Building Cloud Infrastructure

Aaron Davidson

CS 349D
Who am I?

- Early Databricks engineer (4 years)
- Apache Spark committer & PMC member
- Worked on a lot of things @ DB
- Most recently, cloud infrastructure
  - Helping eng produce efficient, secure, and reliable software.
What is Databricks?

- Big Data & Machine Learning in the Cloud
  - Yes - our customers are *data scientists* and *data engineers*
- Thinking about getting into self-driving cars
- Yes, we have some Go and Rust code, but prefer FP
```sql
%sql select browserFamily, count(distinct UserID) as Users, count(1) as Events from accessLogsPrime group by browserFamily order by Users desc limit 5;
```

![Bar chart showing users and events for different browser families.](chart1.png)

```sql
%sql select hour(datetime) as Hour, count(1) as events from accessLogsPrime group by hour(datetime) order by hour(datetime)
```

![Bar chart showing events by hourly datetime.](chart2.png)
Databricks Product

- People love Spark, but:
  - How do I get and maintain a Spark cluster?
  - How do I configure that cluster?
  - How do I run jobs reliably and periodically?
  - How do I interface with Spark?
Databricks Product

- People love Spark, but:
  - How do I get and maintain a Spark cluster?
  - How do I configure that cluster?
  - How do I run jobs reliably and periodically?
  - How do I interface with Spark?

- Enter Databricks...
Databricks Product

- People love Spark, but:
  - How do I get and maintain a Spark cluster?
  - How do I configure that cluster?
  - How do I run jobs reliably and periodically?
  - How do I interface with Spark?

- Enter Databricks…
- What hardware do we have?
What does it mean to be a Cloud Company?

- Most money is **still** in on-premise, but trend is towards Cloud.
- “Enterprise:” Financial institutions, government, health care, etc.
- Berkeley & probably Stanford, too
What does it mean to be a Cloud Company?

- Infrastructure in the Cloud (vs on-prem infrastructure):
  - **Infrastructure is dynamic** -- provisioning new hardware in \( O(\text{minutes}) \) rather than \( O(\text{months}) \).
  - No operations team, but high-level primitives provided instead.
    - Storage (DBs, blob storage), networking (routing/firewalls), etc

- Running Software as a Service (vs on-prem appliance) means:
  - We operate the product on behalf of our customers.
  - Often, the software we run is **multitenant**.
  - **Update often** -- deliver features and fixes faster than 3/6/12 months
In this talk

- We’ll use a real-life motivating example from Databricks to talk about building a **cloud service**.
- Focus on three major aspects:
  - Scaling out a multitenant service
  - Updating services safely
  - Deploying the infrastructure to run our service.
In The Beginning, Databricks provided a single-tenant product

- Easier:
  - Security
  - Isolation
  - Selling

- But:
  - Costly
  - Failures
Databricks Community Edition

- We wanted to make a free, multitenant version
- Use-cases: people playing around with Spark, training/classes, MOOCs (now: all new customers)
- Problems:
  - How do we scale our single-tenant services out?
  - How do we update when there is constant usage?
  - How do we maintain this larger, more dynamic infrastructure?
The Notebook Service

- Collaborative notebook UI
  - Users mainly edit their own notebooks, but sometimes want to collaborate
  - Collaboration requires merging changes from multiple users in real-time.
- Originally: ~10 concurrent users.
- Now: Training of 500 people -- or a 50,000-person MOOC!
- How do we scale this service out?
The Notebook Service

POST /notebook/3/cell/2/insert
{ "char": "s" }
POST /notebook/3/cell/2/insert
{ "char": "e" }
POST /notebook/3/cell/2/insert
{ "char": "l" }

UPDATE notebook_cells
SET text = "sel"
WHERE notebook_id=3
AND cell_id=2;

select m.ClientID, c.CountryCode3, m.SessionId, m.DeviceMake
from mobile_sample m
join countrycodes c
on m.Country = c.CountryName
Service Replication

Database

Notebook Service 1
Notebook Service 2
Notebook Service 3

Load Balancer

{s, e, l}
Replication: Logical Stickiness

I own notebooks 3, 7, and 12.
I own notebooks 3, 7, and 12.
Replication: Logical Stickiness

I own notebooks 3, 7, and 12.

