A Guided Tour of Ray Core

Paco Nathan @pacoid

derwen.ai

$(whoami)

- AI in grad school during mid-1980s
- 7 years R&D in *neural networks* (1980s-90s)
- *network engineering* through the **AI Winter**
- “guinea pig” for AWS (2006-ff)
- led early large **Hadoop instance on EC2**, which became a case study for EMR
- **Apache Spark** community evangelist (2014-15)
- lead committer: **kglab**, **pytextrank**
- current focus: **Graph-Based Data Science**

derwen.ai

part 1:
Ray core

Ray Project

Keynote: The Future of Ray - Robert Nishihara, Anyscale

Ray Project

A kind of **pattern language** for distributed systems,
as a library in Python, Java, C++ (upcoming):

- **task-parallel** – stateless, data independence
- **remote objects** – key/value store
- **actor pattern** – messages among classes, managing state
- **parallel iterators** – lazy, infinite sequences
- **multiprocessing.Pool** – drop-in replacement
- **joblib** – e.g., *scikit-learn* back-end
- Dask, Modin, Mars, etc.

https://ray.io/

Mix and match as needed, without tight coupling
to framework

Closures and Decorators

Ray makes use of *closures* and *decorators* in Python:

- **"Closures and Decorators in Python"**
- **PEP 318**
- **asyncio**

See also:

- **“Ray Design Patterns”** (WIP)
- **patterns.eecs.berkeley.edu**

Pattern: task-parallel

Remote Functions:

- a `@ray.remote` decorator on a function
- properties: *data independence*, *stateless*
- patterns: **Task Parallelism**, **Task Graph**

reference:

Patterns for Parallel Programming
Timothy G. Mattson, **Beverly A. Sanders**, **Berna L. Massingill**
Addison-Wesley (2004)

Pattern: task-parallel

However, by adding the `@ray.remote` decorator, a regular Python function becomes a Ray remote function:

```
@ray.remote
def my_function():
    return 1
```

To invoke this remote function, use the `remote` method. This will immediately return an object ref (a *future* in Python) and then create a task that will be executed on a worker process.

```
obj_ref = my_function.remote()
obj_ref
```

```
ObjectRef(df5a1a828c9685d3ffffffff0100000001000000)
```

The result can be retrieved with `ray.get`

```
ray.get(obj_ref)
```

```
1
```

Pattern: task-parallel

Tutorials:

colab.research.google.com/github/ray-project/tutorial/blob/master/exercises/colab01-03.ipynb

github.com/anyscale/academy/blob/master/ray-crash-course/01-Ray-Tasks.ipynb

Pattern: distributed object store

Remote Objects:

- shared-memory object store
- roughly, akin to parts of **Redis**
- “lives somewhere on the cluster”

reference:

https://en.wikipedia.org/wiki/Shared_memory

Pattern: distributed object store

To start, we'll put an object into the Ray object store...

```
y = 1
obj_ref = ray.put(y)
```

Then get the value of this object reference

```
ray.get(obj_ref)
```

```
1
```

You can also access the values of multiple object references in parallel:

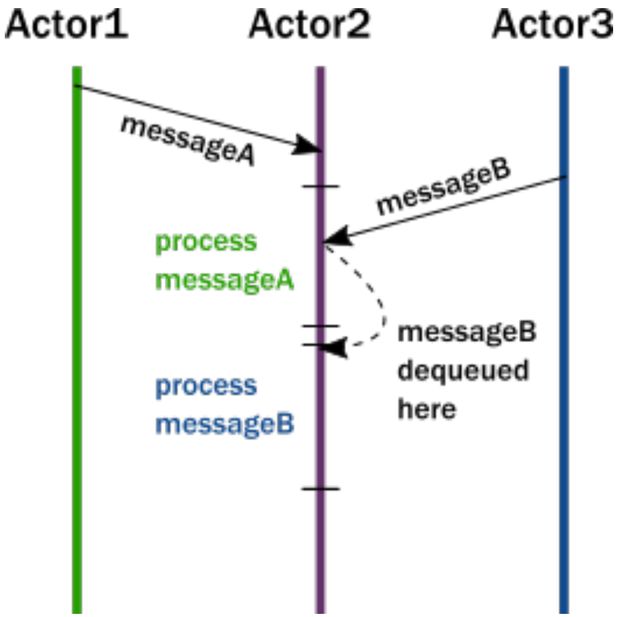
```
ray.get([ray.put(i) for i in range(3)])
```

```
[0, 1, 2]
```

Pattern: actors

Remote Classes:

- a `@ray.remote` decorator on a class
- properties: *stateful, message-passing semantics*
- pattern: **Actors**
- “lives somewhere on the cluster”



reference:

“A Universal Modular Actor Formalism for Artificial Intelligence”
Carl Hewitt, **Peter Bishop**, **Richard Steiger**
IJCAI (1973)
open: **https://www.ijcai.org/Proceedings/73/Papers/027B.pdf**

Pattern: actors

To start, we'll define a class and use the decorator:

```
@ray.remote
class Counter (object):
    def __init__ (self):
        self.value = 0

    def increment (self):
        self.value += 1
        return self.value
```

Now use this class `Counter` to create an actor:

```
counter = Counter.remote()
```

Then call the actor:

```
obj_ref = counter.increment.remote()
ray.get(obj_ref)
```

```
3
```

