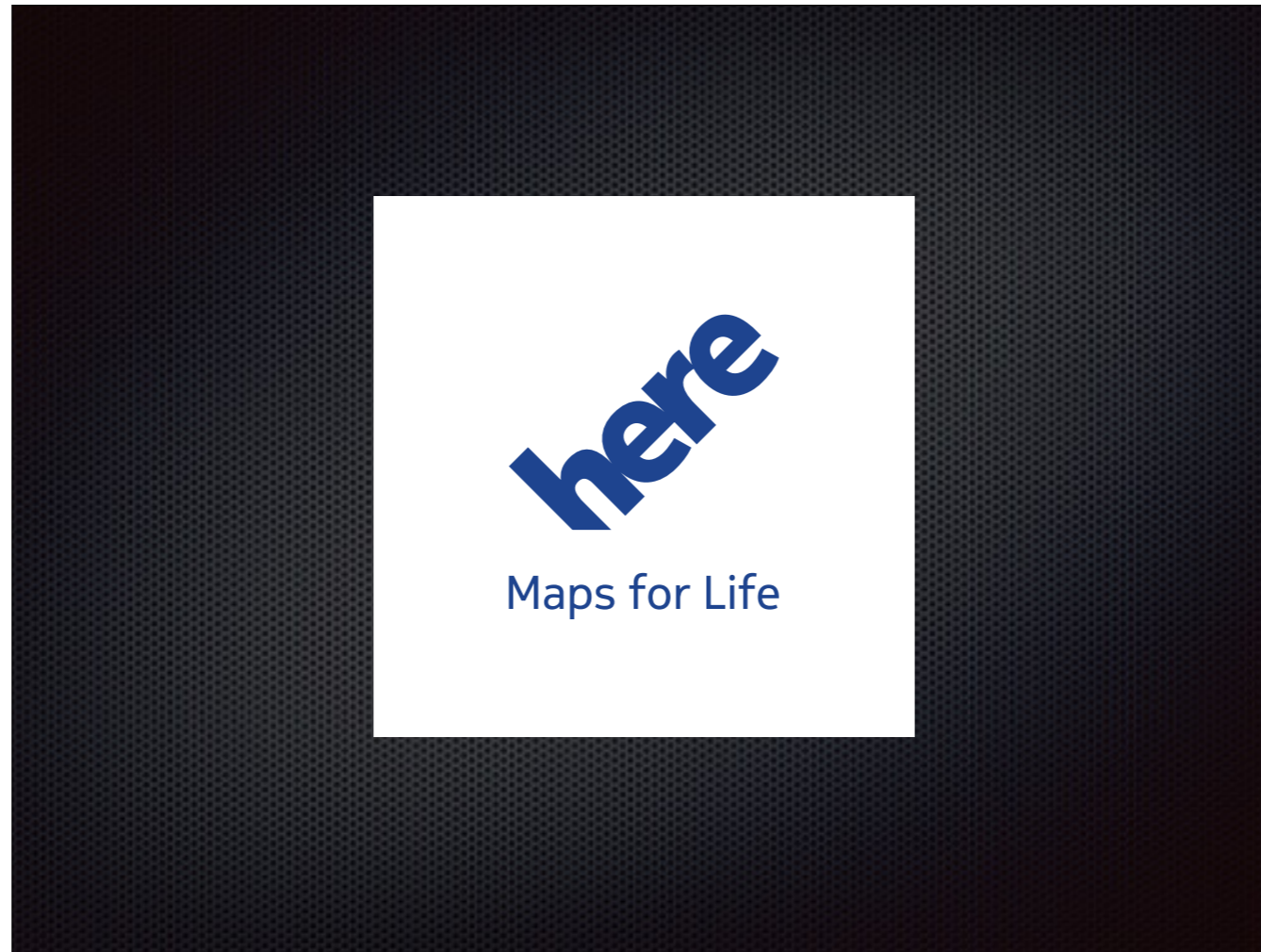


Scientific Workflow DSLs

What are they for anyway?

Twitter: [@irina_guberman](https://twitter.com/irina_guberman)



I work for HERE, name invented by the same marketing geniuses that invented the “Swedish fish”. Trying googling it. Try telling people where you work. It’s a lot of fun... HERE? Where is that? But hey, German car manufacturers: BMW, Audi, and Daimler acquired us nonetheless. The old name was Navteq, the one and only electronic mapping company around for quite a while (since 1985)... until Google became a major competitor. Google is a good thing, nothing drives quality like good competition. And BTW, if you’re using Google Maps on your iPhone or Android, do yourself a favor and download HERE maps instead. You will quickly realize — you’ve been missing out.

While navigable digital maps is HERE’s primary business right now, what is very exciting for HERE future is autonomous driving cars. With that kind of ambition, simply navigable maps with meter resolution won’t do. What’s essential is pre-loaded High Definition maps that a car would be using along with the real-time data.



So high definition maps. Satellite and aerial imagery is a good starting point there. However, what really makes them possible is a fleet of cars equipped with GPS, cameras and LiDAR (“technology that measures distance by illuminating a target with a laser light”). According to numerous sources, LiDAR stands for Light Detection And Ranging. According to Wikipedia, that’s a myth — it was actually coined as a portmanteau of "light" and “radar”. The sensors on these cars (called TRUE cars) were developed by John Ristevski the head of Reality Capture and Processing department, the department where I work.



So high definition maps. Satellite and aerial imagery is a good starting point there. However, what really makes them possible is a fleet of cars equipped with GPS, cameras and LiDAR (“technology that measures distance by illuminating a target with a laser light”). According to numerous sources, LiDAR stands for Light Detection And Ranging. According to Wikipedia, that’s a myth — it was actually coined as a portmanteau of "light" and “radar”. The sensors on these cars (called TRUE cars) were developed by John Ristevski the head of Reality Capture and Processing department, the department where I work.



LiDAR image capturing

So high definition maps. Satellite and aerial imagery is a good starting point there. However, what really makes them possible is a fleet of cars equipped with GPS, cameras and LiDAR (“technology that measures distance by illuminating a target with a laser light”). According to numerous sources, LiDAR stands for Light Detection And Ranging. According to Wikipedia, that’s a myth — it was actually coined as a portmanteau of "light" and “radar”. The sensors on these cars (called TRUE cars) were developed by John Ristevski the head of Reality Capture and Processing department, the department where I work.



LiDAR image capturing

“technology that measures distance by illuminating a target with a laser light”

So high definition maps. Satellite and aerial imagery is a good starting point there. However, what really makes them possible is a fleet of cars equipped with GPS, cameras and LiDAR (“technology that measures distance by illuminating a target with a laser light”). According to numerous sources, LiDAR stands for Light Detection And Ranging. According to Wikipedia, that’s a myth — it was actually coined as a portmanteau of "light" and “radar”. The sensors on these cars (called TRUE cars) were developed by John Ristevski the head of Reality Capture and Processing department, the department where I work.



LiDAR image capturing

“technology that measures distance by illuminating a target with a laser light”

<http://360.here.com/2015/03/24/lidar/>

So high definition maps. Satellite and aerial imagery is a good starting point there. However, what really makes them possible is a fleet of cars equipped with GPS, cameras and LiDAR (“technology that measures distance by illuminating a target with a laser light”). According to numerous sources, LiDAR stands for Light Detection And Ranging. According to Wikipedia, that’s a myth — it was actually coined as a portmanteau of "light" and “radar”. The sensors on these cars (called TRUE cars) were developed by John Ristevski the head of Reality Capture and Processing department, the department where I work.

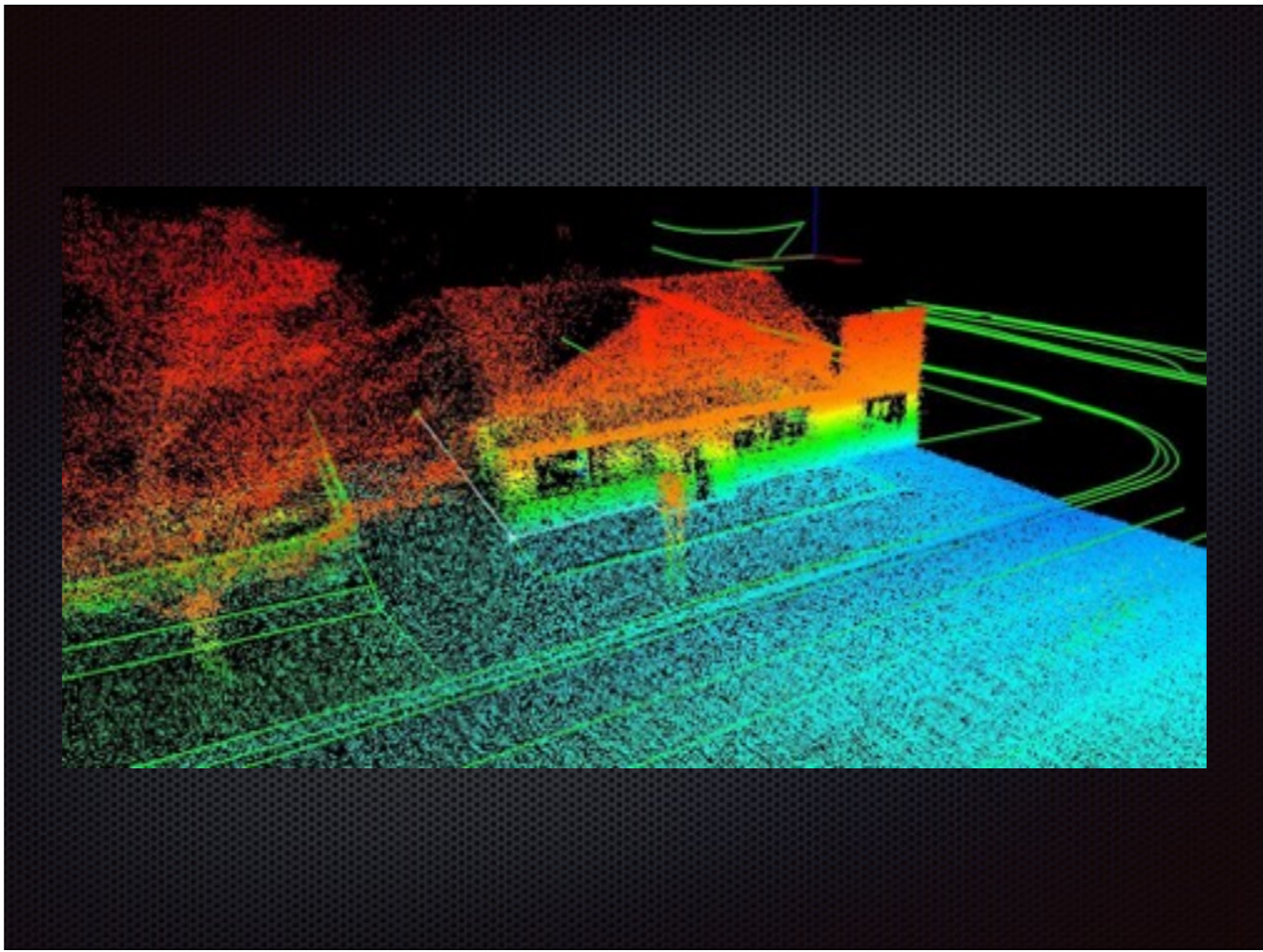


“When the car is in motion, the lidar system—a cylinder about the size of a soda can—spins around, shooting out 32 laser beams and analyzing the light that bounces back. It collects 700,000 points per second”

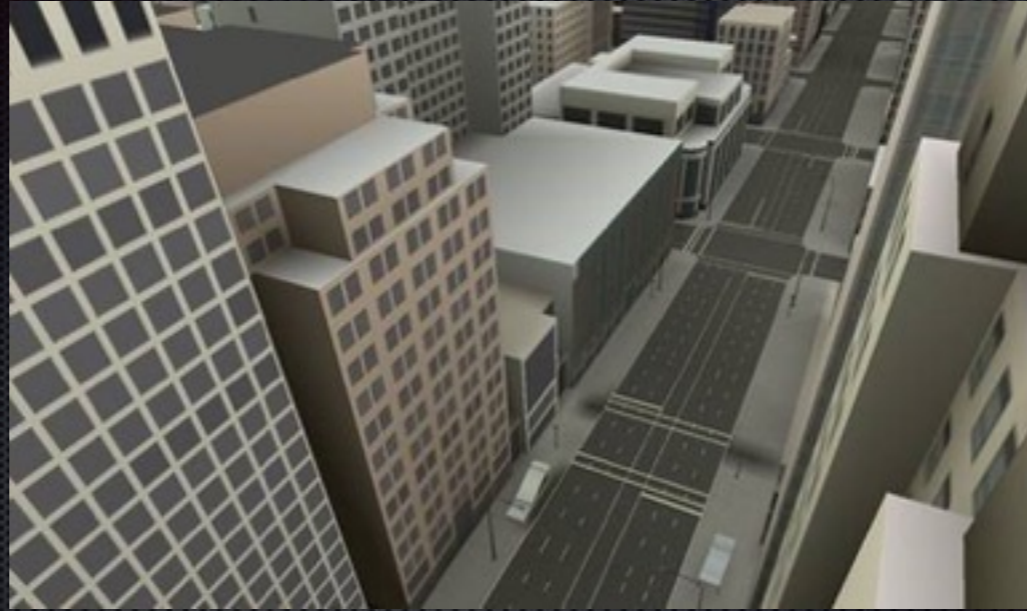
<http://www.wired.com/2014/12/nokia-here-autonomous-car-maps/>

After a car drives for a while, it accumulates dozens of Gigabytes of Lidar and image data to be processed. We get lidar and camera data from these cars driving all over the world.

Which after going through a rather complicated series of Machine Learning algorithms developed by various computer vision teams at HERE and their collective effort turns the drive data from this kind of image (next page)



into this kind of a neat image (Next page):



This kind of High definition map with a few centimeter precision is what driverless cars can actually use.



The original drive data ends up in various Reality Capture and Processing teams that will process different aspects of the data: lidar calibration, optimization, street sign detection, it's a much longer list... Some of these processing stages are completely independent, some depend on the results of the stage that ran before, so what we get here is a pretty complex data processing pipeline, or we can call it a scientific workflow. Members of these teams, typically ML PhDs, are busy enough developing computer vision algorithms not to worry a lot about where and how these things will be running, but considering the amount of data going through the pipeline and the amount of processing power it requires, that in itself is quite a challenge, the challenge called High Throughput Computing. So who are the lucky bunch in charge of this honorable responsibility? That's my team, Architecture & Infrastructure team. We ask the PhD colleagues to put their ML algorithms into a docker container and we take it from there.

So HTC challenges: #1 is Cost -> Efficient use of computing and memory resources -> Resilience, on one hand. Efficient use of developer resources and code maintainability on the other.

HTC

The original drive data ends up in various Reality Capture and Processing teams that will process different aspects of the data: lidar calibration, optimization, street sign detection, it's a much longer list... Some of these processing stages are completely independent, some depend on the results of the stage that ran before, so what we get here is a pretty complex data processing pipeline, or we can call it a scientific workflow. Members of these teams, typically ML PhDs, are busy enough developing computer vision algorithms not to worry a lot about where and how these things will be running, but considering the amount of data going through the pipeline and the amount of processing power it requires, that in itself is quite a challenge, the challenge called High Throughput Computing. So who are the lucky bunch in charge of this honorable responsibility? That's my team, Architecture & Infrastructure team. We ask the PhD colleagues to put their ML algorithms into a docker container and we take it from there.

So HTC challenges: #1 is Cost -> Efficient use of computing and memory resources -> Resilience, on one hand. Efficient use of developer resources and code maintainability on the other.



HTC

Cost

The original drive data ends up in various Reality Capture and Processing teams that will process different aspects of the data: lidar calibration, optimization, street sign detection, it's a much longer list... Some of these processing stages are completely independent, some depend on the results of the stage that ran before, so what we get here is a pretty complex data processing pipeline, or we can call it a scientific workflow. Members of these teams, typically ML PhDs, are busy enough developing computer vision algorithms not to worry a lot about where and how these things will be running, but considering the amount of data going through the pipeline and the amount of processing power it requires, that in itself is quite a challenge, the challenge called High Throughput Computing. So who are the lucky bunch in charge of this honorable responsibility? That's my team, Architecture & Infrastructure team. We ask the PhD colleagues to put their ML algorithms into a docker container and we take it from there.