Database

Notebook Service 1

Notebook Service 2

Notebook Service 3

Load Balancer

{char: "s"}
{char: "e"}
{char: "l"}

{char: "I"}
Replication: Logical Stickiness

I own notebooks 3, 7, and 12.
Replication: Logical Stickiness

I own notebooks 3, 7, and 12.

**Pros**
- Simple programming model
- Efficient

**Cons**
- Requires complex load balancing infrastructure
- Fault recovery complicated
- Scalability constrained
Replication: Stateless

Database

Notebook Service 1

Notebook Service 2

Notebook Service 3

Load Balancer

{s}

{e}

{l}
Replication: Stateless

- Database
  - Notebook Service 1
  - Notebook Service 2
  - Notebook Service 3
    - Load Balancer

{char: "s"}

{char: "e"}

{char: "I"}
Replication: Stateless

Database

Notebook Service 1
Notebook Service 2
Notebook Service 3

Load Balancer

{char: “e”}
{char: “s”}
{char: “l”}
Replication: Stateless

How do we deal?
- Push logic into database
- Take fine-grained locks
Replication: Stateless

How do we deal?
- Push logic into database
- Take fine-grained locks

Pros
- Interchangeable services
- "Trivial" 0-downtime

Cons
- Hardest/least efficient programming model

Load Balancer

Database

Notebook Service 1
{char: "e"}

Notebook Service 2
{char: "l"}

Notebook Service 3
{char: "s"}
Replication: User/Session Stickiness

Database

Notebook Service 1
Notebook Service 2
Notebook Service 3

Load Balancer

{char: “s”}
{char: “e”}
{char: “l”}
Replication: User/Session Stickiness

Diagram:

- Database
- Notebook Service 1
- Notebook Service 2
- Notebook Service 3
- Load Balancer

Symbols:

{char: “s”}
{char: “e”}
{char: “l”}
Replication: User/Session Stickiness
Replication: User/Session Stickiness
Replication: User/Session Stickiness

- **TCP-sticky load balancer**
  - Easy to find -- probably default!

- **HTTP-sticky load balancer**
  - Cookie-based -- a bit more complicated, but also common
Replication: User/Session Stickiness

Database

Notebook Service 1

Notebook Service 2

Notebook Service 3

Load Balancer

TCP-sticky load balancer
Easy to find -- probably default!

HTTP-sticky load balancer
Cookie-based -- a bit more complicated, but also common

<table>
<thead>
<tr>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ Easy to find</td>
<td>- Only supports single-flow/user locality</td>
</tr>
<tr>
<td>+ Built-in fault recovery</td>
<td>- Failures may be harder to reason about</td>
</tr>
</tbody>
</table>
Service replication: How to decide?

- **Review:**
  - Stateless replication: Simplest
    - Simplest ("best") replication model, hardest to program against
  - Session/user stickiness
    - Particularly common replication model -- well-supported by tooling
  - Logical/tenant stickiness
    - Most complicated ("worst") replication model, easiest to program against

- **Considerations:**
  - Higher is better, but have to start thinking from beginning.
  - If not, then the last will be the only option (that’s exactly what we did for notebooks!)
Service replication: How to implement?

- VM-level: Cloud providers have TCP & HTTP load balancers:
  - Static or scalable pool of machines registered with a port & protocol.
  - Health checking mechanism to remove machines from routable pool.
- Container-level: YMMV; Kubernetes also provides TCP- and HTTP-level load balancing, between containers.
Service replication: How to implement?

- Tenant-stickiness?
- Need a consistent, highly-available leader election store
  - ZooKeeper, consul, etcd (Googlers: Chubby)
- Need an HTTP load balancer
  - Probably nginx or go -- not recommended to build your own, in JVM
Recap: Databricks Community Edition

- We wanted to make a free, multitenant version
- Use-cases: people playing around with Spark, training/classes, MOOCs (now: all new customers)

- Problems:
  ✓ How do we scale our single-tenant services out?
  - How do we update when there is constant usage?
  - How do we maintain this larger, more dynamic infrastructure?
Service updates

- Can leverage our earlier work in service replication to perform updates without downtime.
- Update strategies:
  - The ol’ off ‘n’ on
  - Blue-green
  - Rolling
  - Traffic control
Service updates: Blue/green

Version 1
- Notebook Service 1
- Notebook Service 2
- Notebook Service 3
- Load Balancer