Pattern: actors

Tutorials:

colab.research.google.com/github/ray-project/tutorial/blob/master/exercises/colab04-05.ipynb

github.com/anyscale/academy/blob/master/ray-crash-course/02-Ray-Actors.ipynb

Pattern: distributed multiprocessing pool

Distributed multiprocessing.Pool:

- make Python programs distributed, using actors
- easy to scale existing applications that use `multiprocessing.Pool`

```
from ray.util.multiprocessing import Pool

def f(index):
    return index

pool = Pool()
for result in pool.map(f, range(100)):
    print(result)
```

Tutorial:

github.com/anyscale/academy/blob/master/ray-crash-course/04-Ray-Multiprocessing.ipynb

Pattern: joblib

Distributed scikit-learn:

- drop-in replacement to parallelize **JobLib** (scikit-learn backend)

Tutorial:

github.com/anyscale/academy/blob/master/ray-crash-course/04-Ray-Multiprocessing.ipynb

Pattern: joblib

```
import numpy as np
from sklearn.datasets import load_digits
from sklearn.model_selection import RandomizedSearchCV
from sklearn.svm import SVC

digits = load_digits()
param_space = {
    'C': np.logspace(-6, 6, 30),
    'gamma': np.logspace(-8, 8, 30),
    'tol': np.logspace(-4, -1, 30),
    'class_weight': [None, 'balanced'],
}

model = SVC(kernel='rbf')
search = RandomizedSearchCV(model, param_space, cv=5, n_iter=300, verbose=10)

import joblib
from ray.util.joblib import register_ray

register_ray()
with joblib.parallel_backend('ray'):
    search.fit(digits.data, digits.target)
```

Pattern: sharded, lazy iterators

Parallel Iterators:

- API for simple data ingest and processing
- fully serializable
- can operate over infinite sequences of items
- transformations based on method chaining
- passed to remote tasks and actors to represent data shards

Tutorial:

github.com/anyscale/academy/blob/master/ray-crash-course/05-Ray-Parallel-Iterators.ipynb

Pattern: sharded, lazy iterators

```
import ray
import numpy as np

ray.init()

@ray.remote
def train(data_shard):
    for batch in data_shard:
        print("train on", batch)  # perform model update with batch

it = (
    ray.util.iter.from_range(1000000, num_shards=4, repeat=True)
        .batch(1024)
        .for_each(np.array)
)

work = [train.remote(shard) for shard in it.shards()]
ray.get(work)
```

part 2:
“A Berkeley View”

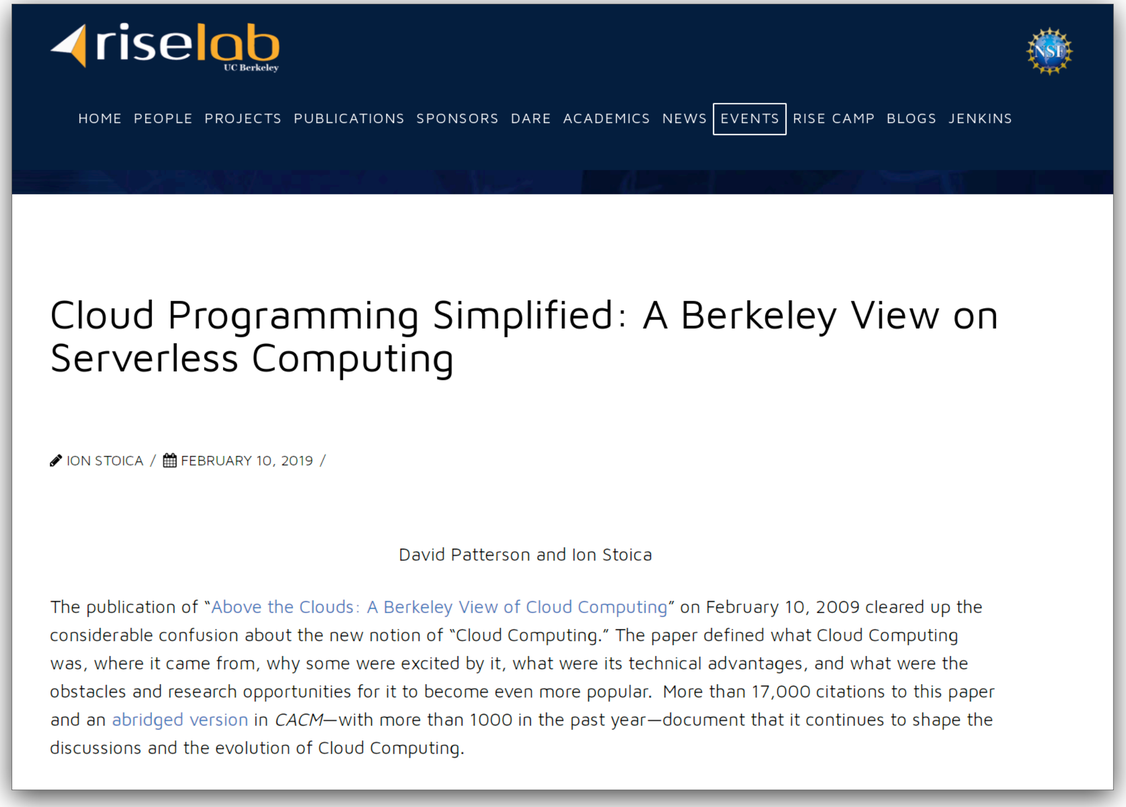
Formal definition of Cloud Computing

Professors **Ion Stoica** and **David Patterson** led EECS grad students to define cloud computing **formally** in 2009

"More than 17,000 citations to this paper…"