So HTC challenges: #1 is Cost -> Efficient use of computing and memory resources -> Resilience, on one hand. Efficient use of developer resources and code maintainability on the other.

HTC

Cost

Efficiency

The original drive data ends up in various Reality Capture and Processing teams that will process different aspects of the data: lidar calibration, optimization, street sign detection, it's a much longer list... Some of these processing stages are completely independent, some depend on the results of the stage that ran before, so what we get here is a pretty complex data processing pipeline, or we can call it a scientific workflow. Members of these teams, typically ML PhDs, are busy enough developing computer vision algorithms not to worry a lot about where and how these things will be running, but considering the amount of data going through the pipeline and the amount of processing power it requires, that in itself is quite a challenge, the challenge called High Throughput Computing. So who are the lucky bunch in charge of this honorable responsibility? That's my team, Architecture & Infrastructure team. We ask the PhD colleagues to put their ML algorithms into a docker container and we take it from there.

So HTC challenges: #1 is Cost -> Efficient use of computing and memory resources -> Resilience, on one hand. Efficient use of developer resources and code maintainability on the other.

HTC

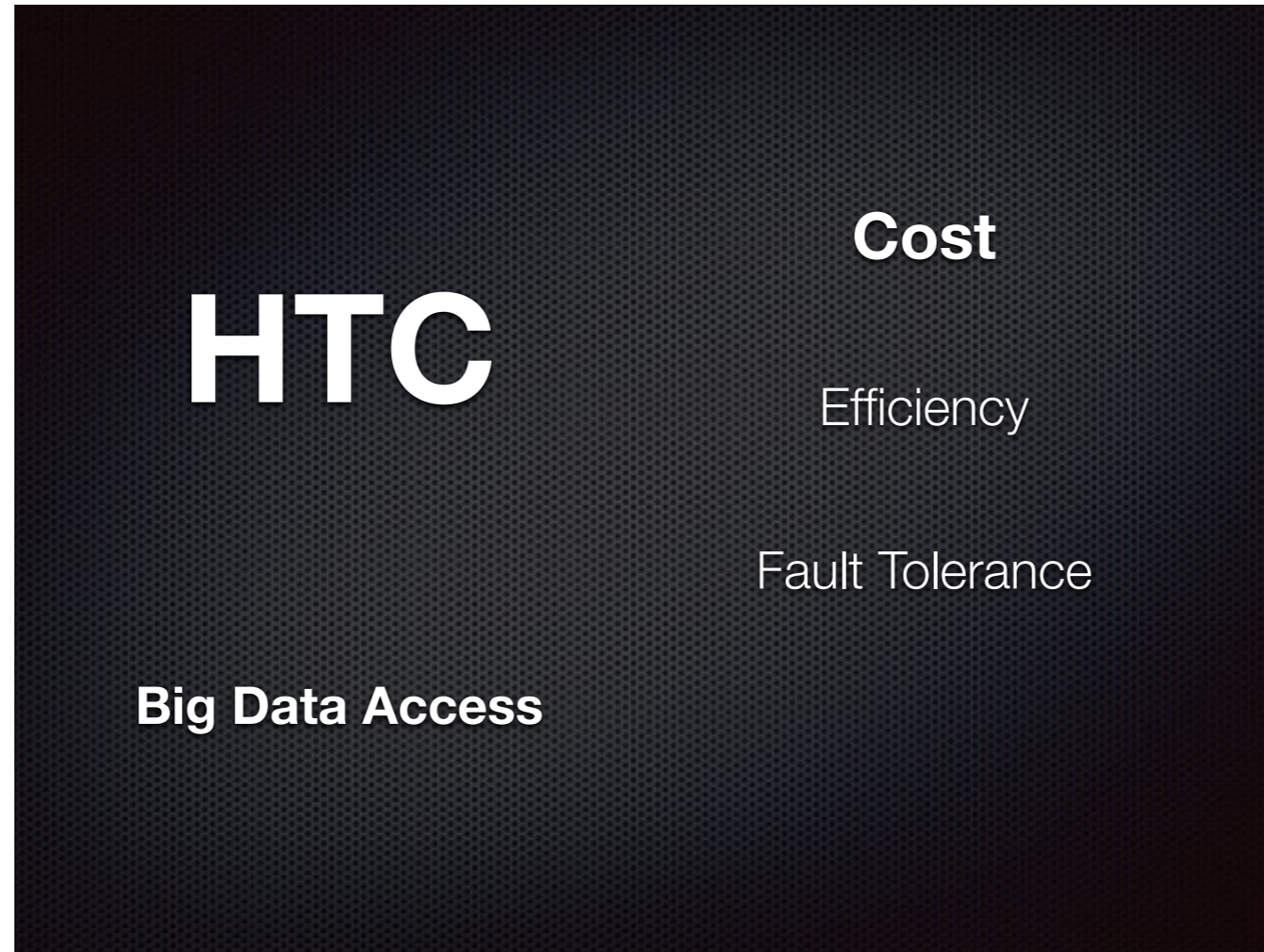
Cost

Efficiency

Fault Tolerance

The original drive data ends up in various Reality Capture and Processing teams that will process different aspects of the data: lidar calibration, optimization, street sign detection, it's a much longer list... Some of these processing stages are completely independent, some depend on the results of the stage that ran before, so what we get here is a pretty complex data processing pipeline, or we can call it a scientific workflow. Members of these teams, typically ML PhDs, are busy enough developing computer vision algorithms not to worry a lot about where and how these things will be running, but considering the amount of data going through the pipeline and the amount of processing power it requires, that in itself is quite a challenge, the challenge called High Throughput Computing. So who are the lucky bunch in charge of this honorable responsibility? That's my team, Architecture & Infrastructure team. We ask the PhD colleagues to put their ML algorithms into a docker container and we take it from there.

So HTC challenges: #1 is Cost -> Efficient use of computing and memory resources -> Resilience, on one hand. Efficient use of developer resources and code maintainability on the other.



The original drive data ends up in various Reality Capture and Processing teams that will process different aspects of the data: lidar calibration, optimization, street sign detection, it's a much longer list... Some of these processing stages are completely independent, some depend on the results of the stage that ran before, so what we get here is a pretty complex data processing pipeline, or we can call it a scientific workflow. Members of these teams, typically ML PhDs, are busy enough developing computer vision algorithms not to worry a lot about where and how these things will be running, but considering the amount of data going through the pipeline and the amount of processing power it requires, that in itself is quite a challenge, the challenge called High Throughput Computing. So who are the lucky bunch in charge of this honorable responsibility? That's my team, Architecture & Infrastructure team. We ask the PhD colleagues to put their ML algorithms into a docker container and we take it from there.

So HTC challenges: #1 is Cost -> Efficient use of computing and memory resources -> Resilience, on one hand. Efficient use of developer resources and code maintainability on the other.

Cost

We don't have any data centers. Everything we do is done in AWS. Cost -> Spot instances... but they go away without warning. Even if with a warning... they just go away. Warning won't help if you don't know how to take advantage of intermediate results and continue on with the job on another spot instance.

Save intermediate results, make work reentrant.

Distribution is a great idea, but not always possible with ML algorithms, they sometimes need to process everything at once, just by the nature of the algorithm, and sometimes they depend on existing libraries which we can't modify.

Cost

AWS Spot Instances

We don't have any data centers. Everything we do is done in AWS. Cost -> Spot instances... but they go away without warning. Even if with a warning... they just go away. Warning won't help if you don't know how to take advantage of intermediate results and continue on with the job on another spot instance.

Save intermediate results, make work reentrant.

Distribution is a great idea, but not always possible with ML algorithms, they sometimes need to process everything at once, just by the nature of the algorithm, and sometimes they depend on existing libraries which we can't modify.

Cost

AWS Spot Instances

Keep work as short as possible.

We don't have any data centers. Everything we do is done in AWS. Cost -> Spot instances... but they go away without warning. Even if with a warning... they just go away. Warning won't help if you don't know how to take advantage of intermediate results and continue on with the job on another spot instance.

Save intermediate results, make work reentrant.

Distribution is a great idea, but not always possible with ML algorithms, they sometimes need to process everything at once, just by the nature of the algorithm, and sometimes they depend on existing libraries which we can't modify.

Cost

AWS Spot Instances

Keep work as short as possible.

Periodically save results

We don't have any data centers. Everything we do is done in AWS. Cost -> Spot instances... but they go away without warning. Even if with a warning... they just go away. Warning won't help if you don't know how to take advantage of intermediate results and continue on with the job on another spot instance.

Save intermediate results, make work reentrant.

Distribution is a great idea, but not always possible with ML algorithms, they sometimes need to process everything at once, just by the nature of the algorithm, and sometimes they depend on existing libraries which we can't modify.

Cost

AWS Spot Instances

Keep work as short as possible.

Periodically save results

Distribute work whenever possible

We don't have any data centers. Everything we do is done in AWS. Cost -> Spot instances... but they go away without warning. Even if with a warning... they just go away. Warning won't help if you don't know how to take advantage of intermediate results and continue on with the job on another spot instance.

Save intermediate results, make work reentrant.

Distribution is a great idea, but not always possible with ML algorithms, they sometimes need to process everything at once, just by the nature of the algorithm, and sometimes they depend on existing libraries which we can't modify.

Distribute work whenever possible

Distribute work! Fun stuff. But what about those dozens of gigabytes of drive data that the algorithms have to operate on???

Keep your data separate from the spot instances. Ideally, use spot instance CPU cycles, but keep the data somewhere they all can access it. So you don't have to copy inputs when you bring spot instance up and you don't lose any results when it goes away prematurely. And in this case, Distributed File System is our great friend.

Distribute work whenever possible

What about huge data?

Distribute work! Fun stuff. But what about those dozens of gigabytes of drive data that the algorithms have to operate on???

Keep your data separate from the spot instances. Ideally, use spot instance CPU cycles, but keep the data somewhere they all can access it. So you don't have to copy inputs when you bring spot instance up and you don't lose any results when it goes away prematurely. And in this case, Distributed File System is our great friend.



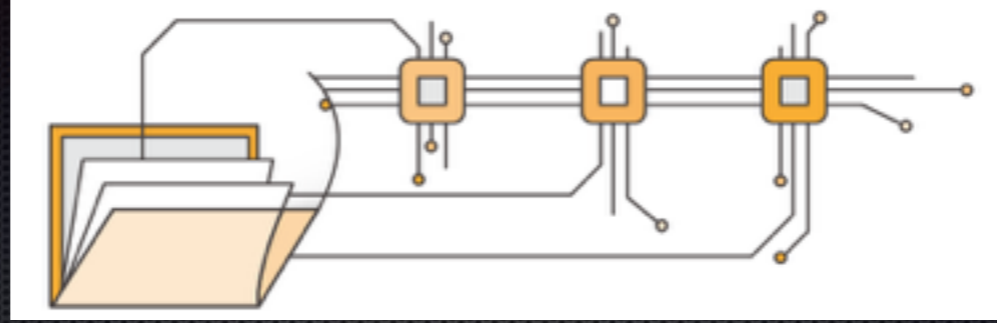
Distribute work! Fun stuff. But what about those dozens of gigabytes of drive data that the algorithms have to operate on???

Keep your data separate from the spot instances. Ideally, use spot instance CPU cycles, but keep the data somewhere they all can access it. So you don't have to copy inputs when you bring spot instance up and you don't lose any results when it goes away prematurely. And in this case, Distributed File System is our great friend.

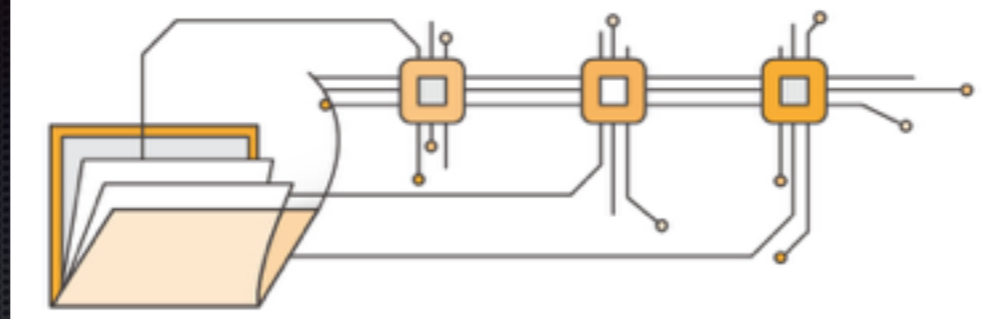


AWS EFS

AWS EFS



AWS EFS

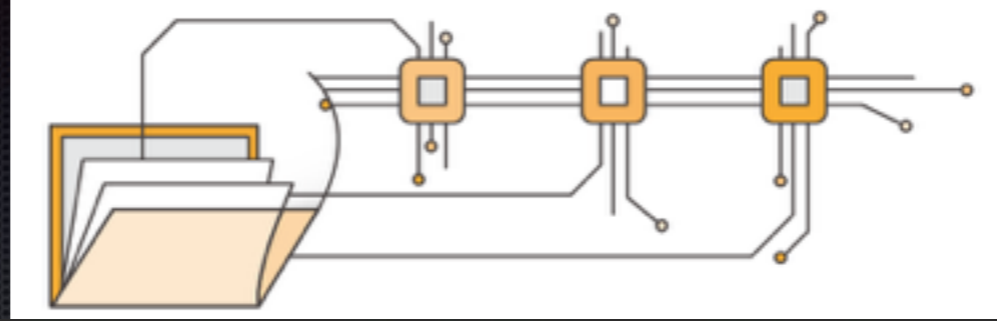


Introducing The Amazon Elastic File System



- A fully managed file system for EC2
- Grows to petabyte scale, elastically
- SSD-based
- Very high throughput and low latency
- Highly available and durable
- Automatic data replication across AZs (like S3)

AWS EFS



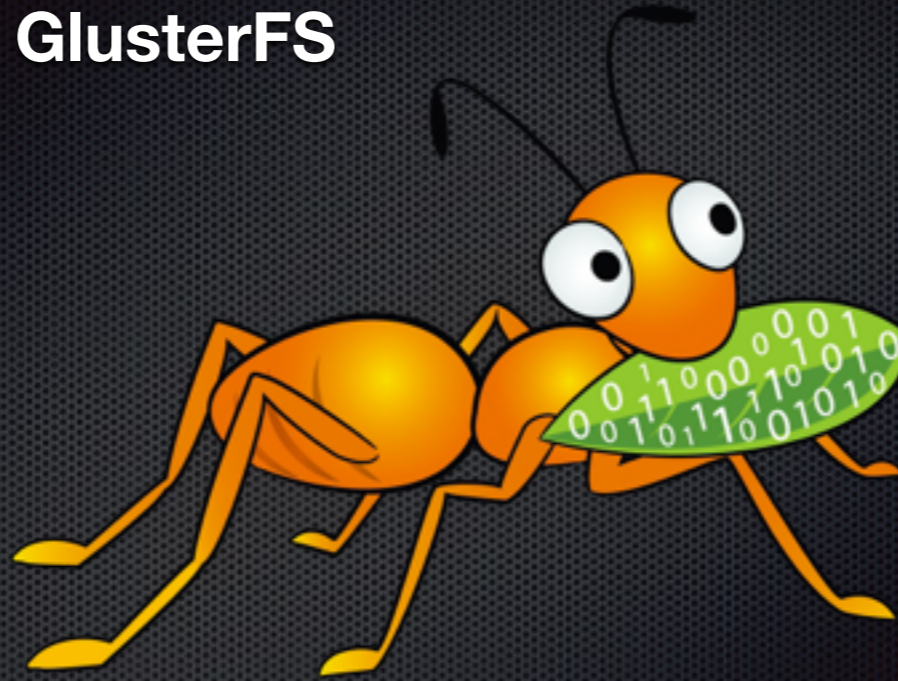
Introducing The Amazon Elastic File System

- A fully managed file system for EC2
- Grows to petabyte scale, automatically
- Base on POSIX
- High throughput and low latency
- Highly available and durable
- Automatic data replication across AZs (like S3)

UNAVAILABLE



GlusterFS



Scalable, general-purpose storage platform
POSIX-y Distributed File System
Object storage (swift) Distributed block storage (qemu) Flexible storage (libgfapi)
No Metadata Server
Heterogeneous Commodity Hardware
Standards-Based – Clients, Applications, Networks
Flexible and Agile Scaling
Capacity – Petabytes and beyond Performance – Thousands of Clients.
Available distribution options: distributed, replicated, striped, replicated striped.