Version 2
- Notebook Service 1
- Notebook Service 2
- Notebook Service 3
- Load Balancer

Load Balancer/DNS

Test!
Service updates: Blue/green

Version 1
- Notebook Service 2
- Notebook Service 1
- Notebook Service 3
- Load Balancer

Version 2
- Notebook Service 2
- Notebook Service 1
- Notebook Service 3
- Load Balancer

Load Balancer/DNS
Service updates: Blue/green

<table>
<thead>
<tr>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ Easy to implement</td>
<td>- Unused infra</td>
</tr>
<tr>
<td>+ Can work with single replica</td>
<td>- All-or-nothing -- bugs exposed immediately</td>
</tr>
</tbody>
</table>

**Pros**

**Cons**

Version 2

- Notebook Service 2
- Notebook Service 1
- Notebook Service 3
- Load Balancer

Load Balancer/DNS
Service updates: Rolling update

- Notebook Service V1
- Notebook Service V1
- Notebook Service V1
- Load Balancer
Service updates: Rolling update

- Notebook Service V1
- Notebook Service V2
- Notebook Service V1
- Load Balancer
Service updates: Rolling update

- Notebook Service V2
- Notebook Service V1
- Load Balancer
Service updates: Rolling update

- Notebook Service V2
- Notebook Service V2
- Notebook Service V2
- Load Balancer
Service updates: Rolling update

- Pros:
  + Gradual roll out
  + All infra used

- Cons:
  - Coarse-grained
Service updates: Traffic control

Version 1

- Notebook Service 1
- Notebook Service 2
- Notebook Service 3
- Load Balancer

Load Balancer/DNS
Service updates: Traffic control

Version 1
- Notebook Service 2
- Notebook Service 1
- Load Balancer

Version 2
- Notebook Service 1
- Load Balancer

Load Balancer/DNS
- 95%
- 5%
Service updates: Traffic control

- Notebook Service 1
- Notebook Service 2
- Notebook Service 3

Load Balancer

Load Balancer/DNS

60% to Notebook Service 2
40% to Notebook Service 1 and Notebook Service 3
Service updates: Traffic control

- Notebook Service 3
- Load Balancer
- Load Balancer/DNS

- Notebook Service 2
- Notebook Service 1
- Notebook Service 3
- Load Balancer
Service updates: Traffic control

<table>
<thead>
<tr>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ Google-scale quality control</td>
<td>- Requires complicated load balancer</td>
</tr>
<tr>
<td>+ Simple extension: shadowing traffic</td>
<td></td>
</tr>
</tbody>
</table>

Gaining traction: Envoy & Istio starting to add support

Pros
- Google-scale quality control
- Simple extension: shadowing traffic

Cons
- Requires complicated load balancer

Notebook Service 1
Notebook Service 2
Notebook Service 3
Load Balancer
Load Balancer/DNS
Update strategy: How to decide?

- Review:
  - Blue/green
    - Useful for stateful applications
    - Useful for acceptance testing
    - Complicated roll-out procedure
  - Rolling update
    - Most common -- simple roll-out procedure
  - Traffic control
    - Best-in-class -- requires complicated load balancer

- Considerations:
  - Design with at least one updates strategy in mind and you can keep downtime minimal, even for unreplicated services.
Update strategy: How to implement?

- VM-level: Cloud providers have (auto)scaling groups.
  - Create a new group for the new version.
  - For blue-green, switch DNS when tested.
  - For rolling update, have load balancer use both groups and increase/decrease replicas.
  - Netflix does this -- see Spinnaker

- Container-level: Kubernetes provides first-class support for rolling updates within one cluster, other stuff is as manual as VM case.
Recap: Databricks Community Edition

- We wanted to make a free, multitenant version
- Use-cases: people playing around with Spark, training/classes, MOOCs (now: all new customers)
- Problems:
  ✓ How do we scale our single-tenant services out?
  ✓ How do we update when there is constant usage?
  - How do we maintain this larger, more dynamic infrastructure?
Infrastructure as Code

- I want to provision 3 VMs for my Notebook Service.
- On-prem: Ask ops team for 3 machines, wait 1-3 months
- Cloud: 

