SCALING UBER BY BREAKING EVERYTHING

MATT RANNEY

UBER

As of October 2015:

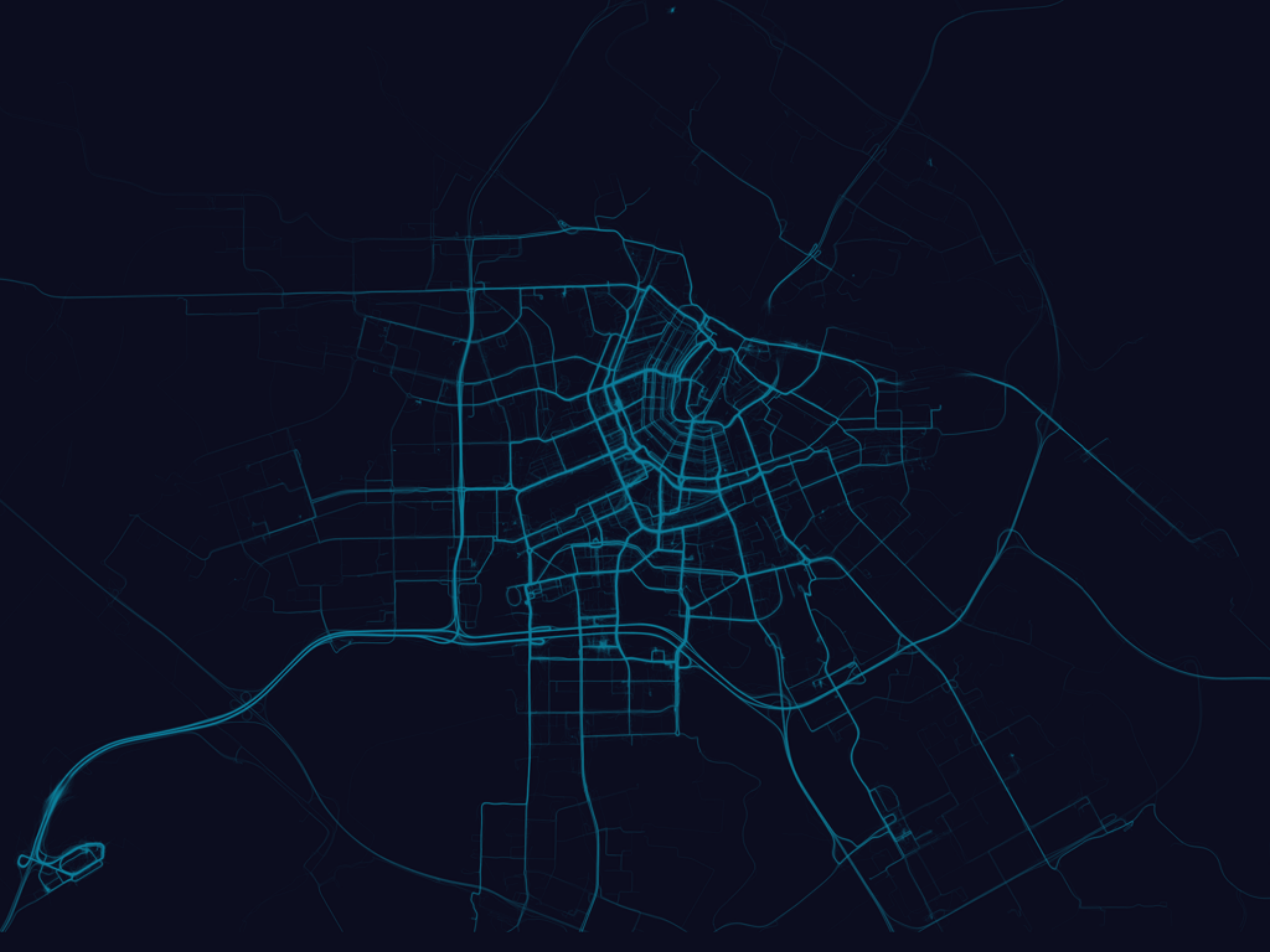
Cities Worldwide: 340

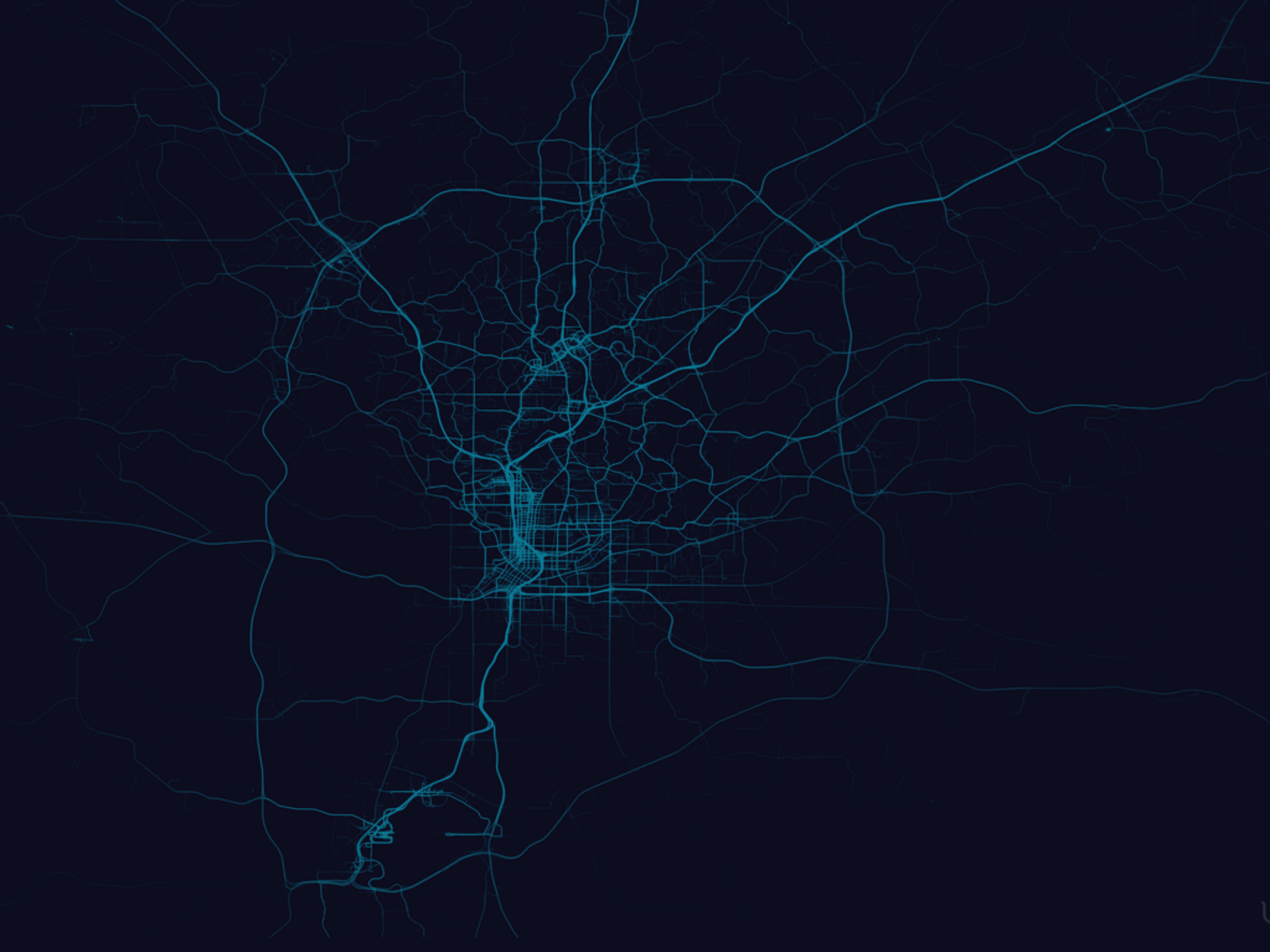
Number of Countries: 61

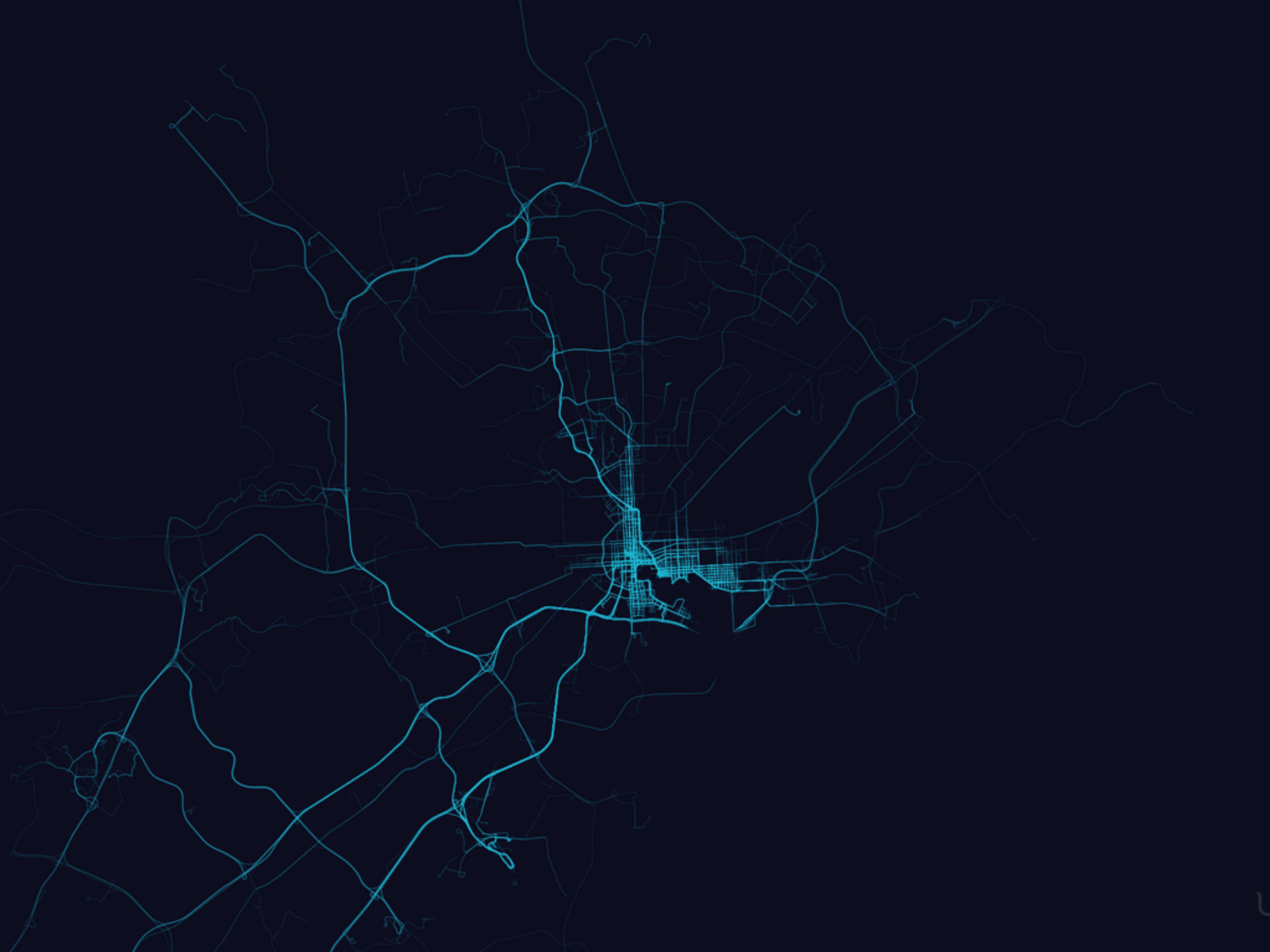
Employees: 4,000+

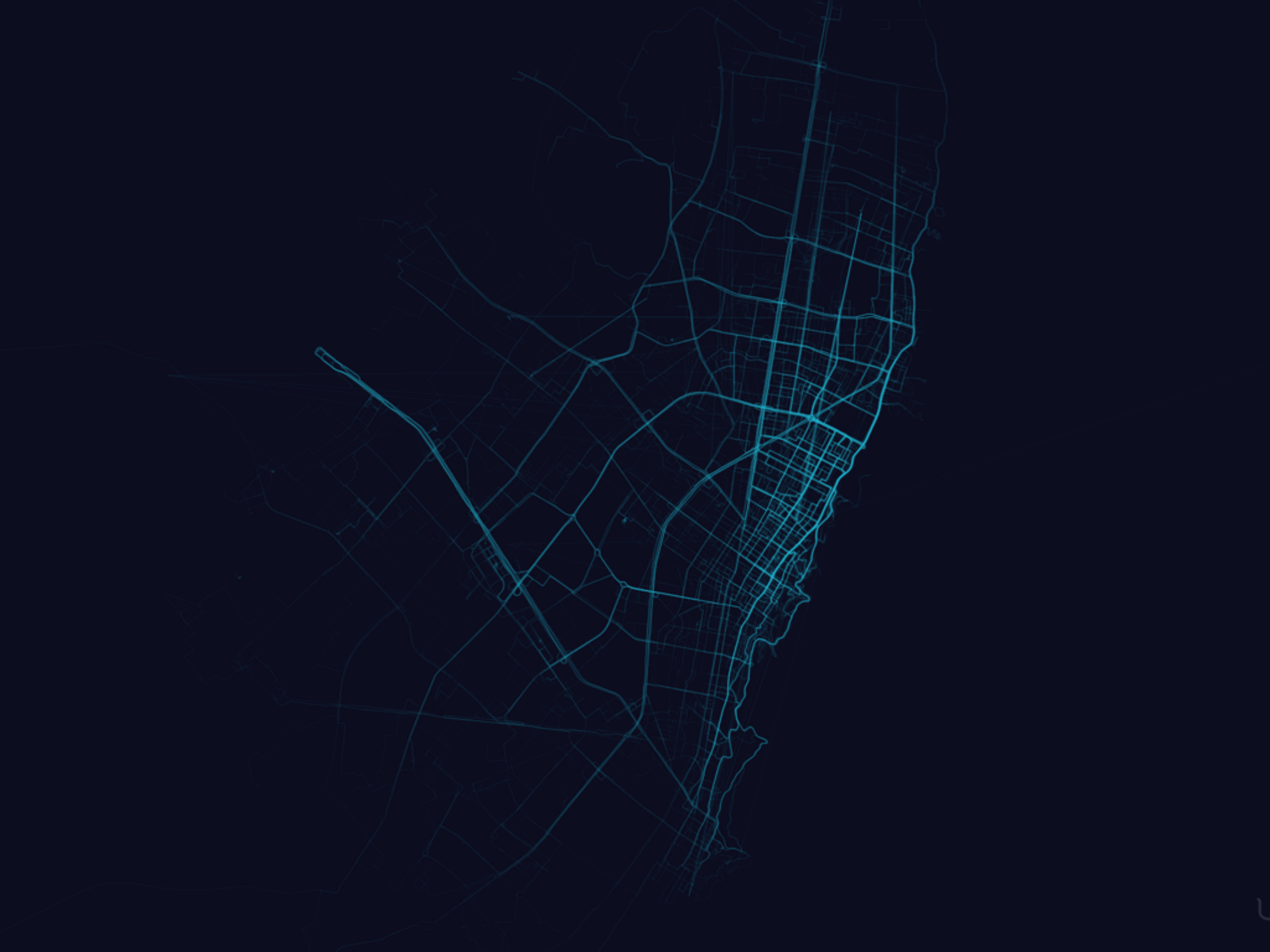
Engineers: 1000+

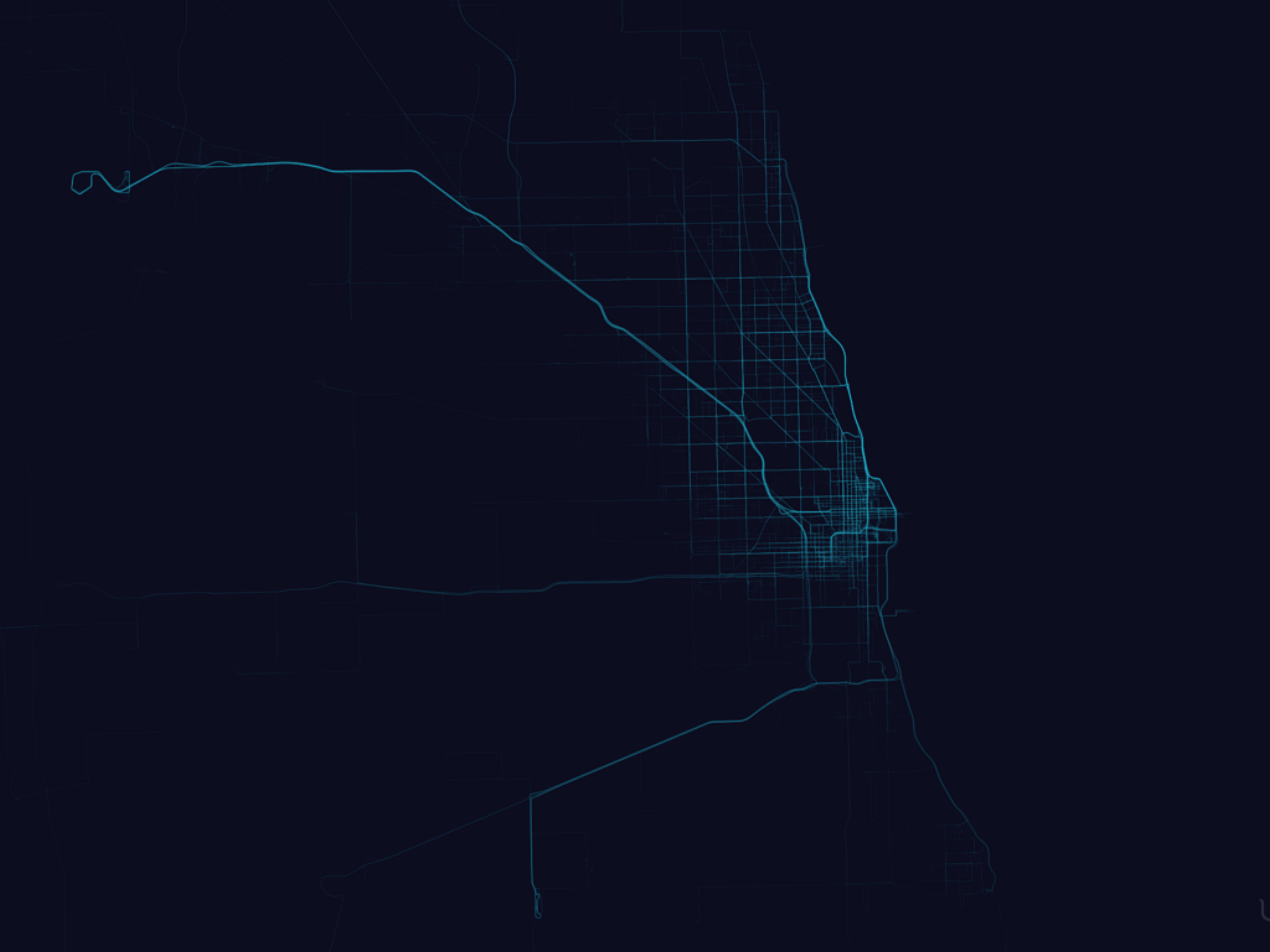
Trips per Day: 3,000,000

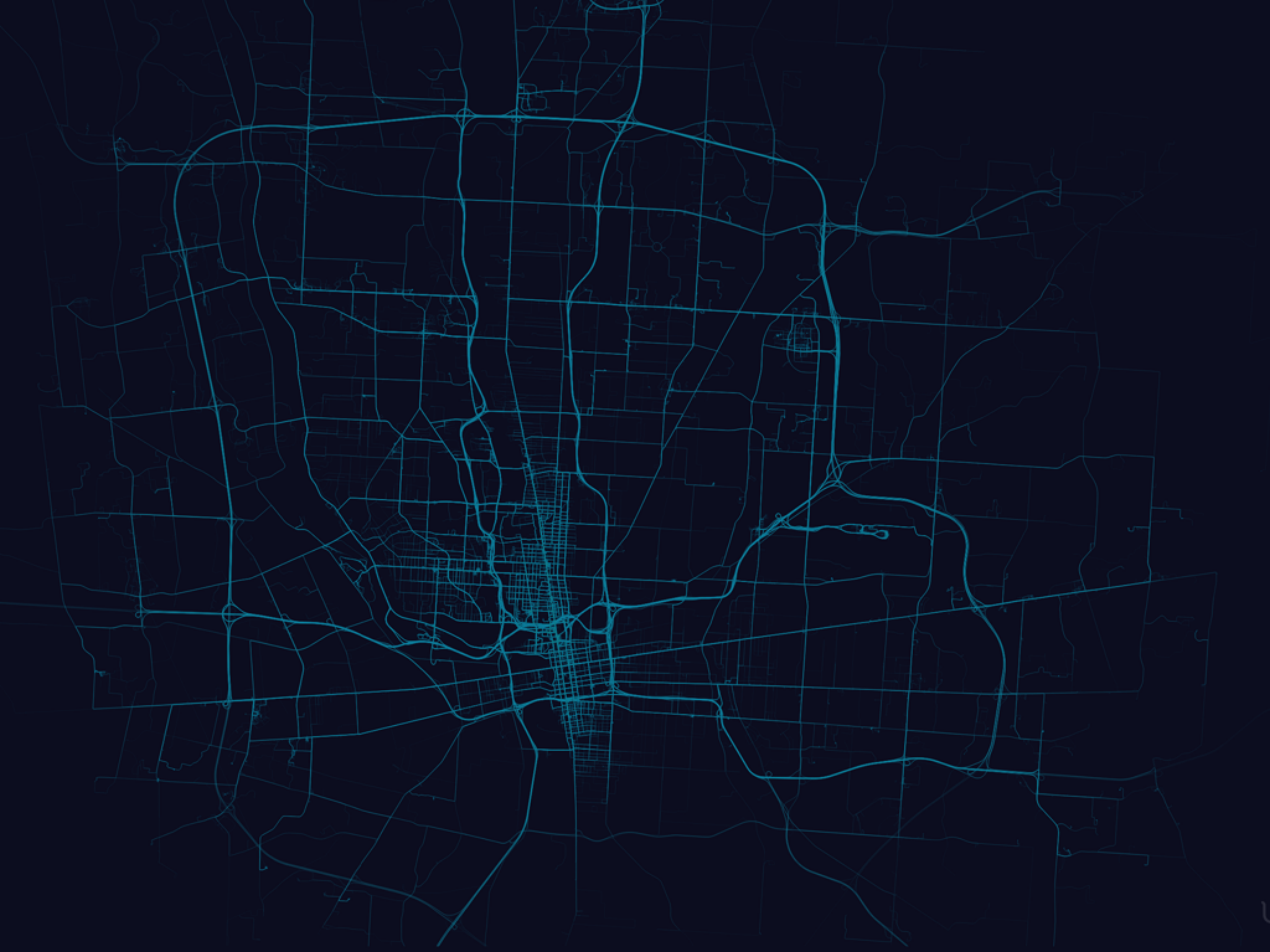


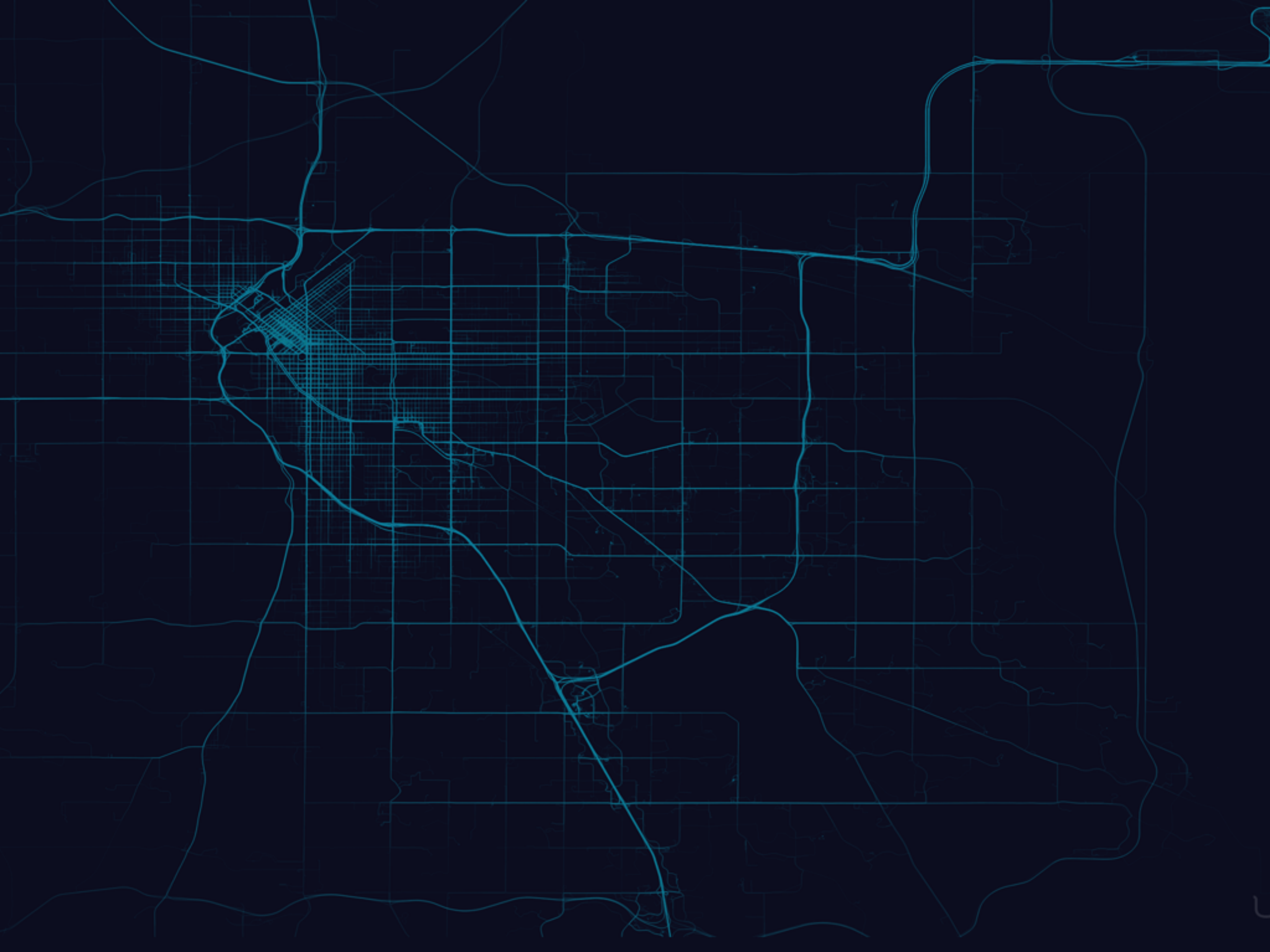


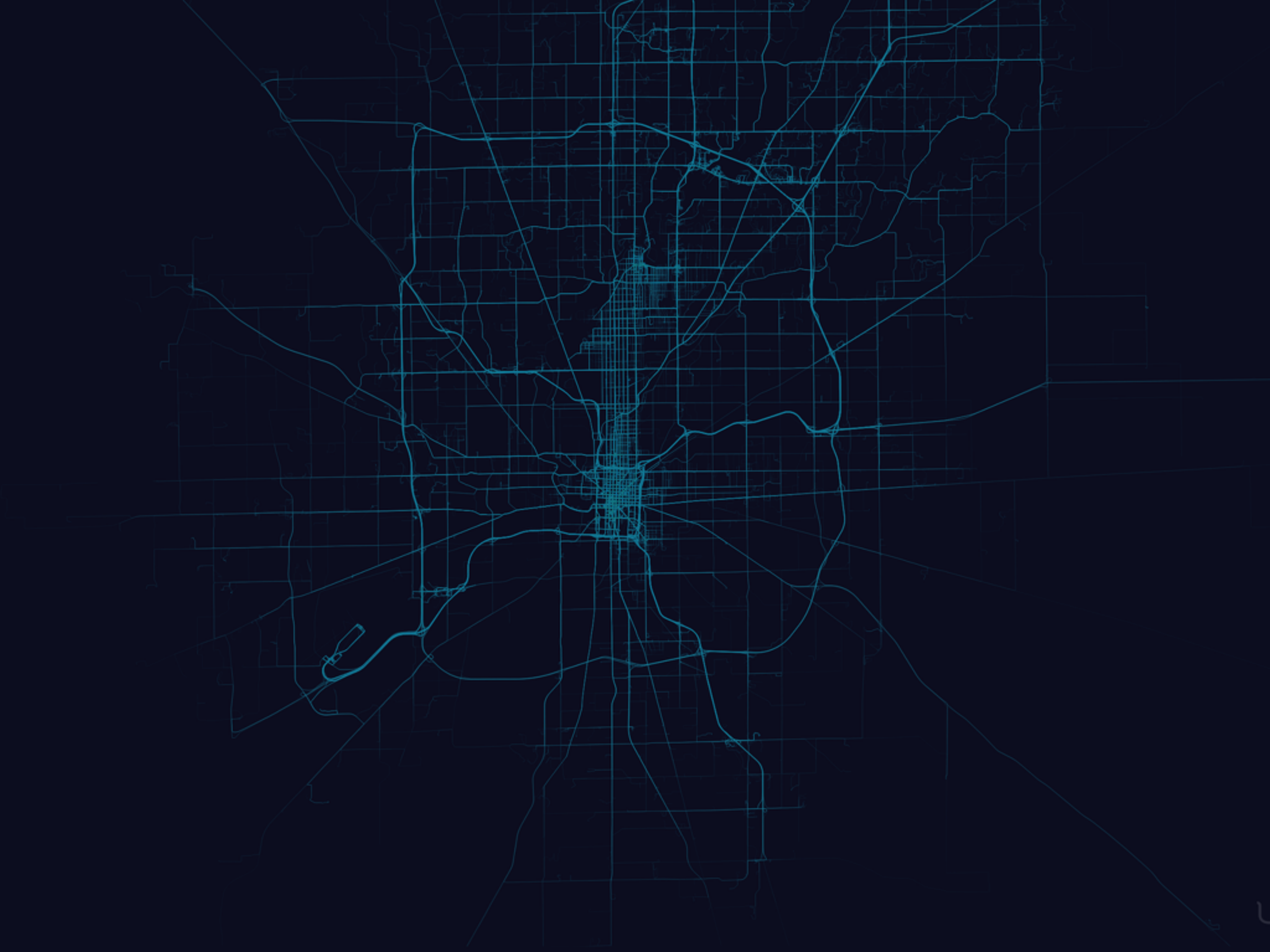


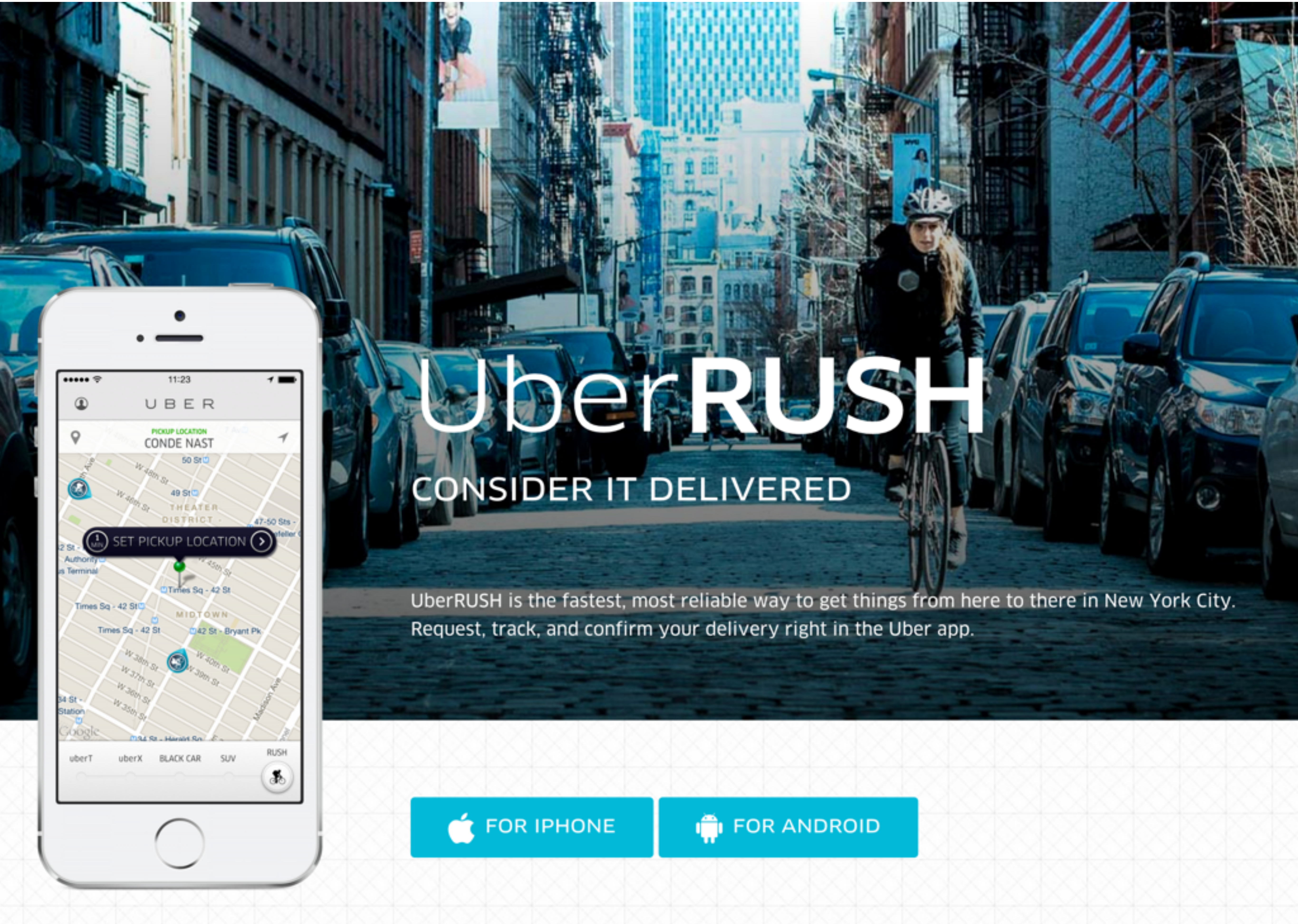








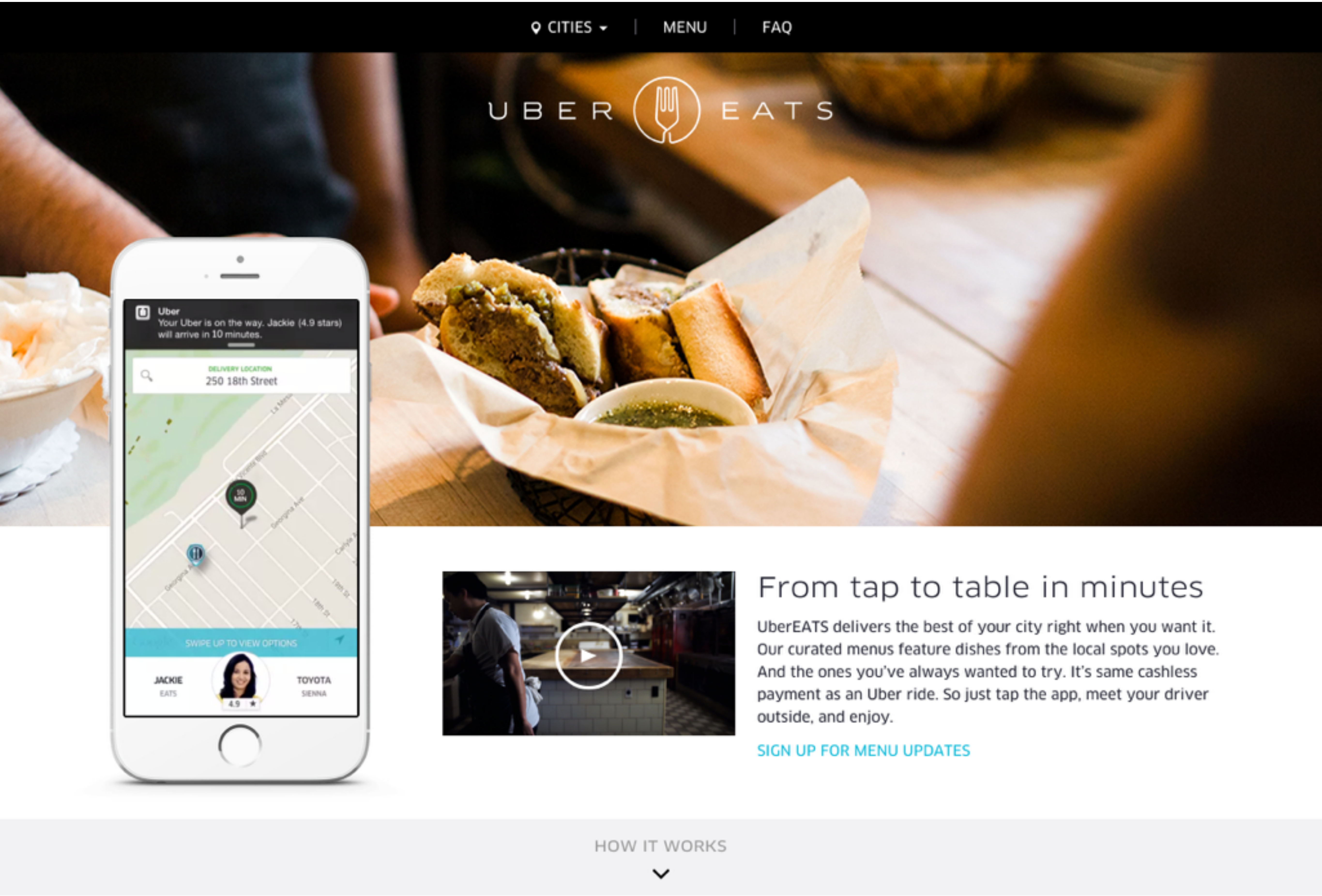




UberRUSH

CONSIDER IT DELIVERED

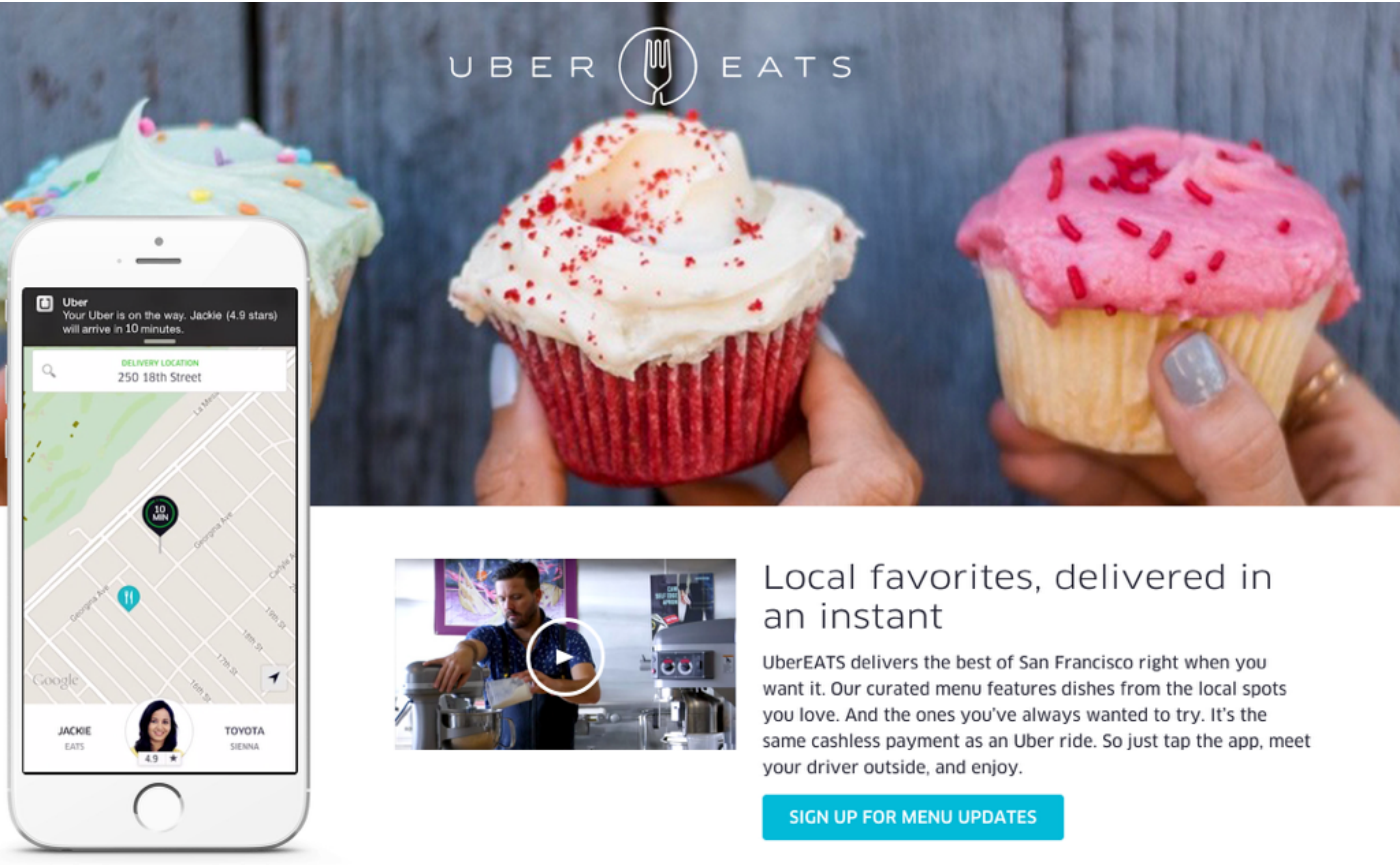
UberRUSH is the fastest, most reliable way to get things from here to there in New York City. Request, track, and confirm your delivery right in the Uber app.