2019 follow-up:

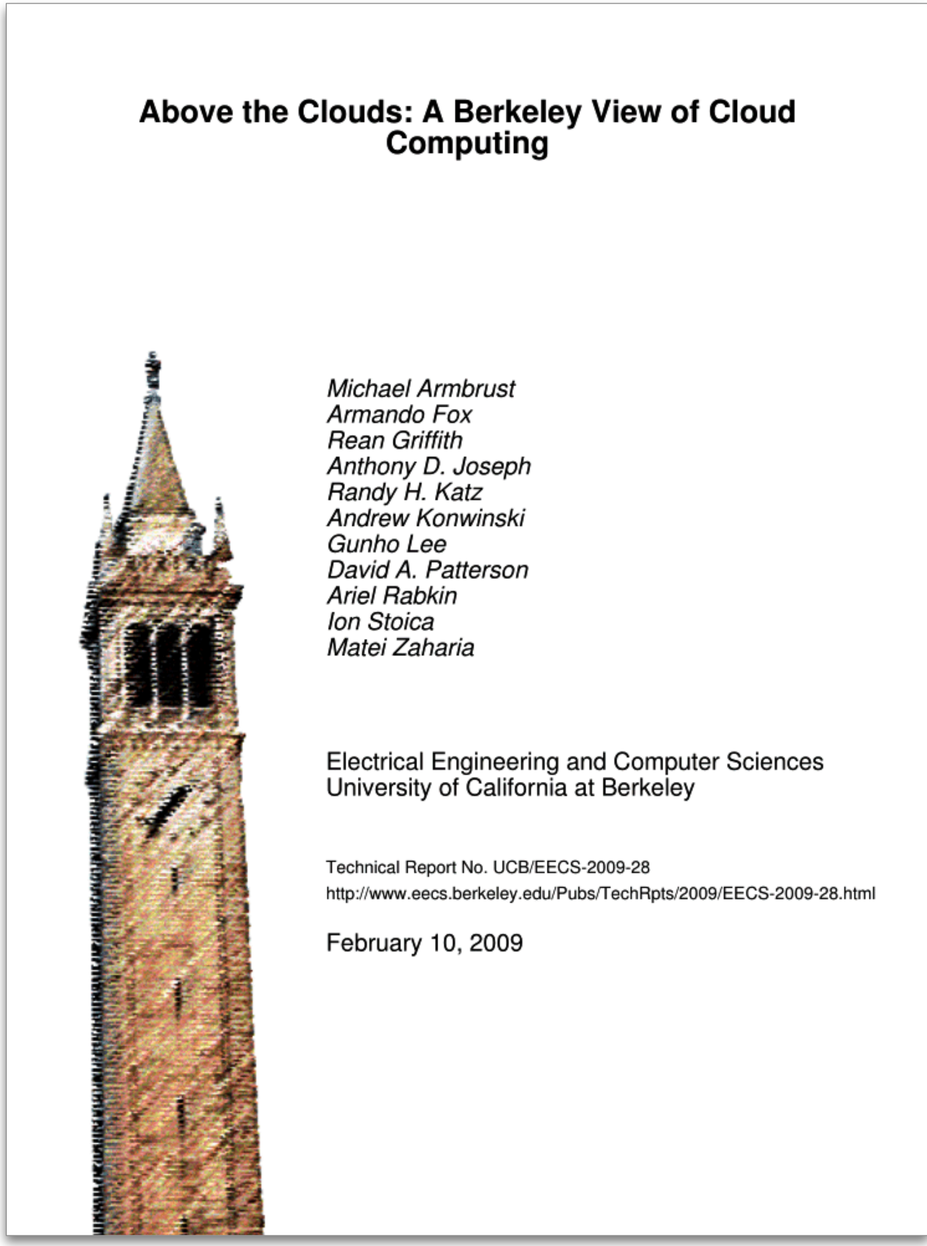
"We now predict … that *Serverless Computing* will grow to dominate the future of cloud computing in the next decade"



Evolution of cloud patterns

Initially, cloud services were simplified to make them more recognizable for IT staff accustomed to VMware

- Patterson, et al., developed industry research methodology and eventually also a **pattern language** to describe distributed systems
- **AMPlab** foresaw how cloud use cases would progress in industry over the next decade, which greatly informed **Apache Spark**, etc.



Above the Clouds: A Berkeley View of Cloud Computing

Michael Armbrust
Armando Fox
Rean Griffith
Anthony D. Joseph
Randy H. Katz
Andrew Konwinski
Gunho Lee
David A. Patterson
Ariel Rabkin
Ion Stoica
Matei Zaharia

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2009-28
http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html

