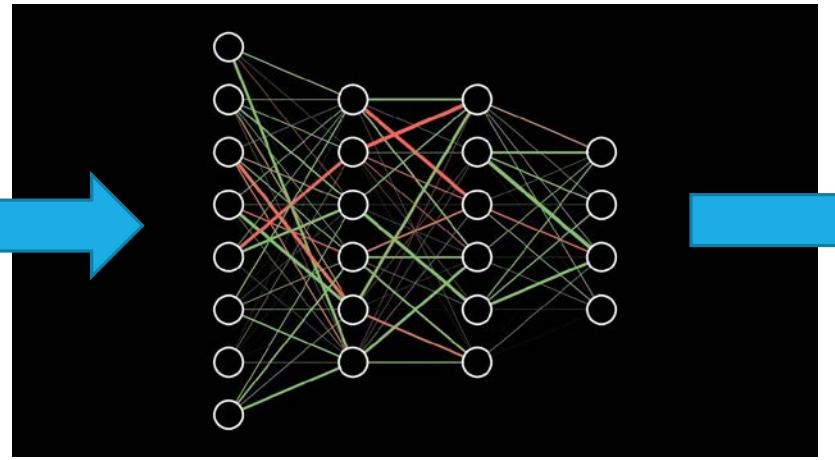
CS5412 Spring 2022
Lecture 10

THE WORLD IS GENERATING A NEW WAVE OF IOT/ML PIPELINES… THERE ARE MANY USE CASES

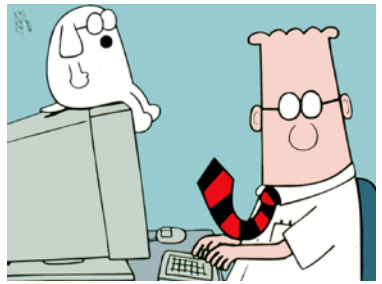
Data sources



Federated ML
Distributed AI

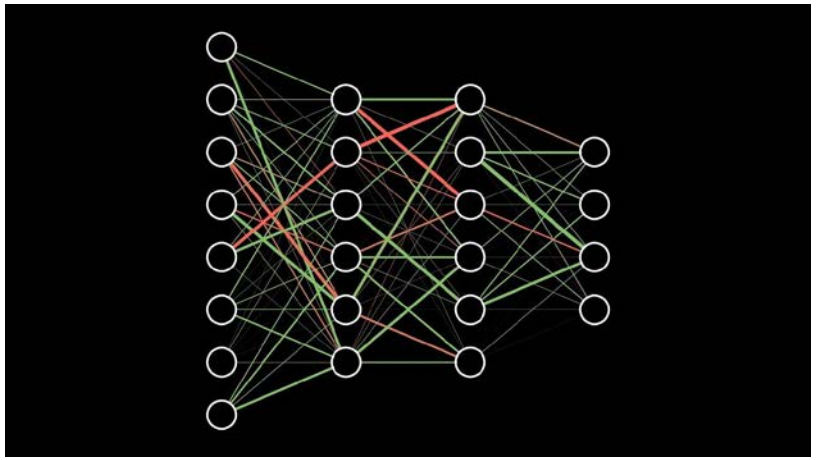


Smart
Queries



How much should I budget for raw milk purchases in March for my yoghurt factory?"

FEDERATED ML



Increasingly seen in robotics, smart homes, 5G, digital twin scenarios.

- The application is a graphical collection of AI classifiers / learners
- Nodes represent computational tasks.
- Edges represent data flow between distinct tasks.

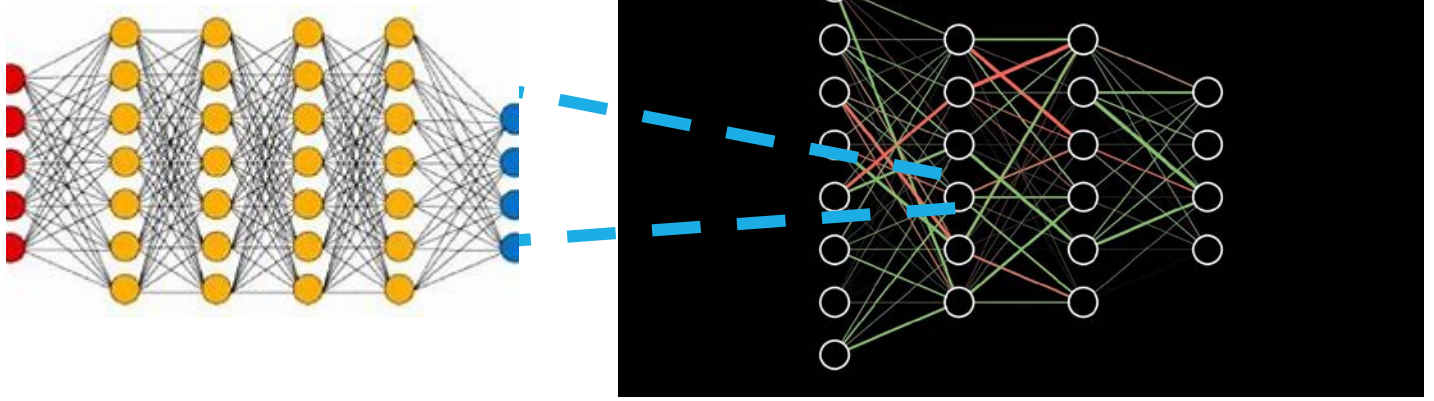
EACH λ REPRESENTS A DISTINCT ML ELEMENT

Many are parallel: Single λ may run on a pool of compute nodes.

Thus, to build a D-AI we build pipelines linking parallel tasks.

Today's cloud platforms have limited support for this model, lack the real-time and consistency guarantees needed for IoT.

DISTRIBUTED AI



A related concept

Used for AI algorithms designed to run on a parallel computer or cluster, for example stochastic gradient descent. Can also refer to the nodes in a DNN or CNN

A federated ML graph could have nodes that themselves are distributed AI components! Our graph would have subgraphs

HIGH LEVEL CASCADE GOALS

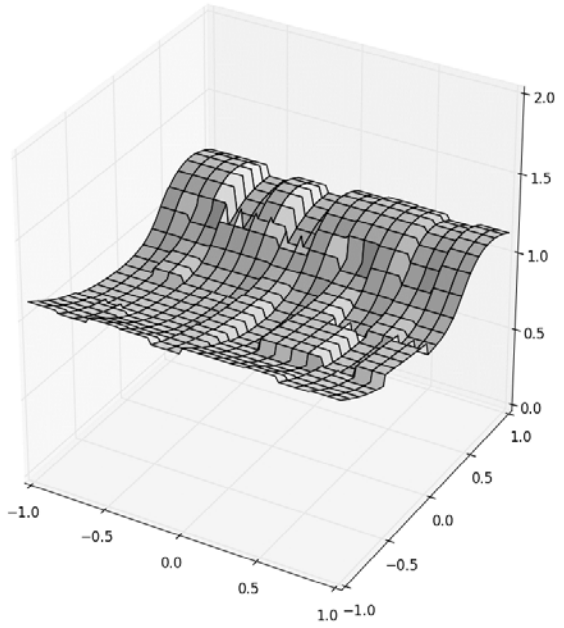
Legacy support: Easy to use with no need to change your code

Much faster than standard platforms: low delay, high bandwidth

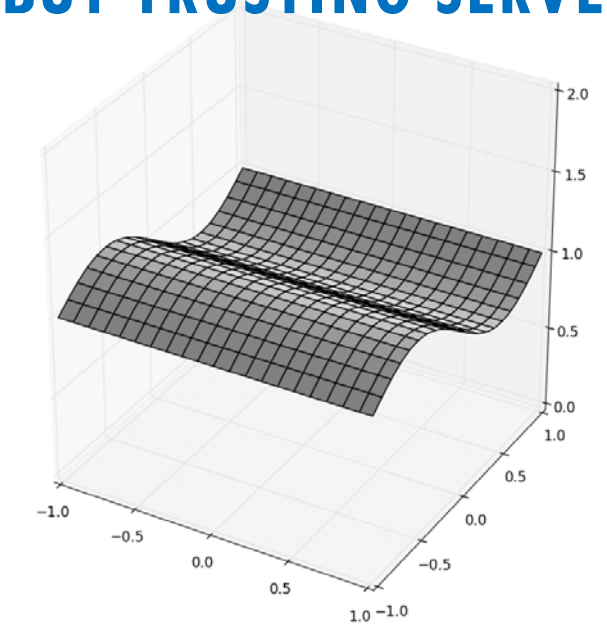
Stronger guarantees: Your ML doesn't fight platform "noise"