Uber
Your Uber is on the way. Jackie (4.9 stars) will arrive in 10 minutes.

From tap to table in minutes

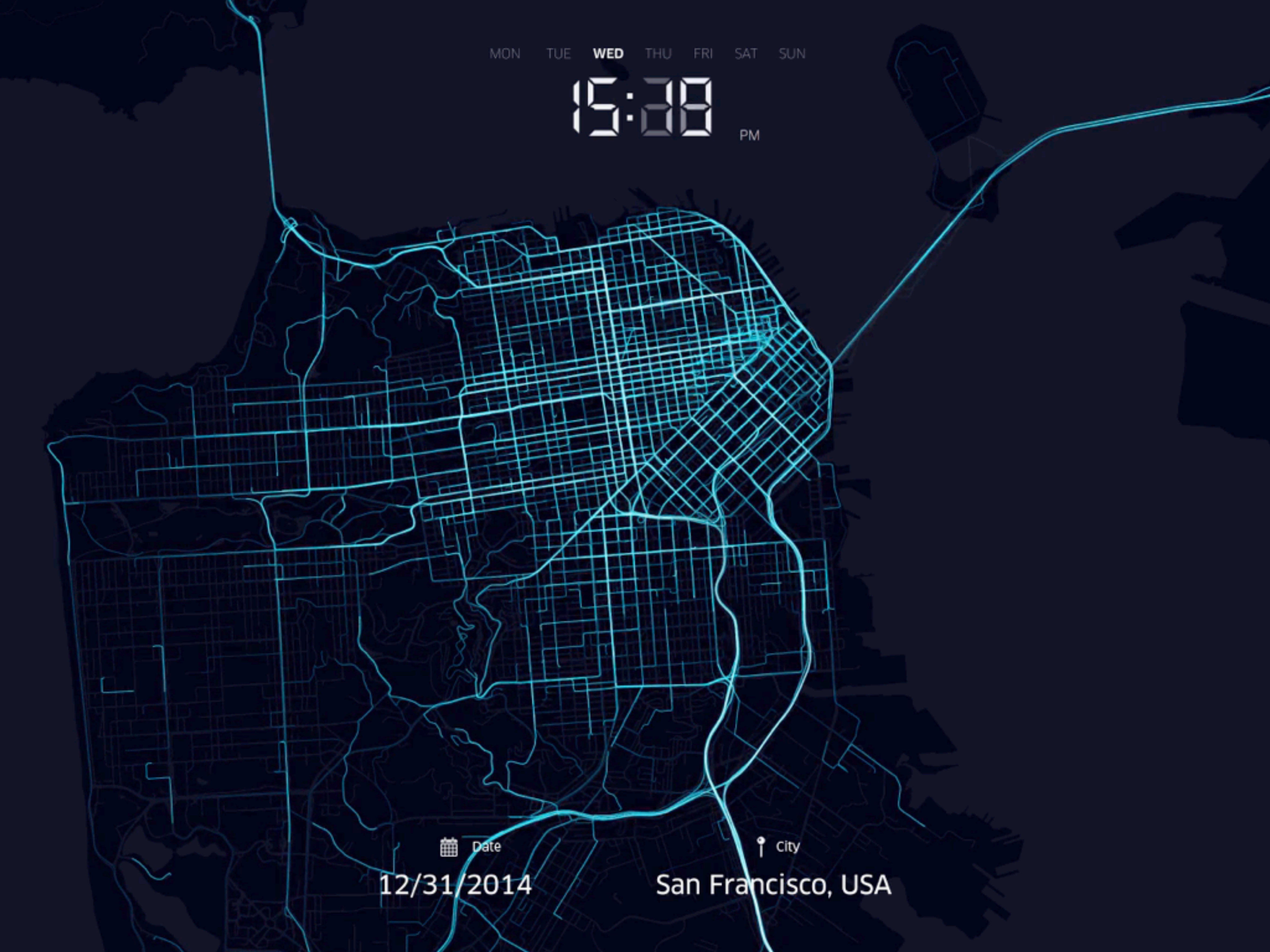
UberEATS delivers the best of your city right when you want it. Our curated menus feature dishes from the local spots you love. And the ones you've always wanted to try. It's same cashless payment as an Uber ride. So just tap the app, meet your driver outside, and enjoy.



Uber
Your Uber is on the way. Jackie (4.9 stars) will arrive in 10 minutes.

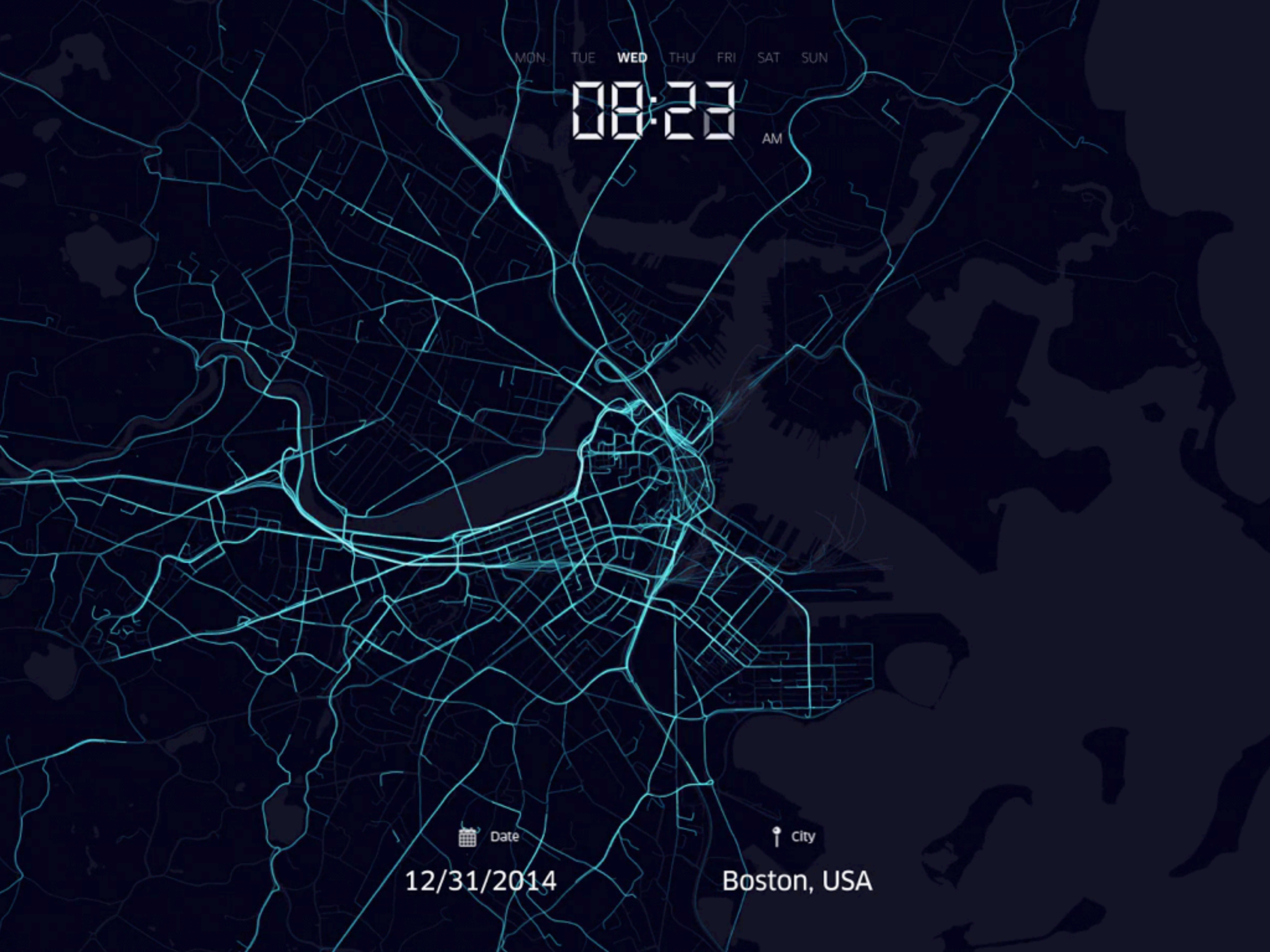
Local favorites, delivered in an instant

UberEATS delivers the best of San Francisco right when you want it. Our curated menu features dishes from the local spots you love. And the ones