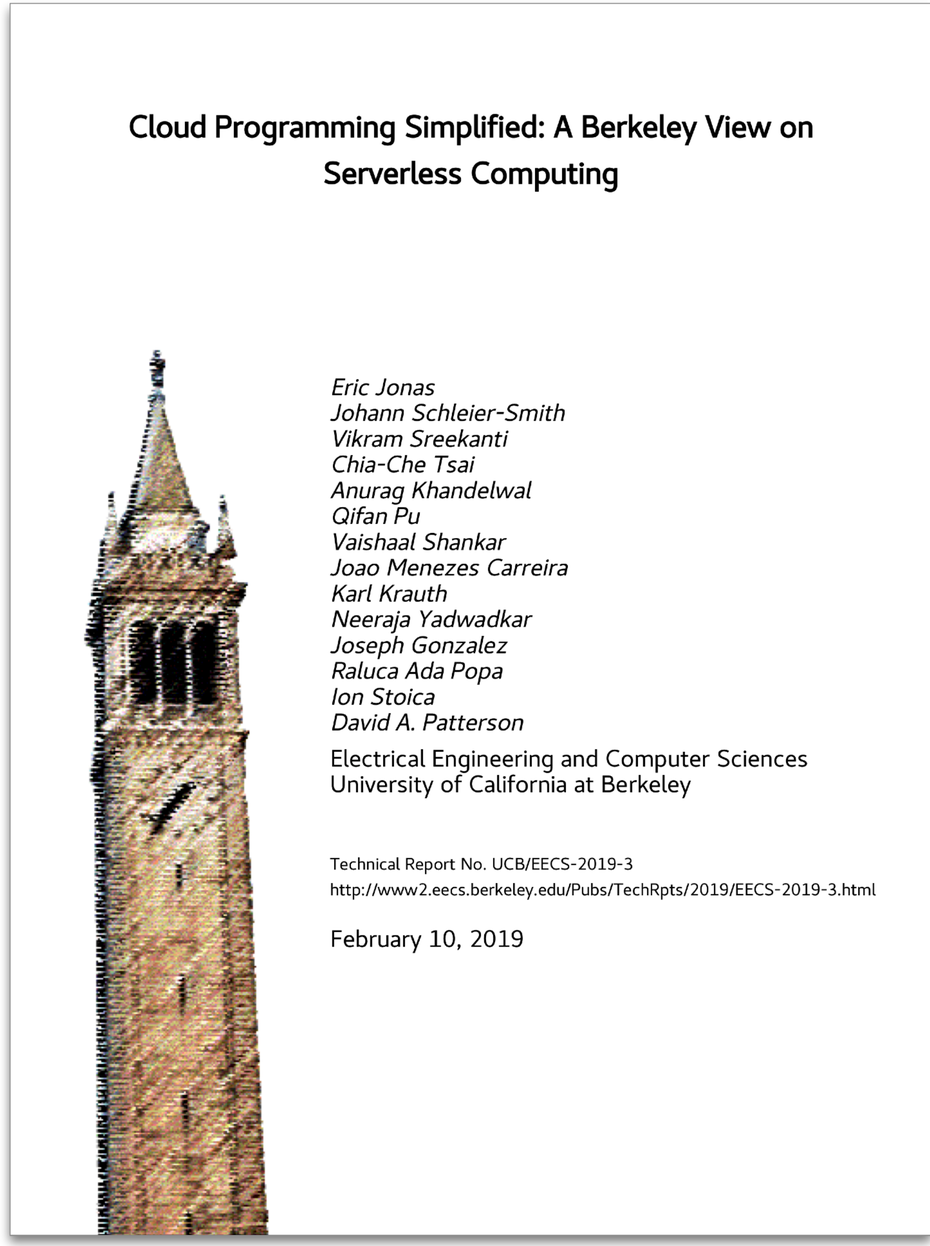
February 10, 2009

2009

Evolution of cloud patterns

Eric Jonas noted >50% of **RISElab** grad students had never used Spark, plus how cloud was evolving in its second decade:

- “Decoupling of computation and storage; they scale separately and are priced independently”
- “The abstraction of executing a piece of code instead of allocating resources on which to execute that code”
- “Paying for the code execution instead of paying for resources you have allocated toward executing the code”



Cloud Programming Simplified: A Berkeley View on Serverless Computing

Eric Jonas
Johann Schleier-Smith
Vikram Sreekanti
Chia-Che Tsai
Anurag Khandelwal
Qifan Pu
Vaishaal Shankar
Joao Menezes Carreira
Karl Krauth
Neeraja Yadwadkar
Joseph Gonzalez
Raluca Ada Popa
Ion Stoica
David A. Patterson

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2019-3
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.html

February 10, 2019

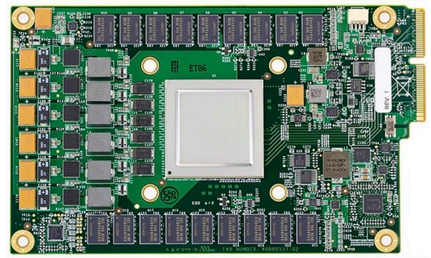
2019

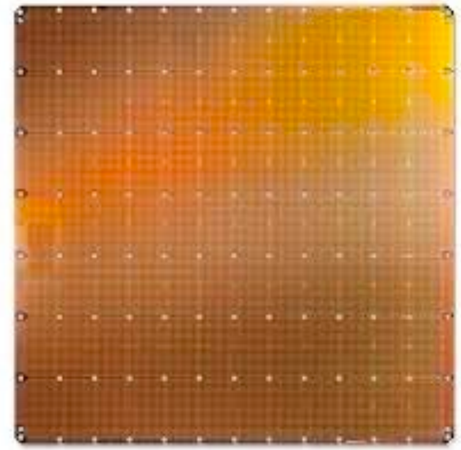
Hardware > Software > Process

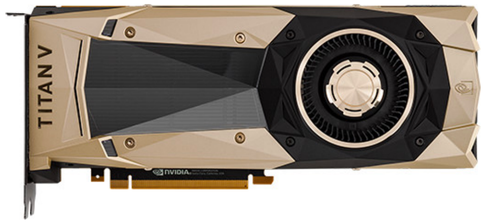
Circa 2005: commodity hardware served Big Data needs: **log file analysis** using **task-parallel**

Circa 2009: data science workloads introduced stateful **actor pattern** through Spark, Akka, etc.