- Scenarios:
  - Scale out to 5 VMs.
  - VM crashes, need to replace it.
  - Change VM parameter (e.g., instance size)
  - Replicate environment to a new region.
  - Create a testing environment.
  - Security breach! Tear it all down and recreate everything.
Infrastructure as Imperative Code

```python
def createInfra():
    for i in range(3):
        ec2.createInstance(
            name = f“NotebookService-$i”,
            type = “m4.xlarge”)
```

- **Scenarios:**
  - Scale out to 5 VMs.
  - VM crashes, need to replace it.
  - Change VM parameter (e.g., instance size)
  - Replicate environment to a new region.
  - Create a testing environment.
  - Security breach! Tear it all down and recreate everything.
Infrastructure as Imperative Code

```python
def createInfra(region):
    for i in range(3):
        ec2.createInstance(
            name = s"NotebookService-$i",
            type = "m4.xlarge",
            region = region)
```

- **Scenarios:**
  - ✗ Scale out to 5 VMs.
  - ✗ VM crashes, need to replace it.
  - ✗ Change VM parameter (e.g., instance size)
  - ✓ Replicate environment to a new region.
  - Create a testing environment.
  - Security breach! Tear it all down and recreate everything.
Infrastructure as Imperative Code

```python
def createInfra(region, accountId):
    for i in range(3):
        ec2.createInstance(
            name = f"NotebookService-$i",
            type = "m4.xlarge",
            region = region,
            accountId = accountId)
```

- Scenarios:
  - ✗ Scale out to 5 VMs.
  - ✗ VM crashes, need to replace it.
  - ✗ Change VM parameter (e.g., instance size)
  - ✓ Replicate environment to a new region.
  - ✓ Create a testing environment.
  - ✓ Security breach! Tear it all down and recreate everything.
def createInfra(region, accountId, oldCount, newCount):
    for i in range(oldCount, newCount):
        ec2.createInstance(
            name = s"NotebookService-$i",
            type = "m4.xlarge",
            region = region,
            accountId = accountId)

- Scenarios:
  ✓ Scale out to 5 VMs.
  ✖ VM crashes, need to replace it.
  ✖ Change VM parameter (e.g., instance size).
  ✓ Replicate environment to a new region.
  ✓ Create a testing environment.
  ✓ Security breach! Tear it all down and recreate everything.
Infrastructure as Imperative Code

- Scenarios:
  - ✓ Scale out to 5 VMs.
  - ❌ VM crashes, need to replace it.
  - ❌ Change VM parameter (e.g., instance size).
  - ✓ Replicate environment to a new region.
  - ✓ Create a testing environment.
  - ✓ Security breach! Tear it all down and recreate everything.

- Problems:
  - Specific: Each scenario needs new code, new parameters. Not necessarily shared between use-cases, either (e.g., create a database)
  - Stateful: Correctness requires either maintaining state, writing state resolution logic, or having a human enter the state.
  - Fallible: Did you spot the incorrect error handling?
Infrastructure as Declarative Code

```json
[
  { kind: "EC2::Instance",
    type: "m4.xlarge",
    name: "NotebookService-0",
    region: "oregon",
    accountId: 1234567,
  }, ...
]
```

- **Scenarios:**
  - ✓ Scale out to 5 VMs.
  - ✓ VM crashes, need to replace it.
  - ✓ Change VM parameter (e.g., instance size).
  - ✓ Replicate environment to a new region.
  - ✓ Create a testing environment.
  - ✓ Security breach! Tear it all down and recreate everything.
Infrastructure as Declarative Code

- **Scenarios:**
  - ✓ Scale out to 5 VMs.
  - ✓ VM crashes, need to replace it.
  - ✓ Change VM parameter (e.g., instance size).
  - ✓ Replicate environment to a new region.
  - ✓ Create a testing environment.
  - ✓ Security breach! Tear it all down and recreate everything.

- **Benefits:** State, API, and error handling are all managed for us
  - Difficult to manage large, dynamic infrastructure due to duplication.
    (One solution here is to introduce a layer of templating)
  - Needs an implementation of “Declarative Deployer”
    - All cloud providers have a native way of doing this (e.g., [CloudFormation](#))
    - [Terraform](#) is a cloud semi-agnostic tool
    - [Quilt](#)?
Recap: Databricks Community Edition

- We wanted to make a free, multitenant version
- Use-cases: people playing around with Spark, training/classes, MOOCs (now: all new customers)
- Problems:
  ✓ How do we scale our single-tenant services out?
  ✓ How do we update when there is constant usage?
  ✓ How do we maintain this larger, more dynamic infrastructure?
Summary

- Cloud infrastructure is dynamic
  - Replicate multitenant services for scale-out
  - Automate deployment (imperatively or declaratively)
  - Leverage cloud provider abstractions (VMs, load balancers, databases)

- Software as a Service allows us to move quickly
  - Deliver updates on weekly cadence rather than 3/6/12-monthly
  - Reduce friction of use by taking over operational burden
  - Just make sure your updates aren’t breaking things too often!
Thank you!