CONSISTENCY AND FAULT TOLERANCE

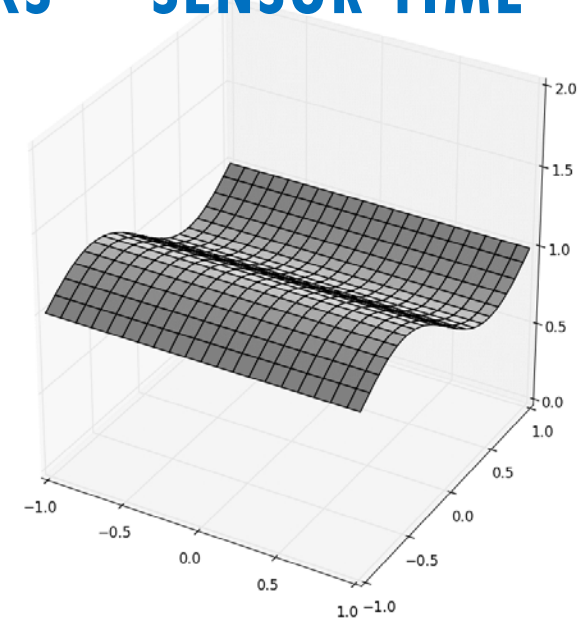
HDFS



CASCADE USING CONSISTENT CUTS BUT TRUSTING SERVER CLOCKS

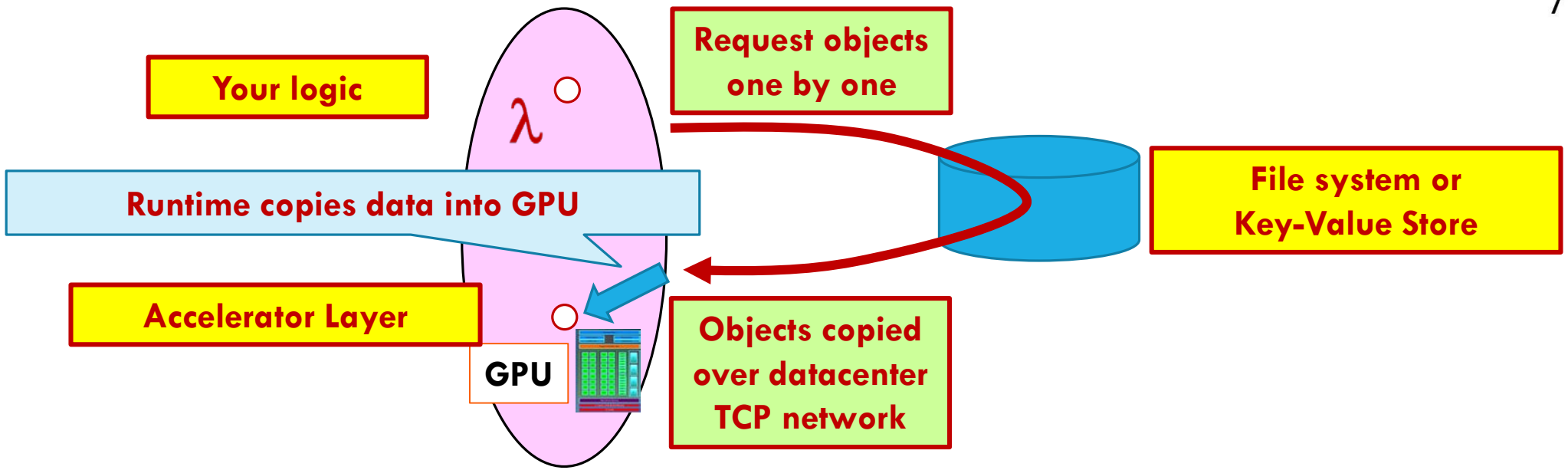


CASCADE WITH SENSOR TIME



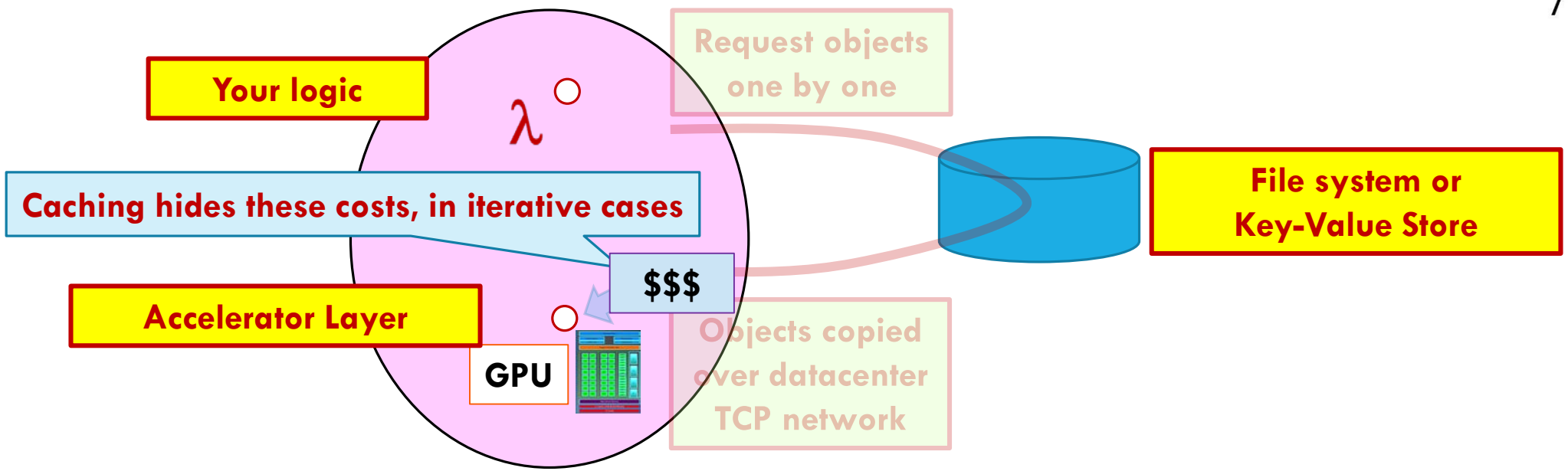
We provide high availability, auto-repair after failures, and strong consistency.
Apache HDFS (used by Spark) becomes noisy under time pressure.
Cascade (middle and right) supports “clean” temporal data access.

TRADITIONAL APPROACH



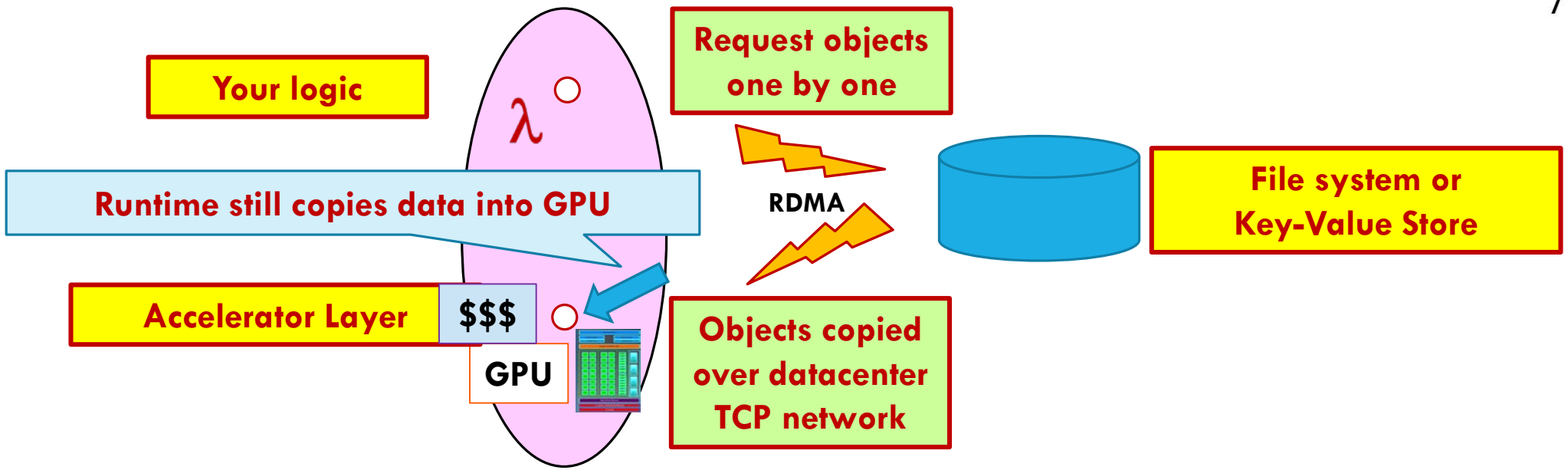
Your code is in its own address space, maybe on a different computer

TRADITIONAL APPROACH



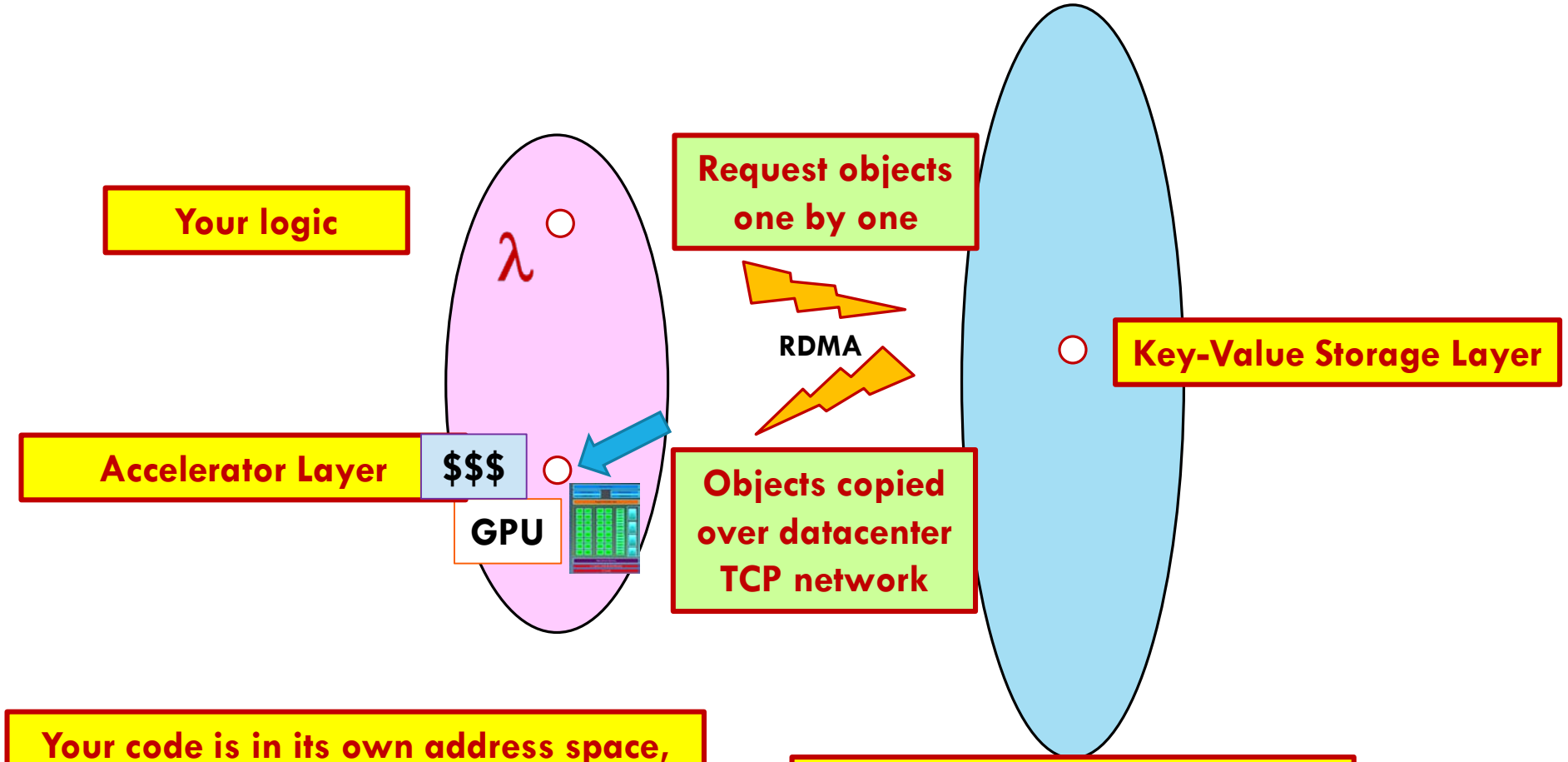
Your code is in its own address space, maybe on a different computer