Circa 2013: deep learning placed more demands on s/w + h/w, **differentiating gradients within the context of networked data**







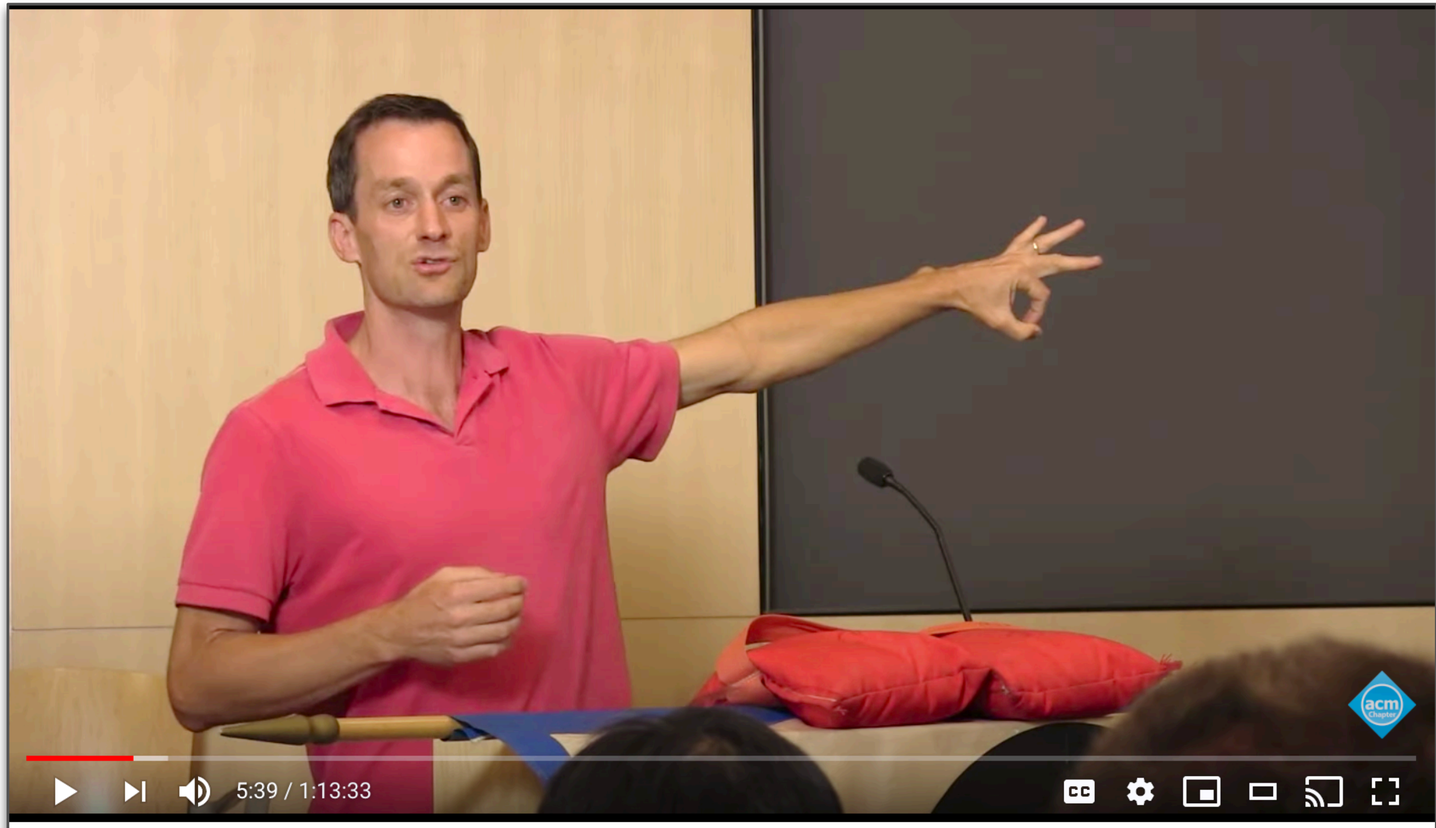
Networks and Gradients

"Taming Latency"
Jeff Dean (2013)

youtu.be/S9twUcX1Zp0

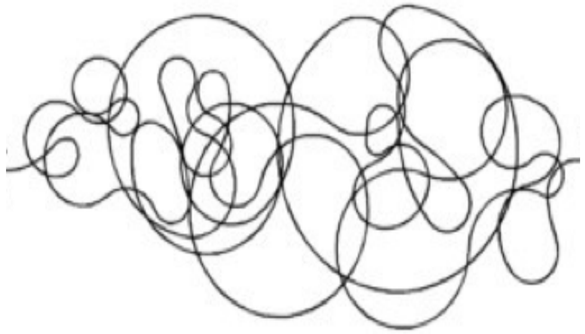
Describing factors within their latest datacenters that drove the design of **TensorFlow**

Optimized to train **networks** (graphs/tensors)

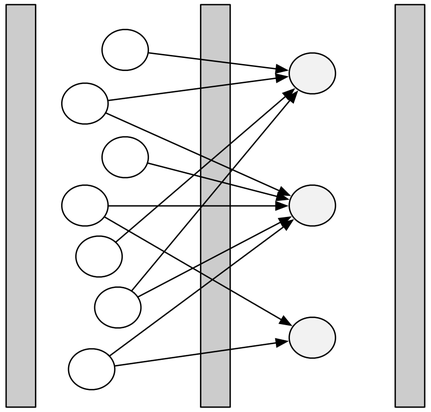


Taming Latency Variability and Scaling Deep Learning, by Jeff Dean, Google, 20131016