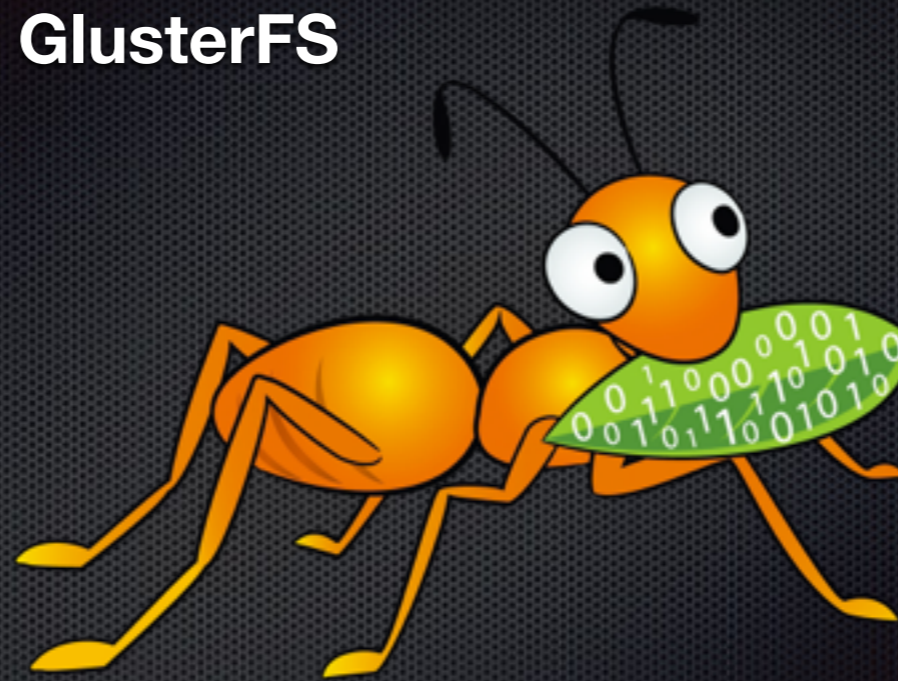
Following volume types are supported in glusterfs:

- a) Distribute
- b) Stripe
- c) Replication
- d) Distributed Replicate
- e) Striped Replicate
- f) Distributed Striped Replicate

Software only, runs on commodity hardware
No external metadata servers

GlusterFS

\$\$\$



Scalable, general-purpose storage platform POSIX-y Distributed File System
Object storage (swift) Distributed block storage (qemu) Flexible storage (libgfapi)
No Metadata Server
Heterogeneous Commodity Hardware
Standards-Based – Clients, Applications, Networks
Flexible and Agile Scaling
Capacity – Petabytes and beyond Performance – Thousands of Clients.
Available distribution options: distributed, replicated, striped, replicated striped.

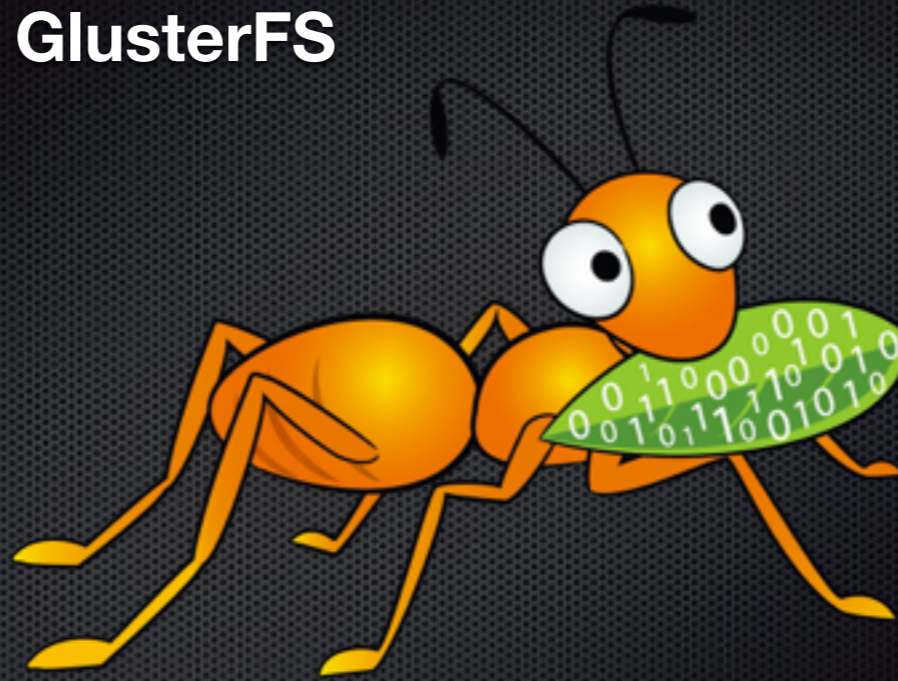
Following volume types are supported in glusterfs:

- a) Distribute
- b) Stripe
- c) Replication
- d) Distributed Replicate
- e) Striped Replicate
- f) Distributed Striped Replicate

Software only, runs on commodity hardware
No external metadata servers

GlusterFS

\$\$\$



[https://gluster.readthedocs.org/en/latest/presentations/
GlusterFS Architecture & Roadmap-Vijay Bellur-LinuxCon EU 2013.pdf](https://gluster.readthedocs.org/en/latest/presentations/GlusterFS%20Architecture%20&%20Roadmap-Vijay%20Bellur-LinuxCon%20EU%202013.pdf)

Scalable, general-purpose storage platform POSIX-y Distributed File System
Object storage (swift) Distributed block storage (qemu) Flexible storage (libgfapi)
No Metadata Server
Heterogeneous Commodity Hardware
Standards-Based – Clients, Applications, Networks
Flexible and Agile Scaling
Capacity – Petabytes and beyond Performance – Thousands of Clients.
Available distribution options: distributed, replicated, striped, replicated striped.

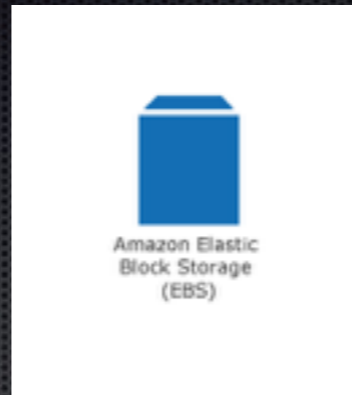
Following volume types are supported in glusterfs:

- a) Distribute
- b) Stripe
- c) Replication
- d) Distributed Replicate
- e) Striped Replicate
- f) Distributed Striped Replicate

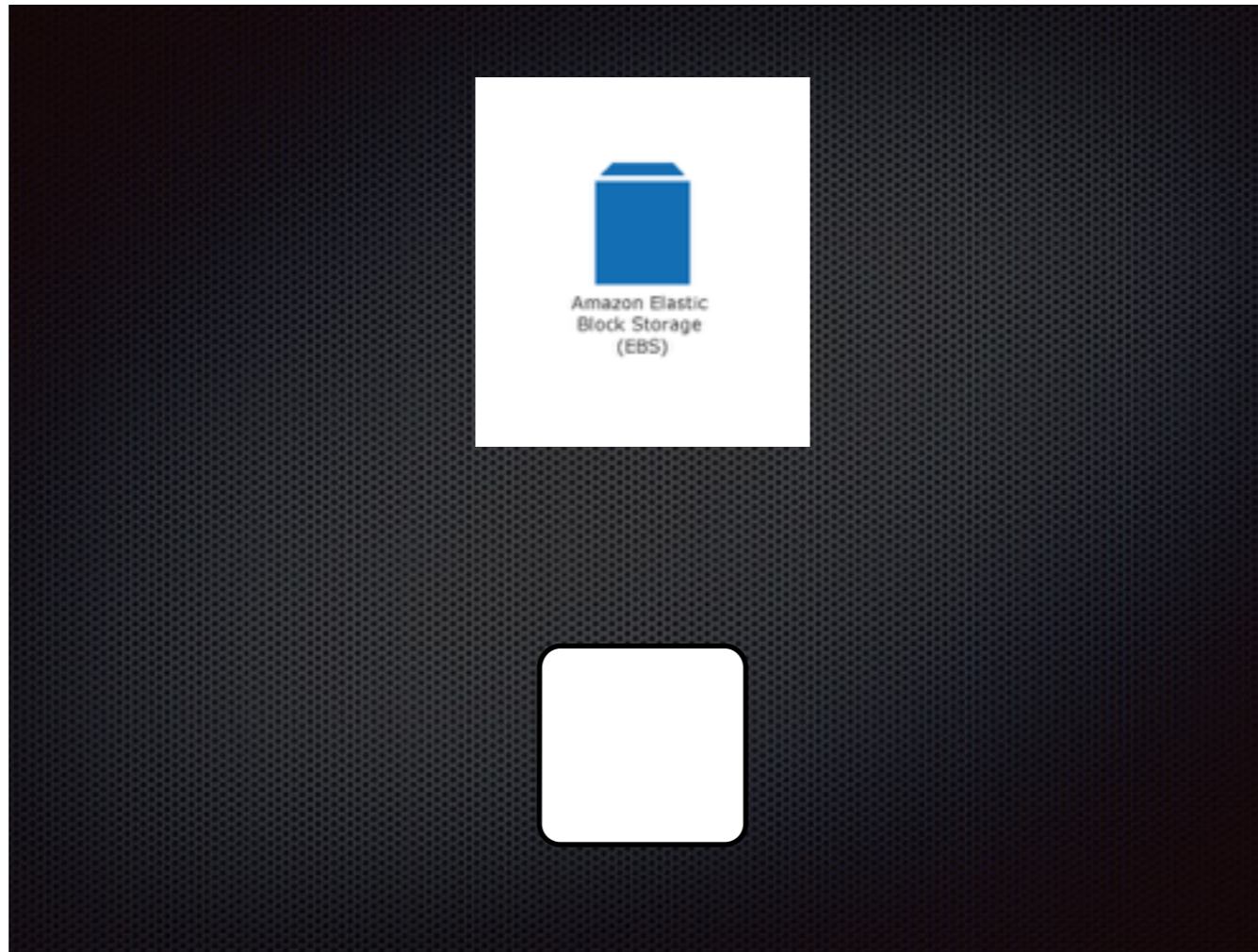
Software only, runs on commodity hardware
No external metadata servers



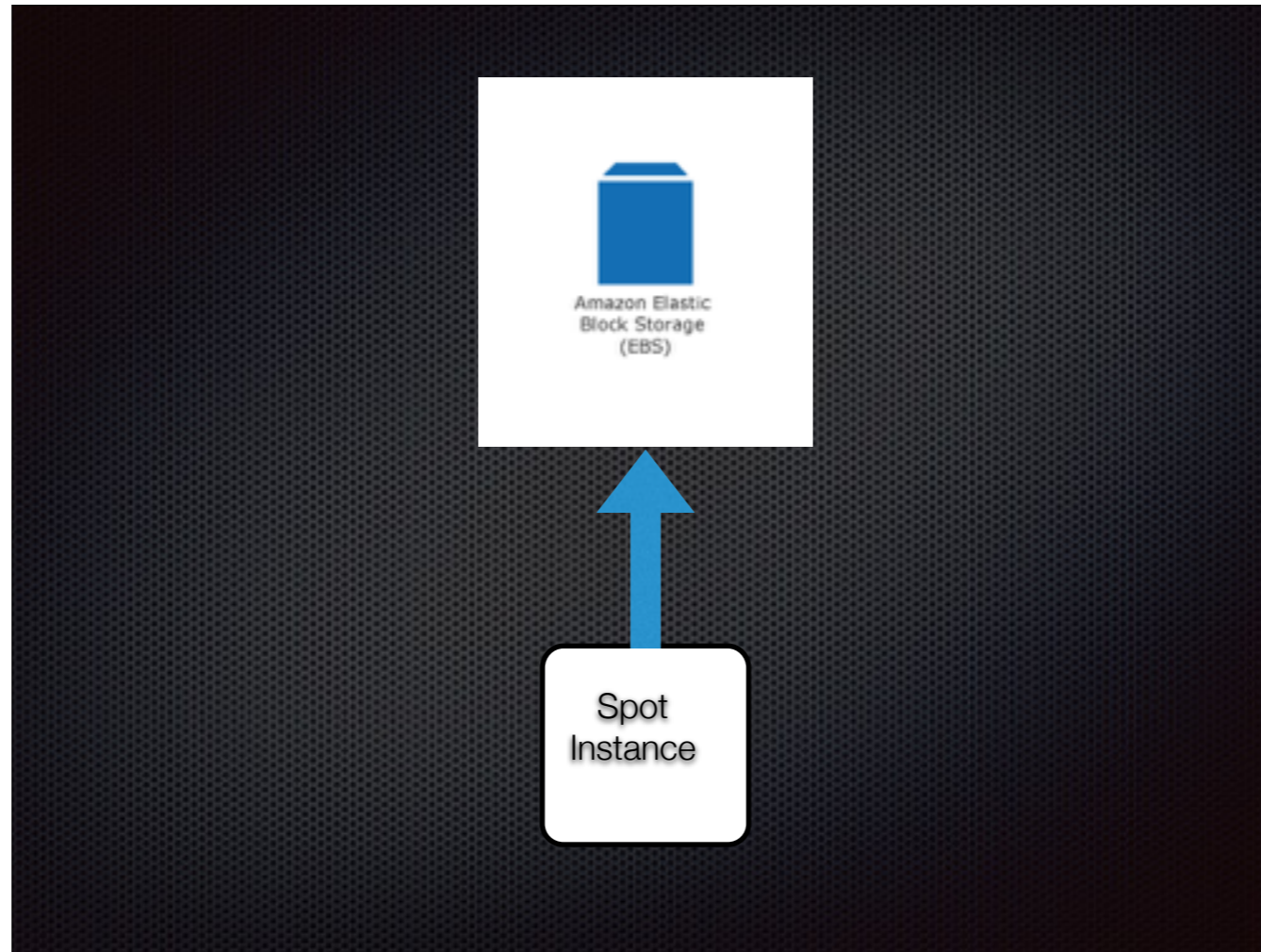
So GlusterFS is much pricier than EFS would've been and it's much pricier than just a simple EBS volume attached to a spot instance. Well, maybe it will prove to be negligible in the larger scheme of things maybe it won't be. We're in an experimental state of things and we're playing with our options. If for some reason we decide to not use Gluster FS and just go with simple EBS attach/reattach, we'll also have to un-distribute processing of the stages that would've been otherwise perfectly distributable, or else — we'll end up re-implementing some semblance of a DFS ourselves. Too messy and time consuming to bother with. And un-distribute might be ok if it saves us a lot of money. And maybe EFS will become available at some point and some brave hearts will even use it in their production env before we do. The reason why I'm bringing up the challenges and potential fluidity of our architecture and demonstrating to you all these scenarios is actually not to educate you about all this stuff, though if you found it interesting — awesome! If you didn't — thanks for your patience! The reason is — we run scientific workflows on top of it. That is running complexity on top of complexity. Equivalent of entering a perfect storm.