RDMA CAN HELP… A LITTLE

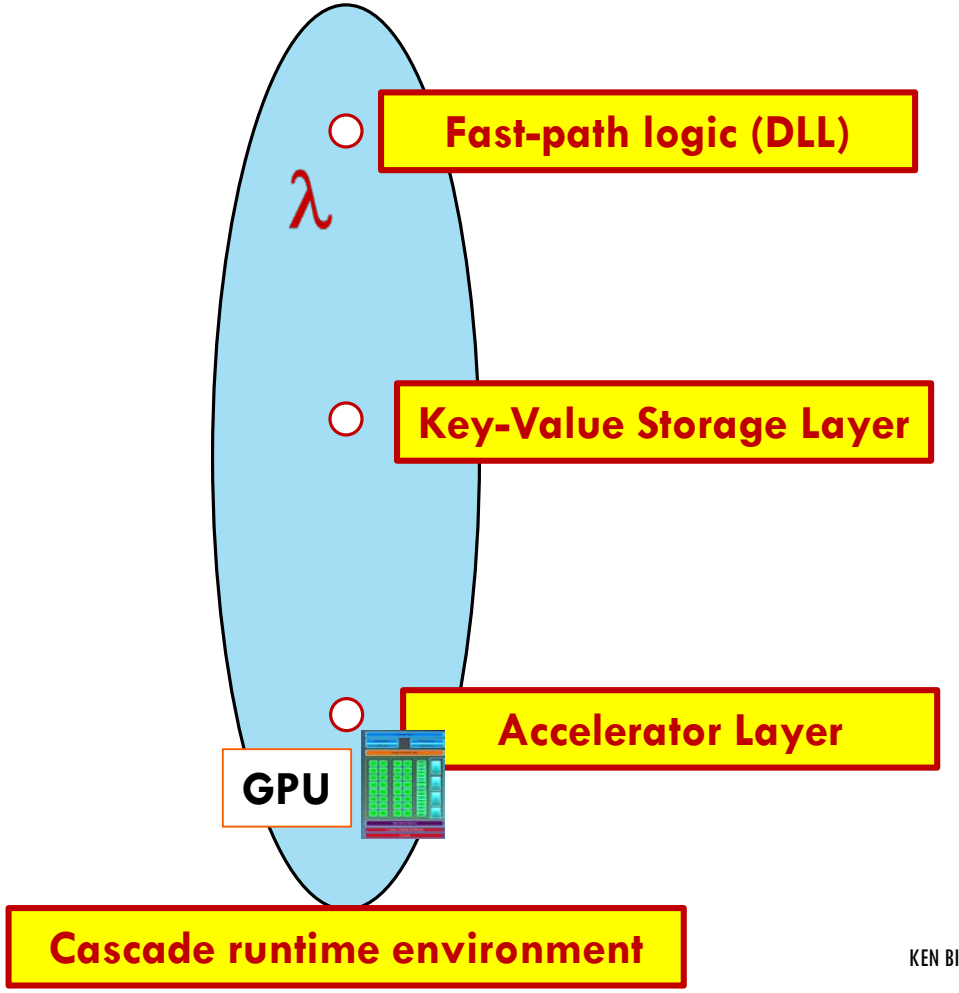


Your code is in its own address space, maybe on a different computer

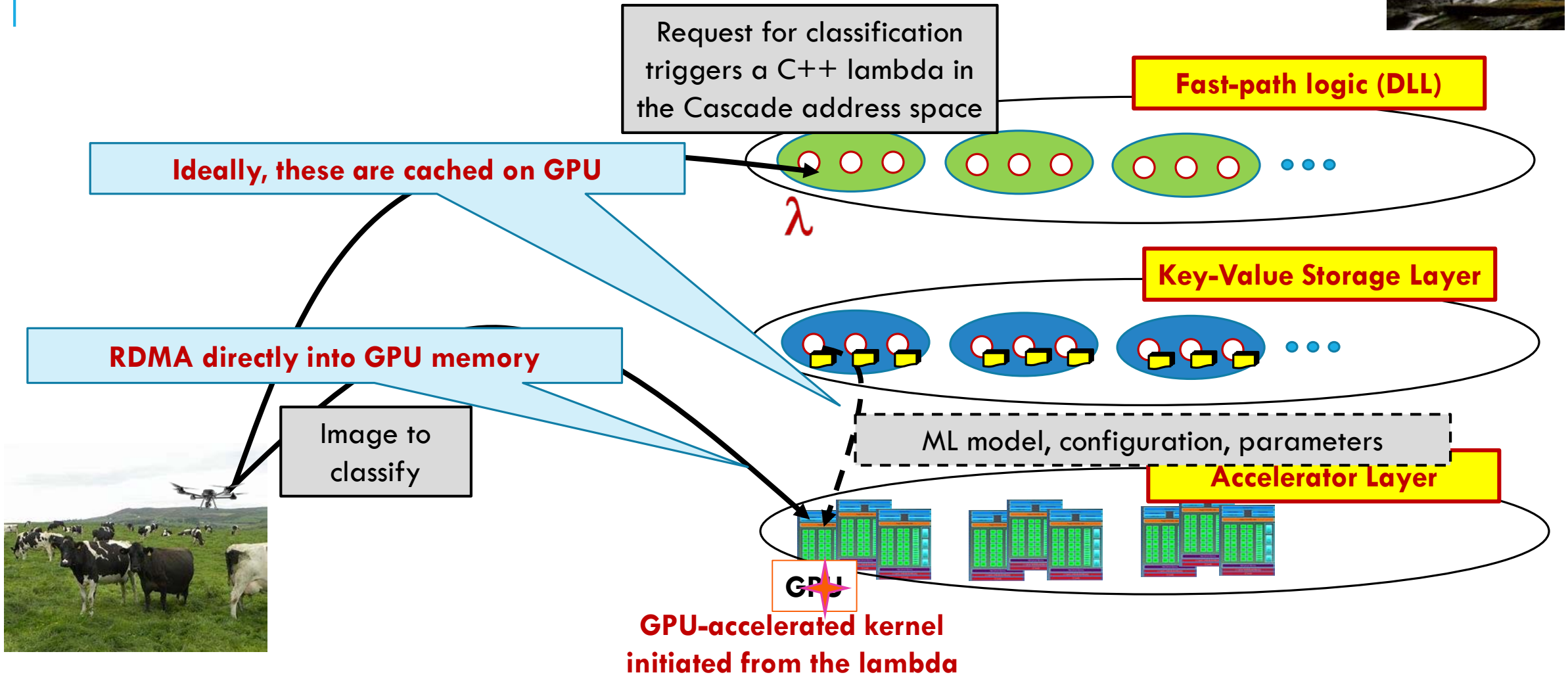
CASCADE CAN BE USED THIS WAY…



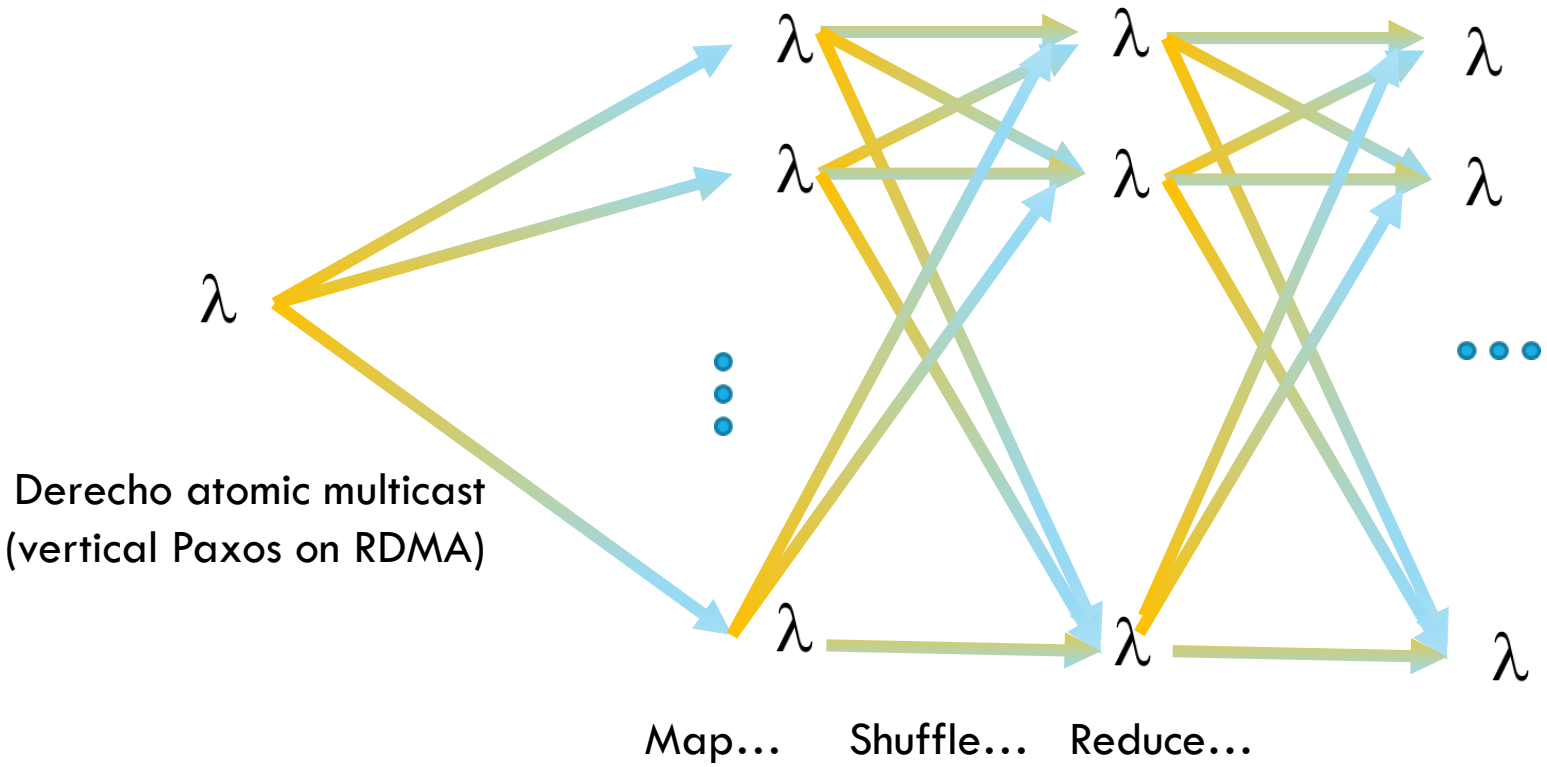
… BUT CAN ALSO HOST USER CODE



CASCADE: CUSTOMIZED SMART SERVICES



… AND WE CAN EVEN RUN MAPREDUCE



SO… WHAT MAKES IT HARD?

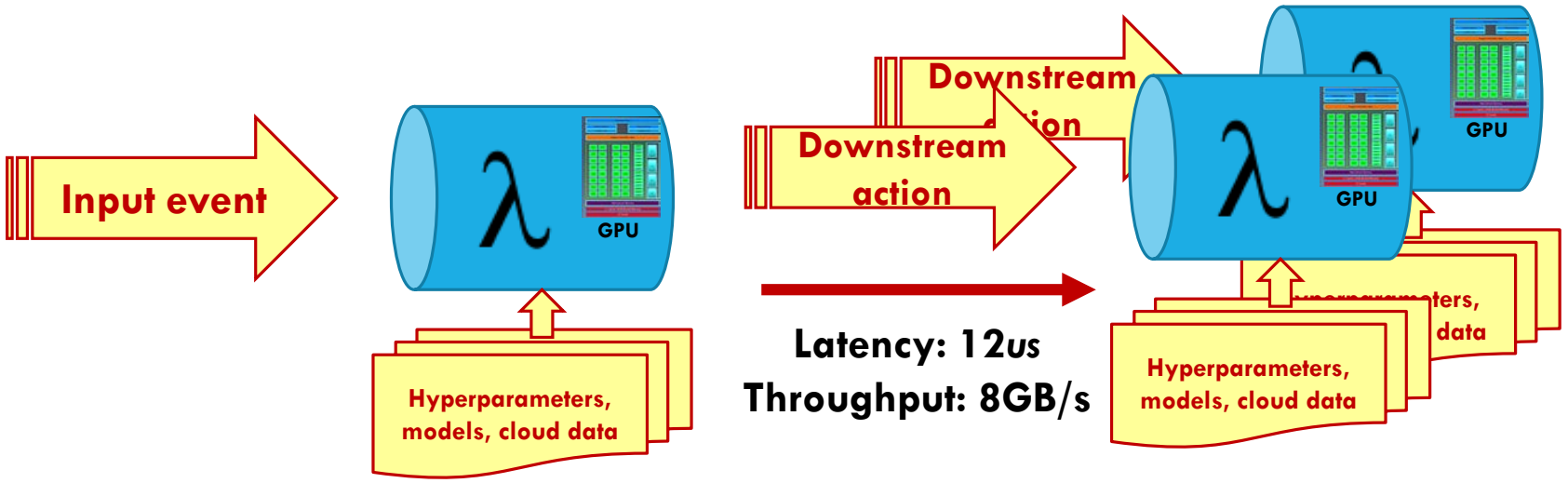
When storing objects, use the federation graph to anticipate how they will be used. ***Placement decisions*** should collocate objects at nodes where computations that need them will run.

Later, ***an event occurs***. Launch the computational pipeline on nodes that will (all) have the required inputs.

Metrics: delay, throughput, balanced workload, resource utilization.

FAST-PATH PERFORMANCE

A simple federated ML pipeline



Cascade is close to ideal efficiency on our hardware and 100 to 10,000x faster than common options like Apache Flink

DAIRY INTELLIGENCE WITH CASCADE

Integrate daily data

Date	cow_id	daily_yield	daily_fat	…	daily_protein
12/3/20	1	14	3.96	…	2.89
…	…	…	…	…	…
1/10/21	237	20	4.42	…	4.55

<field>/<cow_id>{<ts(ver)>}
daily_protein/cow_id1{ver_1} = 2.89

Download blobs from Azure &
Store to Cascade VCSS subgroup

```
client = cascade_py.ServiceClientAPI()
daily_yield_pool = client.get_object_pool("VolatileCascadeStore","daily_yield")
daily_yield_pool.put(cow_id,val)
```

Upload daily date to Azure Blob Storage

Date	cow_id	…
12/3/20	1	…
…	…	…
12/3/20	237	…

daily_fat/cow_id237{ver_38} = 4.42

Filtered image frames

Store to subgroup VCSS
Trigger image analysis

Action = black_cow_infer

```
white_cow_net = load_cow_model( \
    "cow_symbol.json", \
    'white_cow.params')
black_cow_net = load_cow_model( \
    "cow_symbol.json", \
    'black_cow.params')
...
action = cascade_context.get_next_action()
if action.type == "white_cow":
    pred = white_cow_net(action.frame)
else
    pred = black_cow_net(action.frame)
```