We’re hiring -- come intern with us!

Aaron Davidson - aaron@databricks.com

Try Community Edition:
https://databricks.com/try-databricks
Appendix: Container Engines (Kubernetes)
What problem are we trying to solve?

- I want to run my code on a remote server.
- How do I get my code there?
  - What about my code’s dependencies (e.g., library A)?
  - What about my code’s system dependencies (e.g., curl or ntp)?
- How do I know what’s going on?
  - Logging?
  - SSHing into the machine?
- How do I update my code? How do I roll back?
World V1: Ansible and “bare-metal”

- I want to run my code on a remote server.
- How do I get my code there?
  - Script which copies my JAR and any dependent jars.
  - Script also can install dependencies on target host.
- How do I know what’s going on?
  - SSH in and find out.
- How do I update my code? How do I roll back?
  - Rerun script (how to undo dependencies?)
- Problems:
  - Script is not very general! New one per service.
  - Have to manually place services on hosts (what about node failure?)
World V2: Ansible and Docker

- I want to run my code on a remote server.
- How do I get my code there?
  - I build a Docker container which contains all my dependencies!
  - I run a script which starts that script.
- How do I know what’s going on?
  - SSH in and find out.
- How do I update my code? How do I roll back?
  - Rerun script -- dependencies inside container so can roll back.
- Problems:
  - Script is now pretty general, service-specific stuff is in container.
  - Still have to manually place services on hosts (node failures)
World V3: Kubernetes (w/Docker)

- I want to run my code on a remote server.
- How do I get my code there?
  - I build a Docker container which contains all my dependencies!
  - I ask Kubernetes to find somewhere to put that container.
- How do I know what’s going on?
  - I ask Kubernetes for logs or to SSH into the container directly.
- How do I update my code? How do I roll back?
  - I ask Kubernetes to do a rolling update.
- Problems:
  - Kubernetes replaces my custom script entirely
  - Kubernetes deals with placement of containers within a cluster, and with node failure.
Other Kubernetes Features

- In addition to managing containers, Kubernetes helps with:
  - Exposing services to the outside world via Load Balancers
  - Maintaining a fixed set of replicas of a node.
  - Health checking and restarting services (provided service-specific health checks).
  - Managing network-attached storage.
  - Providing cross-cloud abstractions.
  - (And more!)
- Similar systems: DC/OS, Docker Swarm, Google’s Borg
Container Engines: Unsolved Problems

- Solid, authn/authz inter-service networking
  - Envoy & istio approach problem from proxy layer
  - Calico approaches problem from network layer (BGP!)
- Geo-replicated (multi-cluster) services
- Easy-to-use logical stickiness abstraction (e.g., notebooks)
Appendix: Terraform
Terraform Operating Model

```json
{
  kind: "EC2::Instance",
  type: "m4.xlarge",
  name: "NotebookService-0",
  region: "oregon",
  accountId: 1234567
}
```

- **Input**: Template, state file, and cloud resources
- **Output**: Plan of how to converge state
Terraform Operating Model

- Input: Template, state file, and cloud resources
- Output: Plan of how to converge state
- Different properties require different change procedures.
  - Changing EC2 VM instance size requires tearing down and recreating.
  - Changing RDS database instance size requires just restarting.
Terraform Operating Model

```json
{
  kind: "EC2::Instance",
  type: "m4.xlarge",
  name: "NotebookService-100",
  region: "oregon",
  accountId: 1234567
}
```

- State file used so Terraform knows when it should delete objects.
- Otherwise, we would just create a second instance and keep the old one around.
Declarative Deploy: Unsolved Problems

- Cloud agnostic terminology & semantics is elusive
- Declaring different classes of resources (e.g., cloud provider versus Kubernetes objects) requires different systems
- Enacting a certain change may require several intermediate templates
- No standard for templating.