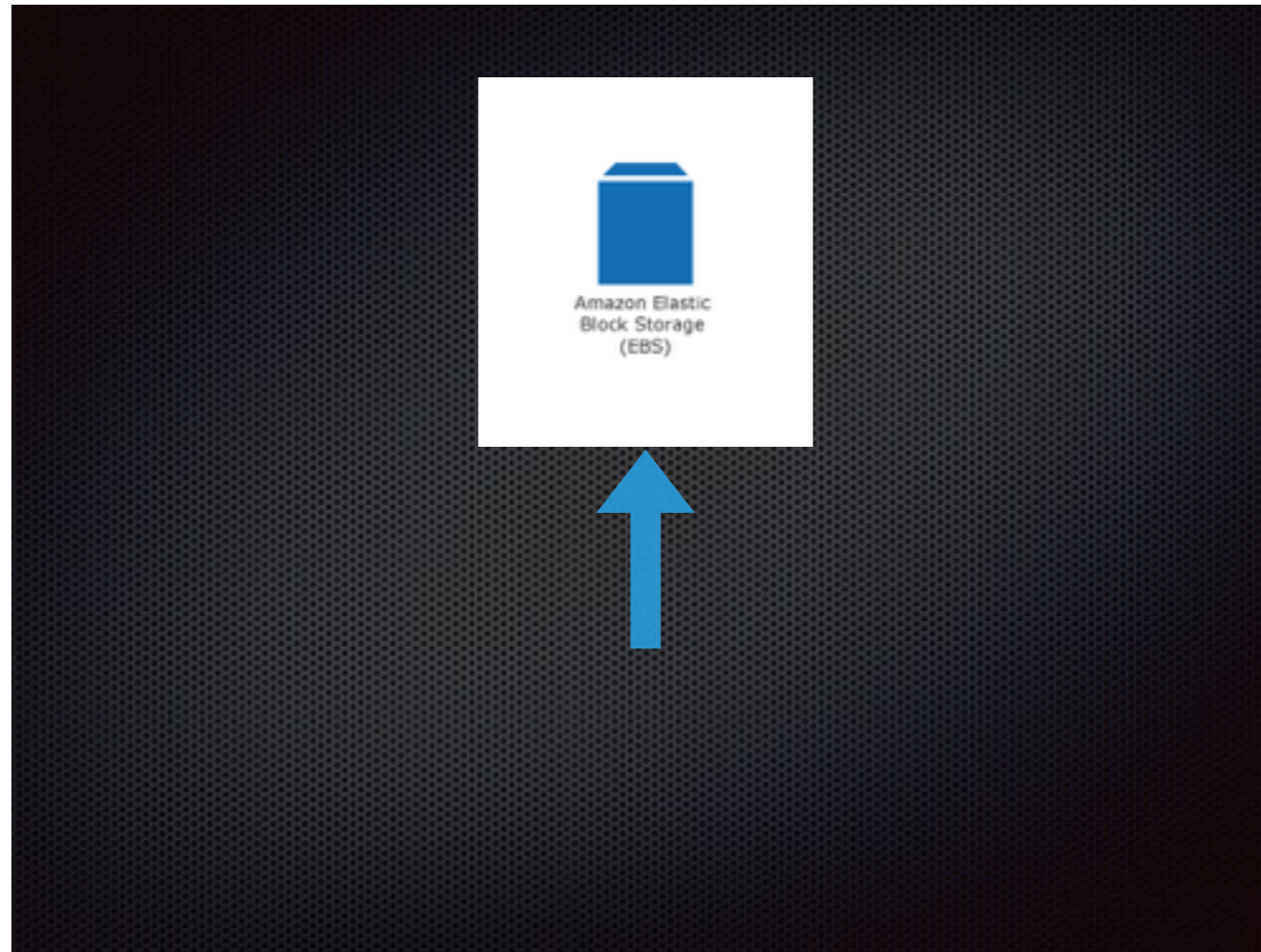
So GlusterFS is much pricier than EFS would've been and it's much pricier than just a simple EBS volume attached to a spot instance. Well, maybe it will prove to be negligible in the larger scheme of things maybe it won't be. We're in an experimental state of things and we're playing with our options. If for some reason we decide to not use Gluster FS and just go with simple EBS attach/reattach, we'll also have to un-distribute processing of the stages that would've been otherwise perfectly distributable, or else — we'll end up re-implementing some semblance of a DFS ourselves. Too messy and time consuming to bother with. And un-distribute might be ok if it saves us a lot of money. And maybe EFS will become available at some point and some brave hearts will even use it in their production env before we do. The reason why I'm bringing up the challenges and potential fluidity of our architecture and demonstrating to you all these scenarios is actually not to educate you about all this stuff, though if you found it interesting — awesome! If you didn't — thanks for your patience! The reason is — we run scientific workflows on top of it. That is running complexity on top of complexity. Equivalent of entering a perfect storm.



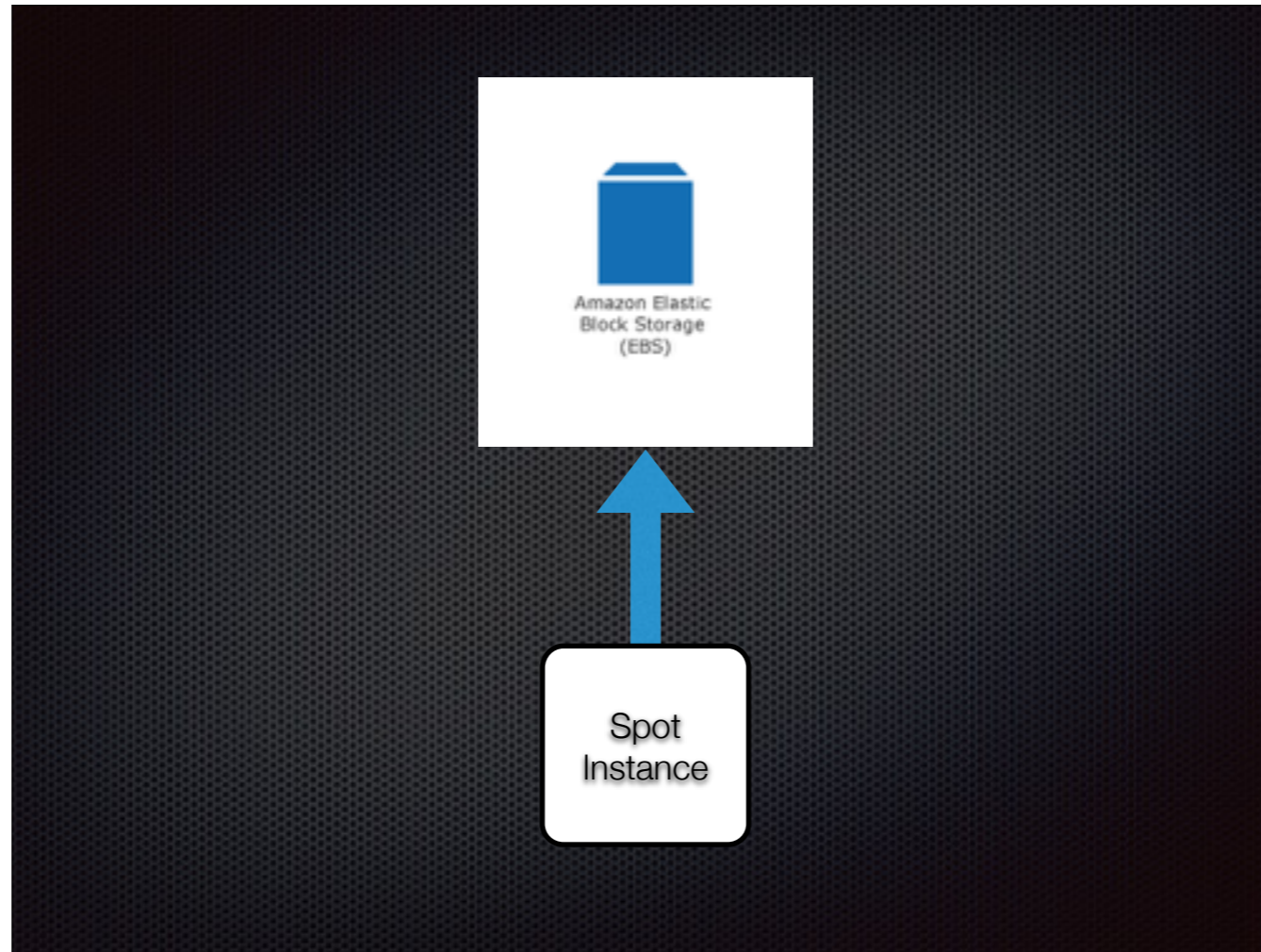
So GlusterFS is much pricier than EFS would've been and it's much pricier than just a simple EBS volume attached to a spot instance. Well, maybe it will prove to be negligible in the larger scheme of things maybe it won't be. We're in an experimental state of things and we're playing with our options. If for some reason we decide to not use Gluster FS and just go with simple EBS attach/reattach, we'll also have to un-distribute processing of the stages that would've been otherwise perfectly distributable, or else — we'll end up re-implementing some semblance of a DFS ourselves. Too messy and time consuming to bother with. And un-distribute might be ok if it saves us a lot of money. And maybe EFS will become available at some point and some brave hearts will even use it in their production env before we do. The reason why I'm bringing up the challenges and potential fluidity of our architecture and demonstrating to you all these scenarios is actually not to educate you about all this stuff, though if you found it interesting — awesome! If you didn't — thanks for your patience! The reason is — we run scientific workflows on top of it. That is running complexity on top of complexity. Equivalent of entering a perfect storm.



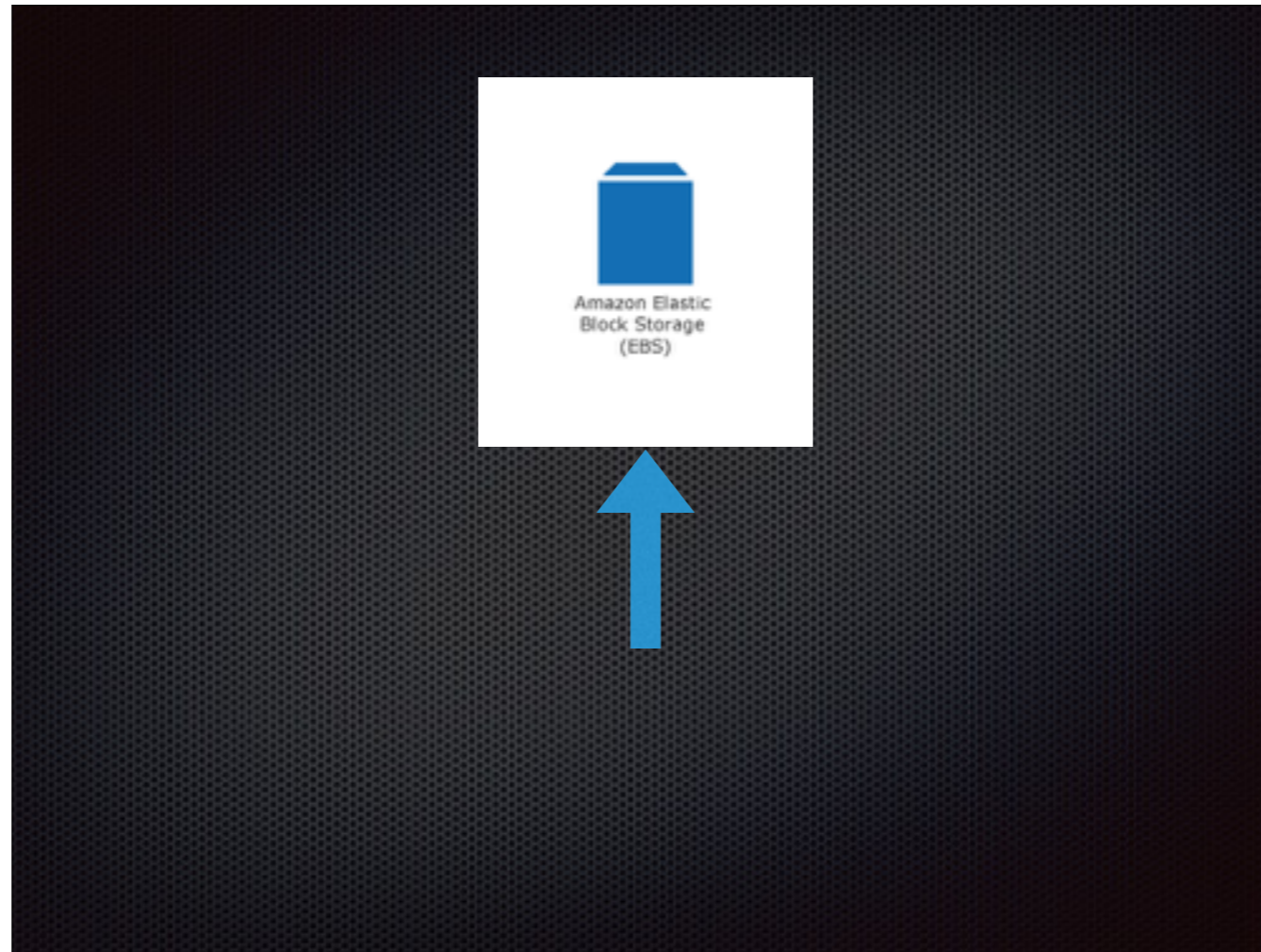
So GlusterFS is much pricier than EFS would've been and it's much pricier than just a simple EBS volume attached to a spot instance. Well, maybe it will prove to be negligible in the larger scheme of things maybe it won't be. We're in an experimental state of things and we're playing with our options. If for some reason we decide to not use Gluster FS and just go with simple EBS attach/reattach, we'll also have to un-distribute processing of the stages that would've been otherwise perfectly distributable, or else — we'll end up re-implementing some semblance of a DFS ourselves. Too messy and time consuming to bother with. And un-distribute might be ok if it saves us a lot of money. And maybe EFS will become available at some point and some brave hearts will even use it in their production env before we do. The reason why I'm bringing up the challenges and potential fluidity of our architecture and demonstrating to you all these scenarios is actually not to educate you about all this stuff, though if you found it interesting — awesome! If you didn't — thanks for your patience! The reason is — we run scientific workflows on top of it. That is running complexity on top of complexity. Equivalent of entering a perfect storm.



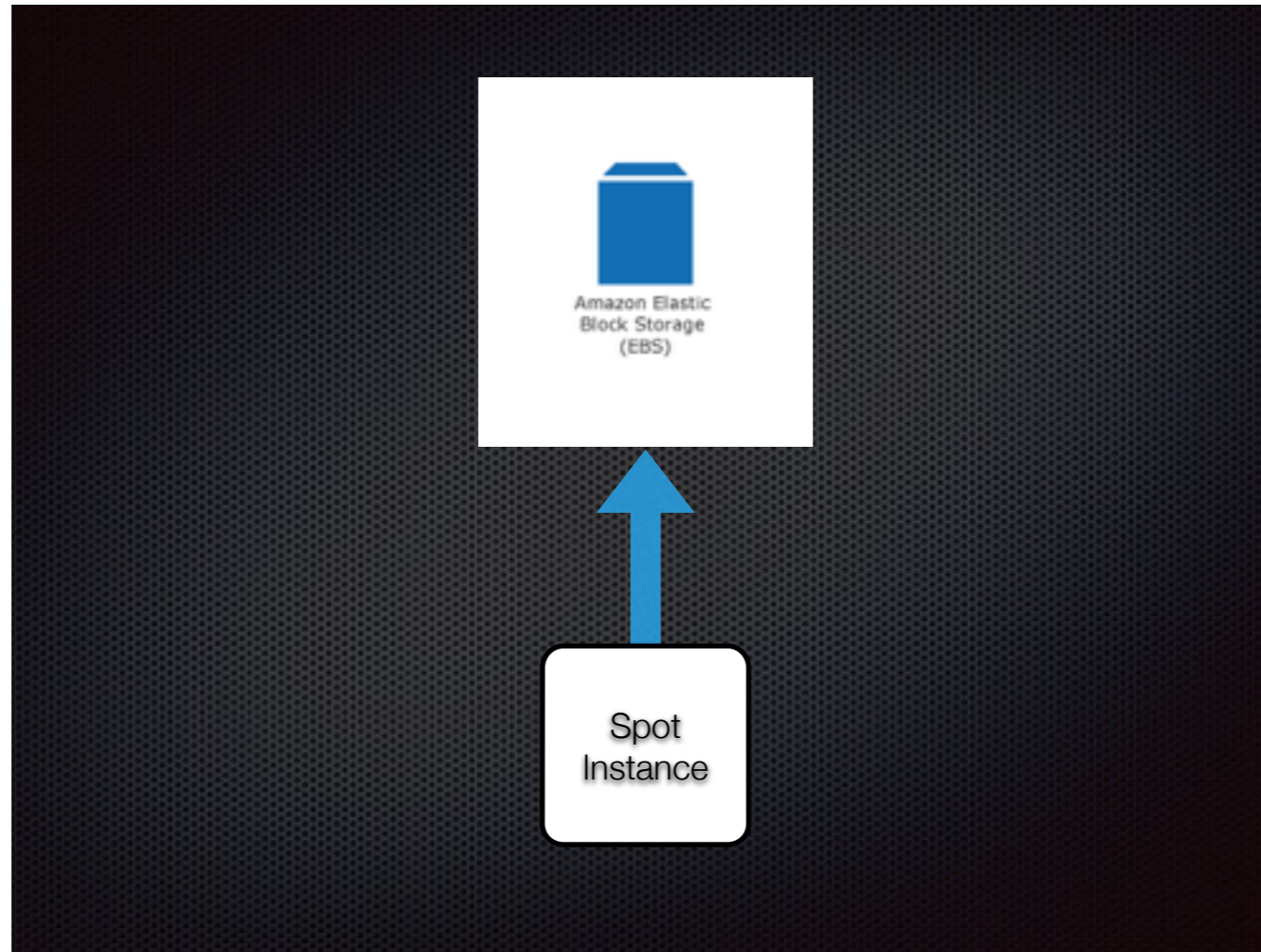
So GlusterFS is much pricier than EFS would've been and it's much pricier than just a simple EBS volume attached to a spot instance. Well, maybe it will prove to be negligible in the larger scheme of things maybe it won't be. We're in an experimental state of things and we're playing with our options. If for some reason we decide to not use Gluster FS and just go with simple EBS attach/reattach, we'll also have to un-distribute processing of the stages that would've been otherwise perfectly distributable, or else — we'll end up re-implementing some semblance of a DFS ourselves. Too messy and time consuming to bother with. And un-distribute might be ok if it saves us a lot of money. And maybe EFS will become available at some point and some brave hearts will even use it in their production env before we do. The reason why I'm bringing up the challenges and potential fluidity of our architecture and demonstrating to you all these scenarios is actually not to educate you about all this stuff, though if you found it interesting — awesome! If you didn't — thanks for your patience! The reason is — we run scientific workflows on top of it. That is running complexity on top of complexity. Equivalent of entering a perfect storm.