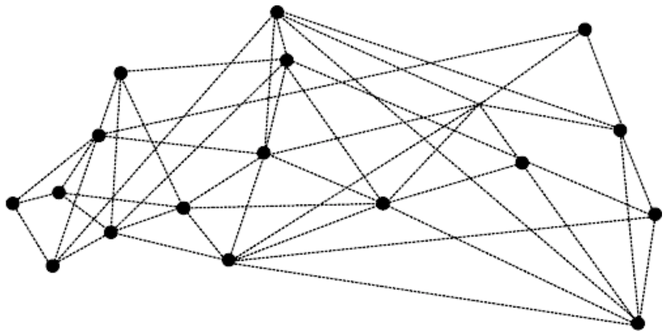
Cluster topologies, by generation



1990s



2000s



current

part 3:
resources

Examples in notebooks…

README.md

A Guided Tour of Ray Core

An introductory tutorial about leveraging Ray core features for distributed patterns.

Note: these examples have been tested using Python 3.7+ on:

- Ubuntu 18.04 LTS
- macOS 10.13

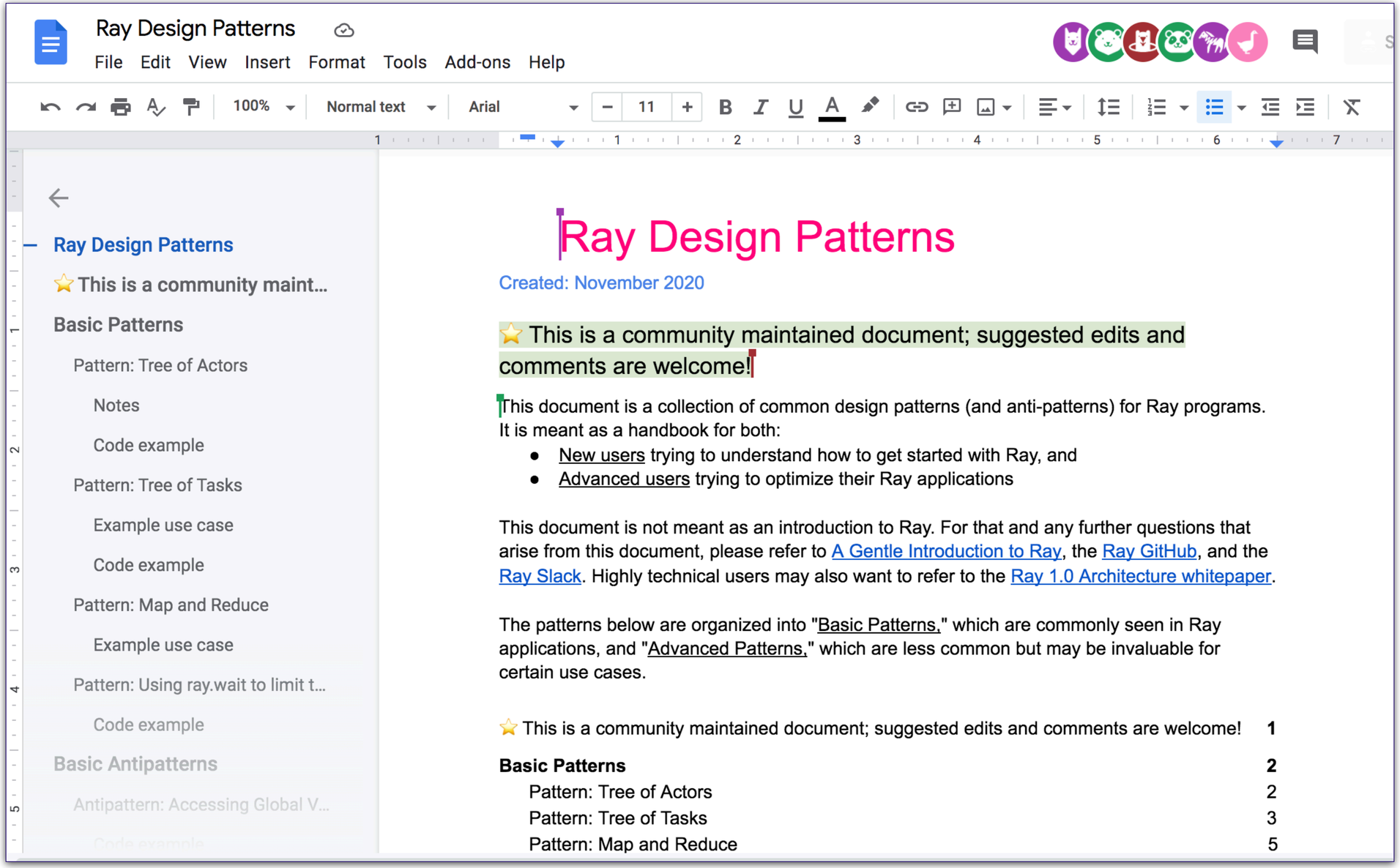
Getting Started

To get started use `git` to clone this public repository:

```
git clone https://github.com/DerwenAI/ray_tutorial.git
cd ray_tutorial
```

https://github.com/DerwenAI/ray_tutorial

A tour through “Ray Design Patterns”