Farm server

External Client to Cascade

Streaming image frames through TCP portal

CV model
Image analysis

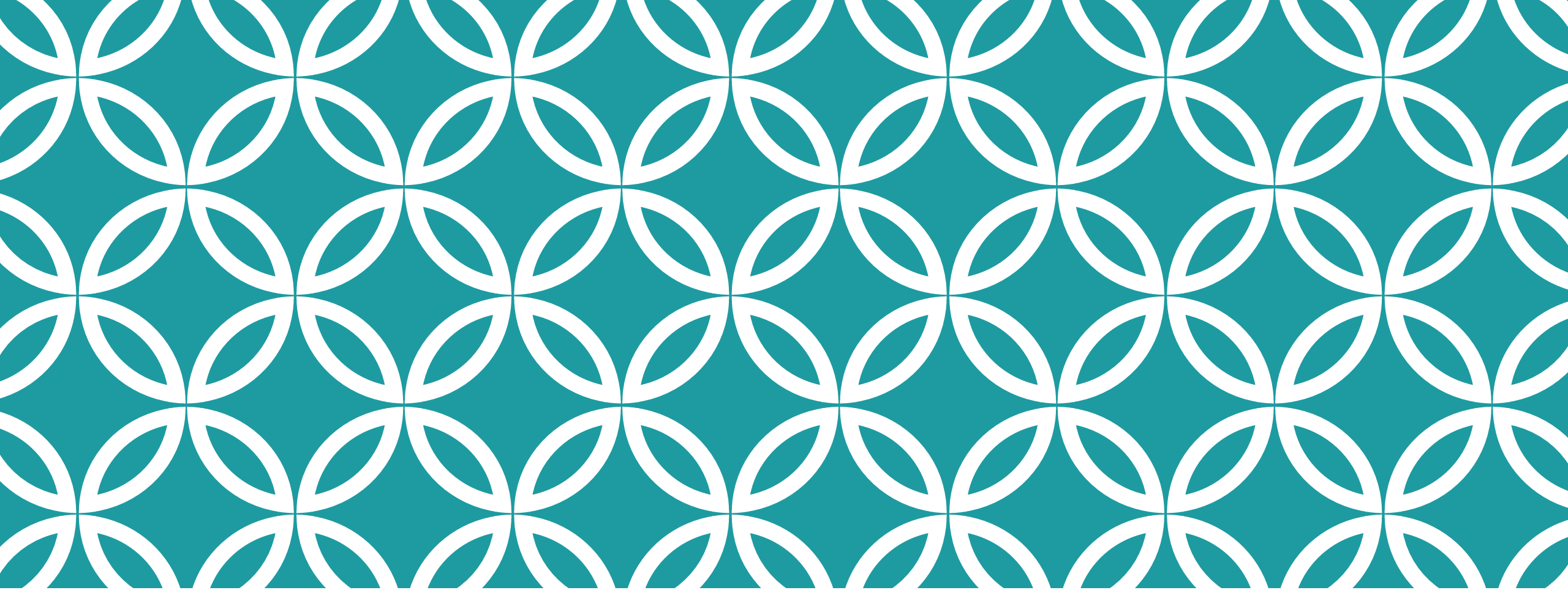
cow id: 127

Probability of calving
in next 8h is: 80%

ML Model
birth prediction

LINQ query to retrieve data of most recent 10 days from Cascade about cow 128

```
client = cascade_py.ServiceClientAPI()
cow_id = "cow_id128"
recent_data = []
for field in ["daily_yeild", "daily_fat", "daily_protein"]:
    field_pool = client.get_object_pool("VolatileCascadeStore",field)
    query_result = field_pool \
                   .where(lambda x:x.endswith("/"+cow_id)) \
                   .order_by(lambda x:x.version) \
                   .reverse()
                   .take(10)
    recent_data.append(query_result)
```

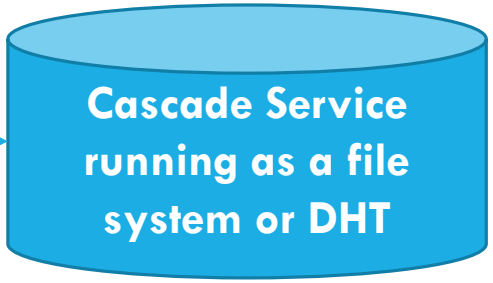


THE CASCADE MODEL

Is it a service? A library?

CASCADE IS A SERVICE

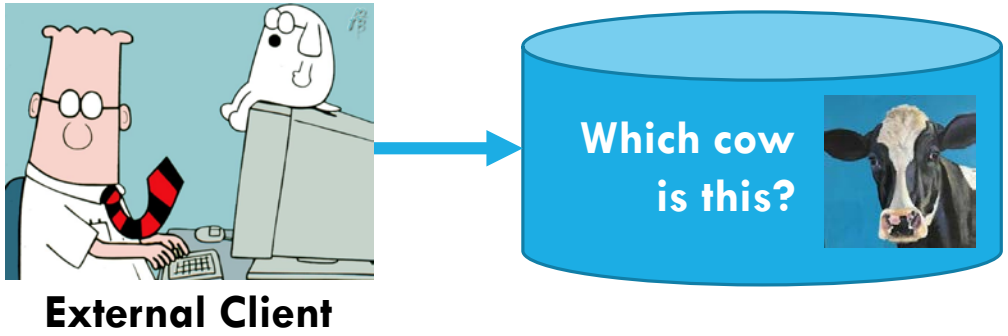
External Client



Although created using Derecho (a C++ library), Cascade runs on a set of nodes (machines or VMs) where it controls some resources (cores, RDMA interfaces, GPUs/FPGAs, memory).

Users can build applications that access Cascade from "outside". We call those "external clients". The put/get API would work, and RDMA is supported.

CASCADE IS AN EXTENSIBLE SERVICE



But this slide deck is mostly about users who *extend* the Cascade service by adding logic that runs inside of Cascade.

Cascade behaves as a customized service: a "smart" service.

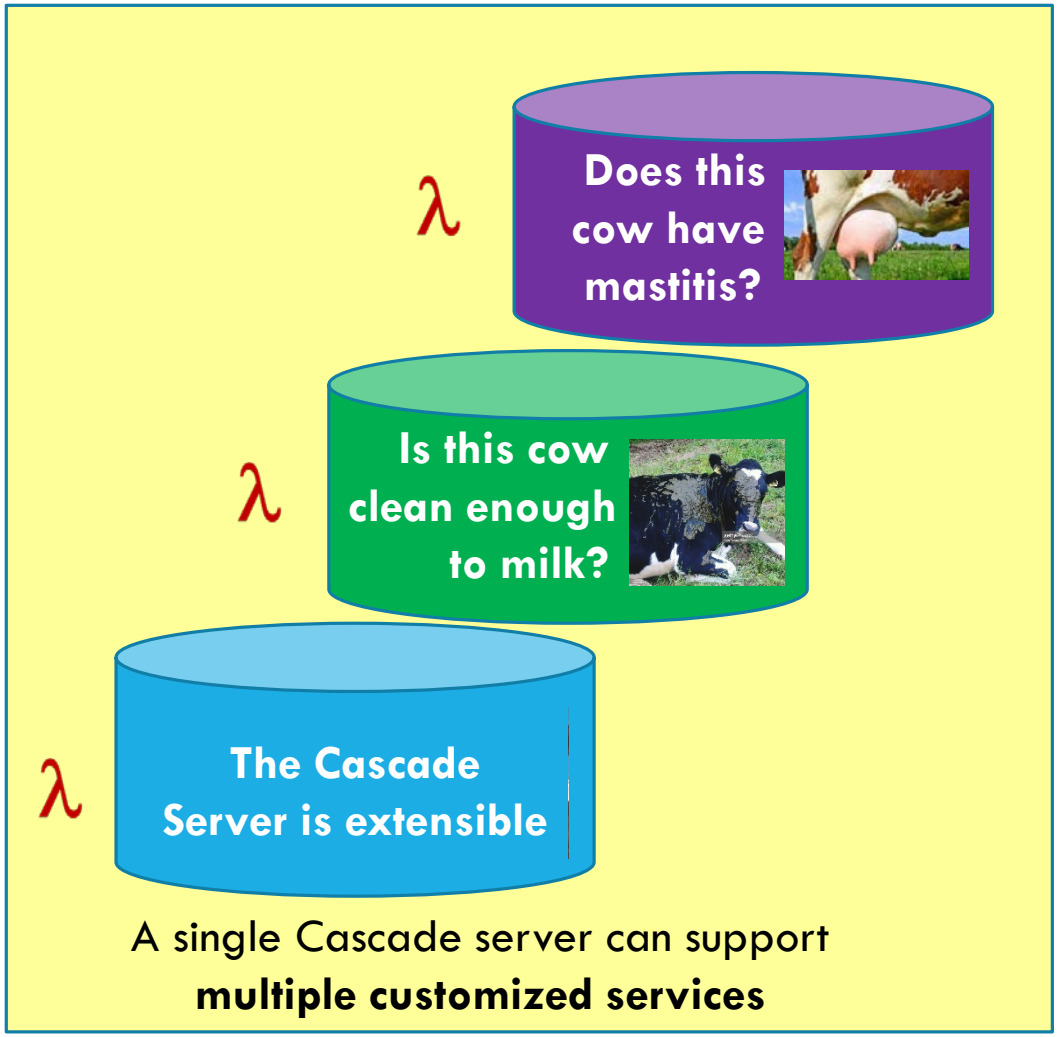
We will explain how this is done… it involves extra APIs, but allows us to also treat our service as a kind of library.

EXTENSION CONCEPT

Cascade is one service

But when you supply customization it acts like many specialized services, one per application

So it becomes a platform for new microservices, like these!



KEY CONCEPT: EXTENSIBILITY

What does it mean to be an "extensible platform as a service"?

Think about devices you plug into your computer. For example, Ken uses a Garmin GPS to track bike rides. When he plugs the Garmin into his Dell computer, it becomes an expert on all the bike rides he has taken since 2008.

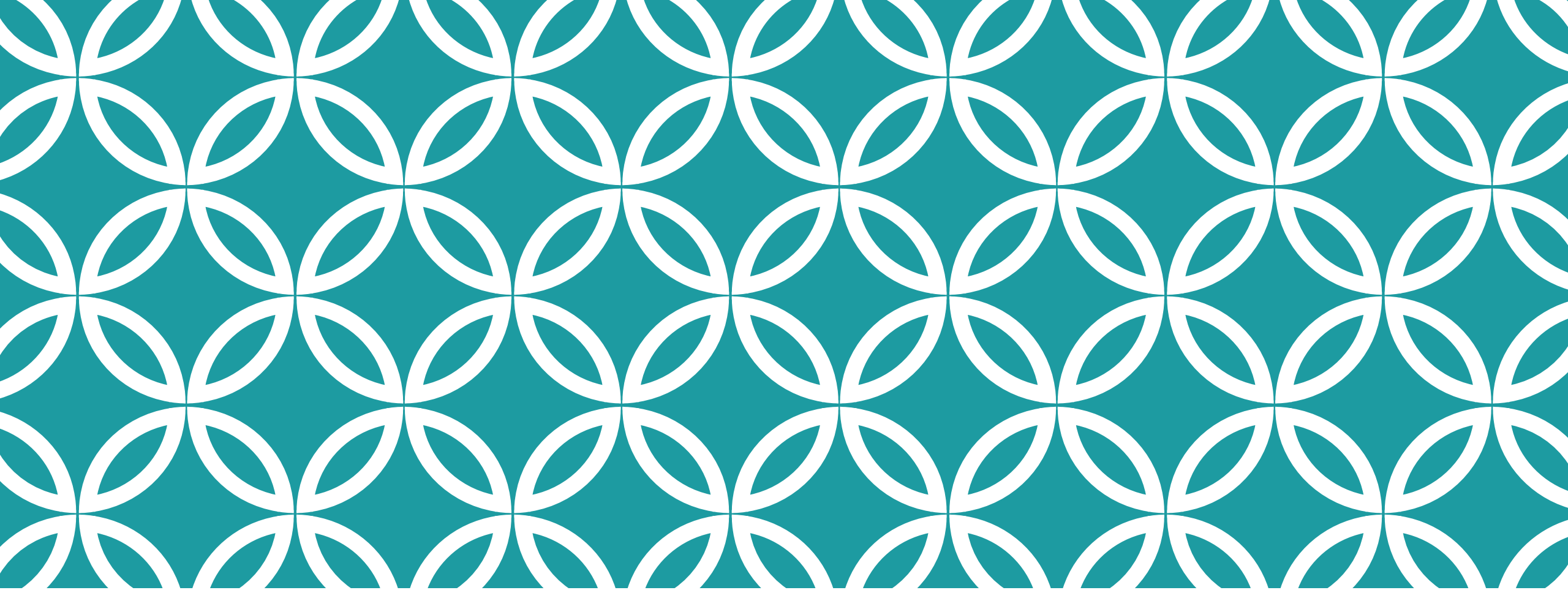
In some sense, the Garmin+Dell pair is a "new thing"

HOW DOES A PERSON EXTEND CASCADE?