So GlusterFS is much pricier than EFS would've been and it's much pricier than just a simple EBS volume attached to a spot instance. Well, maybe it will prove to be negligible in the larger scheme of things maybe it won't be. We're in an experimental state of things and we're playing with our options. If for some reason we decide to not use Gluster FS and just go with simple EBS attach/reattach, we'll also have to un-distribute processing of the stages that would've been otherwise perfectly distributable, or else — we'll end up re-implementing some semblance of a DFS ourselves. Too messy and time consuming to bother with. And un-distribute might be ok if it saves us a lot of money. And maybe EFS will become available at some point and some brave hearts will even use it in their production env before we do. The reason why I'm bringing up the challenges and potential fluidity of our architecture and demonstrating to you all these scenarios is actually not to educate you about all this stuff, though if you found it interesting — awesome! If you didn't — thanks for your patience! The reason is — we run scientific workflows on top of it. That is running complexity on top of complexity. Equivalent of entering a perfect storm.



So GlusterFS is much pricier than EFS would've been and it's much pricier than just a simple EBS volume attached to a spot instance. Well, maybe it will prove to be negligible in the larger scheme of things maybe it won't be. We're in an experimental state of things and we're playing with our options. If for some reason we decide to not use Gluster FS and just go with simple EBS attach/reattach, we'll also have to un-distribute processing of the stages that would've been otherwise perfectly distributable, or else — we'll end up re-implementing some semblance of a DFS ourselves. Too messy and time consuming to bother with. And un-distribute might be ok if it saves us a lot of money. And maybe EFS will become available at some point and some brave hearts will even use it in their production env before we do. The reason why I'm bringing up the challenges and potential fluidity of our architecture and demonstrating to you all these scenarios is actually not to educate you about all this stuff, though if you found it interesting — awesome! If you didn't — thanks for your patience! The reason is — we run scientific workflows on top of it. That is running complexity on top of complexity. Equivalent of entering a perfect storm.

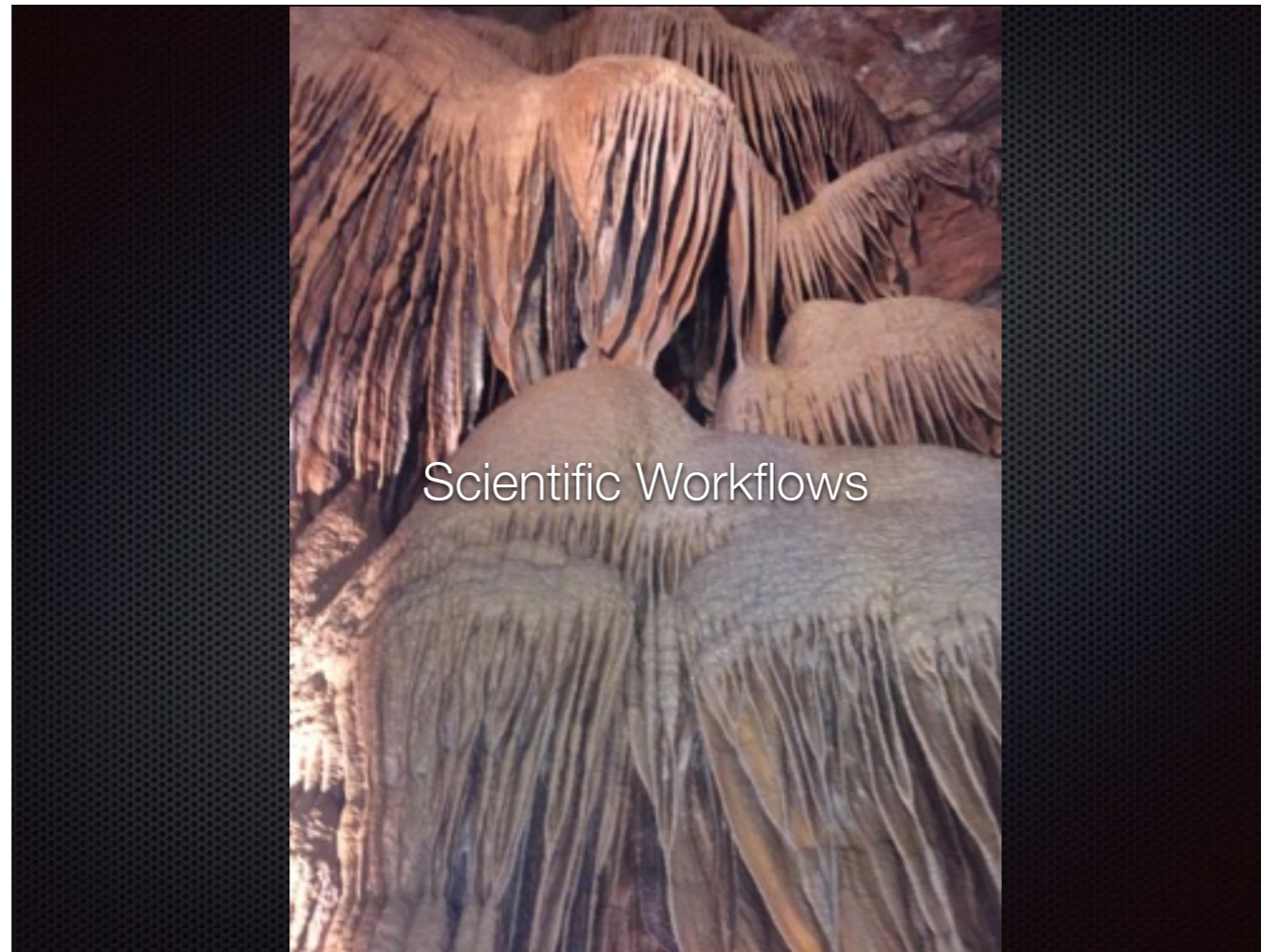


So GlusterFS is much pricier than EFS would've been and it's much pricier than just a simple EBS volume attached to a spot instance. Well, maybe it will prove to be negligible in the larger scheme of things maybe it won't be. We're in an experimental state of things and we're playing with our options. If for some reason we decide to not use Gluster FS and just go with simple EBS attach/reattach, we'll also have to un-distribute processing of the stages that would've been otherwise perfectly distributable, or else — we'll end up re-implementing some semblance of a DFS ourselves. Too messy and time consuming to bother with. And un-distribute might be ok if it saves us a lot of money. And maybe EFS will become available at some point and some brave hearts will even use it in their production env before we do. The reason why I'm bringing up the challenges and potential fluidity of our architecture and demonstrating to you all these scenarios is actually not to educate you about all this stuff, though if you found it interesting — awesome! If you didn't — thanks for your patience! The reason is — we run scientific workflows on top of it. That is running complexity on top of complexity. Equivalent of entering a perfect storm.



Scientific Workflows

Scientific workflows can get very complex. For some reason when I think about scientific workflows, this picture I took in a cave in Branson, Missouri always comes to mind. All these cascading map-reduces-feeding into more cascading map-reduces. If you don't get what this cave picture has to do with scientific workflows. Here is maybe a better example: Invocation graph for variant calling workflow I stole from Cuneiform paper. This stuff is complex, so if you're a scientist involved in creating the workflow, you have enough to worry about not to worry about how it will be distributed and how that part works. You need to abstract the backend part away so you can focus on your stuff. Or even if you're an architect helping out the scientists. You need to separate the architecture concerns and the workflow logic otherwise you'll end with a huge code-mess that's very difficult to support and troubleshoot. And lets say a decision to un-distribute a part of your workflow, whether based on your architecture-cost concerns or because it somehow messes up your results, it should be a matter of a small code-change in your workflow code. These things are actually sometimes very difficult to decide, so at least it shouldn't a pain in the neck to implement once your direction is clear.

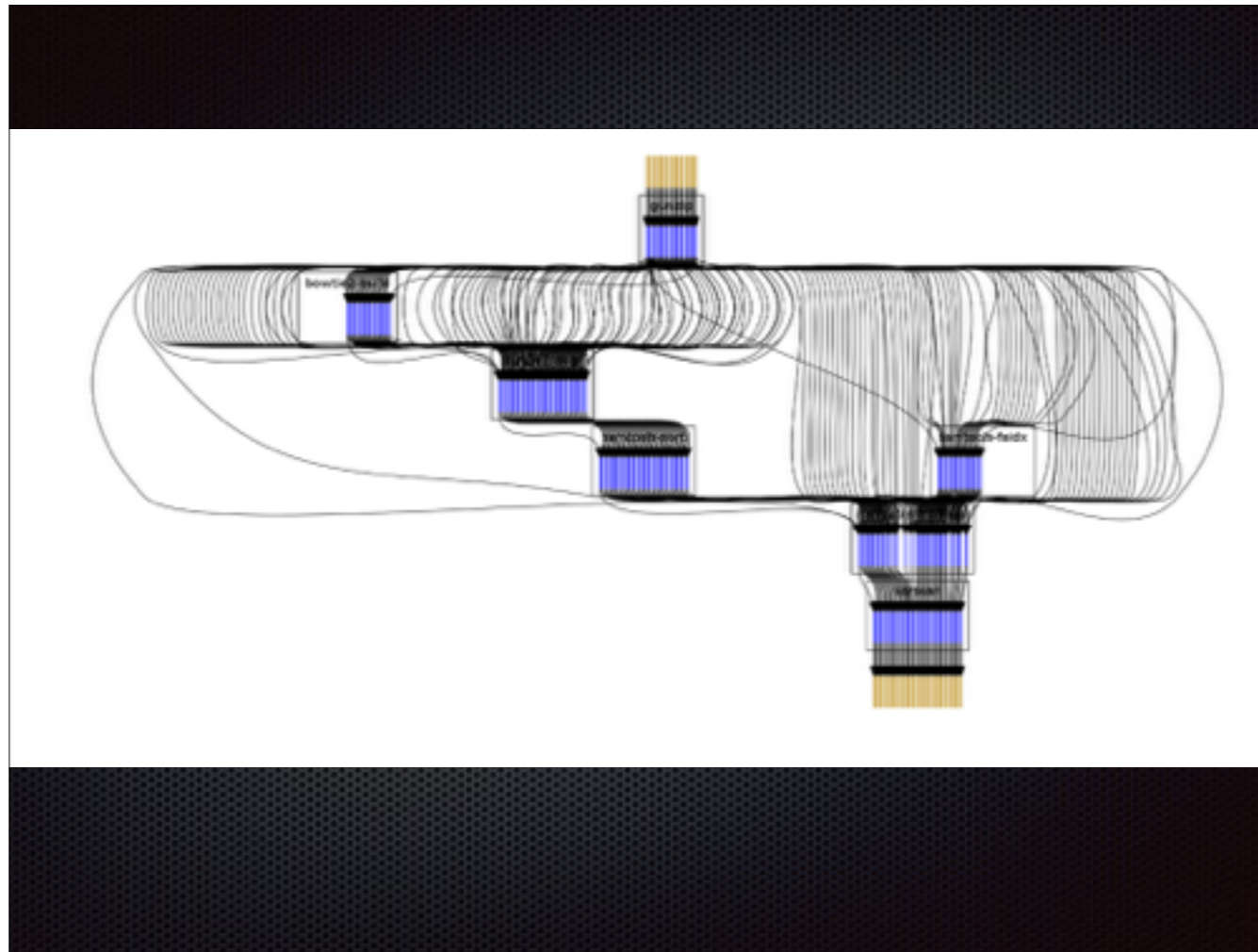


Scientific workflows can get very complex. For some reason when I think about scientific workflows, this picture I took in a cave in Branson, Missouri always comes to mind. All these cascading map-reduces-feeding into more cascading map-reduces. If you don't get what this cave picture has to do with scientific workflows. Here is maybe a better example: Invocation graph for variant calling workflow I stole from Cuneiform paper. This stuff is complex, so if you're a scientist involved in creating the workflow, you have enough to worry about not to worry about how it will be distributed and how that part works. You need to abstract the backend part away so you can focus on your stuff. Or even if you're an architect helping out the scientists. You need to separate the architecture concerns and the workflow logic otherwise you'll end with a huge code-mess that's very difficult to support and troubleshoot. And lets say a decision to un-distribute a part of your workflow, whether based on your architecture-cost concerns or because it somehow messes up your results, it should be a matter of a small code-change in your workflow code. These things are actually sometimes very difficult to decide, so at least it shouldn't a pain in the neck to implement once your direction is clear.