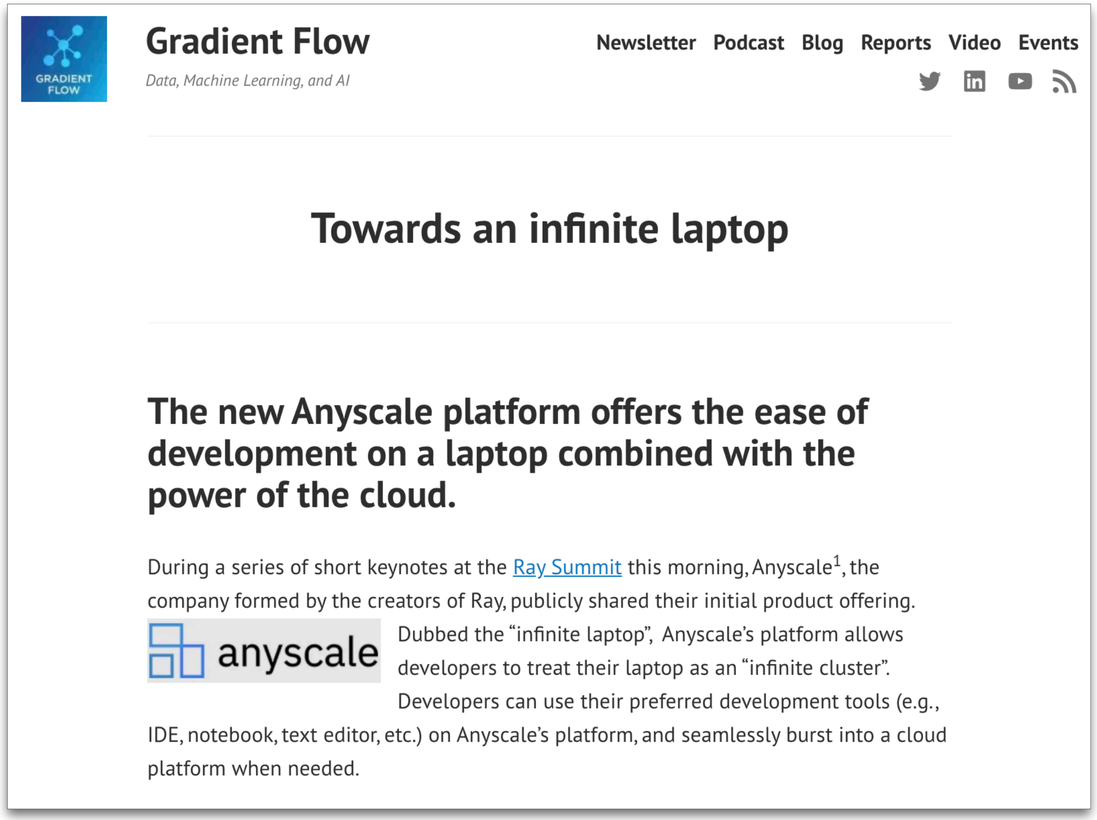


Infinite Laptop

While working, is your **attention** focused on your laptop, or focused somewhere in the cloud?

Why not **both**?

- Ben Lorica explores the emerging architectural pattern of an **Infinite Laptop**
- What RISElab means by the claim “serverless will dominate cloud”