you've always wanted to try. It's the same cashless payment as an Uber ride. So just tap the app, meet your driver outside, and enjoy.



Date
12/31/2014

City
San Francisco, USA



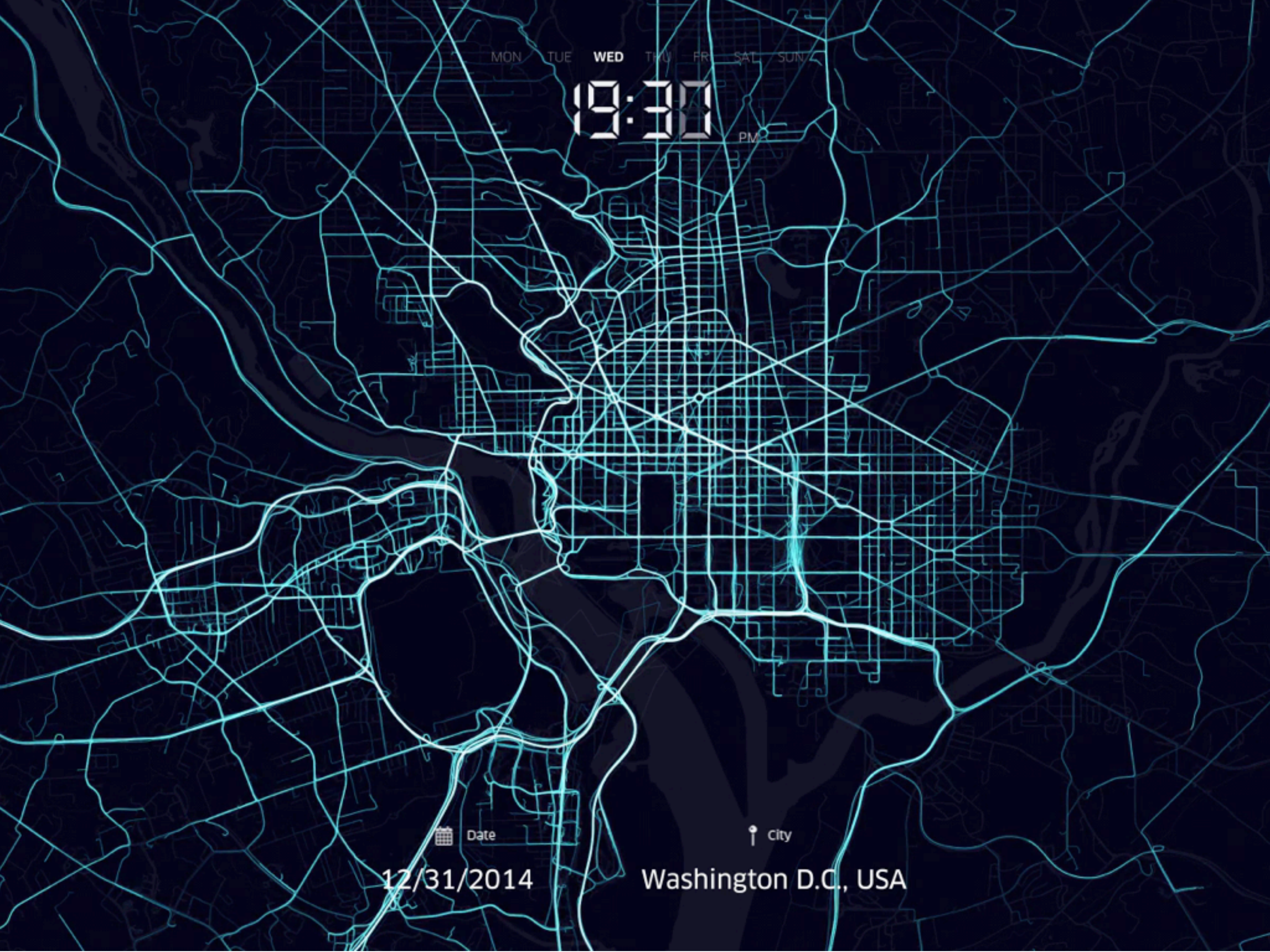
08:23 AM

Date

12/31/2014

City

Boston, USA



Date
12/31/2014

City
Washington D.C., USA

UBER ENGINEERING HISTORY

2009-2010	Outsourced PHP + MySQL
Jan 2011	"dispatch" - Node.JS/MongoDB
Jan 2011	“API” - Python/SQLAlchemy/MySQL
Feb 2012	Dispatch swaps MongoDB for Redis
May 2012	Dispatch adds ON fallback
Jan 2013	First non-API Python services
Feb 2013	API switched to Postgres
Mar 2014	New Python services use MySQL
Mar 2014	Schemaless begins, must finish before pg collapse
Sep 2014	First Schemaless - trips out of Postgres
Aug 2015	Dispatch X.0 / Ringpop / Riak