We will look at this later in the class, but the idea is to plug in new software written in C++, and to tell Cascade when this software should run.

The new code watches for IoT events specific to the application, by monitoring keys. Updates will trigger the λ to run.

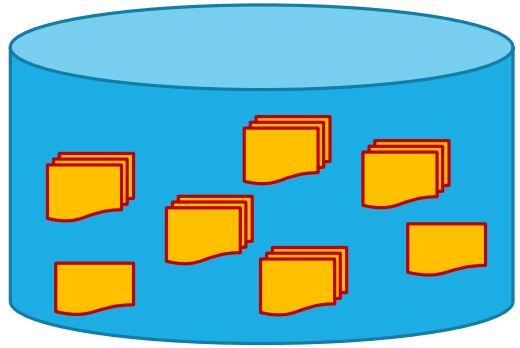
For example, your plug-in could handle "cow-washing events"



THE CASCADE STORAGE MODEL

How should Cascade data be managed?

FILE SYSTEM "EXTENSION"



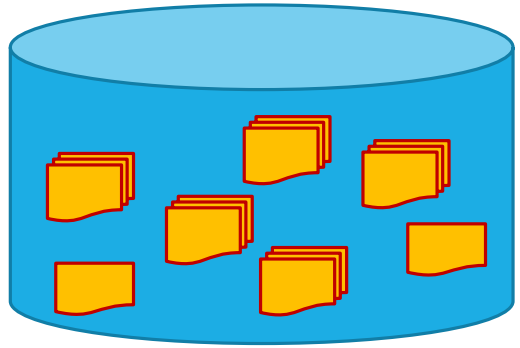
User sees scalable, "private" object pools

Cascade has a built-in file system extension:

- Every object has a pathname.
- The file system extension supports normal file operations.
- You can access it just like any file system.

Yet Cascade isn't a file system. It is a key-value **get**/**put**/**watch** store. Moreover, it is automatically sharded for scalability.

UPDATES: CREATE A NEW FILE OR APPENDING TO AN EXISTING FILE



User sees scalable, "private" object pools

Any key-value object lives in one shard (but that same shard may have many keys that map to it!).

- A *key* is a string. A *value* is an object serialized as a byte vector.
- Updates are log appends using Paxos. Each object has a log of versions that evolved over time.
- Queries run on the *stable prefix* of the log.

VERSIONED UPDATES

Each time you write to an object, Cascade creates a new version, with a unique (incrementing) version number.

If you read an object, modify it, and then write it back, you can tell Cascade which version you modified. The **put** will double check to be sure that this is still the current version before replacing it, and otherwise returns an error (then you can loop)

VERSIONED/TEMPORAL QUERIES

Accessed via **get.**

In the volatile case, Cascade only keeps the most recent version. With persistent objects, Cascade keeps a log of past versions.

- By default, applications see the most current version
- Indexed access allows the application to query any version (by version number or time), or fetch any data range.

VERSIONED OBJECTS

We configure the object store to track versions. **put** creates a new version:

- *key:* The object store *always* tracks information on a per-object basis
- *version-number:* Just an integer
- *time:* If the object itself lacks a timestamp, we just use "platform" time.

Now **get** can lookup most current version, or a specific one, even by time.

The object store is optimized to leverage non-volatile memory hardware.

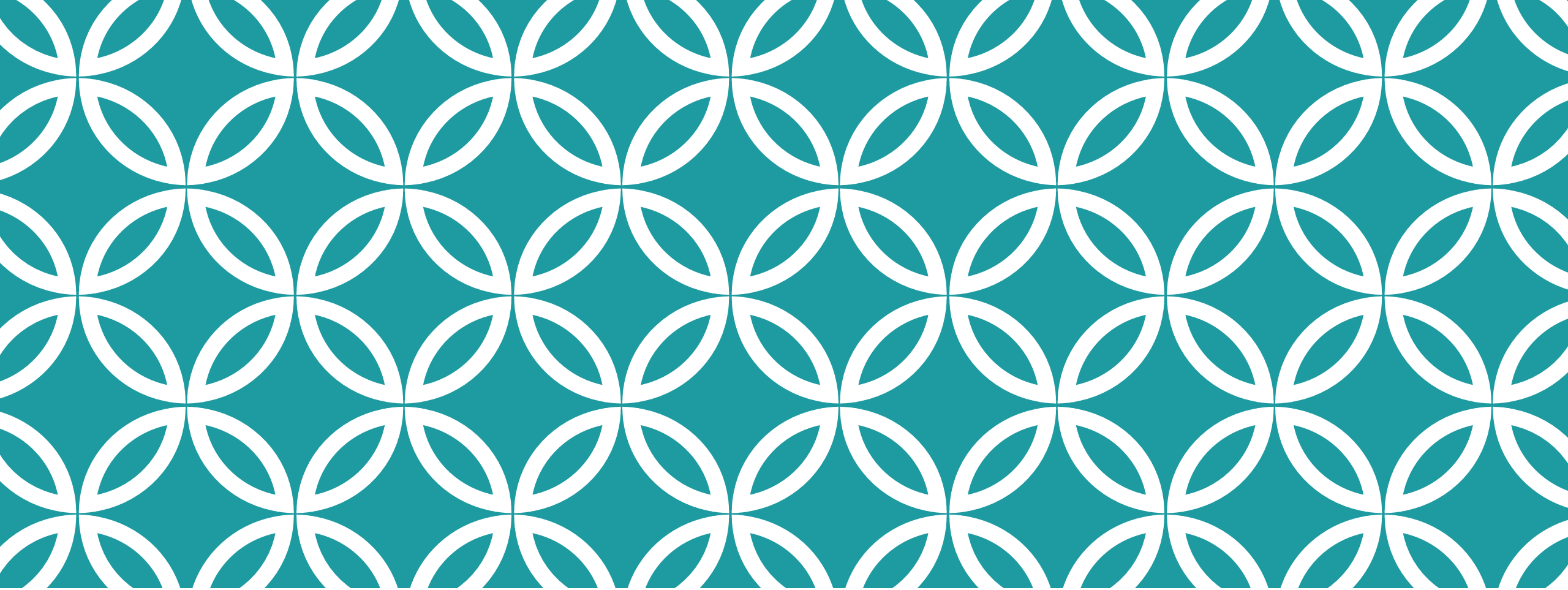
STORING DELTAS

Existing DHTs lack support for versioned data.

We implemented a highly optimized versioned data structure

We implement a temporal index, and cache frequently accessed data.

- A server still manages a map (since many keys map to it), but you can think of the values for a specific key as being versioned.
- Sometimes deltas are more efficient. If you have a function to compute the delta, we won't even create a new version unless you tell us to.
- Values (or deltas) are saved on NVMe & replicated for fault-tolerance.



THE CASCADE COMPUTE MODEL

Lambdas, coded in your favorite language

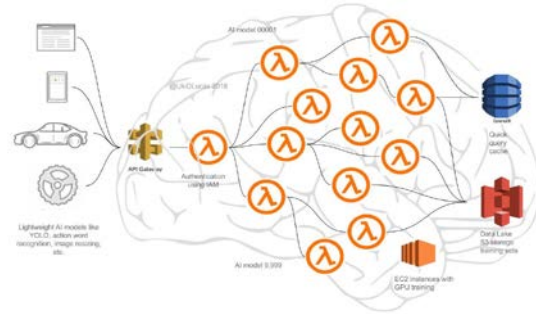
THE CENTRAL PUZZLE

The very fastest data paths require compilation, ideally in languages like C++.

But we want Cascade to run as a *service*, so it would often already be running when a new user comes along and wishes to create and launch some completely new service.

How can we extend a running system? Actually… it isn't so hard

FIRST QUESTION: WHAT'S IN A λ?



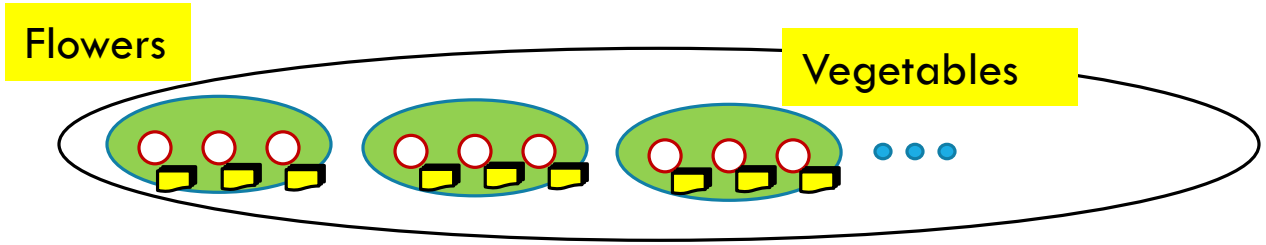
We support many languages. Native APIs are Python with various packages (including LINQ) and C++ with LINQ.

Code is concise – LINQ pioneered a style that mixes "kernel" invocations with embedded SQL. Maps cleanly to GPU, FPGAs.

Cascade manages GPUs and can cache data in GPU memory.

HOW CAN A KEY-VALUE STORE "BE" A CLASSIFIER SERVICE OR AN ANALYTIC SERVICE?

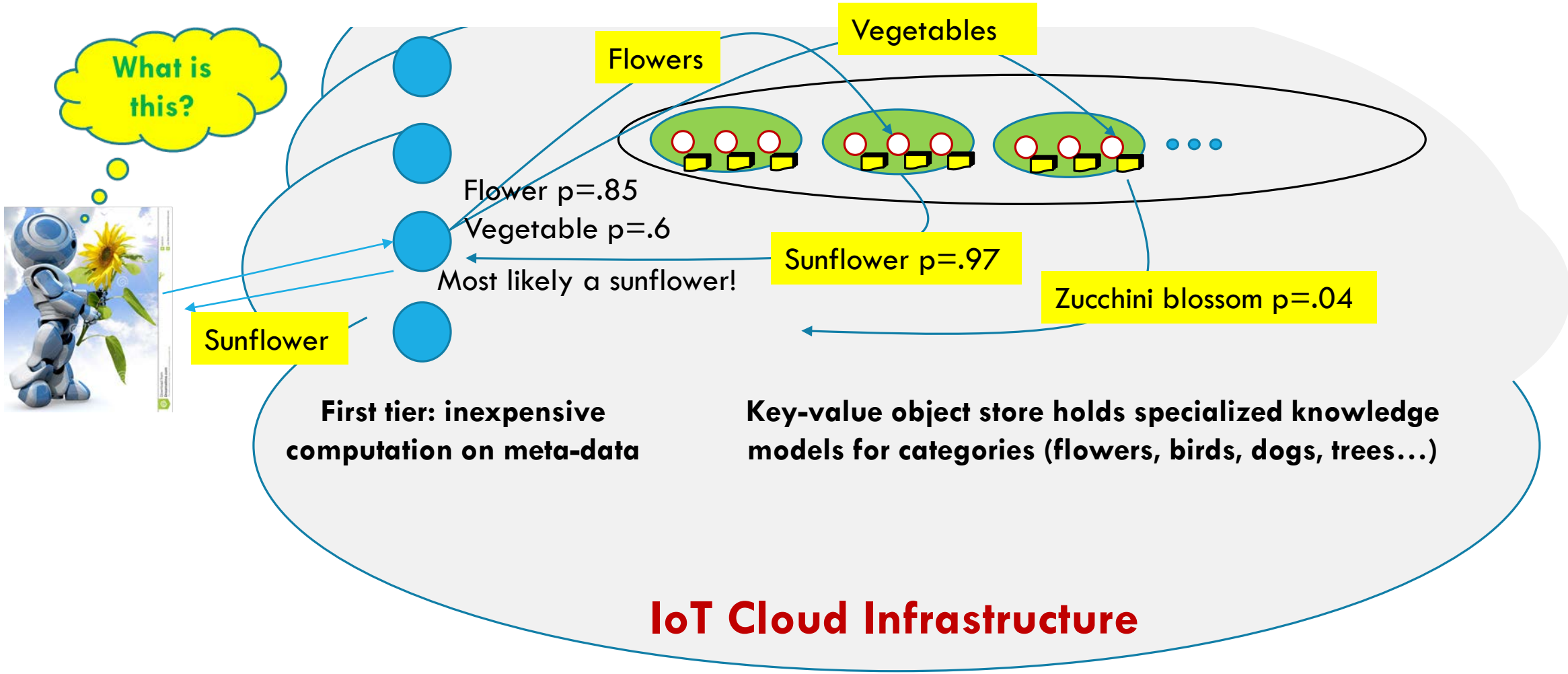
We run Cascade on a set of nodes. Here we see nine nodes in three shards.