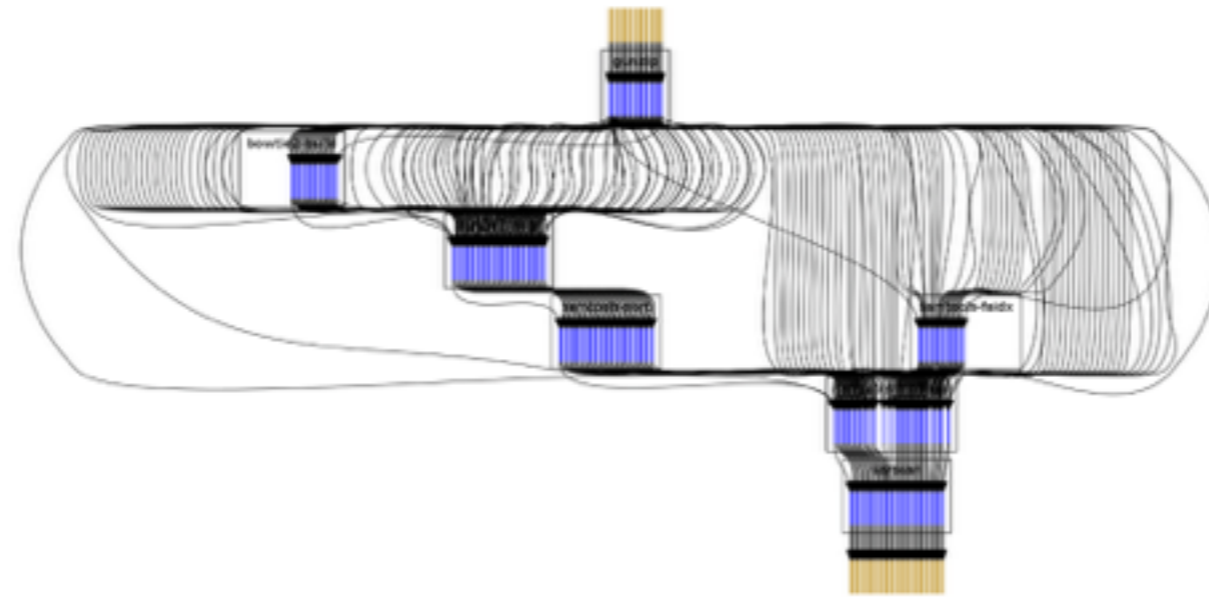


Scientific Workflows

Scientific workflows can get very complex. For some reason when I think about scientific workflows, this picture I took in a cave in Branson, Missouri always comes to mind. All these cascading map-reduces-feeding into more cascading map-reduces. If you don't get what this cave picture has to do with scientific workflows. Here is maybe a better example: Invocation graph for variant calling workflow I stole from Cuneiform paper. This stuff is complex, so if you're a scientist involved in creating the workflow, you have enough to worry about not to worry about how it will be distributed and how that part works. You need to abstract the backend part away so you can focus on your stuff. Or even if you're an architect helping out the scientists. You need to separate the architecture concerns and the workflow logic otherwise you'll end with a huge code-mess that's very difficult to support and troubleshoot. And lets say a decision to un-distribute a part of your workflow, whether based on your architecture-cost concerns or because it somehow messes up your results, it should be a matter of a small code-change in your workflow code. These things are actually sometimes very difficult to decide, so at least it shouldn't a pain in the neck to implement once your direction is clear.



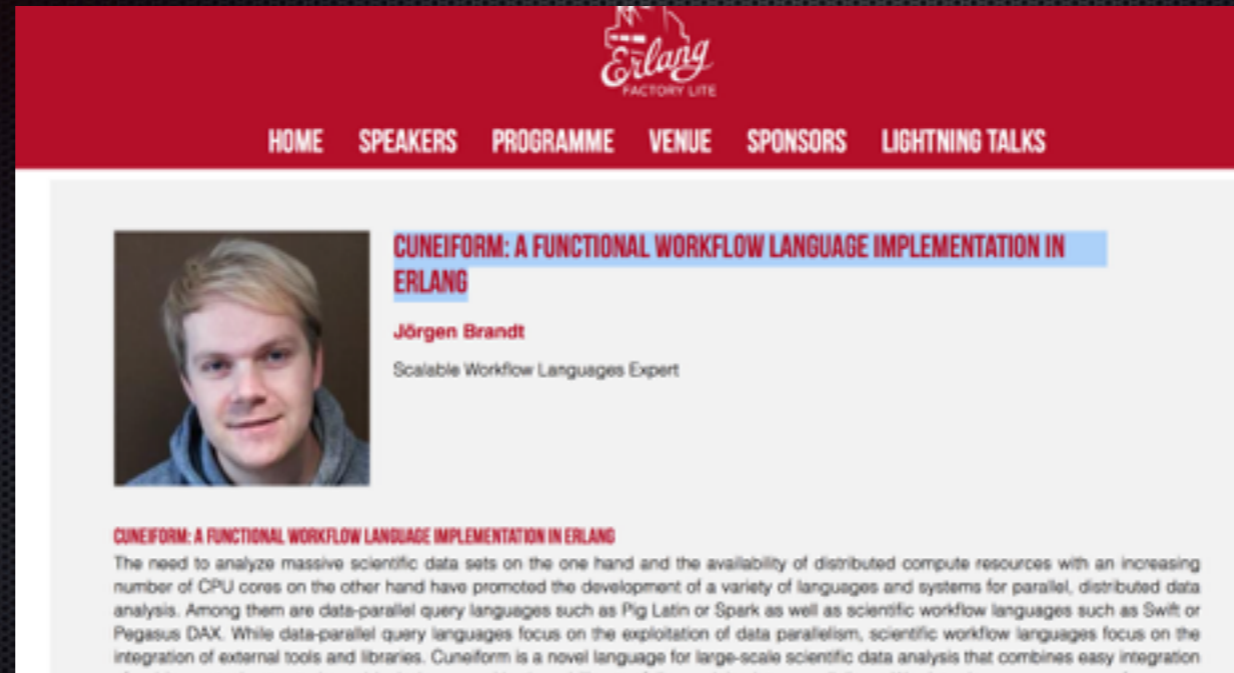
Scientific workflows can get very complex. For some reason when I think about scientific workflows, this picture I took in a cave in Branson, Missouri always comes to mind. All these cascading map-reduces-feeding into more cascading map-reduces. If you don't get what this cave picture has to do with scientific workflows. Here is maybe a better example: Invocation graph for variant calling workflow I stole from Cuneiform paper. This stuff is complex, so if you're a scientist involved in creating the workflow, you have enough to worry about not to worry about how it will be distributed and how that part works. You need to abstract the backend part away so you can focus on your stuff. Or even if you're an architect helping out the scientists. You need to separate the architecture concerns and the workflow logic otherwise you'll end with a huge code-mess that's very difficult to support and troubleshoot. And lets say a decision to un-distribute a part of your workflow, whether based on your architecture-cost concerns or because it somehow messes up your results, it should be a matter of a small code-change in your workflow code. These things are actually sometimes very difficult to decide, so at least it shouldn't a pain in the neck to implement once your direction is clear.



<http://ceur-ws.org/Vol-1330/paper-03.pdf>

Scientific workflows can get very complex. For some reason when I think about scientific workflows, this picture I took in a cave in Branson, Missouri always comes to mind. All these cascading map-reduces-feeding into more cascading map-reduces. If you don't get what this cave picture has to do with scientific workflows. Here is maybe a better example: Invocation graph for variant calling workflow I stole from Cuneiform paper. This stuff is complex, so if you're a scientist involved in creating the workflow, you have enough to worry about not to worry about how it will be distributed and how that part works. You need to abstract the backend part away so you can focus on your stuff. Or even if you're an architect helping out the scientists. You need to separate the architecture concerns and the workflow logic otherwise you'll end with a huge code-mess that's very difficult to support and troubleshoot. And lets say a decision to un-distribute a part of your workflow, whether based on your architecture-cost concerns or because it somehow messes up your results, it should be a matter of a small code-change in your workflow code. These things are actually sometimes very difficult to decide, so at least it shouldn't a pain in the neck to implement once your direction is clear.

CUNEIFORM



The screenshot shows the website for Erlang Factory Lite. At the top, there is a red navigation bar with the Erlang Factory Lite logo and menu items: HOME, SPEAKERS, PROGRAMME, VENUE, SPONSORS, and LIGHTNING TALKS. Below the navigation bar, there is a section for a talk by Jörgen Brandt. The talk title is "CUNEIFORM: A FUNCTIONAL WORKFLOW LANGUAGE IMPLEMENTATION IN ERLANG". The speaker's name is "Jörgen Brandt" and his title is "Scalable Workflow Languages Expert". There is a small portrait photo of Jörgen Brandt. Below the photo, there is a short paragraph of text describing the talk.

CUNEIFORM: A FUNCTIONAL WORKFLOW LANGUAGE IMPLEMENTATION IN ERLANG

Jörgen Brandt
Scalable Workflow Languages Expert

CUNEIFORM: A FUNCTIONAL WORKFLOW LANGUAGE IMPLEMENTATION IN ERLANG

The need to analyze massive scientific data sets on the one hand and the availability of distributed compute resources with an increasing number of CPU cores on the other hand have promoted the development of a variety of languages and systems for parallel, distributed data analysis. Among them are data-parallel query languages such as Pig Latin or Spark as well as scientific workflow languages such as Swift or Pegasus DAX. While data-parallel query languages focus on the exploitation of data parallelism, scientific workflow languages focus on the integration of external tools and libraries. Cuneiform is a novel language for large-scale scientific data analysis that combines easy integration

<http://www.erlang-factory.com/berlin2015/jorgen-brandt>

So one gloomy cloudy winter Sunday in Chicago I am running some very long-running, boring, and clumsy shell scripts which in turn are invoking all the fascinating stuff our lidar-and-image-processing teams create. And to entertain myself somehow I browse through my twitter feed and I run into a link to Berlin Erlang Factory Lite talk by Jörgen Brandt, Scalable Workflow Languages Expert. Cuneiform: A Functional Workflow Language Implementation in Erlang. So I watch the talk, I read up about it to understand what it's all about and slowly start getting fascinated. I feel like the sun is coming out and the winter is not so bad and life is not so bad either! There are other workflow DSLs out there and they didn't yet appeal to me, but this one definitely stands out.



1. I don't think I have to do much work to convince this audience about "Why Functional Programming Matters". I think someone in this conference already took care of this part ;)
2. Erlang. So, it can be trusted not to be the one thin abstraction layer on top of everything else that makes things much easier to develop and maintain, but then causes trouble in production — thank you very much! We have plenty of other reasons to fail, like hardware issues and such, we don't want the DSL to be the cause of problems. Even it might have bugs initially, I know they can be fixed and the implementation can quickly become bulletproof, and even if support of it is poor — I can fix it myself! But, support isn't poor, Joergen and his cuneiform colleagues are working really hard on this thing. Just like Phoenix and Elixir, Cuneiform has awesome support, at least in my experience. Though Erlang implementation is not production-ready at the moment, but it's very quickly getting there. It was originally written in Java, and because Java, IMHO, isn't the most suitable language for implementing a DSL, nor it is the best choice for supervising backend progress, the code base is a huge mess — at least comparing to Erlang. Once rewritten in Erlang it resulted in 80% less code.
3. And what really did it for me — one of the backends it supports is HTCondor! What is HTCondor. That's the the job scheduler we decided to use in our team to orchestrate our workflows. Why we chose that, and not Highway or Mesos or something else?
Let's take brief tour of HTCondor, before we continue on to Cuneiform.

1. It's functional!

1. I don't think I have to do much work to convince this audience about "Why Functional Programming Matters". I think someone in this conference already took care of this part ;)
2. Erlang. So, it can be trusted not to be the one thin abstraction layer on top of everything else that makes things much easier to develop and maintain, but then causes trouble in production — thank you very much! We have plenty of other reasons to fail, like hardware issues and such, we don't want the DSL to be the cause of problems. Even it might have bugs initially, I know they can be fixed and the implementation can quickly become bulletproof, and even if support of it is poor — I can fix it myself! But, support isn't poor, Joergen and his cuneiform colleagues are working really hard on this thing. Just like Phoenix and Elixir, Cuneiform has awesome support, at least in my experience. Though Erlang implementation is not production-ready at the moment, but it's very quickly getting there. It was originally written in Java, and because Java, IMHO, isn't the most suitable language for implementing a DSL, nor it is the best choice for supervising backend progress, the code base is a huge mess — at least comparing to Erlang. Once rewritten in Erlang it resulted in 80% less code.
3. And what really did it for me — one of the backends it supports is HTCondor! What is HTCondor. That's the the job scheduler we decided to use in our team to orchestrate our workflows. Why we chose that, and not Highway or Mesos or something else?
Let's take brief tour of HTCondor, before we continue on to Cuneiform.

1. It's functional!

**2. It's implemented in Erlang.
It can be trusted!**

1. I don't think I have to do much work to convince this audience about "Why Functional Programming Matters". I think someone in this conference already took care of this part ;)

2. Erlang. So, it can be trusted not to be the one thin abstraction layer on top of everything else that makes things much easier to develop and maintain, but then causes trouble in production — thank you very much! We have plenty of other reasons to fail, like hardware issues and such, we don't want the DSL to be the cause of problems. Even it might have bugs initially, I know they can be fixed and the implementation can quickly become bulletproof, and even if support of it is poor — I can fix it myself! But, support isn't poor, Joergen and his cuneiform colleagues are working really hard on this thing. Just like Phoenix and Elixir, Cuneiform has awesome support, at least in my experience. Though Erlang implementation is not production-ready at the moment, but it's very quickly getting there. It was originally written in Java, and because Java, IMHO, isn't the most suitable language for implementing a DSL, nor it is the best choice for supervising backend progress, the code base is a huge mess — at least comparing to Erlang. Once rewritten in Erlang it resulted in 80% less code.

3. And what really did it for me — one of the backends it supports is HTCondor! What is HTCondor. That's the the job scheduler we decided to use in our team to orchestrate our workflows. Why we chose that, and not Highway or Mesos or something else?

Let's take brief tour of HTCondor, before we continue on to Cuneiform.

1. It's functional!

**2. It's implemented in Erlang.
It can be trusted!**

3. It supports **HTCondor backend!**

1. I don't think I have to do much work to convince this audience about "Why Functional Programming Matters". I think someone in this conference already took care of this part ;)

2. Erlang. So, it can be trusted not to be the one thin abstraction layer on top of everything else that makes things much easier to develop and maintain, but then causes trouble in production — thank you very much! We have plenty of other reasons to fail, like hardware issues and such, we don't want the DSL to be the cause of problems. Even it might have bugs initially, I know they can be fixed and the implementation can quickly become bulletproof, and even if support of it is poor — I can fix it myself! But, support isn't poor, Joergen and his cuneiform colleagues are working really hard on this thing. Just like Phoenix and Elixir, Cuneiform has awesome support, at least in my experience. Though Erlang implementation is not production-ready at the moment, but it's very quickly getting there. It was originally written in Java, and because Java, IMHO, isn't the most suitable language for implementing a DSL, nor it is the best choice for supervising backend progress, the code base is a huge mess — at least comparing to Erlang. Once rewritten in Erlang it resulted in 80% less code.

3. And what really did it for me — one of the backends it supports is HTCondor! What is HTCondor. That's the the job scheduler we decided to use in our team to orchestrate our workflows. Why we chose that, and not Highway or Mesos or something else?

Let's take brief tour of HTCondor, before we continue on to Cuneiform.