Gradient Flow

Data, Machine Learning, and AI

Newsletter Podcast Blog Reports Video Events

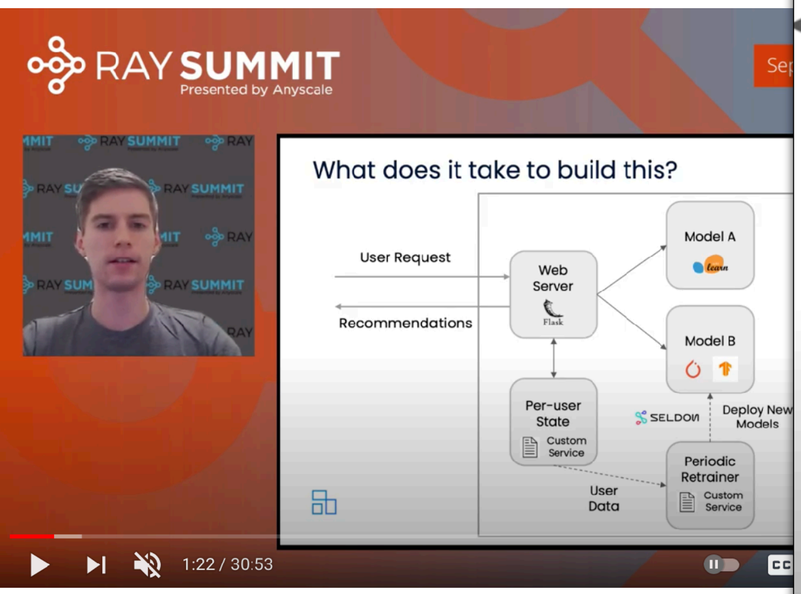
Towards an infinite laptop

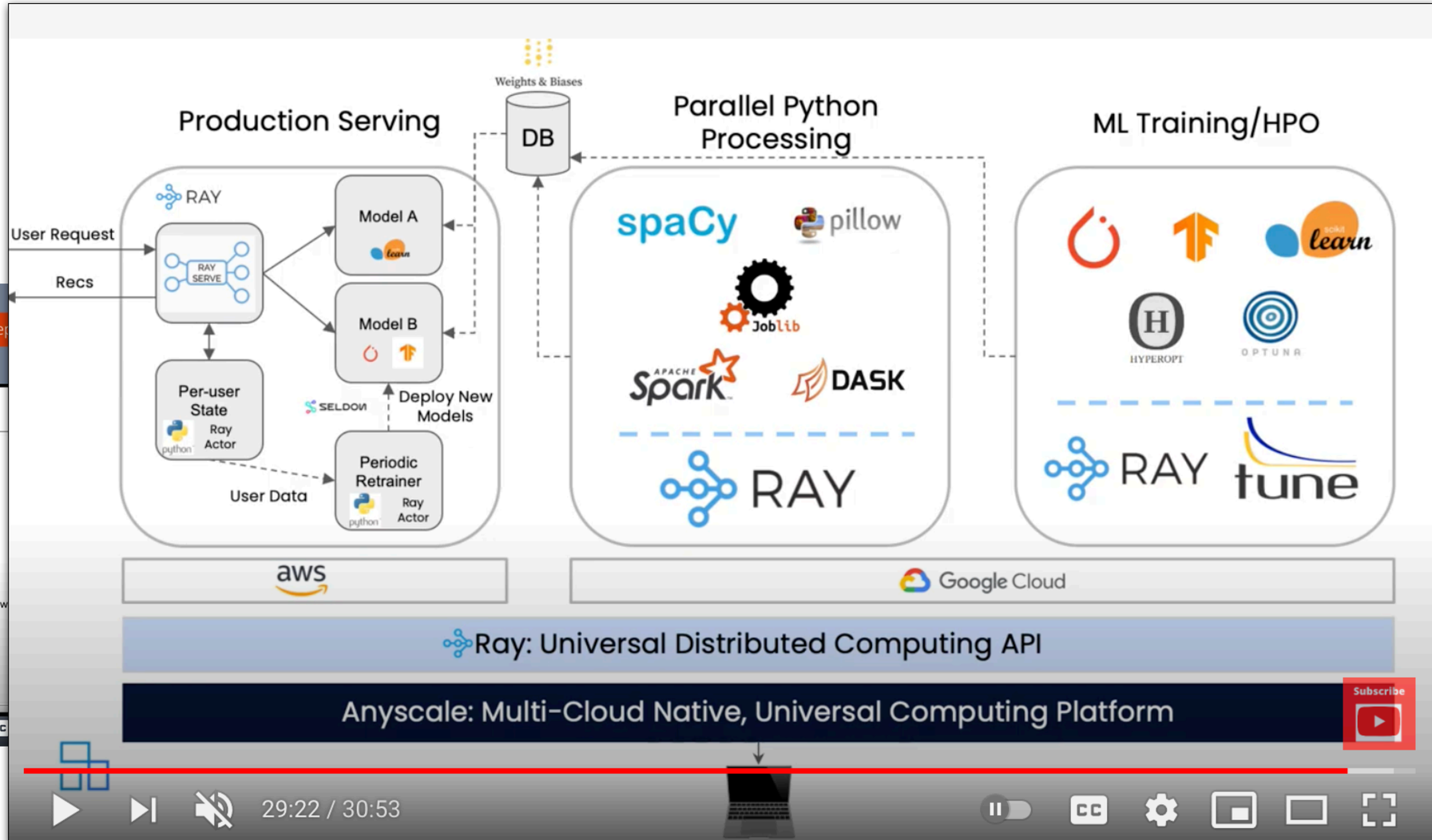
The new Anyscale platform offers the ease of development on a laptop combined with the power of the cloud.

During a series of short keynotes at the Ray Summit this morning, Anyscale[1], the company formed by the creators of Ray, publicly shared their initial product offering. Dubbed the “infinite laptop”, Anyscale’s platform allows developers to treat their laptop as an “infinite cluster”. Developers can use their preferred development tools (e.g., IDE, notebook, text editor, etc.) on Anyscale’s platform, and seamlessly burst into a cloud platform when needed.

Edward Oakes demo

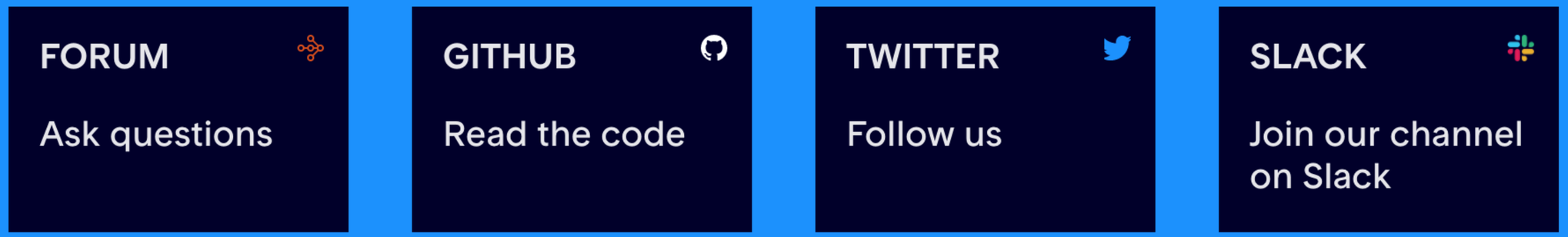
youtu.be/8GTd8Y_JGTQ





anyscale.com/ray-summit-2021

Get involved with the Ray community



- https://discuss.ray.io/
- https://github.com/ray-project/ray
- https://twitter.com/raydistributed
- https://tinyurl.com/rayslack

publications, interviews, conference summaries…

https://derwen.ai/paco
@pacoid

Thank you!

derwen.ai