A shard identically replicates (key,value) tuples, using Paxos.

Here, an object with the key "Flowers" was stored in shard 0. "Vegetables" ended up in shard 2.

… ENABLING THIS KIND OF SOLUTION



HOW DOES "WATCH" WORK?

On a given Cascade server node, it will issue an upcall to user-specified code if the key(s) the user wants to watch change.

Cascade's name space is best understood as a global file system namespace. The keys are the file pathnames.

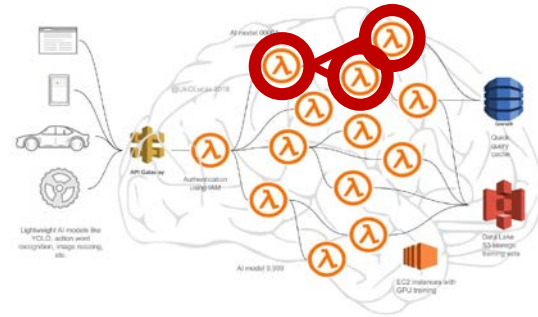
Watch thus is monitoring a file or directory…

SO… A λ IS JUST A PROGRAM DESIGNED FOR PARALLEL EXECUTION INSIDE A KEY-VALUE STORE

Our idea:

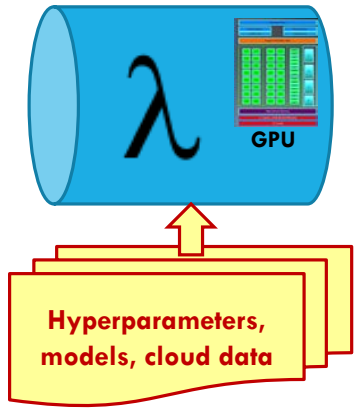
- Cascade hosts the key-value data (or file system, like Ceph)
- The user's code is treated like a dynamically linked library.
- The user creates this DLL, saves it into Cascade, then tells us where to run it. Cascade loads and launches it there.
- DLLs have zero overhead, once loaded. So now the user's logic is efficiently callable from Cascade!
- And we use the **watch** feature to initiate those upcalls!

CREATING AND INSTALLING A NEW λ ON SOME CASCADE NODES



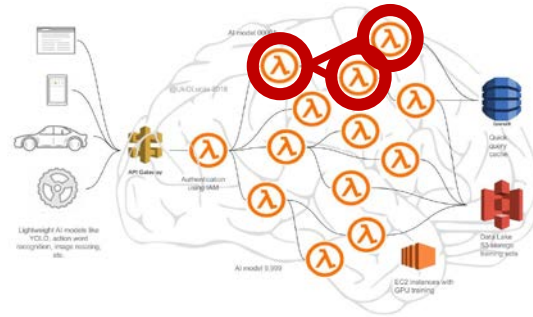
Developer builds a new ML program, designed for parallel execution directly "in" a key-value store.

Cascade is already running in the cloud. She tells Cascade to load this DLL.



On the designated compute nodes, Cascade loads the DLL and activates it. The DLL initializes itself and register some "watch" upcalls.

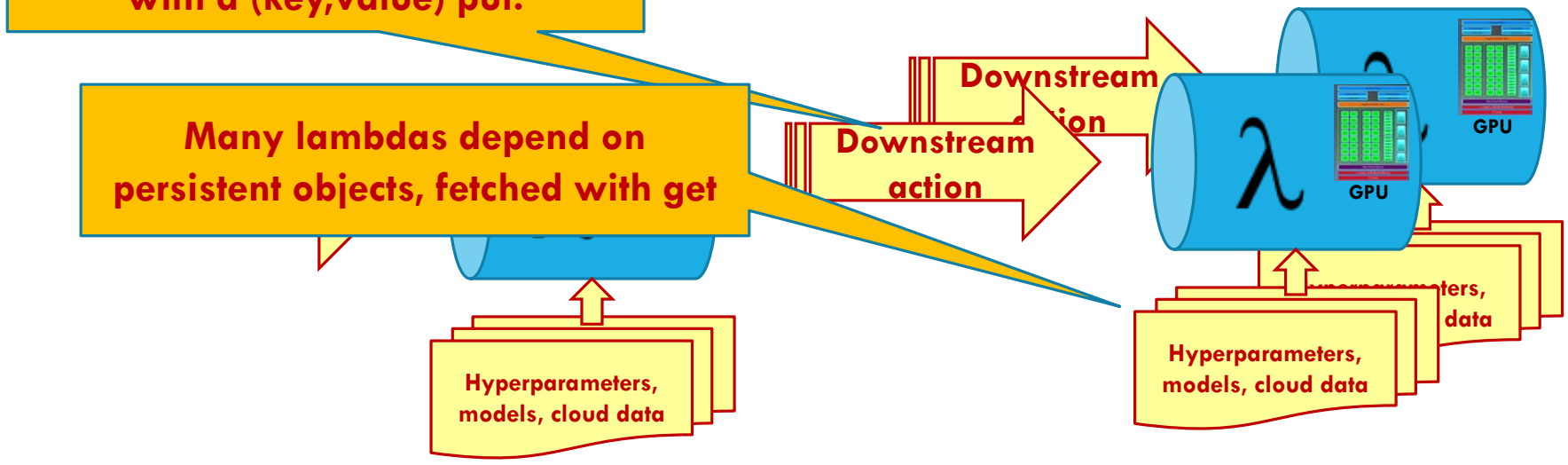
Recall that a key-value store is sharded. Each node will host a different set of keys.



D-AI PIPELINES WILL BE COMMON

One way to trigger a lambda is with a (key,value) put.

Many lambdas depend on persistent objects, fetched with get



Each of these lambda stages potentially runs on a pool of machines.

ANYTHING, ANYWHERE, ANY TIME

With permissions, any code can access any object, or the associated time-series if the object is a persisted history.

Obviously, performance is best if we can minimize data movement and compute instantly when new events occur.

This all yields a sophisticated programming model

CASCADE λs EXECUTE IN CONSISTENT CUTS

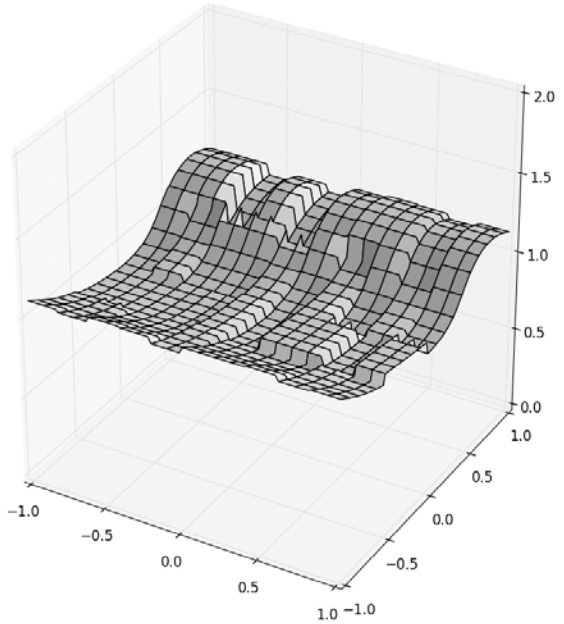
Recall: Cascade has a built-in temporal indexing feature. Suppose our distributed AI is triggered by event ε at time τ.

We run all the lambdas triggered by ε along a consistent cut "optimally close" to time τ (and selected deterministically).

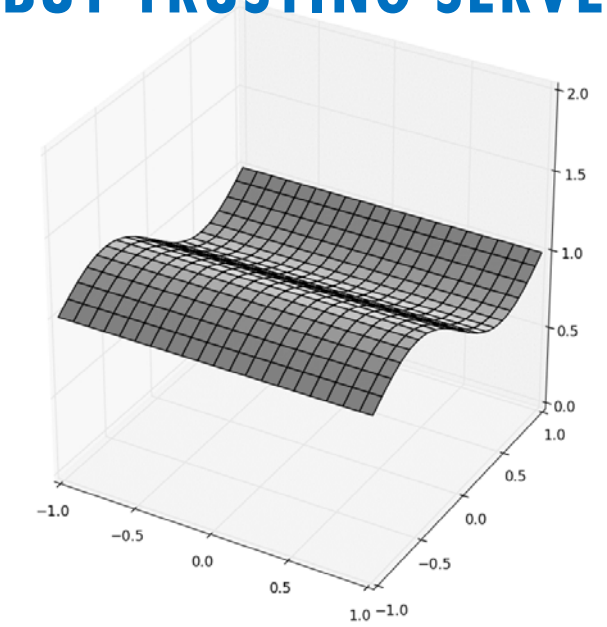
Effect: The lambda won't see platform-induced inconsistencies.

VISUALIZATION OF CASCADE CONSISTENCY

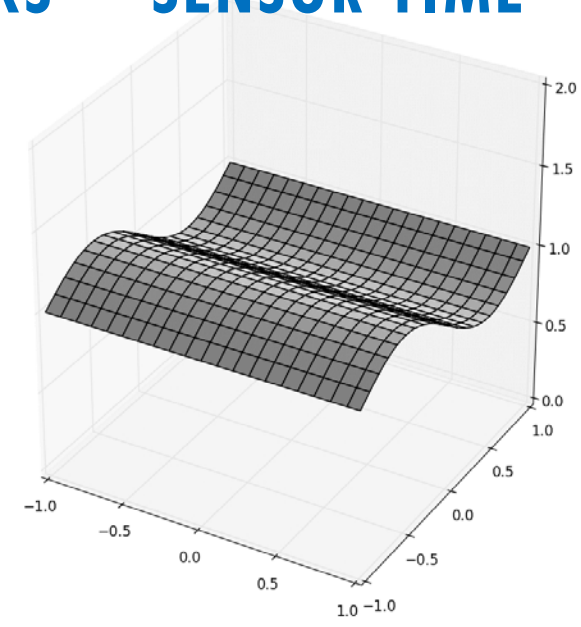
HDFS



CASCADE USING CONSISTENT CUTS BUT TRUSTING SERVER CLOCKS



CASCADE WITH SENSOR TIME



Cascade consistent cuts + GPS-timestamped sensor data result in clean input to the D-AI algorithm (in this case, a simple visualization)

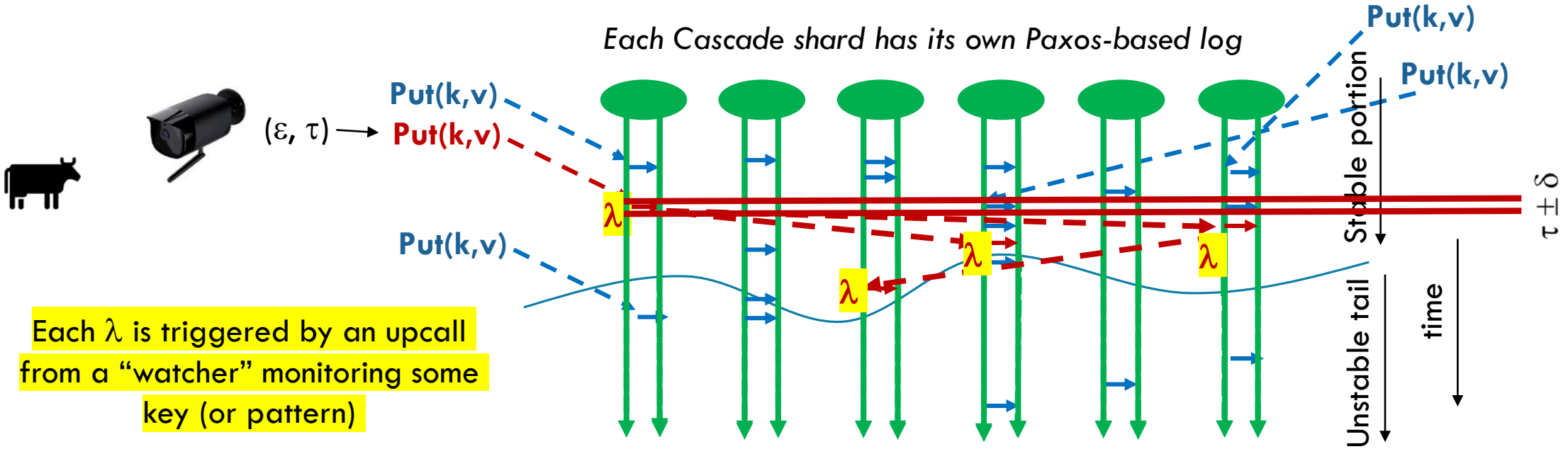
CENTRAL CONCEPTUAL INSIGHT

One event may trigger many lambdas.