TECHNICAL DEBT

BREAKAGE

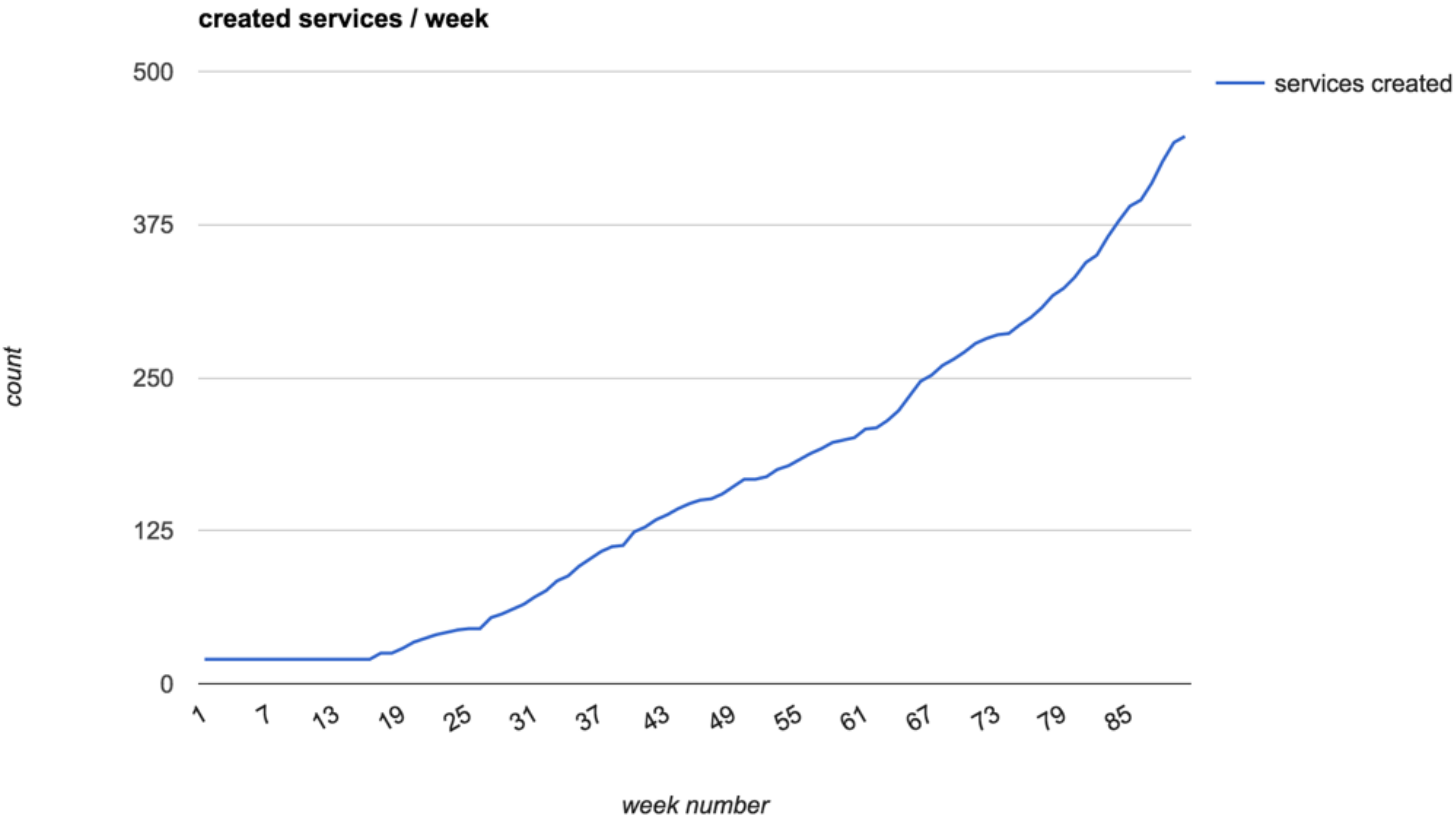
RESPONSIBLE CHOICES

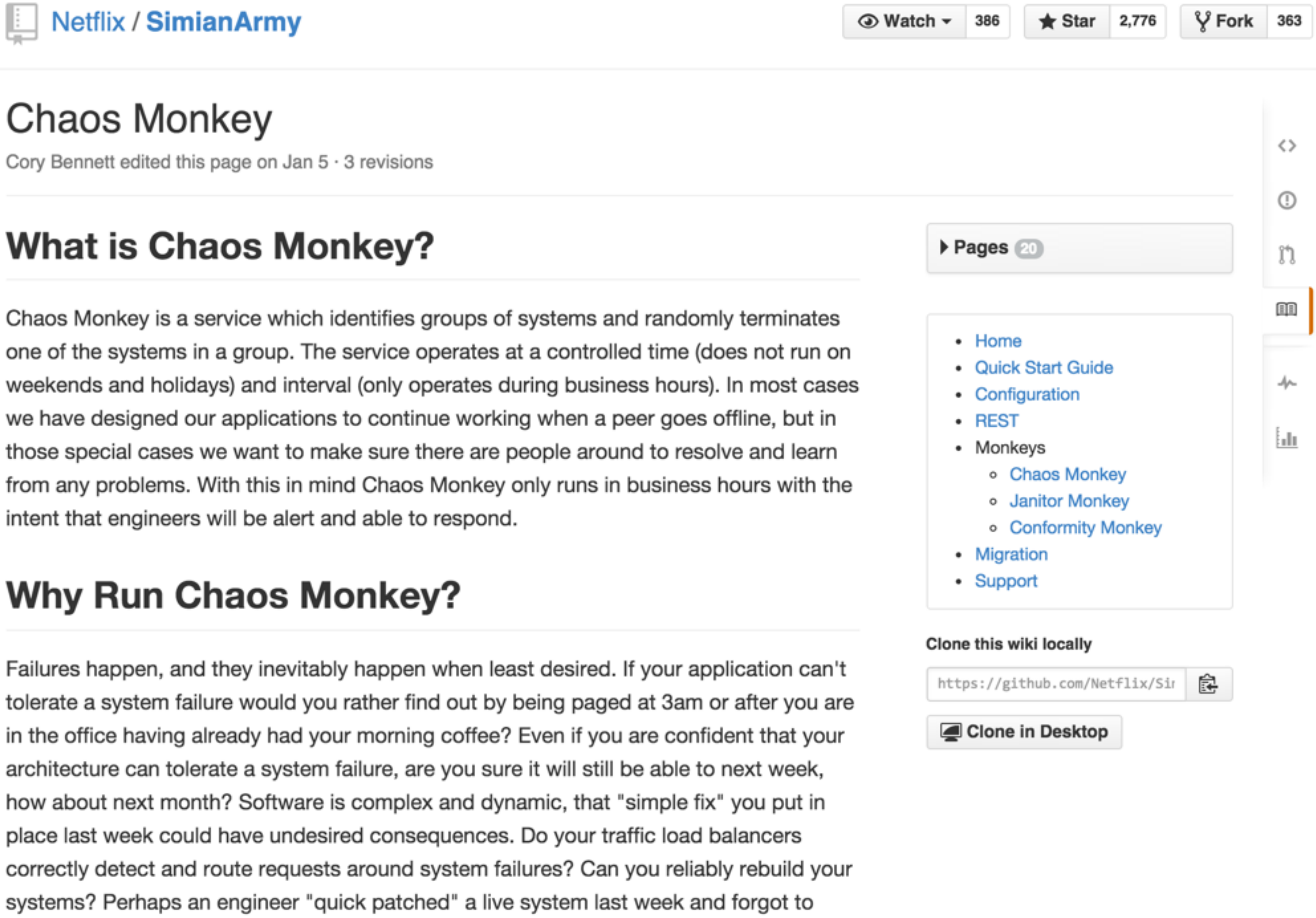
CUMULATIVE FAILURE

TRADEOFFS

TRADEOFFS

MICROSERVICES


created services / week
500
375
250
125
0
count
1
7
13
19
25
31
37
43
49
55
61
67
73
79
85
week number
services created



Chaos Monkey

Cory Bennett edited this page on Jan 5 · 3 revisions

- Home
- Quick Start Guide
- Configuration
- REST
- Monkeys
  - Chaos Monkey
  - Janitor Monkey
  - Conformity Monkey
- Migration
- Support

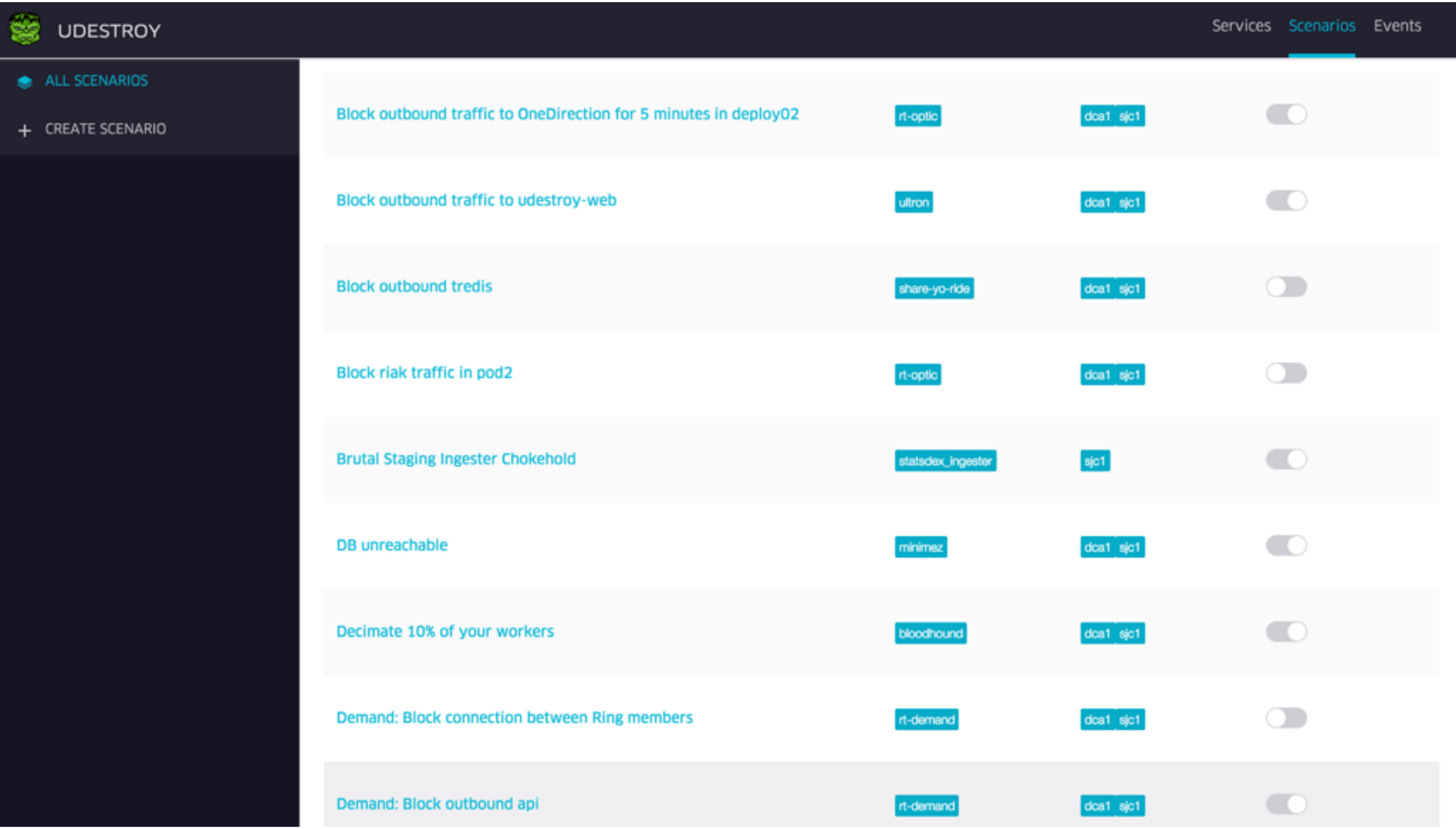
What is Chaos Monkey?

Chaos Monkey is a service which identifies groups of systems and randomly terminates one of the systems in a group. The service operates at a controlled time (does not run on weekends and holidays) and interval (only operates during business hours). In most cases we have designed our applications to continue working when a peer goes offline, but in those special cases we want to make sure there are people around to resolve and learn from any problems. With this in mind Chaos Monkey only runs in business hours with the intent that engineers will be alert and able to respond.

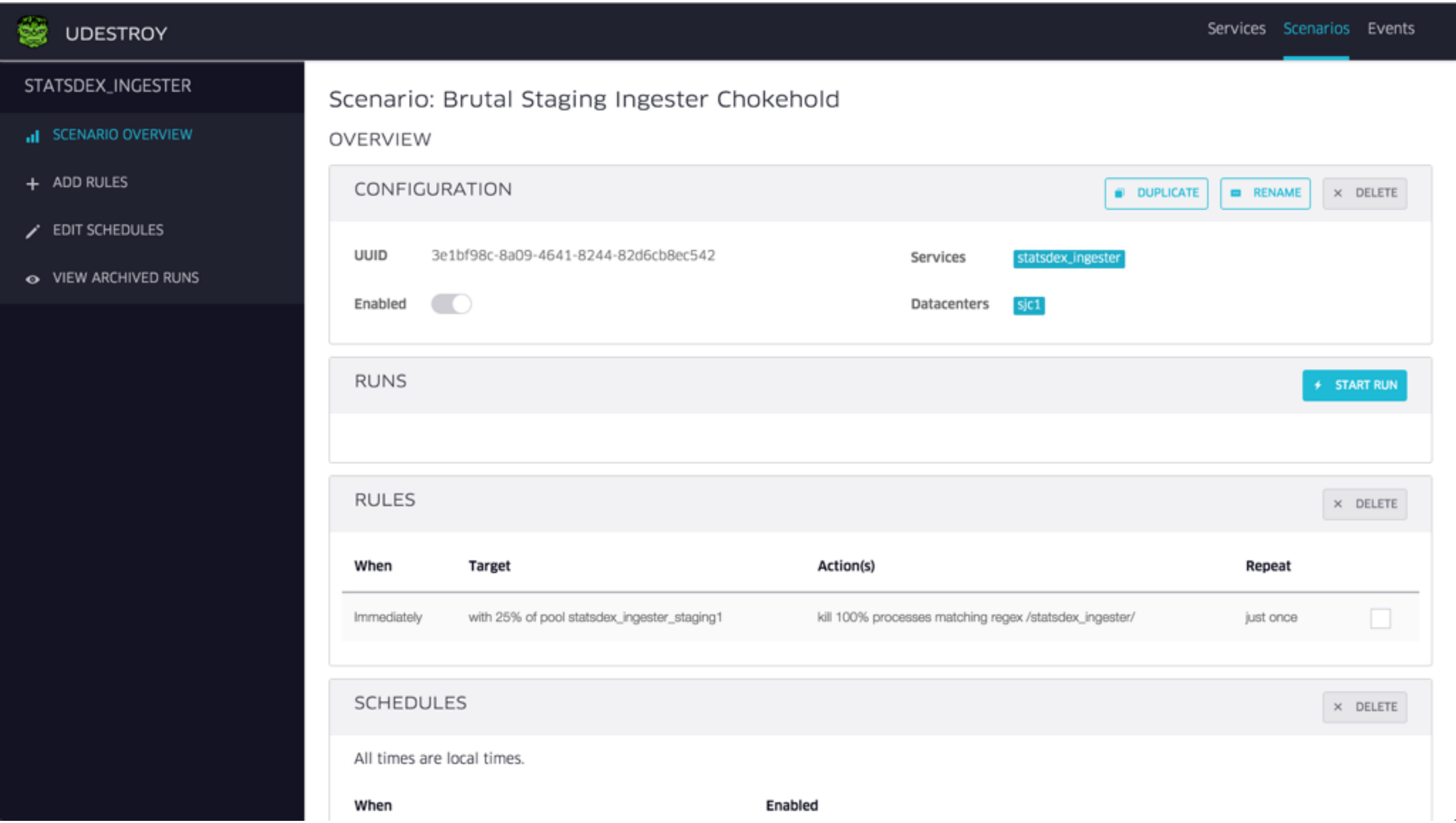
Why Run Chaos Monkey?