HTCondor

1. HTCondor is a specialized workload management system for compute-intensive jobs. Like other full-featured batch systems, HTCondor provides a job queueing mechanism, scheduling policy, priority scheme, resource monitoring, and resource management. Users submit their serial or parallel jobs to HTCondor, HTCondor places them into a queue, chooses when and where to run the jobs based upon a policy, carefully monitors their progress, and ultimately informs the user upon completion.
2. To put in simpler terms: you're a scientist in need of running high-intensity workflows it can scavenge an entire university system for you to get the CPU cycles and memory you need to get job done. Thanks to its amazing match-making mechanism called Class Ads. A worker can be anything that chooses to advertise itself as a worker "Use me all you want as long as nobody is at the keyboard!" As for the job "Class Add", "Hey I need 8 cpus and 256M of memory, you got that for me?" And there is a negotiator to negotiate between the jobs and the machines, and there is a scheduler to schedule jobs when upstreams are complete and a suitable machine becomes available. And the master to keep the logs. Quite a theater! And in my experience with condor so far, the actors rehearsed their roles really well, the system is so darn reliable. So, here is a fun fact — no wonder LIGO lab used HTCondor

HTCondor

- a specialized workload management system for compute-intensive jobs

1. HTCondor is a specialized workload management system for compute-intensive jobs. Like other full-featured batch systems, HTCondor provides a job queueing mechanism, scheduling policy, priority scheme, resource monitoring, and resource management. Users submit their serial or parallel jobs to HTCondor, HTCondor places them into a queue, chooses when and where to run the jobs based upon a policy, carefully monitors their progress, and ultimately informs the user upon completion.
2. To put in simpler terms: you're a scientist in need of running high-intensity workflows it can scavenge an entire university system for you to get the CPU cycles and memory you need to get job done. Thanks to its amazing match-making mechanism called Class Ads. A worker can be anything that chooses to advertise itself as a worker "Use me all you want as long as nobody is at the keyboard!" As for the job "Class Add", "Hey I need 8 cpus and 256M of memory, you got that for me?" And there is a negotiator to negotiate between the jobs and the machines, and there is a scheduler to schedule jobs when upstreams are complete and a suitable machine becomes available. And the master to keep the logs. Quite a theater! And in my experience with condor so far, the actors rehearsed their roles really well, the system is so darn reliable. So, here is a fun fact — no wonder LIGO lab used HTCondor

HTCondor

- a specialized workload management system for compute-intensive jobs

Mesos and
Highway

1. HTCondor is a specialized workload management system for compute-intensive jobs. Like other full-featured batch systems, HTCondor provides a job queueing mechanism, scheduling policy, priority scheme, resource monitoring, and resource management. Users submit their serial or parallel jobs to HTCondor, HTCondor places them into a queue, chooses when and where to run the jobs based upon a policy, carefully monitors their progress, and ultimately informs the user upon completion.
2. To put in simpler terms: you're a scientist in need of running high-intensity workflows it can scavenge an entire university system for you to get the CPU cycles and memory you need to get job done. Thanks to its amazing match-making mechanism called Class Ads. A worker can be anything that chooses to advertise itself as a worker "Use me all you want as long as nobody is at the keyboard!" As for the job "Class Add", "Hey I need 8 cpus and 256M of memory, you got that for me?" And there is a negotiator to negotiate between the jobs and the machines, and there is a scheduler to schedule jobs when upstreams are complete and a suitable machine becomes available. And the master to keep the logs. Quite a theater! And in my experience with condor so far, the actors rehearsed their roles really well, the system is so darn reliable. So, here is a fun fact — no wonder LIGO lab used HTCondor

HTCondor

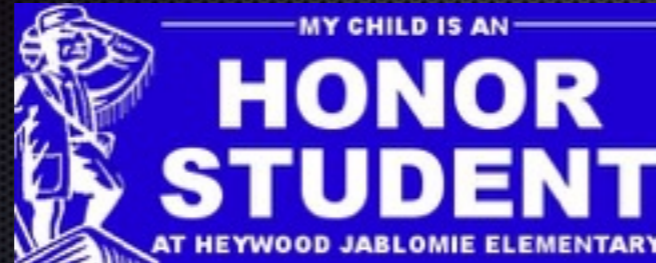
- a specialized workload management system for compute-intensive jobs



1. HTCondor is a specialized workload management system for compute-intensive jobs. Like other full-featured batch systems, HTCondor provides a job queueing mechanism, scheduling policy, priority scheme, resource monitoring, and resource management. Users submit their serial or parallel jobs to HTCondor, HTCondor places them into a queue, chooses when and where to run the jobs based upon a policy, carefully monitors their progress, and ultimately informs the user upon completion.
2. To put in simpler terms: you're a scientist in need of running high-intensity workflows it can scavenge an entire university system for you to get the CPU cycles and memory you need to get job done. Thanks to its amazing match-making mechanism called Class Ads. A worker can be anything that chooses to advertise itself as a worker "Use me all you want as long as nobody is at the keyboard!" As for the job "Class Add", "Hey I need 8 cpus and 256M of memory, you got that for me?" And there is a negotiator to negotiate between the jobs and the machines, and there is a scheduler to schedule jobs when upstreams are complete and a suitable machine becomes available. And the master to keep the logs. Quite a theater! And in my experience with condor so far, the actors rehearsed their roles really well, the system is so darn reliable. So, here is a fun fact — no wonder LIGO lab used HTCondor

HTCondor

- a specialized workload management system for compute-intensive jobs

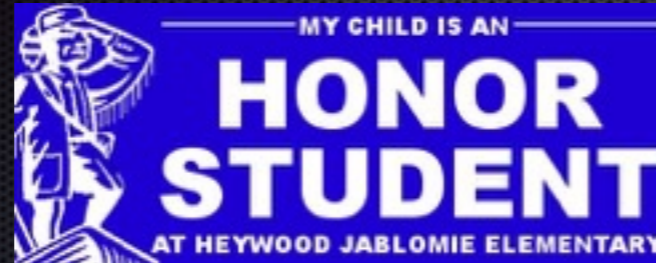


← Mesos and Highway

1. HTCondor is a specialized workload management system for compute-intensive jobs. Like other full-featured batch systems, HTCondor provides a job queueing mechanism, scheduling policy, priority scheme, resource monitoring, and resource management. Users submit their serial or parallel jobs to HTCondor, HTCondor places them into a queue, chooses when and where to run the jobs based upon a policy, carefully monitors their progress, and ultimately informs the user upon completion.
2. To put in simpler terms: you're a scientist in need of running high-intensity workflows it can scavenge an entire university system for you to get the CPU cycles and memory you need to get job done. Thanks to its amazing match-making mechanism called Class Ads. A worker can be anything that chooses to advertise itself as a worker "Use me all you want as long as nobody is at the keyboard!" As for the job "Class Add", "Hey I need 8 cpus and 256M of memory, you got that for me?" And there is a negotiator to negotiate between the jobs and the machines, and there is a scheduler to schedule jobs when upstreams are complete and a suitable machine becomes available. And the master to keep the logs. Quite a theater! And in my experience with condor so far, the actors rehearsed their roles really well, the system is so darn reliable. So, here is a fun fact — no wonder LIGO lab used HTCondor

HTCondor

- a specialized workload management system for compute-intensive jobs



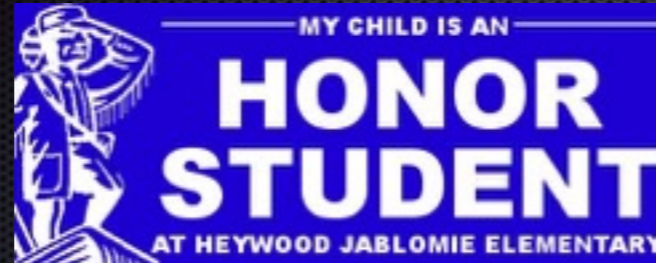
← Mesos and Highway



1. HTCondor is a specialized workload management system for compute-intensive jobs. Like other full-featured batch systems, HTCondor provides a job queueing mechanism, scheduling policy, priority scheme, resource monitoring, and resource management. Users submit their serial or parallel jobs to HTCondor, HTCondor places them into a queue, chooses when and where to run the jobs based upon a policy, carefully monitors their progress, and ultimately informs the user upon completion.
2. To put in simpler terms: you're a scientist in need of running high-intensity workflows it can scavenge an entire university system for you to get the CPU cycles and memory you need to get job done. Thanks to its amazing match-making mechanism called Class Ads. A worker can be anything that chooses to advertise itself as a worker "Use me all you want as long as nobody is at the keyboard!" As for the job "Class Add", "Hey I need 8 cpus and 256M of memory, you got that for me?" And there is a negotiator to negotiate between the jobs and the machines, and there is a scheduler to schedule jobs when upstreams are complete and a suitable machine becomes available. And the master to keep the logs. Quite a theater! And in my experience with condor so far, the actors rehearsed their roles really well, the system is so darn reliable. So, here is a fun fact — no wonder LIGO lab used HTCondor

HTCondor

- a specialized workload management system for compute-intensive jobs



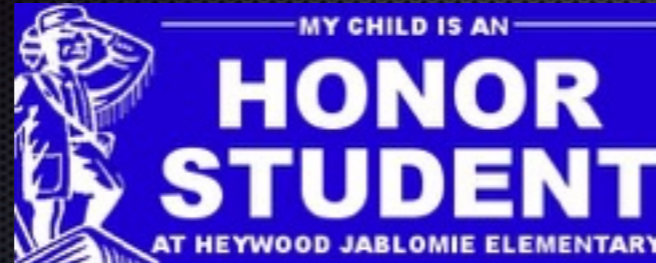
← Mesos and Highway



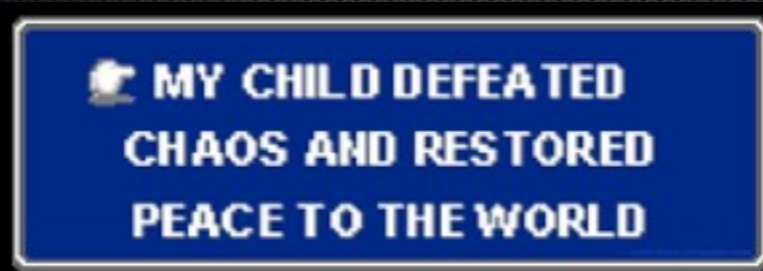
1. HTCondor is a specialized workload management system for compute-intensive jobs. Like other full-featured batch systems, HTCondor provides a job queueing mechanism, scheduling policy, priority scheme, resource monitoring, and resource management. Users submit their serial or parallel jobs to HTCondor, HTCondor places them into a queue, chooses when and where to run the jobs based upon a policy, carefully monitors their progress, and ultimately informs the user upon completion.
2. To put in simpler terms: you're a scientist in need of running high-intensity workflows it can scavenge an entire university system for you to get the CPU cycles and memory you need to get job done. Thanks to its amazing match-making mechanism called Class Ads. A worker can be anything that chooses to advertise itself as a worker "Use me all you want as long as nobody is at the keyboard!" As for the job "Class Add", "Hey I need 8 cpus and 256M of memory, you got that for me?" And there is a negotiator to negotiate between the jobs and the machines, and there is a scheduler to schedule jobs when upstreams are complete and a suitable machine becomes available. And the master to keep the logs. Quite a theater! And in my experience with condor so far, the actors rehearsed their roles really well, the system is so darn reliable. So, here is a fun fact — no wonder LIGO lab used HTCondor

HTCondor

- a specialized workload management system for compute-intensive jobs



← Mesos and Highway



1. HTCondor is a specialized workload management system for compute-intensive jobs. Like other full-featured batch systems, HTCondor provides a job queueing mechanism, scheduling policy, priority scheme, resource monitoring, and resource management. Users submit their serial or parallel jobs to HTCondor, HTCondor places them into a queue, chooses when and where to run the jobs based upon a policy, carefully monitors their progress, and ultimately informs the user upon completion.
2. To put in simpler terms: you're a scientist in need of running high-intensity workflows it can scavenge an entire university system for you to get the CPU cycles and memory you need to get job done. Thanks to its amazing match-making mechanism called Class Ads. A worker can be anything that chooses to advertise itself as a worker "Use me all you want as long as nobody is at the keyboard!" As for the job "Class Add", "Hey I need 8 cpus and 256M of memory, you got that for me?" And there is a negotiator to negotiate between the jobs and the machines, and there is a scheduler to schedule jobs when upstreams are complete and a suitable machine becomes available. And the master to keep the logs. Quite a theater! And in my experience with condor so far, the actors rehearsed their roles really well, the system is so darn reliable. So, here is a fun fact — no wonder LIGO lab used HTCondor

HTCondor

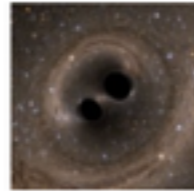
- a specialized workload management system for compute-intensive jobs



1. HTCondor is a specialized workload management system for compute-intensive jobs. Like other full-featured batch systems, HTCondor provides a job queueing mechanism, scheduling policy, priority scheme, resource monitoring, and resource management. Users submit their serial or parallel jobs to HTCondor, HTCondor places them into a queue, chooses when and where to run the jobs based upon a policy, carefully monitors their progress, and ultimately informs the user upon completion.
2. To put in simpler terms: you're a scientist in need of running high-intensity workflows it can scavenge an entire university system for you to get the CPU cycles and memory you need to get job done. Thanks to its amazing match-making mechanism called Class Ads. A worker can be anything that chooses to advertise itself as a worker "Use me all you want as long as nobody is at the keyboard!" As for the job "Class Add", "Hey I need 8 cpus and 256M of memory, you got that for me?" And there is a negotiator to negotiate between the jobs and the machines, and there is a scheduler to schedule jobs when upstreams are complete and a suitable machine becomes available. And the master to keep the logs. Quite a theater! And in my experience with condor so far, the actors rehearsed their roles really well, the system is so darn reliable. So, here is a fun fact — no wonder LIGO lab used HTCondor



OSG helps LIGO scientists confirm Einstein's unproven theory



Albert Einstein first posed the idea of gravitational waves in his **general theory of relativity** just over a century ago. But until now, they had never been observed directly. For the first time, scientists with the Laser Interferometer Gravitational-Wave Observatory (LIGO) Scientific Collaboration (LSC) have observed ripples in the fabric of spacetime called **gravitational waves**.

Image Courtesy Caltech/MIT/LIGO Laboratory

LIGO consists of two observatories within the United States—one in Hanford, Washington and the other in Livingston, Louisiana—separated by 1,865 miles. LIGO's detectors search for gravitational waves from deep space. With two detectors, researchers can use differences in the wave's arrival times to constrain the source location in the sky. LIGO's first data run of its advanced gravitational wave detectors began in September 2015 and ran through January 12, 2016. The first gravitational waves were detected on September 14, 2015 by both detectors.

The LIGO project employs many concepts that the Open Science Grid (OSG) promotes—resource sharing, aggregating opportunistic use across a variety of resources—and adds two twists: First, this experiment ran across LIGO Data Grid (LDG), OSG and Extreme Science and Engineering Discovery Environment (XSEDE)-based resources, all managed from a single HTCondor-based system to take advantage of dedicated LDG, opportunistic OSG and NSF eXtreme Digital (XD) allocations. Second, workflows analyzing LIGO detector data proved more data-intensive than many opportunistic OSG workflows. Despite these challenges, LIGO scientists were able to manage workflows with the same tools they use to run on dedicated LDG systems—Pegasus and HTCondor.

Peter Couvares, data analysis computing manager for the Advanced LIGO project at Caltech, specializes in distributed computing problems. He and colleagues James Clark (Georgia Tech) and Lame Pekowsky (Syracuse University) explained LIGO's computing needs and environment: The main focus is on optimization of data analysis codes, where optimization is broadly defined to encompass the overall performance and efficiency of their computing. While they use traditional optimization techniques to make things run faster, they also pursue more efficient resource management, and opportunistic resources—if there are computers available, they try to use them—thus the collaboration with OSG.

<http://www.opensciencegrid.org/osg-helps-ligo-scientists-confirm-einsteins-last-unproven-theory/>

1. Laser Interferometer Gravitational-Wave Observatory (LIGO) Scientific Collaboration (LSC) have observed ripples in the fabric of spacetime called gravitational waves.



To enable checkpointing, the user must link the program with the Condor system call library (libcondorsyscall.a), using the `condor_compile` command

2. HTC Condor is a 15-year-old system, the product of years of research by the Center for High Throughput Computing in the Department of Computer Sciences at the University of Wisconsin-Madison (UW-Madison)
 1. Condor doesn't optimize for fast job startup, but for high throughput. That's important to remember when you consider it for your use case.
 2. It's old, but it's very modern as it has great support. For instance, it has many different universes the job can run in, and one of the newest supported universes is "Docker". It's new kid on the block and condor supports it.
 3. One and only downside: there is a lot to learn, somewhat unintuitive syntax and setup at first. A lot of people hate DAGMAN the workflow DSL condor provides. Thousands of pages of tutorials (though that is justified — condor is a powerhouse of functionality, it's got to be documented somewhere).
 4. Super cool feature, Condor's Checkpoint Mechanism. Checkpointing is taking a snapshot of the current state of a program in such a way that the program can be restarted from that state at a later time. Checkpointing gives the Condor scheduler the freedom to reconsider scheduling decisions through preemptive-resume scheduling. Remember we mentioned job re-entrance essential if running on AWS Spot instances?



University of Wisconsin-Madison

To enable checkpointing, the user must link the program with the Condor system call library (libcondorsyscall.a), using the `condor_compile` command

2. HTC Condor is a 15-year-old system, the product of years of research by the Center for High Throughput Computing in the Department of Computer Sciences at the University of Wisconsin-Madison (UW-Madison)
 1. Condor doesn't optimize for fast job startup, but for high throughput. That's important to remember when you consider it for your use case.
 2. It's old, but it's very modern as it has great support. For instance, it has many different universes the job can run in, and one of the newest supported universes is "Docker". It's new kid on the block and condor supports it.
 3. One and only downside: there is a lot to learn, somewhat unintuitive syntax and setup at first. A lot of people hate DAGMAN the workflow DSL condor provides. Thousands of pages of tutorials (though that is justified — condor is a powerhouse of functionality, it's got to be documented somewhere).
 4. Super cool feature, Condor's Checkpoint Mechanism. Checkpointing is taking a snapshot of the current state of a program in such a way that the program can be restarted from that state at a later time. Checkpointing gives the Condor scheduler the freedom to reconsider scheduling decisions through preemptive-resume scheduling. Remember we mentioned job re-entrance essential if running on AWS Spot instances?