These lambdas may need to run on multiple nodes… yet will share the same temporal index (τ from the trigger event ε).

A Cascade query always sees a "consistent state snapshot."

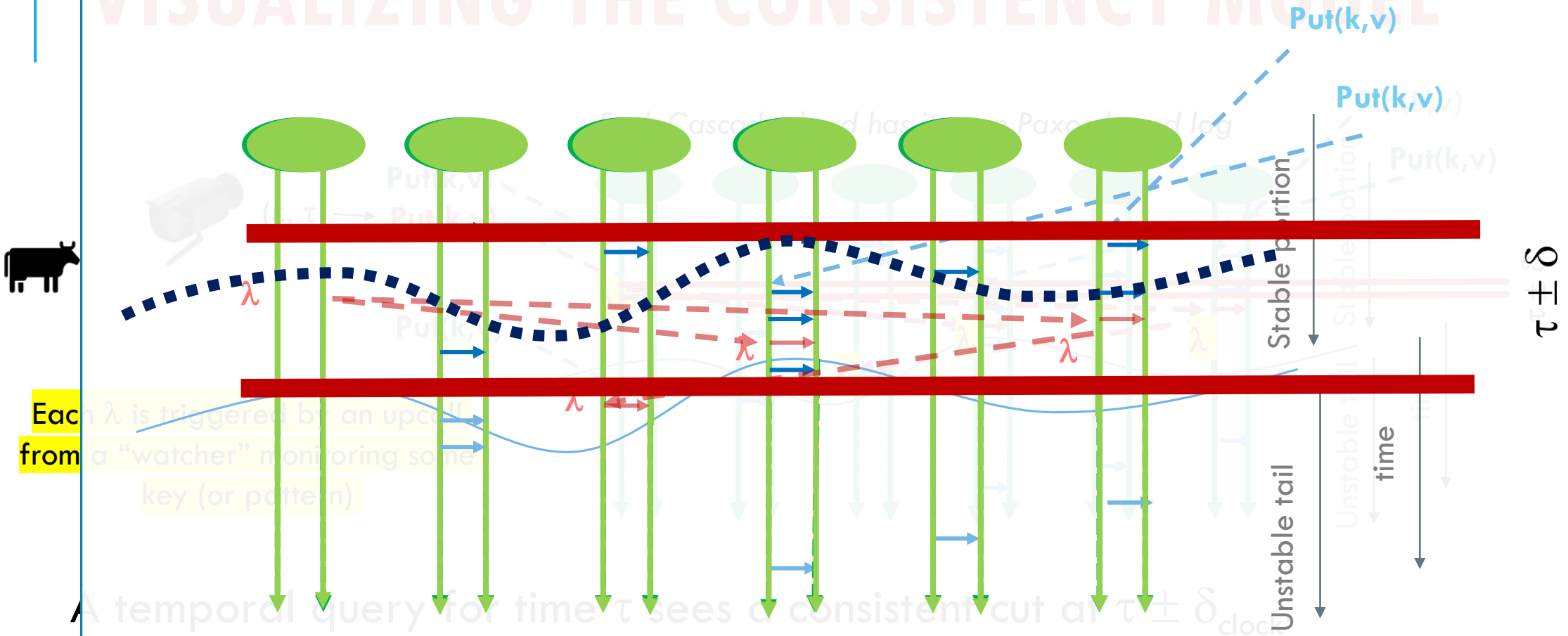
VISUALIZING THE CONSISTENCY MODEL



A temporal query for time τ sees a consistent cut at $\tau \pm \delta_{clock}$.

Queries to unstable data must wait, but updates are stable within 50us.

Even if δ is small, there could be a few events at each process in the $t \pm \delta$ time window. By selecting a consistent cut, Cascade avoids the "mashup" issue we saw in the HDFS animation.

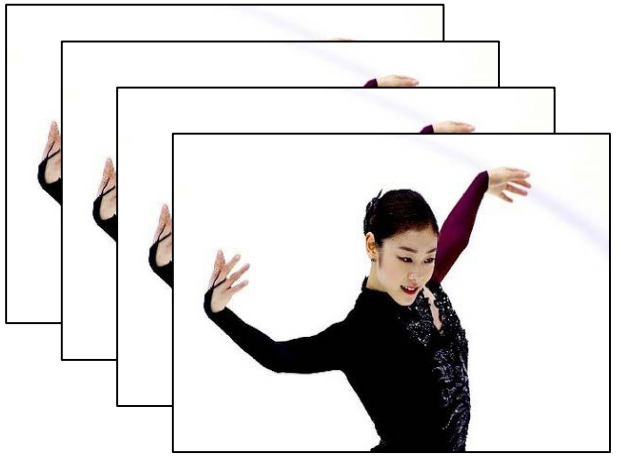


Each λ is triggered by an update from a "watcher" monitoring some key (or pattern)

A temporal query for time τ sees a consistent cut at $\tau \pm \delta_{clock}$

Queries to unstable data must wait, but updates are stable within 50us.

BUT THE JOB IS STILL HARD!

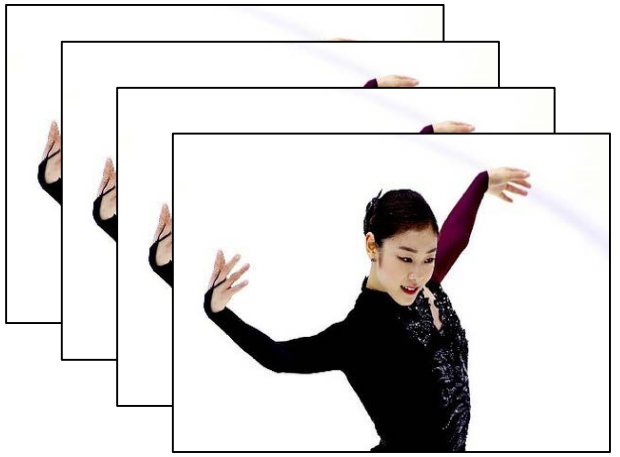


Cascade makes it easy to extract a *tensor* with a temporal dimension: A stack of frames

But writing ML code that can recognize patterns over time is not easy! Even identifying movement trajectories is a hard vision task.

Cascade is just the platform. You need to write the application!

COOL OBJECT ORIENTED IDEA



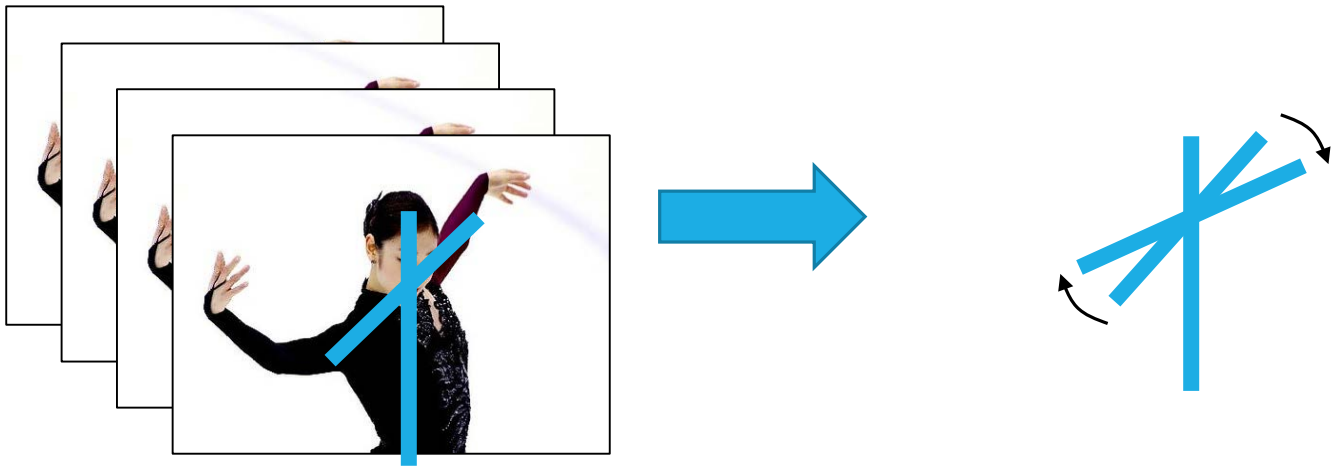
Suppose that you have a device that captures images of size w,h and you want to form a 3-D tensor for times t0 .. t1.

You can define your sensor as an object with *getter* methods that turn around and fetch the appropriate images. Now your code is written in terms of my_tensor[x,y,t] and yet Cascade handles fetching and caching the data.

You can even use LINQ to do this in one line of Python or C++ code (we will see this later in the semester)

COOL OBJECT ORIENTED IDEA

Now, if you have a computer vision algorithm that can recognize the orientation of the skater frame by frame, you can write a function that will "represent" the pattern of how her body is spinning over time.



"MACHINE LEARNING" A SPIN

In this model, we can think of a trajectory as the "motion trace" of some key points such as the skater's hands, arms, face, etc.

Each traces out a path in time and space.

In effect, our system is learning a collection of high dimensional splines that fit the observed data, and can be used to predict future movement

REALISTIC "USE CASES?" FOR SKATING

A skating judge might be interested in measured properties of the spin, like speed, number of spins, steadiness.

A coach might be trying to diagnose the root cause of a small wobble.

The skater may be wondering what would have happened if her left hand was just a tiny bit higher

HOW WOULD YOUR CODE WORK?

First retrieve a tensor: one axis for time, and then 3 spatial axes.

Now you can write code that finds the best match layer by layer relative to the prior layer. That tells you the trajectory of her hand movements.

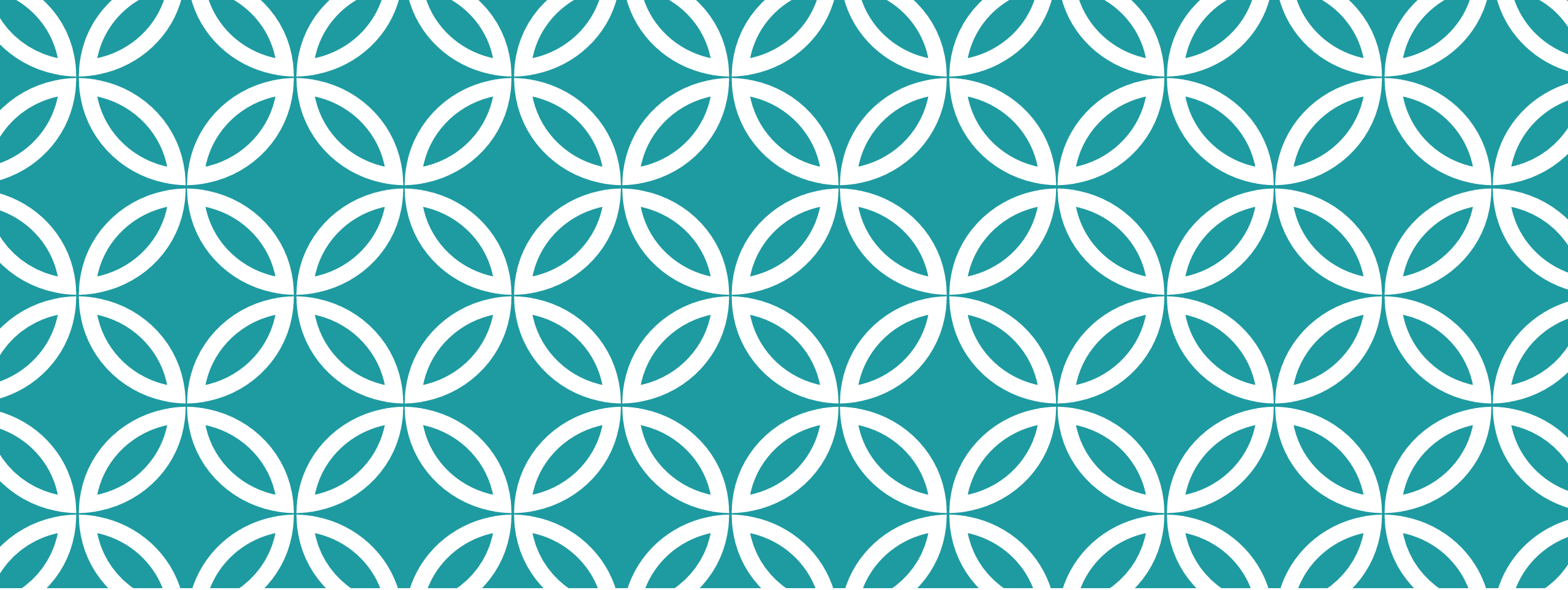
Last, you might apply a model to this and highlight small errors.