Failures happen, and they inevitably happen when least desired. If your application can't tolerate a system failure would you rather find out by being paged at 3am or after you are in the office having already had your morning coffee? Even if you are confident that your architecture can tolerate a system failure, are you sure it will still be able to next week, how about next month? Software is complex and dynamic, that "simple fix" you put in place last week could have undesired consequences. Do your traffic load balancers correctly detect and route requests around system failures? Can you reliably rebuild your systems? Perhaps an engineer "quick patched" a live system last week and forgot to

UDESTROY



Block outbound traffic to OneDirection for 5 minutes in deploy02	rt-optic	dca1 sjc1
Block outbound traffic to udestroy-web	ultron	dca1 sjc1
Block outbound tredis	share-yo-ride	dca1 sjc1
Block riak traffic in pod2	rt-optic	dca1 sjc1
Brutal Staging Ingester Chokehold	statsdex_ingester	sjc1
DB unreachable	minimez	dca1 sjc1
Decimate 10% of your workers	bloodhound	dca1 sjc1
Demand: Block connection between Ring members	rt-demand	dca1 sjc1
Demand: Block outbound api	rt-demand	dca1 sjc1



Scenario: Brutal Staging Ingester Chokehold

OVERVIEW

CONFIGURATION

UUID 3e1bf98c-8a09-4641-8244-82d6cb8ec542

Enabled

Services statsdex_ingester

Datacenters sjc1

RUNS

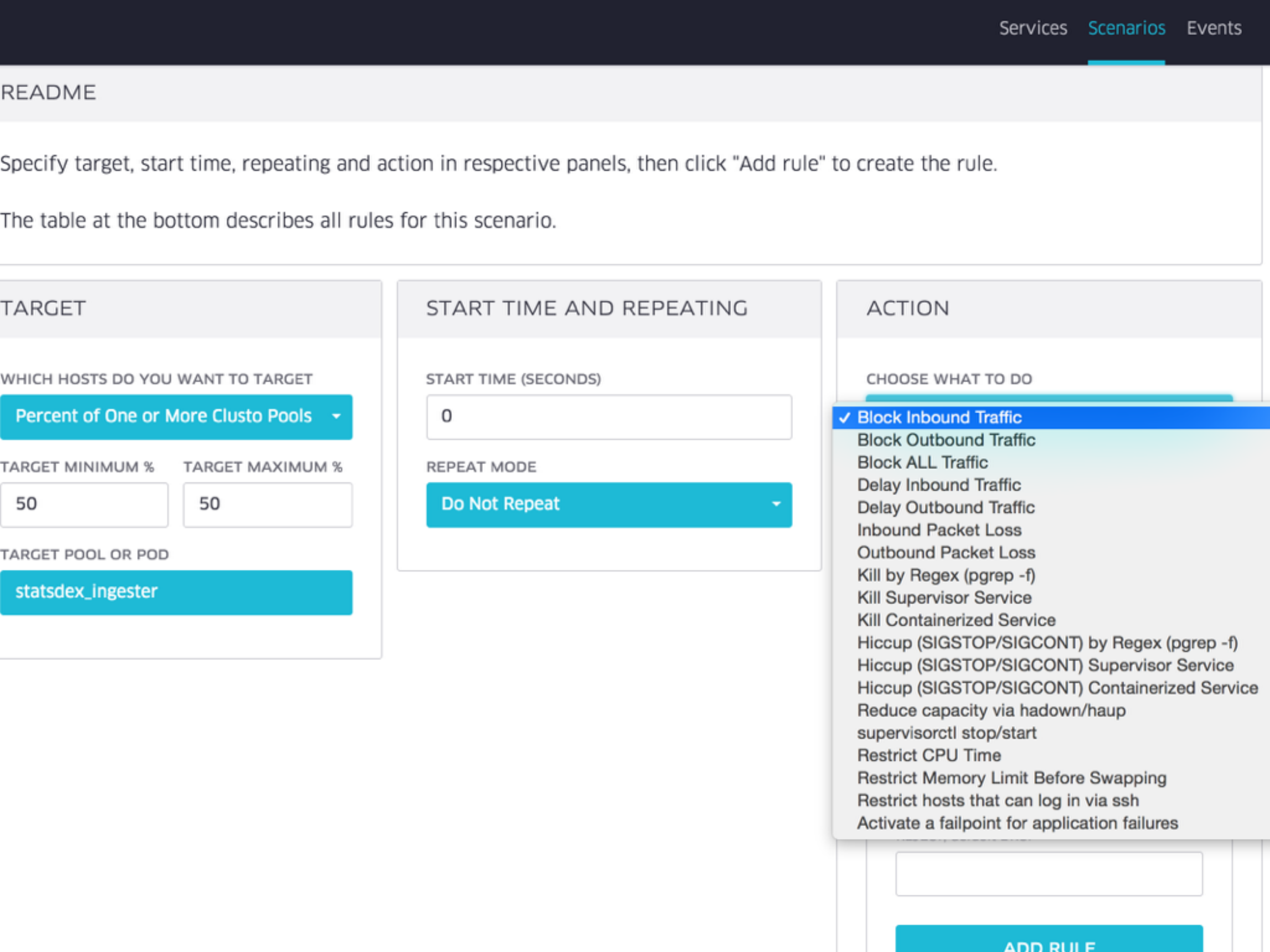
RULES

When	Target	Action(s)	Repeat	
Immediately	with 25% of pool statsdex_ingester_staging1	kill 100% processes matching regex /statsdex_ingester/	just once	□

SCHEDULES

All times are local times.

When	Enabled



README

Specify target, start time, repeating and action in respective panels, then click "Add rule" to create the rule.

The table at the bottom describes all rules for this scenario.

TARGET

WHICH HOSTS DO YOU WANT TO TARGET: Percent of One or More Clusto Pools

TARGET MINIMUM %: 50

TARGET MAXIMUM %: 50

TARGET POOL OR POD: statsdex_ingester

START TIME AND REPEATING

START TIME (SECONDS): 0

REPEAT MODE: Do Not Repeat

ACTION

CHOOSE WHAT TO DO

✓ Block Inbound Traffic
Block Outbound Traffic
Block ALL Traffic
Delay Inbound Traffic
Delay Outbound Traffic
Inbound Packet Loss
Outbound Packet Loss
Kill by Regex (pgrep -f)
Kill Supervisor Service
Kill Containerized Service
Hiccup (SIGSTOP/SIGCONT) by Regex (pgrep -f)
Hiccup (SIGSTOP/SIGCONT) Supervisor Service
Hiccup (SIGSTOP/SIGCONT) Containerized Service
Reduce capacity via hadown/haup
supervisorctl stop/start
Restrict CPU Time
Restrict Memory Limit Before Swapping
Restrict hosts that can log in via ssh
Activate a failpoint for application failures
ADD RULE

FAILURE TESTING

DATABASES

REPLICATION PARTNERS

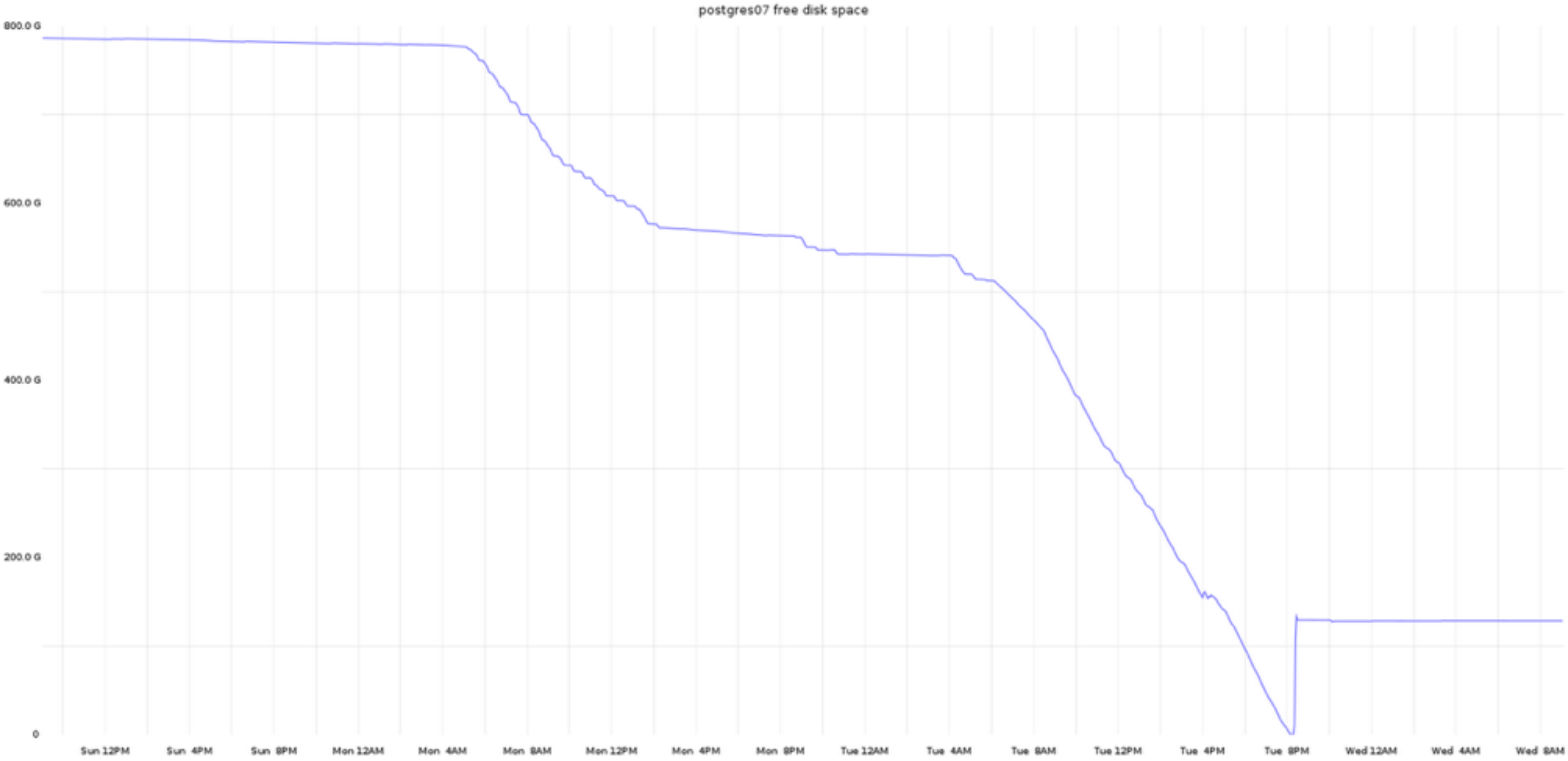
- master and “slave”
- leader and follower
- primary and secondary
- queen and princess?

WORST OUTAGE EVER

The API did not have an available Postgres master database for approximately 16 hours, and had inadequate read capacity for an additional 24 hours. The core trip flow continued to work during this outage, but most other capabilities were degraded or non-functional during this period.

May 11, 4:10AM

<patch> is landed to upgrade the s3cmd program from v1.5.0 to v1.5.2, which adds support for China, but also starts requiring new IAM permissions. This caused the program which backs up write-ahead log segments to S3 to fail and (to prevent losing data) store all segments on disk, greatly increasing the rate of database disk consumption.


postgres07 free disk space
800.0 G
600.0 G
400.0 G
200.0 G
0
Sun 12PM
Sun 4PM
Sun 8PM
Mon 12AM
Mon 4AM
Mon 8AM
Mon 12PM
Mon 4PM
Mon 8PM
Tue 12AM
Tue 4AM
Tue 8AM
Tue 12PM
Tue 4PM
Tue 8PM
Wed 12AM
Wed 4AM
Wed 8AM

6:11 PM - Initial Pagerduty alert fires on Postgres07-sjc1, the API Postgres master database at that time : “DISK CRITICAL - free space: / 81 GB (2% inode=99%)”.

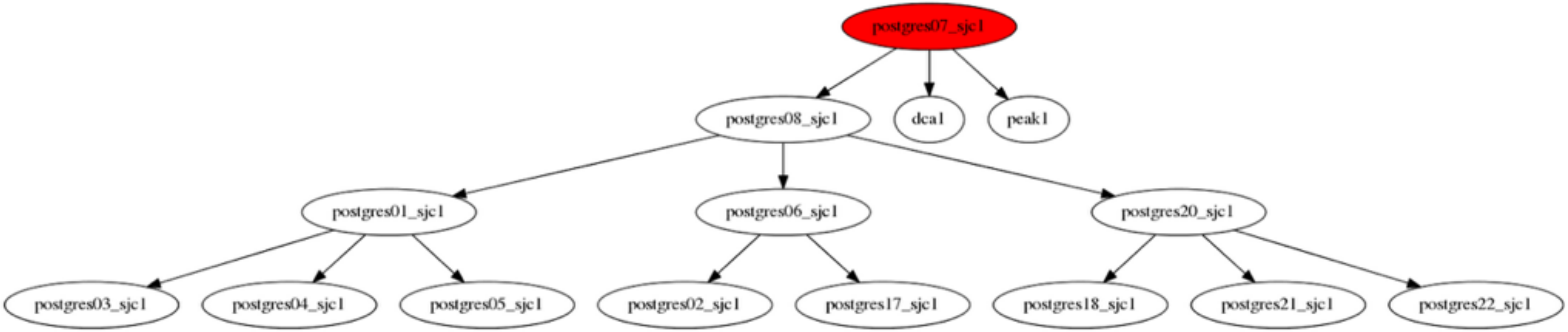
6:11 PM - Alert was acknowledged by oncall engineer.

7:11 PM - Second Pagerduty alert fires on Postgres07-sjc1. "DISK CRITICAL - free space: / 36 GB (1% inode=99%)".

7:17 PM - Alert acknowledged by oncall engineer.

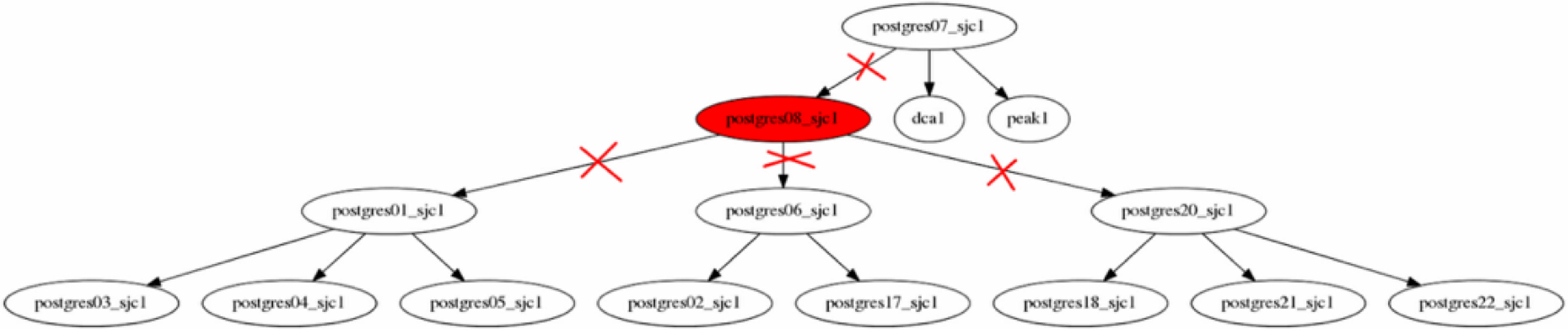
8:11 PM - Third Pagerduty alert fires on postgres07-sjc1. "DISK CRITICAL - free space: / 0 GB (0% inode=99%)"

8:14 PM - API master database becomes unavailable and Postgres master process crashes.

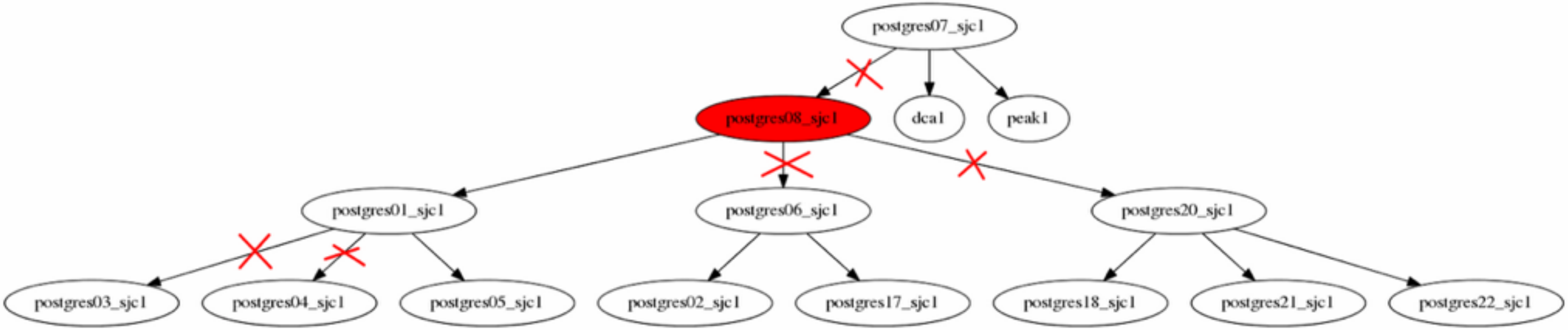

postgres07_sjc1
postgres08_sjc1
dca1
peak1
postgres01_sjc1
postgres06_sjc1
postgres20_sjc1
postgres03_sjc1
postgres04_sjc1
postgres05_sjc1
postgres02_sjc1
postgres17_sjc1
postgres18_sjc1
postgres21_sjc1
postgres22_sjc1

8:20 PM - Issue is identified as being caused by a huge number of unarchived WAL files, and <engineer> tries to resolve the issue by deleting old WAL issues and restarting Postgres. This is done incorrectly and Postgres fails to restart.