University of Wisconsin-Madison

Condor doesn't optimize for fast job startup,
but for high throughput

To enable checkpointing, the user must link the program with the Condor system call library (libcondorsyscall.a), using the `condor_compile` command

2. HTC Condor is a 15-year-old system, the product of years of research by the Center for High Throughput Computing in the Department of Computer Sciences at the University of Wisconsin-Madison (UW-Madison)
1. Condor doesn't optimize for fast job startup, but for high throughput. That's important to remember when you consider it for your use case.
2. It's old, but it's very modern as it has great support. For instance, it has many different universes the job can run in, and one of the newest supported universes is "Docker". It's new kid on the block and condor supports it.
3. One and only downside: there is a lot to learn, somewhat unintuitive syntax and setup at first. A lot of people hate DAGMAN the workflow DSL condor provides. Thousands of pages of tutorials (though that is justified — condor is a powerhouse of functionality, it's got to be documented somewhere).
4. Super cool feature, Condor's Checkpoint Mechanism. Checkpointing is taking a snapshot of the current state of a program in such a way that the program can be restarted from that state at a later time. Checkpointing gives the Condor scheduler the freedom to reconsider scheduling decisions through preemptive-resume scheduling. Remember we mentioned job re-entrance essential if running on AWS Spot instances?



University of Wisconsin-Madison

Condor doesn't optimize for fast job startup,
but for high throughput

It's 15 years old, yet very modern

To enable checkpointing, the user must link the program with the Condor system call library (libcondorsyscall.a), using the `condor_compile` command

2. HTC Condor is a 15-year-old system, the product of years of research by the Center for High Throughput Computing in the Department of Computer Sciences at the University of Wisconsin-Madison (UW-Madison)
1. Condor doesn't optimize for fast job startup, but for high throughput. That's important to remember when you consider it for your use case.
2. It's old, but it's very modern as it has great support. For instance, it has many different universes the job can run in, and one of the newest supported universes is "Docker". It's new kid on the block and condor supports it.
3. One and only downside: there is a lot to learn, somewhat unintuitive syntax and setup at first. A lot of people hate DAGMAN the workflow DSL condor provides. Thousands of pages of tutorials (though that is justified — condor is a powerhouse of functionality, it's got to be documented somewhere).
4. Super cool feature, Condor's Checkpoint Mechanism. Checkpointing is taking a snapshot of the current state of a program in such a way that the program can be restarted from that state at a later time. Checkpointing gives the Condor scheduler the freedom to reconsider scheduling decisions through preemptive-resume scheduling. Remember we mentioned job re-entrance essential if running on AWS Spot instances?



University of Wisconsin-Madison

Condor doesn't optimize for fast job startup,
but for high throughput

It's 15 years old, yet very modern

Condor's Checkpoint Mechanism

To enable checkpointing, the user must link the program with the Condor system call library (libcondorsyscall.a), using the `condor_compile` command

2. HTCCondor is a 15-year-old system, the product of years of research by the Center for High Throughput Computing in the Department of Computer Sciences at the University of Wisconsin-Madison (UW-Madison)
1. Condor doesn't optimize for fast job startup, but for high throughput. That's important to remember when you consider it for your use case.
2. It's old, but it's very modern as it has great support. For instance, it has many different universes the job can run in, and one of the newest supported universes is "Docker". It's new kid on the block and condor supports it.
3. One and only downside: there is lot to learn, somewhat unintuitive syntax and setup at first. A lot of people hate DAGMAN the workflow DSL condor provides. Thousands of pages of tutorials (though that is justified — condor is a powerhouse of functionality, it's got to be documented somewhere).
4. Super cool feature, Condor's Checkpoint Mechanism. Checkpointing is taking a snapshot of the current state of a program in such a way that the program can be restarted from that state at a later time. Checkpointing gives the Condor scheduler the freedom to reconsider scheduling decisions through preemptive-resume scheduling. Remember we mentioned job re-entrance essential if running on AWS Spot instances?

- 1. It's functional!**
- 2. It's implemented in Erlang.
It can be trusted!**
- 3. It supports HTCondor backend!**

So where were we? It supports HTCondor backend... but also Highway and a local backend, and possibly some cuneiform-native Erlang distributed backend, and maybe Mesos in the future. Anything is possible with cuneiform modular and pluggable architecture.

Beautiful logical syntax — I know you can't wait to see it, but no worries, we'll be looking at that stuff for the rest of the presentation. And honestly, there isn't much to it, it's dead simple. All of the heavy lifting is in:

5. Foreign language support... Bash, Python, R... what have you. Well, Bash, Python, R is what we have. So I don't care much about others at this point, but there are inquiries about adding Ruby, etc., I'm sure it would be easy as pie to add elixir and especially erlang support in the future. Cuneiform architecture is very modular and pluggable, let's take a look:

- 1. It's functional!**
- 2. It's implemented in Erlang.
It can be trusted!**
- 3. It supports HTCondor backend!**

So where were we? It supports HTCondor backend... but also Highway and a local backend, and possibly some cuneiform-native Erlang distributed backend, and maybe Mesos in the future. Anything is possible with cuneiform modular and pluggable architecture.

Beautiful logical syntax — I know you can't wait to see it, but no worries, we'll be looking at that stuff for the rest of the presentation. And honestly, there isn't much to it, it's dead simple. All of the heavy lifting is in:

5. Foreign language support... Bash, Python, R... what have you. Well, Bash, Python, R is what we have. So I don't care much about others at this point, but there are inquiries about adding Ruby, etc., I'm sure it would be easy as pie to add elixir and especially erlang support in the future. Cuneiform architecture is very modular and pluggable, let's take a look:

- 1. It's functional!**
- 2. It's implemented in Erlang.
It can be trusted!**
- 3. It supports HTCondor backend!**
- 4. Beautiful simple logical syntax.**

So where were we? It supports HTCondor backend... but also Highway and a local backend, and possibly some cuneiform-native Erlang distributed backend, and maybe Mesos in the future. Anything is possible with cuneiform modular and pluggable architecture.

Beautiful logical syntax — I know you can't wait to see it, but no worries, we'll be looking at that stuff for the rest of the presentation. And honestly, there isn't much to it, it's dead simple. All of the heavy lifting is in:

5. Foreign language support... Bash, Python, R... what have you. Well, Bash, Python, R is what we have. So I don't care much about others at this point, but there are inquiries about adding Ruby, etc., I'm sure it would be easy as pie to add elixir and especially erlang support in the future. Cuneiform architecture is very modular and pluggable, let's take a look:

- 1. It's functional!**
- 2. It's implemented in Erlang.
It can be trusted!**
- 3. It supports HTCondor backend!**
- 4. Beautiful simple logical syntax.**
- 5. Foreign language support: Bash,
Python, R...**

So where were we? It supports HTCondor backend... but also Highway and a local backend, and possibly some cuneiform-native Erlang distributed backend, and maybe Mesos in the future. Anything is possible with cuneiform modular and pluggable architecture.

Beautiful logical syntax — I know you can't wait to see it, but no worries, we'll be looking at that stuff for the rest of the presentation. And honestly, there isn't much to it, it's dead simple. All of the heavy lifting is in:

5. Foreign language support... Bash, Python, R... what have you. Well, Bash, Python, R is what we have. So I don't care much about others at this point, but there are inquiries about adding Ruby, etc., I'm sure it would be easy as pie to add elixir and especially erlang support in the future. Cuneiform architecture is very modular and pluggable, let's take a look:

Cuneiform Architecture

EFFI

CUNEIFORM

CRE

Python

Bash

R

Cuneiform DSL

HTcondor

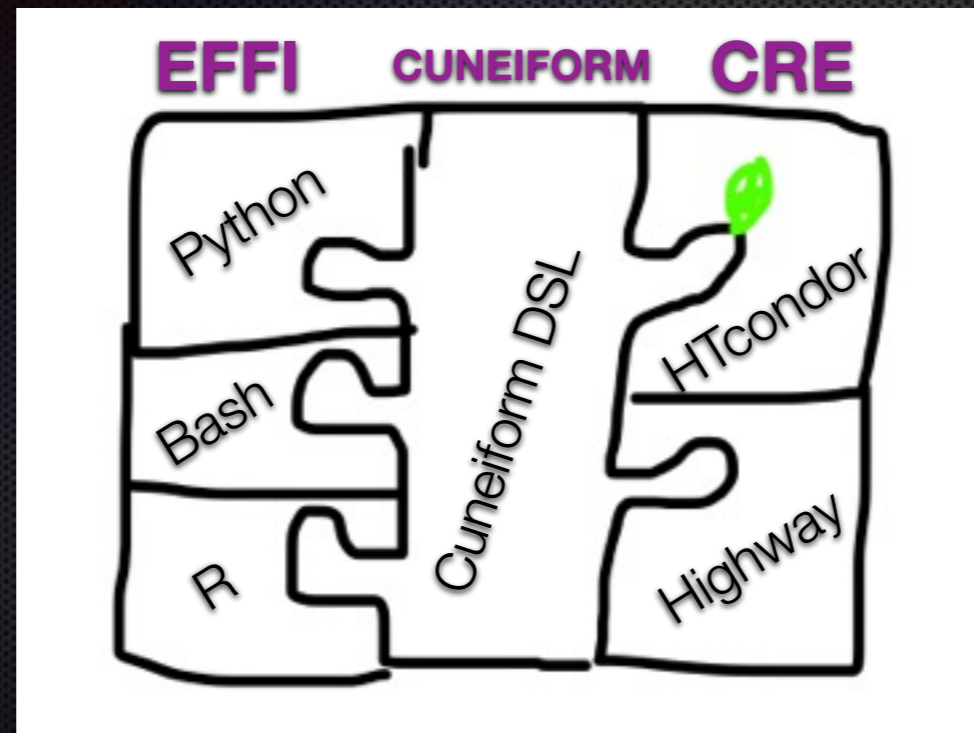
Highway

Three repos in GitHub

1. Cuneiform itself, that has the actual DSL semantics and parsing.
2. EFFI: Foreign Function Interface. For foreign language support.
3. CRE: Distributed backends. Local, Highway, HTCondor, maybe Mesos and native Erlang backend in the future?

So, whew, I am done prepping you for the actual thing. Hope you're very excited, because I'm about to show you the super-awesome Cuneiform stuff. Let's open the curtains!

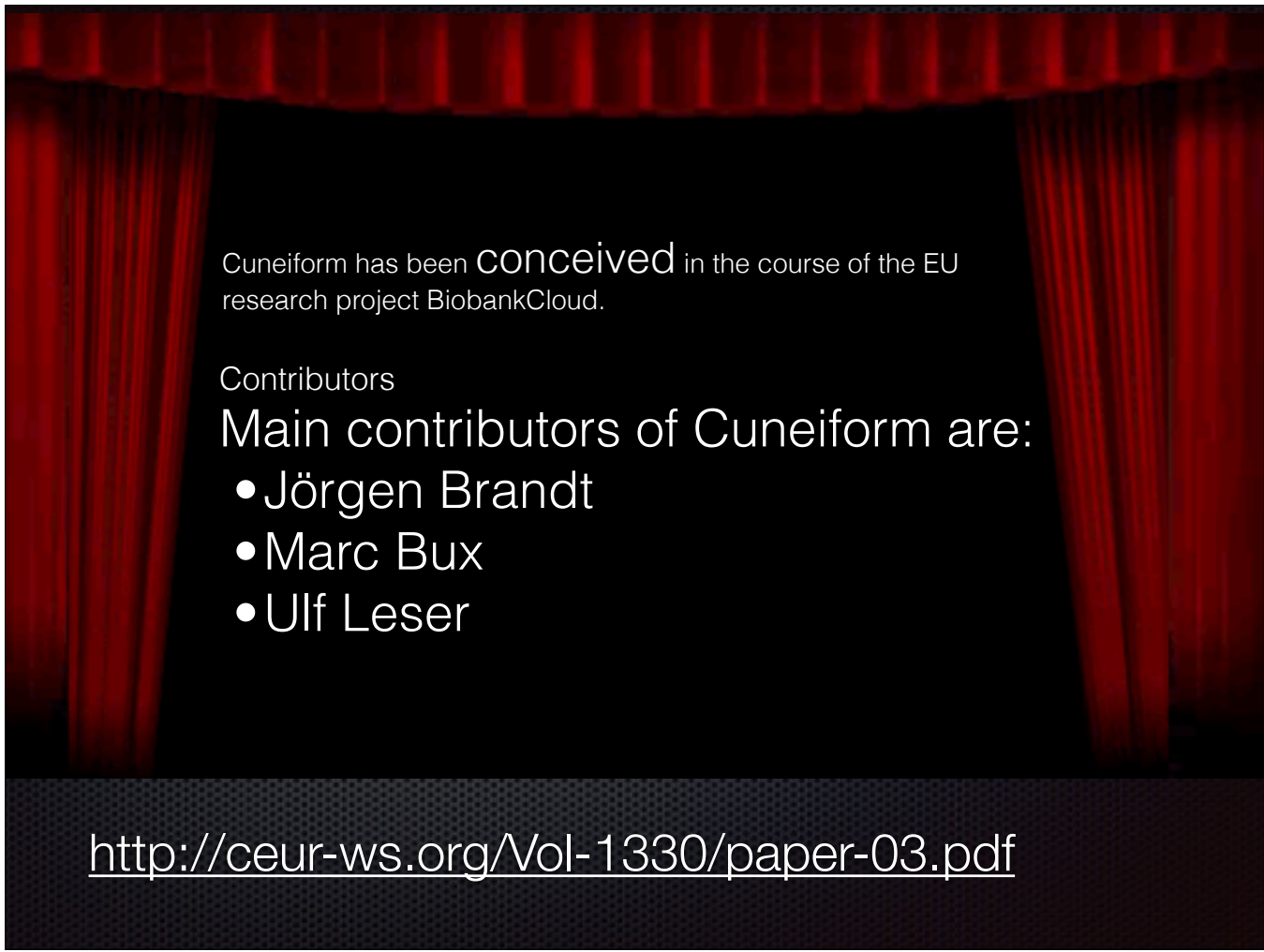
Cuneiform Architecture



Three repos in GitHub

1. Cuneiform itself, that has the actual DSL semantics and parsing.
2. EFFI: Foreign Function Interface. For foreign language support.
3. CRE: Distributed backends. Local, Highway, HTCondor, maybe Mesos and native Erlang backend in the future?

So, whew, I am done prepping you for the actual thing. Hope you're very excited, because I'm about to show you the super-awesome Cuneiform stuff. Let's open the curtains!



Cuneiform has been **conceived** in the course of the EU research project BiobankCloud.

Contributors

Main contributors of Cuneiform are:

- Jörgen Brandt
- Marc Bux
- Ulf Leser

<http://ceur-ws.org/Vol-1330/paper-03.pdf>

Switch to cuneiform editor for syntax demo...

Condor submit file example

```
executable = cf-generated-submit-script
universe = vanilla
input = test.data
output = script.stdout
error = script.stderr
log = condor.log

initialdir = run_0
queue
```

```
> condor_submit submitfile
```

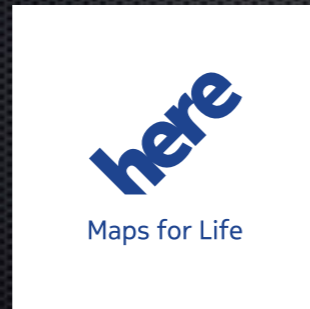
Cuneiform will capture any custom parameters from the CF file and generate the submit file.

If no parameters present it'll just go with the defaults, i.e. universe=vanilla. As for executable that gets wrapped by Cuneiform depending on what language it is in and will be named automatically (i.e. cfsubmitfile) and added to this generated file.

Thank you!

Jürgen Brandt

Josh Flachsbart and the A&I team.



Twitter: @irina_guberman