GROUPING OBJECTS

Often a lambda will need to access several objects that should ideally all have their own keys, yet you want them grouped on the same shard.

For this, Cascade supports "affinity grouping". Each object has a second key, used for placement.

Even if A and B have different keys – "names" – they will be stored on the same shard if you assign them the same affinity key.



HOW TO CREATE A NEW λ

Cascade is an *extensible* service!

TRIGGERED ACTIONS: THE CODE ITSELF

The lambda is created as code that implements a static API.

- User places the DLL (or the source file) in a Cascade object
- A command tells Cascade which nodes should load it.
- The DLL has an **initialize** method. Cascade calls it.

Notice that the user-supplied code can define new object types. We only require that all Cascade objects be byte-serializable.

TRIGGERED ACTIONS: API IS SIMPLE

Cascade implements has three primary APIs

- The main Cascade APIs: (key,value) **put** and **get.**
- **watch** upcalls to trigger user logic when some key is updated.

Watch can do exact key match, or path-prefix match, or arbitrary regular-expression matches.

TRIGGERED ACTIONS: THE CODE ITSELF

The **initialize** method calls **watch** to register the user's lambdas

- A lambda is a *closure*. This associates "context" with the lambda.
- Example: keys identifying hyperparameter and model objects.

Watch monitors for keys that match.

If a match occurs, Cascade passes the matching key to the lambda.

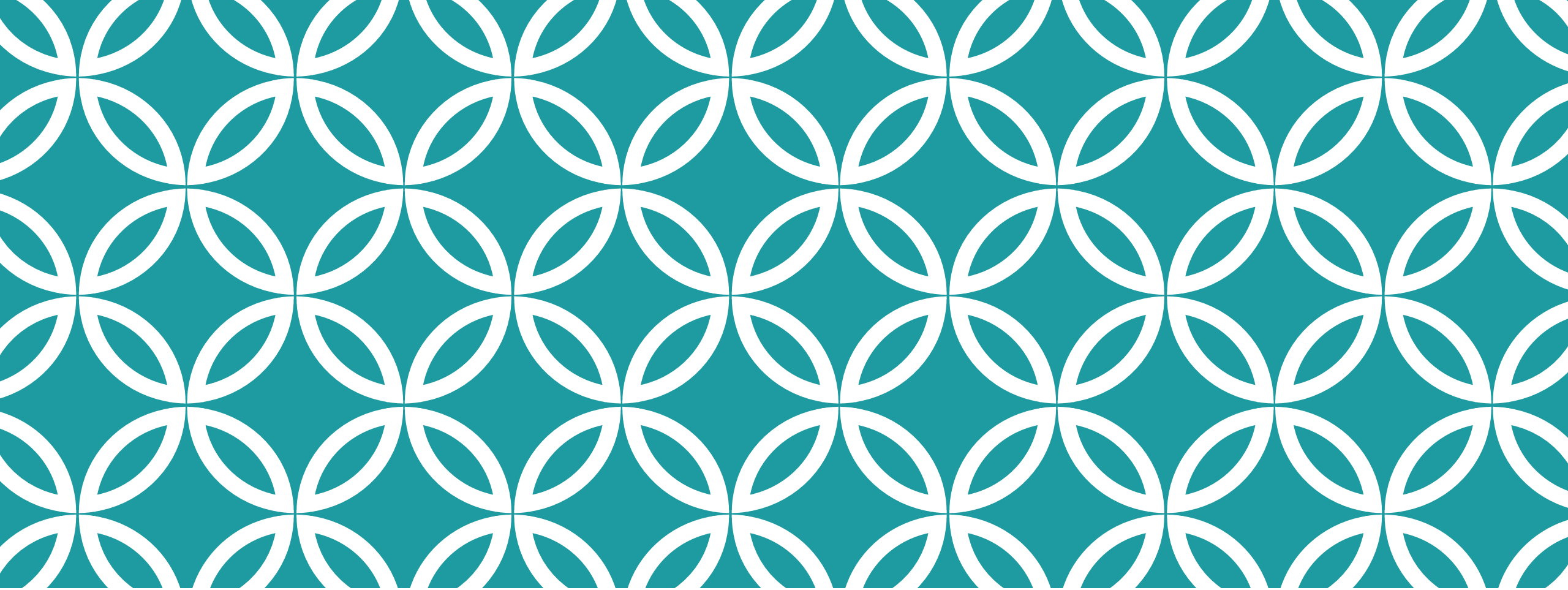
PATTERN MATCHING

Just like with file names (key == file name)

/data-center/instance/users/Alicia/SmartDairy/ImageRec/cow2716/model

Many watch patterns will seek exact match.

Some are a prefix followed by a glob-style pattern, like “trigger if any change occurs in this folder”



WHAT DOES A REAL APPLICATION LOOK LIKE TODAY?

Example, courtesy of Weijia, Alicia and Thompson

DAIRY IMAGE PIPELINE: FRONT END

Dairy Farm

The Farm Server (IoT Edge)

Frame Extractor

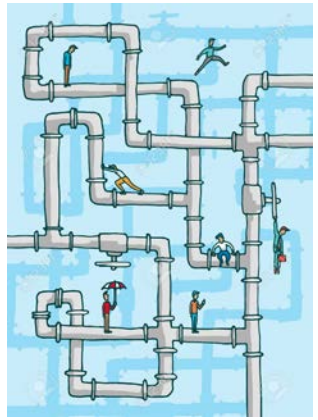
Frame Sampler

Video clip store

Frame Server

Data Center

Image Pipeline Front End (As an external client)



Cascade Image pipeline

WAN

… DETAILED VERSION (PyLINQ ON MSFT AZURE)

Integrate daily data

Date	cow_id	daily_yield	daily_fat	…	daily_protein
12/3/20	1	14	3.96	…	2.89
…	…	…	…	…	…
1/10/21	237	20	4.42	…	4.55

<field>/<cow_id>{<ts(ver)>}
daily_protein/cow_id1{ver_1} = 2.89

Download blobs from Azure &
Store to Cascade VCSS subgroup

```
client = cascade_py.ServiceClientAPI()
daily_yield_pool = client.get_object_pool("VolatileCascadeStore","daily_yield")
daily_yield_pool.put(cow_id,val)
```

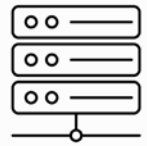
Upload daily date to Azure Blob Storage

Date	cow_id	…
12/3/20	1	…
…	…	…
12/3/20	237	…

daily_fat/cow_id237{ver_38} = 4.42

Farm server

Filtered image frames



External Client to Cascade

Store to subgroup VCSS
Trigger image analysis

Action = black_cow_infer

```
white_cow_net = load_cow_model( \
    "cow_symbol.json", \
    'white_cow.params')
black_cow_net = load_cow_model( \
    "cow_symbol.json", \
    'black_cow.params')
...
action = cascade_context.get_next_action()
if action.type == "white_cow":
    pred = white_cow_net(action.frame)
else
    pred = black_cow_net(action.frame)
```

Streaming image frames through TCP portal

Probability of calving
in next 8h is: 80%

ML Model
birth prediction

CV model
Image analysis

cow id: 127

LINQ query to retrieve data of most recent 10 days from Cascade about cow 128

```
client = cascade_py.ServiceClientAPI()
cow_id = "cow_id128"
recent_data = []
for field in ["daily_yeild", "daily_fat", "daily_protein"]:
    field_pool = client.get_object_pool("VolatileCascadeStore",field)
    query_result = field_pool \
                   .where(lambda x:x.endswith("/"+cow_id)) \
                   .order_by(lambda x:x.version) \
                   .reverse()
                   .take(10)
    recent_data.append(query_result)
```

C++ IS SIMILAR (BUT MORE EFFICIENT)

Integrate daily data

Date	cow_id	daily_yield	daily_fat	…	daily_protein
12/3/20	1	14	3.96	…	2.89
…	…	…	…	…	…
1/10/21	237	20	4.42	…	4.55

<field>/<cow_id>{<ts(ver)>}
daily_protein/cow_id1{ver_1} = 2.89

Download blobs from Azure &
Store to Cascade VCSS subgroup

```
ServiceClientAPI capi;
uint32_t num_subgroups = capi.template<SubgroupType> get_number_of_subgroups();
auto sg_idx = std::hash<std::string>{} (daily_yield) % num_subgroups;
uint32_t num_shards = capi.template<SubgroupType> get_number_of_shards(sg_idx);
capi.template put<SubgroupType>(val,sg_idx,cow_id % num_shards);
```

Upload daily date to Azure Blob Storage

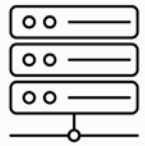
Date	cow_id	…
12/3/20	1	…
…	…	…
12/3/20	237	…

daily_fat/cow_id237{ver_38} = 4.42

Cascade backend

Farm server

Filtered image frames



External Client to Cascade

Store to subgroup VCSS
Trigger image analysis

Action = black_cow_infer

```
InferenceEngine white_cow_ie;
InferenceEngine black_cow_ie;
auto frame = dynamic_cast<ImageFrame*>(action.action_data.get());
if (action.action_type == WHITE_COW) {
    std::tie(name,soft_max) = white_cow_ie.infer(*frame);
} else {
    std::tie(name,soft_max) = black_cow_ie.infer(*frame);
}
PCSS::ObjectType obj(frame->key,name.c_str(),name.size());
```

Streaming image frames through TCP portal

Probability of calving
in next 8h is: 80%

ML Model
birth prediction

CV model
Image analysis

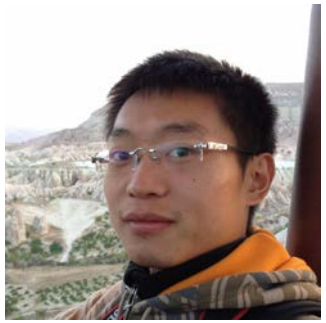
cow id: 127

LINQ query to retrieve data of most recent 10 days from Cascade about cow

```
std::vector<std::string> fields{"daily_yield", "daily_fat", "daily_protein"};
std::vector<std::vector<double>> recent_data;
for (std::string& field: fields)
{
    auto sg_idx = std::hash<std::string>{} (field) % num_subgroups;
    auto sh_idx = 128 % num_shards;
    std::string key_quering(field + "/cow_id128");
    auto query_result = from_shard<VCSS,ServiceClientAPI>(keys,capi,sg_idx,sh_idx,-1)
                        .where([](typename SubgroupType::ObjectType obj) { return !obj.key.compare(key_quering); })
                        .orderBy([](typename SubgroupType::ObjectType obj) { return obj.version; })
                        .reverse()
                        .take(10)
                        .toStdVector();
    recent_data.emplace_back(query_result);
}
```

THE CASCADE AND DERECHO TEAM

Weijia Song



Alicia Yang

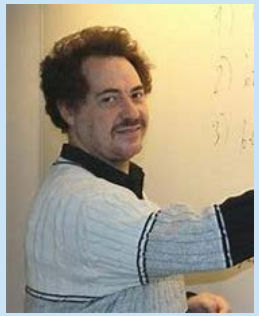
Ken Birman

Sagar Jha

Lorenzo Rosa

Mae Milano
(post-doc at Berkeley)

Andrea Merlina



Roman Vitenberg

Cornell undergrads:
Aahil Awatramani, Ben Posnick, Max Charlamb, Archishman Sravankumar, Aaron Weiss, Peter Zheng

Thompson Liu

Edward Tremel
(faculty at Augusta Univ.)