~9:00 PM - Promoted postgres08-sjc1 using pg_ctl promote, (note that the runbook failed to specify that the archive_cmd must be changed on the promoted slave before promoting it).


postgres07_sjc1
postgres08_sjc1
dca1
peak1
postgres01_sjc1
postgres06_sjc1
postgres20_sjc1
postgres03_sjc1
postgres04_sjc1
postgres05_sjc1
postgres02_sjc1
postgres17_sjc1
postgres18_sjc1
postgres21_sjc1
postgres22_sjc1

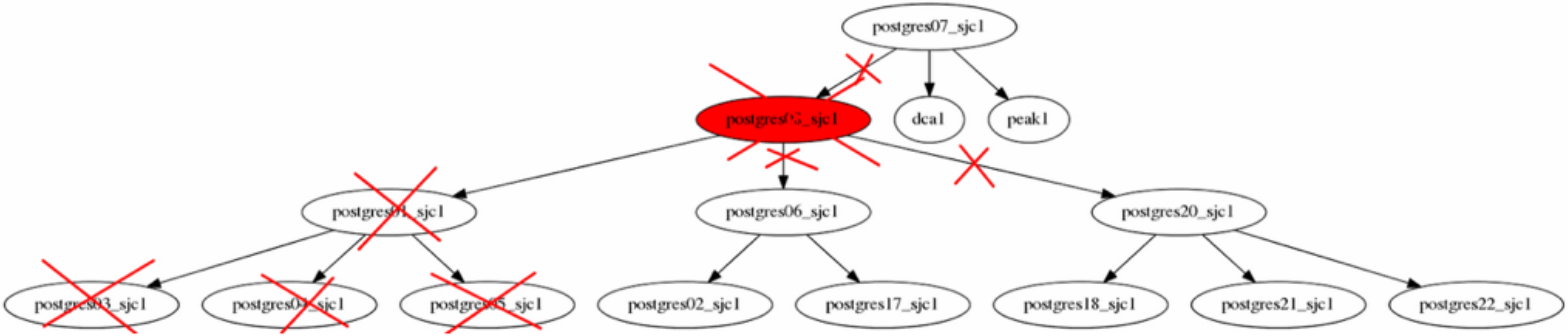
9:20PM - We decide to restart postgres01-sjc1 and postgres05-sjc1 pointing to postgres08-sjc1


postgres07_sjc1
postgres08_sjc1
dca1
peak1
postgres01_sjc1
postgres06_sjc1
postgres20_sjc1
postgres03_sjc1
postgres04_sjc1
postgres05_sjc1
postgres02_sjc1
postgres17_sjc1
postgres18_sjc1
postgres21_sjc1
postgres22_sjc1

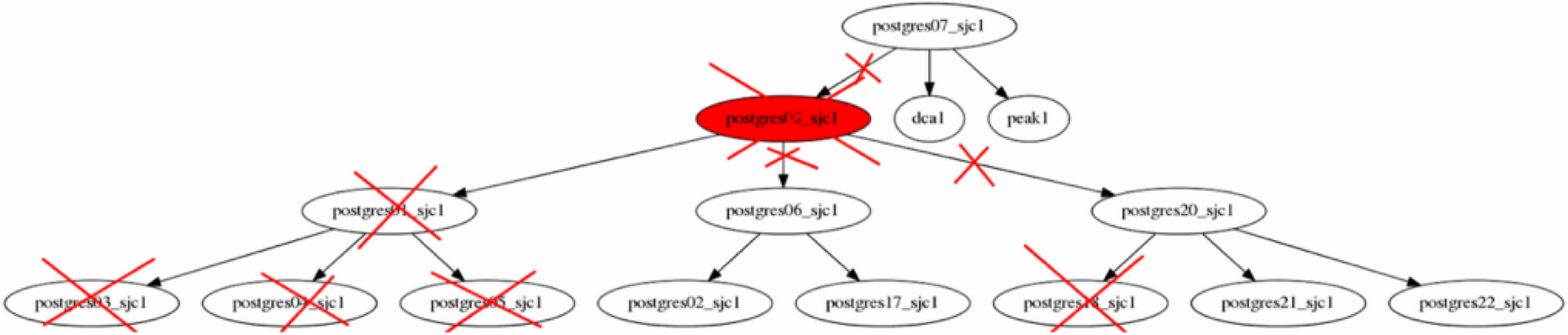
9:35 PM - The new timeline is copied from postgres08-sjc1 to S3, and followers begin trying to catch up, but very slowly, there is a lot to recover. API Read-Write is stopped to reduce potential data loss and amount of writes to recover (however autovacuum is still writing to the WAL, a lot).

11:43 PM - Identify that at least one WAL segment is corrupt in S3, i.e. archived while it was pre-allocated, but not completely written. The archiver was failing on this because it existed, so the script was changed to no-op if it already exists, and postgres removed the good copy. At this point, postgres08-sjc1 is up as a master, but any follower of postgres08-sjc1 at a WAL location prior to the corrupt segment will not be able to cross the gap (i.e. all of them).

postgres08-sjc1, postgres05-sjc1, and postgres01-sjc1, as well as all children of postgres01-sjc1 are effectively broken at this point.

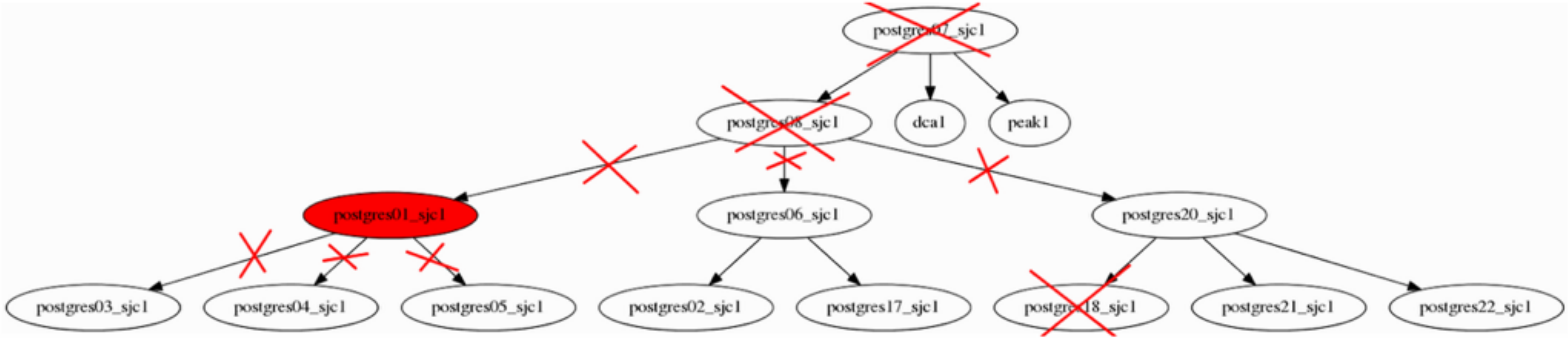

postgres07_sjc1
postgres[illegible]_sjc1
dca1
peak1
postgres01_sjc1
postgres06_sjc1
postgres20_sjc1
postgres03_sjc1
postgres04_sjc1
postgres05_sjc1
postgres02_sjc1
postgres17_sjc1
postgres18_sjc1
postgres21_sjc1
postgres22_sjc1

11:45 PM - <consultants> suggest that we may be able to rsync the data from postgres08-sjc1 to the other follower since the data changes are minimal (caveat: autovacuum), and we decide to use postgres18-sjc1 as the follower to try to sync. This effectively takes postgres18-sjc1 out of commission.


postgres07_sjc1
postgres05_sjc1
dca1
peak1
postgres01_sjc1
postgres06_sjc1
postgres20_sjc1
postgres03_sjc1
postgres04_sjc1
postgres15_sjc1
postgres02_sjc1
postgres17_sjc1
postgres18_sjc1
postgres21_sjc1
postgres22_sjc1

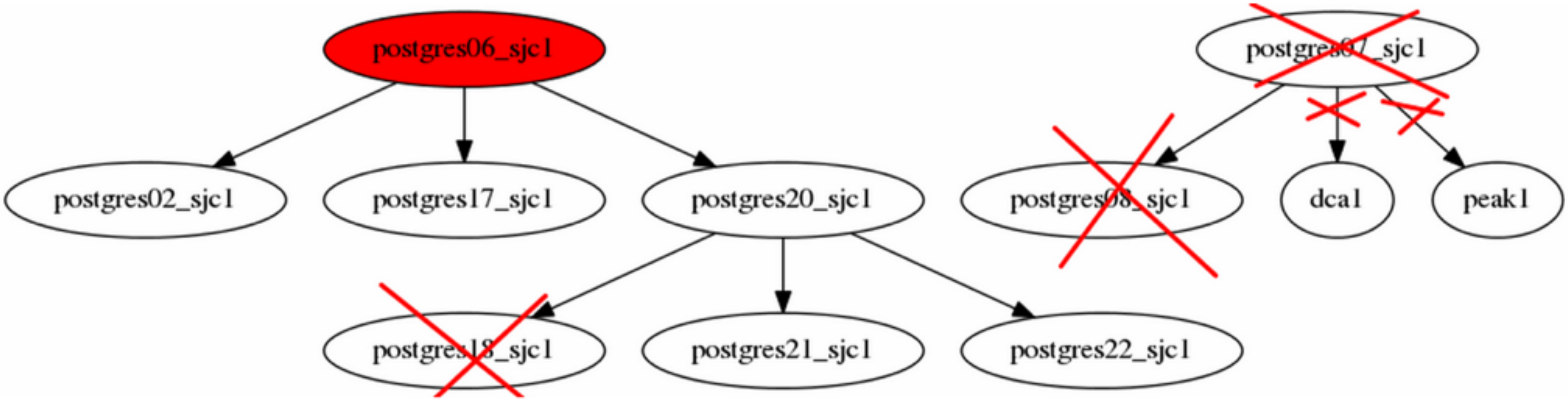
12:30 AM - <consultants> believe that they may be able to write a C program that would allow us to promote postgres05-sjc1 in a way that would skip the corrupted WAL file on postgres08-sjc1. This program would bypass the normal safety mechanisms built into PostgreSQL.

3:20 AM - The C program allows us to issue commands to postgres01-sjc1. We are able to promote it, but postgres05-sjc1 is unable to follow the promotion.

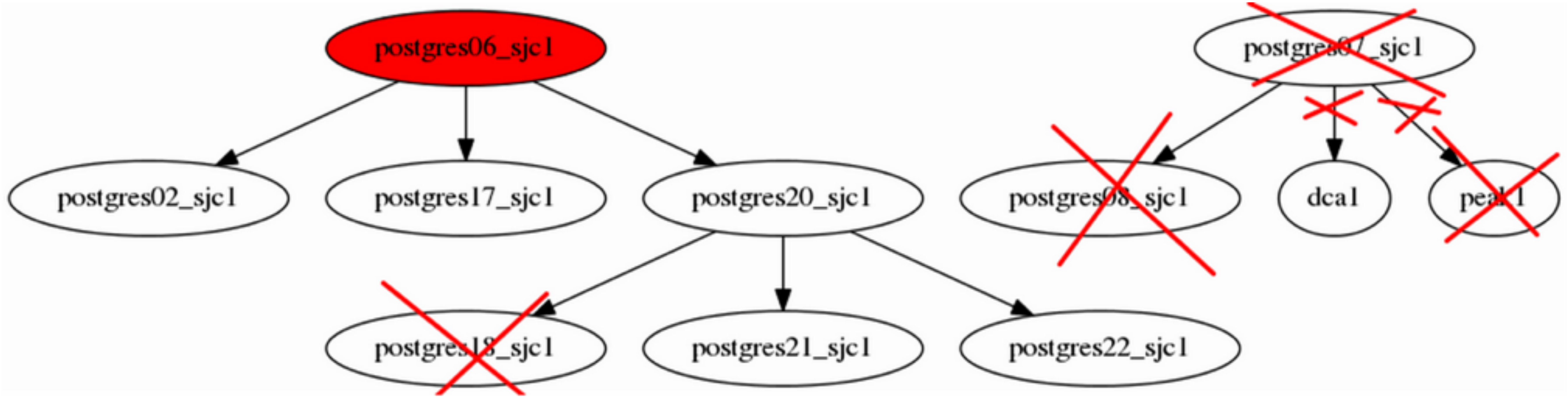

postgres07_sjc1
postgres08_sjc1
dca1
peak1
postgres01_sjc1
postgres06_sjc1
postgres20_sjc1
postgres03_sjc1
postgres04_sjc1
postgres05_sjc1
postgres02_sjc1
postgres17_sjc1
postgres18_sjc1
postgres21_sjc1
postgres22_sjc1

~6:00 AM - We decide to give up on postgres01, and decide to promote postgres06-sjc1, which is immediately successful, but getting followers replicating from it is taking too long to recover via S3, so archive/recover scripts are modified to use cephfs on our local network instead.

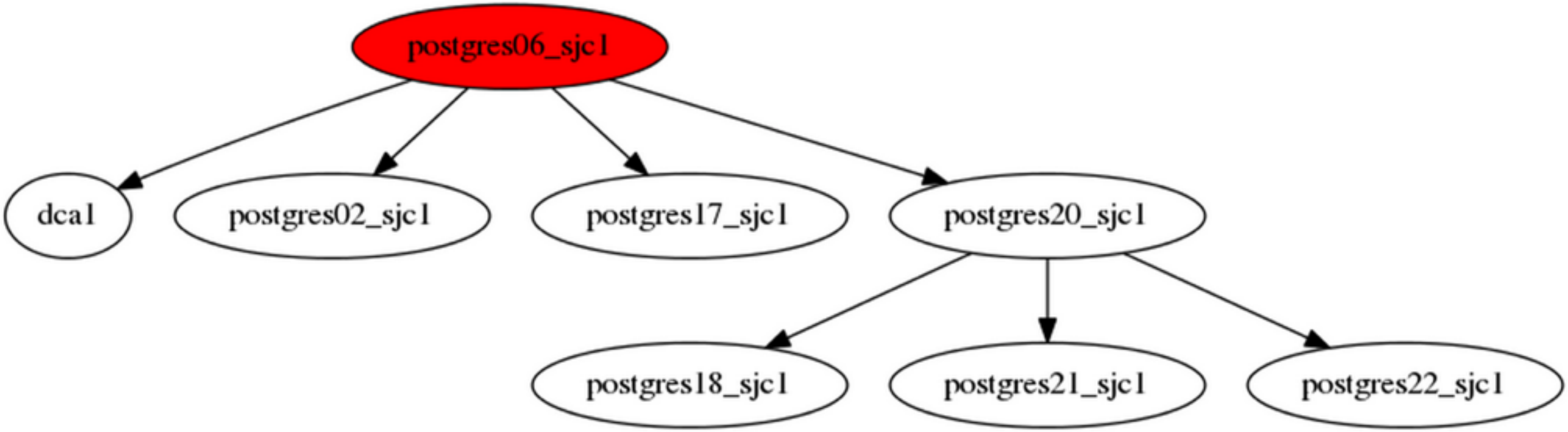
~7:00 AM - We successfully reparent postgres20-sjc1 under postgres06-sjc1, which brings us about 5 followers.


postgres06_sjc1
postgres02_sjc1
postgres17_sjc1
postgres20_sjc1
postgres18_sjc1
postgres21_sjc1
postgres22_sjc1
postgres07_sjc1
postgres08_sjc1
dca1
peak1

~7:30 AM - We attempt to reparent peak1 to postgres06-sjc1 using the same mechanism used to reparent postgres20-sjc1. This does not work, and peak1 is left unrecoverable.


postgres06_sjc1
postgres07_sjc1
postgres02_sjc1
postgres17_sjc1
postgres20_sjc1
postgres08_sjc1
dca1
peak1
postgres18_sjc1
postgres21_sjc1
postgres22_sjc1

May 13, ~9:00 AM - We are able to successfully reparent DCA1 to postgres06-sjc1.


postgres06_sjc1
dca1
postgres02_sjc1
postgres17_sjc1
postgres20_sjc1
postgres18_sjc1
postgres21_sjc1
postgres22_sjc1

May 14, 10:00 AM - All systems returned to normal level of operations.

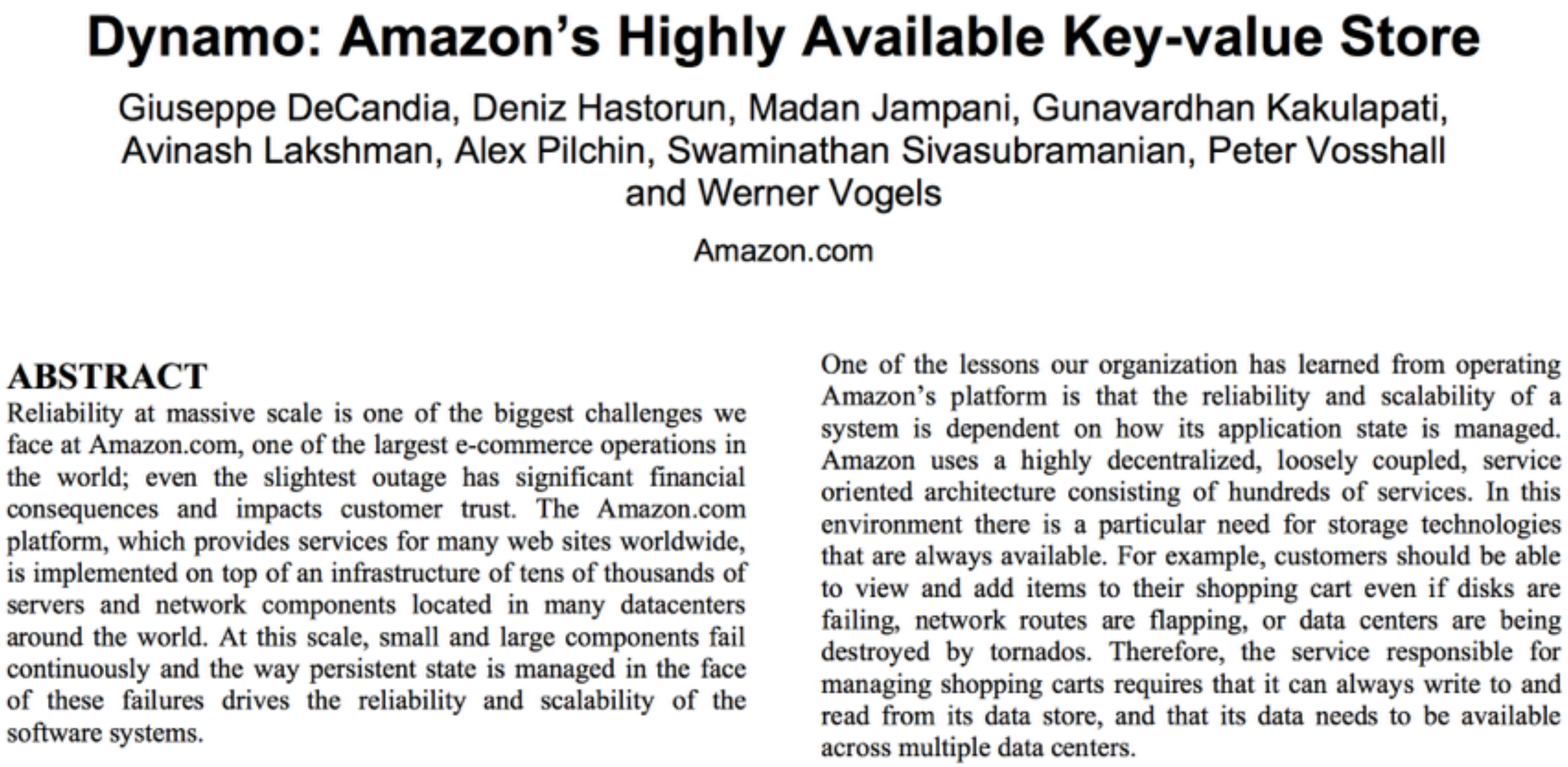
LEARNING THE HARD WAY

Alert titles can obscure critical information

Thresholds aren’t as useful as derivatives

Make a lot of copies of your critical data

Practice failure scenarios



Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati,
Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall
and Werner Vogels

Amazon.com

ABSTRACT

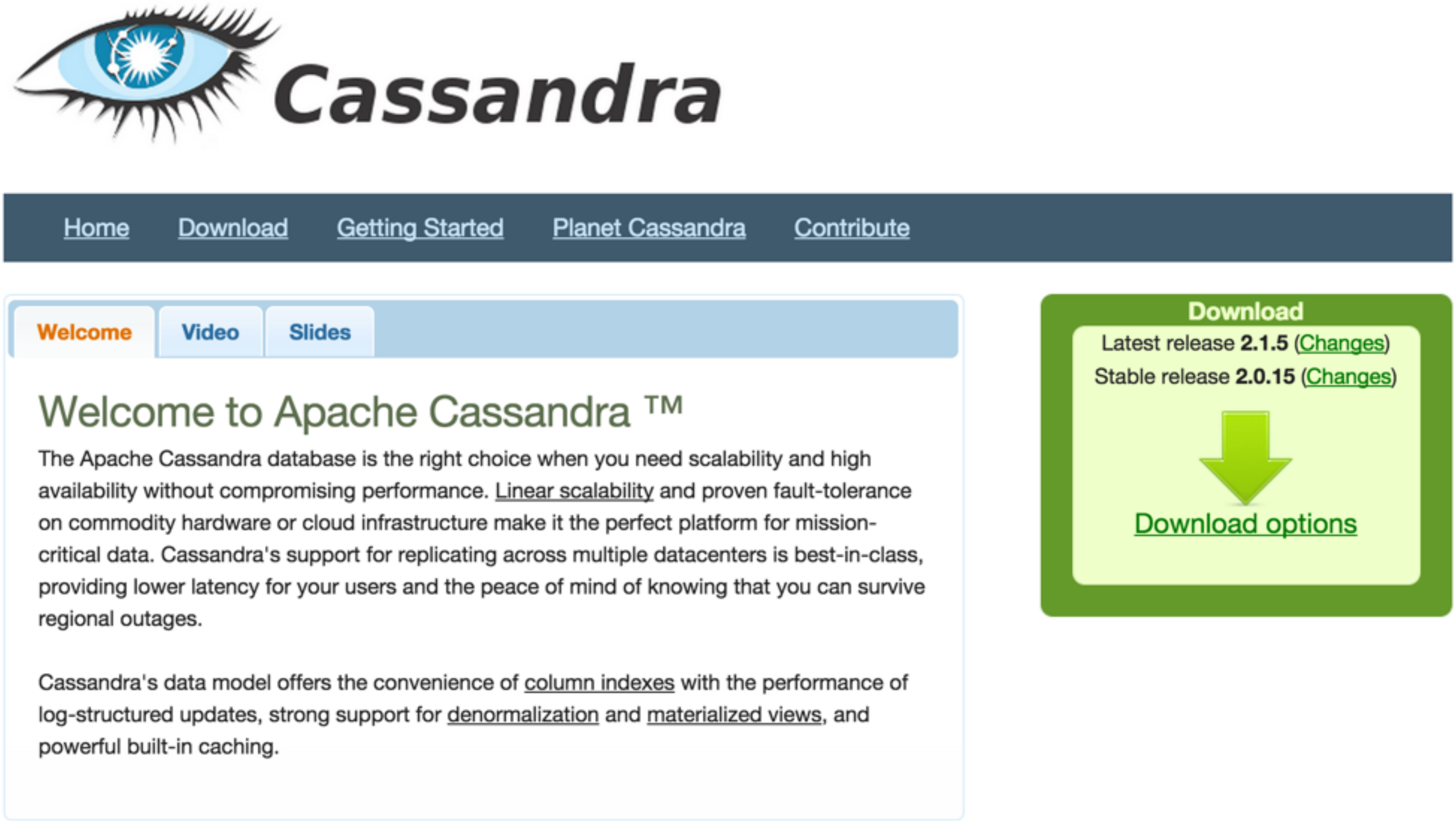
Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world; even the slightest outage has significant financial consequences and impacts customer trust. The Amazon.com platform, which provides services for many web sites worldwide, is implemented on top of an infrastructure of tens of thousands of servers and network components located in many datacenters around the world. At this scale, small and large components fail continuously and the way persistent state is managed in the face of these failures drives the reliability and scalability of the software systems.

One of the lessons our organization has learned from operating Amazon's platform is that the reliability and scalability of a system is dependent on how its application state is managed. Amazon uses a highly decentralized, loosely coupled, service oriented architecture consisting of hundreds of services. In this environment there is a particular need for storage technologies that are always available. For example, customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornados. Therefore, the service responsible for managing shopping carts requires that it can always write to and read from its data store, and that its data needs to be available across multiple data centers.

Amazo

ABSTRACT

Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world; even the slightest outage has significant financial consequences and impacts customer trust. The Amazon.com platform, which provides services for many web sites worldwide, is implemented on top of an infrastructure of tens of thousands of servers and network components located in many datacenters around the world. At this scale, small and large components fail continuously and the way persistent state is managed in the face of these failures drives the reliability and scalability of the software systems.



Welcome to Apache Cassandra ™

The Apache Cassandra database is the right choice when you need scalability and high availability without compromising performance. Linear scalability and proven fault-tolerance on commodity hardware or cloud infrastructure make it the perfect platform for mission-critical data. Cassandra's support for replicating across multiple datacenters is best-in-class, providing lower latency for your users and the peace of mind of knowing that you can survive regional outages.

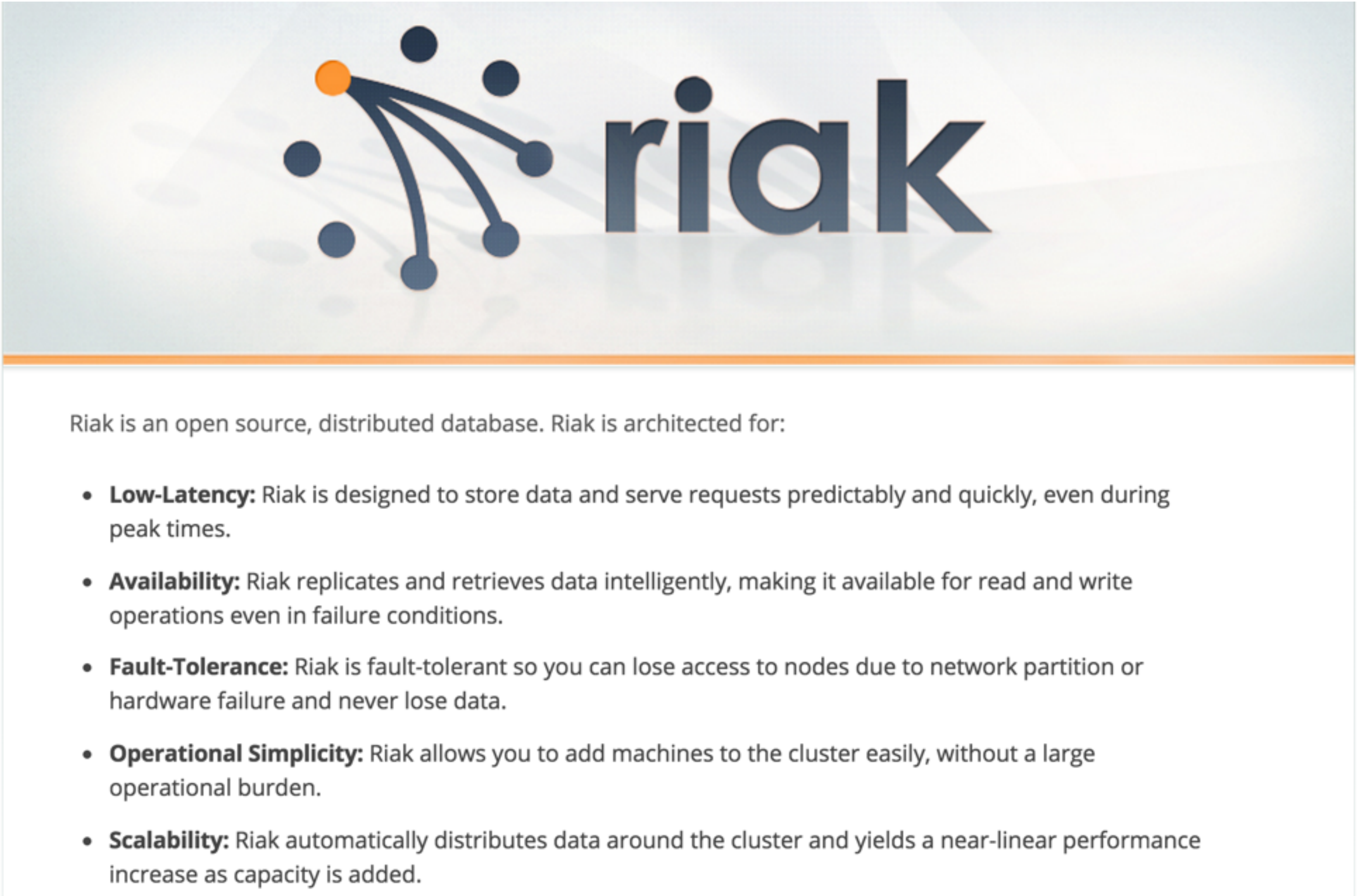
Cassandra's data model offers the convenience of column indexes with the performance of log-structured updates, strong support for denormalization and materialized views, and powerful built-in caching.

Download

Latest release **2.1.5** (Changes)

Stable release **2.0.15** (Changes)

Download options



Riak is an open source, distributed database. Riak is architected for:

- **Low-Latency:** Riak is designed to store data and serve requests predictably and quickly, even during peak times.
- **Availability:** Riak replicates and retrieves data intelligently, making it available for read and write operations even in failure conditions.
- **Fault-Tolerance:** Riak is fault-tolerant so you can lose access to nodes due to network partition or hardware failure and never lose data.
- **Operational Simplicity:** Riak allows you to add machines to the cluster easily, without a large operational burden.
- **Scalability:** Riak automatically distributes data around the cluster and yields a near-linear performance increase as capacity is added.

August 21 ~3:00 PM - 10 node Riak cluster “rt-riak-supply” experiences an intermittent and as yet undiagnosed network problem resulting in multi-second TCP delays.

These delays are masked by Riak, but due to a misconfiguration many siblings are created.

3:41 PM - Delays increase and Riak is no longer able to mask this latency. Clients respond by opening more connections.

3:45 PM - Additional connections, sibling management, and client retries combine to exhaust file descriptors on all 10 Riak nodes.

3:47 PM - Bad timeout handling in client service cascades through SOA to front ends. All dispatch services are degraded or down.

4:06 PM - Traffic is routed to legacy dispatch system.

LEARNING THE HARD WAY

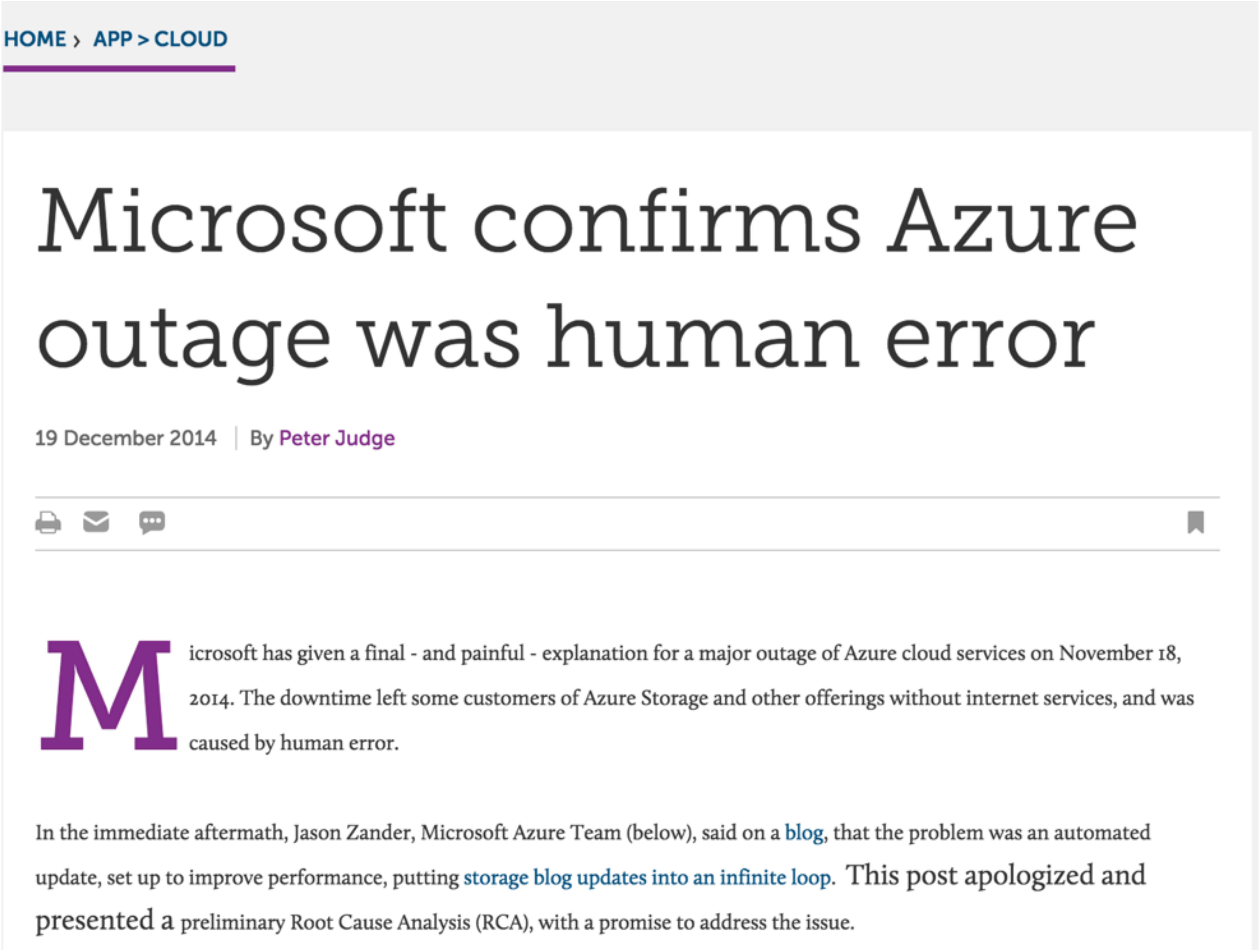
Delay testing would have caught this

Proper back pressure is hard

Understanding the metrics is crucial

CRASH ONLY

HUMAN ERROR



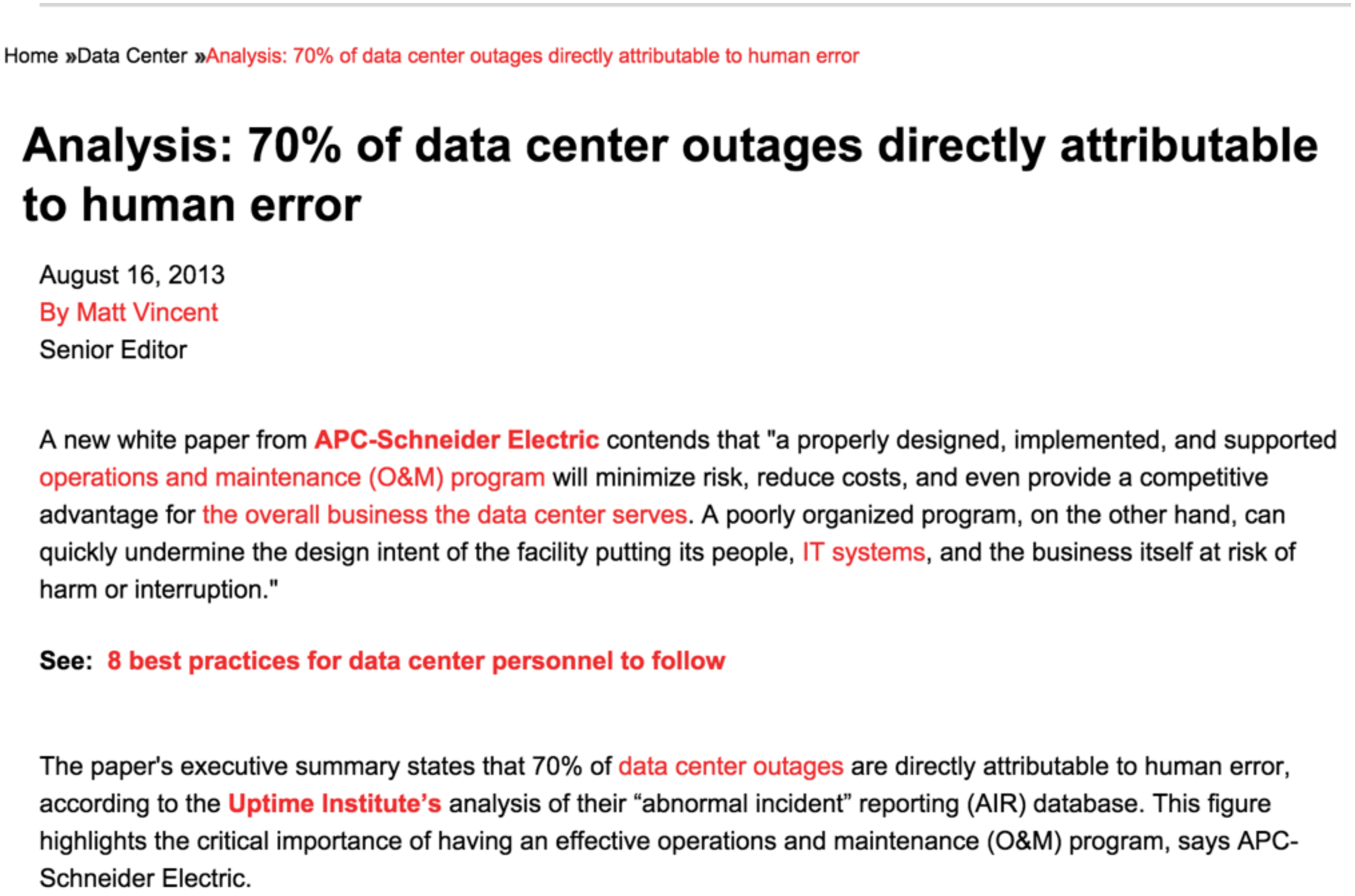
HOME › APP > CLOUD

Microsoft confirms Azure outage was human error

19 December 2014 | By Peter Judge

Microsoft has given a final - and painful - explanation for a major outage of Azure cloud services on November 18, 2014. The downtime left some customers of Azure Storage and other offerings without internet services, and was caused by human error.

In the immediate aftermath, Jason Zander, Microsoft Azure Team (below), said on a blog, that the problem was an automated update, set up to improve performance, putting storage blog updates into an infinite loop. This post apologized and presented a preliminary Root Cause Analysis (RCA), with a promise to address the issue.



Home »Data Center »Analysis: 70% of data center outages directly attributable to human error

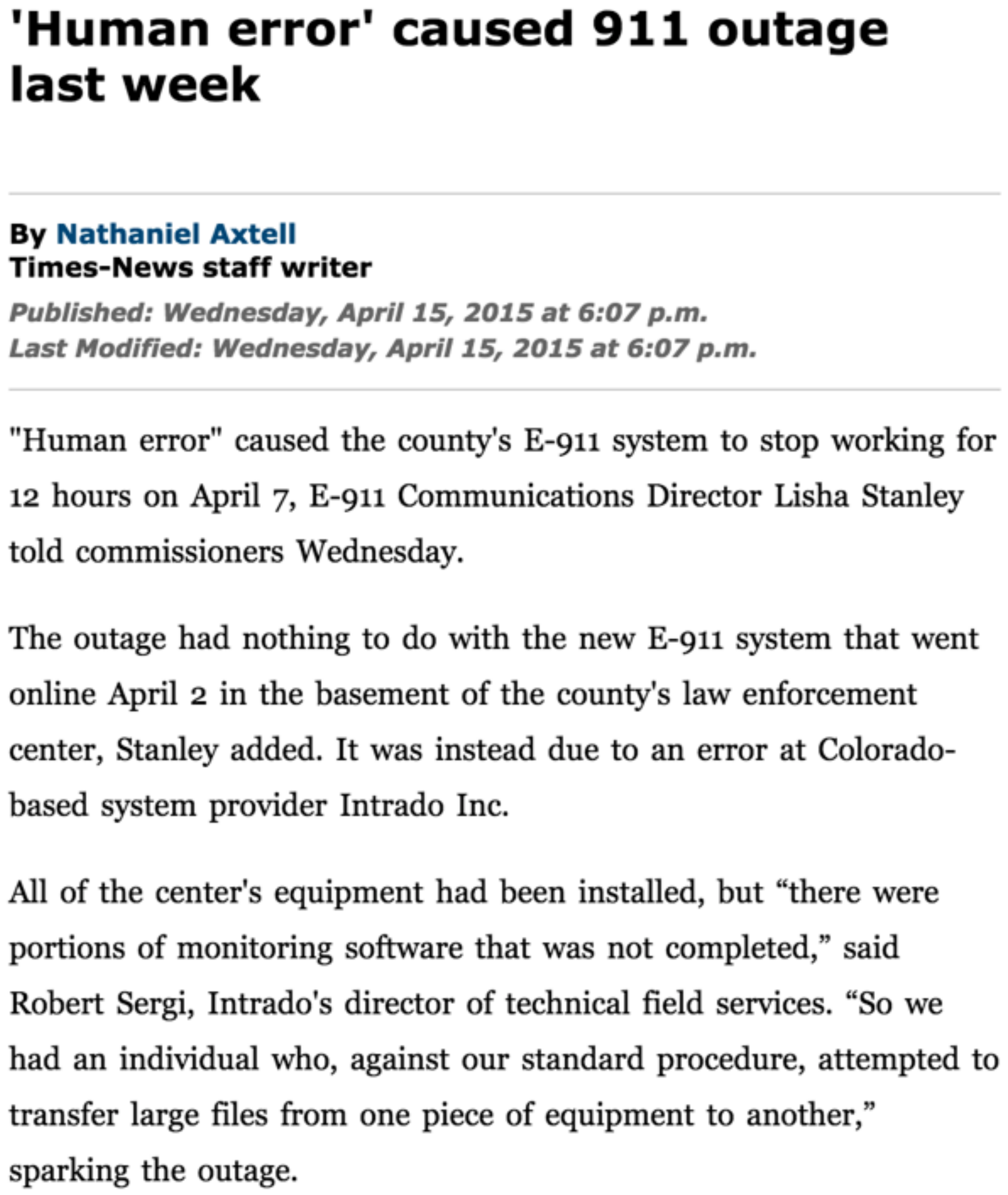
Analysis: 70% of data center outages directly attributable to human error

August 16, 2013
By Matt Vincent
Senior Editor

A new white paper from **APC-Schneider Electric** contends that "a properly designed, implemented, and supported operations and maintenance (O&M) program will minimize risk, reduce costs, and even provide a competitive advantage for the overall business the data center serves. A poorly organized program, on the other hand, can quickly undermine the design intent of the facility putting its people, IT systems, and the business itself at risk of harm or interruption."

See: 8 best practices for data center personnel to follow

The paper's executive summary states that 70% of data center outages are directly attributable to human error, according to the **Uptime Institute's** analysis of their "abnormal incident" reporting (AIR) database. This figure highlights the critical importance of having an effective operations and maintenance (O&M) program, says APC-Schneider Electric.



'Human error' caused 911 outage last week

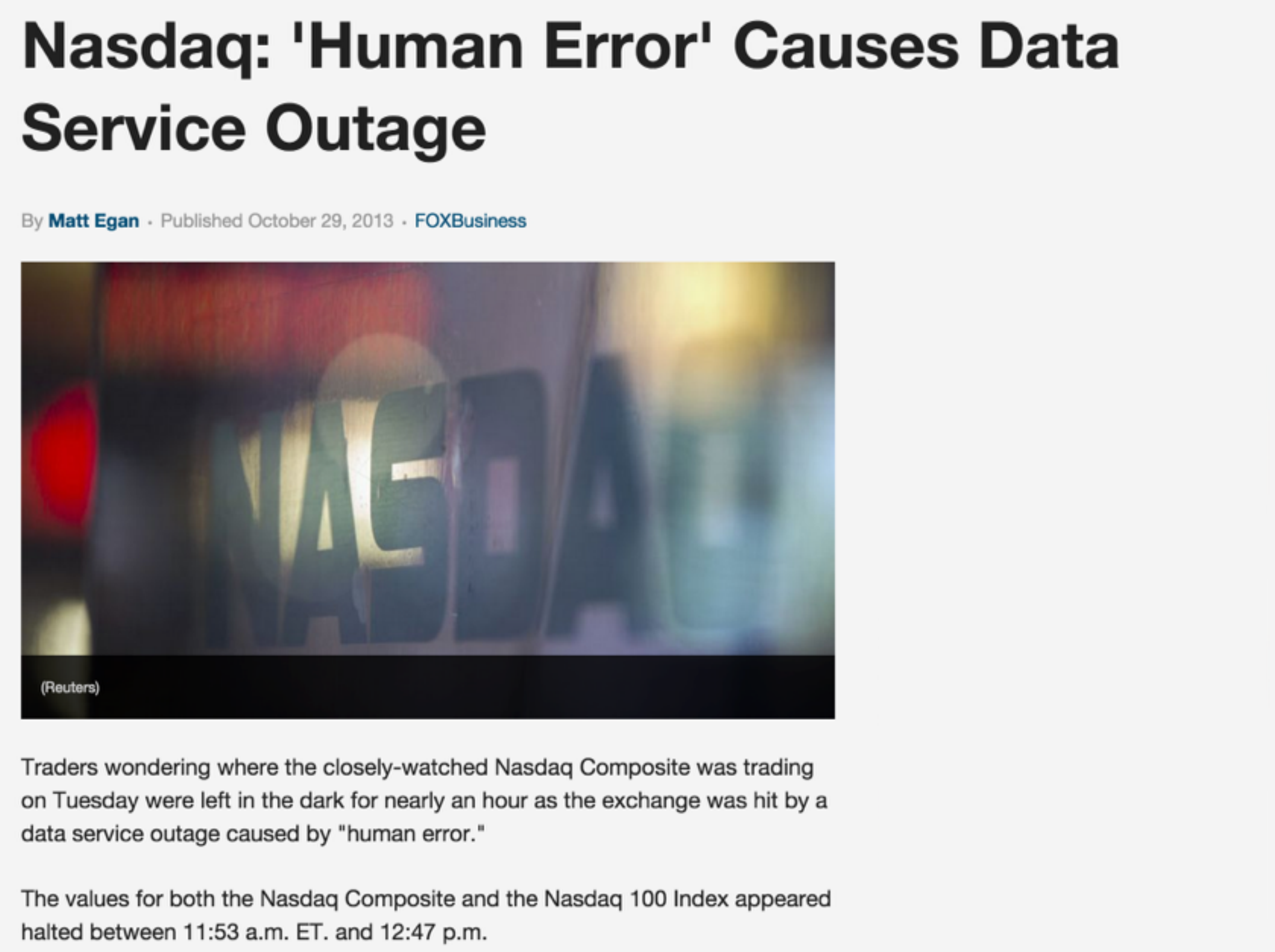
By Nathaniel Axtell
Times-News staff writer

Published: Wednesday, April 15, 2015 at 6:07 p.m.
Last Modified: Wednesday, April 15, 2015 at 6:07 p.m.

"Human error" caused the county's E-911 system to stop working for 12 hours on April 7, E-911 Communications Director Lisha Stanley told commissioners Wednesday.

The outage had nothing to do with the new E-911 system that went online April 2 in the basement of the county's law enforcement center, Stanley added. It was instead due to an error at Colorado-based system provider Intrado Inc.

All of the center's equipment had been installed, but “there were portions of monitoring software that was not completed,” said Robert Sergi, Intrado's director of technical field services. “So we had an individual who, against our standard procedure, attempted to transfer large files from one piece of equipment to another,” sparking the outage.



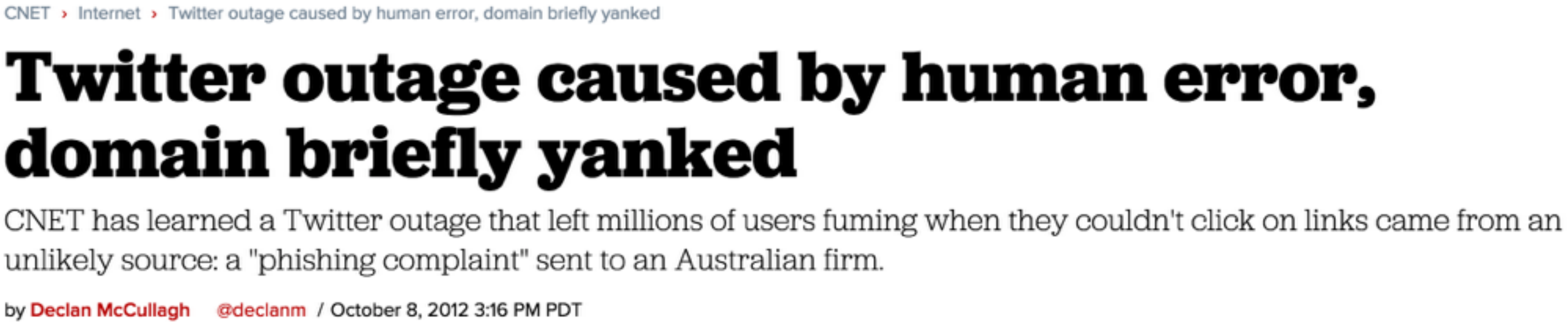
Nasdaq: 'Human Error' Causes Data Service Outage

By Matt Egan · Published October 29, 2013 · FOXBusiness

(Reuters)

Traders wondering where the closely-watched Nasdaq Composite was trading on Tuesday were left in the dark for nearly an hour as the exchange was hit by a data service outage caused by "human error."

The values for both the Nasdaq Composite and the Nasdaq 100 Index appeared halted between 11:53 a.m. ET. and 12:47 p.m.

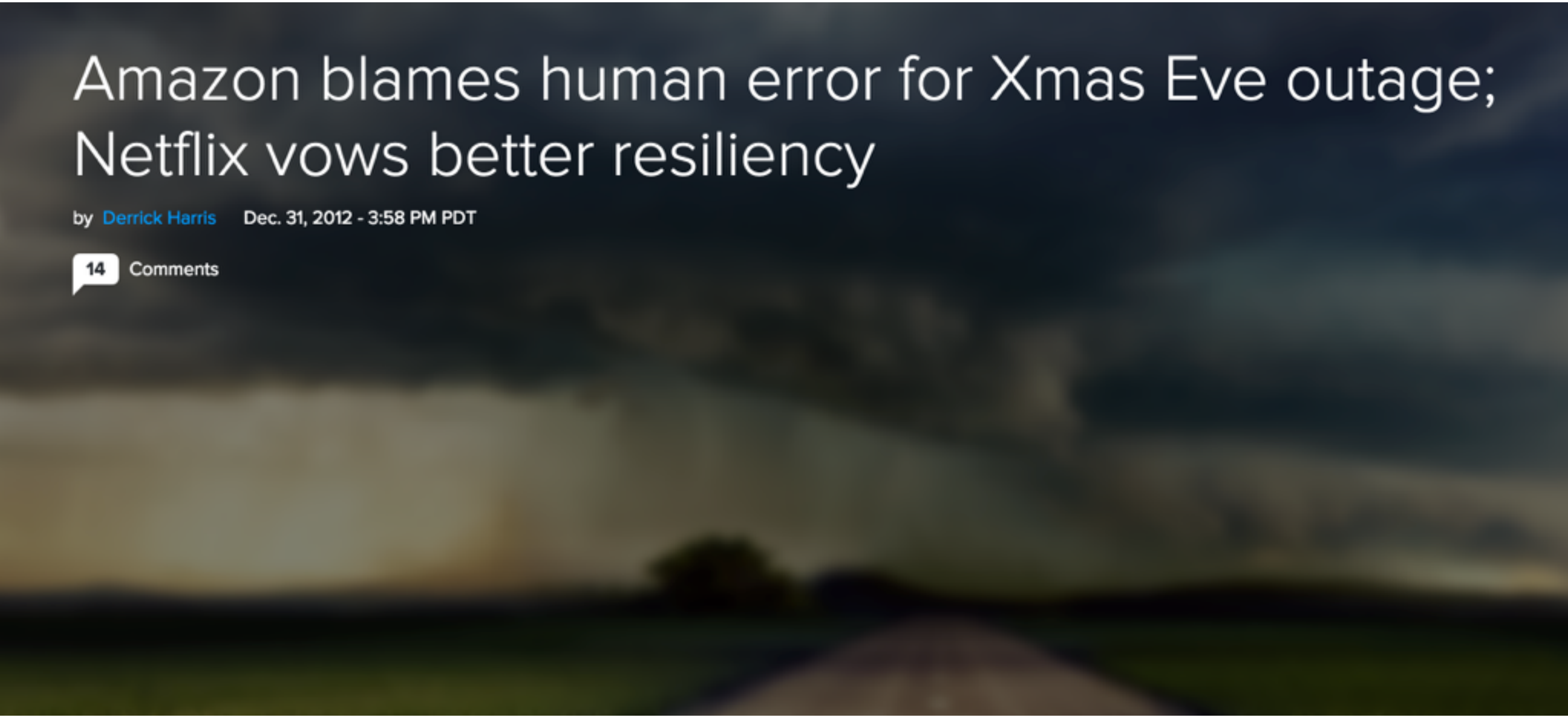


CNET › Internet › Twitter outage caused by human error, domain briefly yanked

Twitter outage caused by human error, domain briefly yanked

CNET has learned a Twitter outage that left millions of users fuming when they couldn't click on links came from an unlikely source: a "phishing complaint" sent to an Australian firm.

by Declan McCullagh @declanm / October 8, 2012 3:16 PM PDT



Amazon blames human error for Xmas Eve outage; Netflix vows better resiliency

by Derrick Harris Dec. 31, 2012 - 3:58 PM PDT

GMAIL OUTAGE NOT APOCALYPSE - JUST HUMAN ERROR

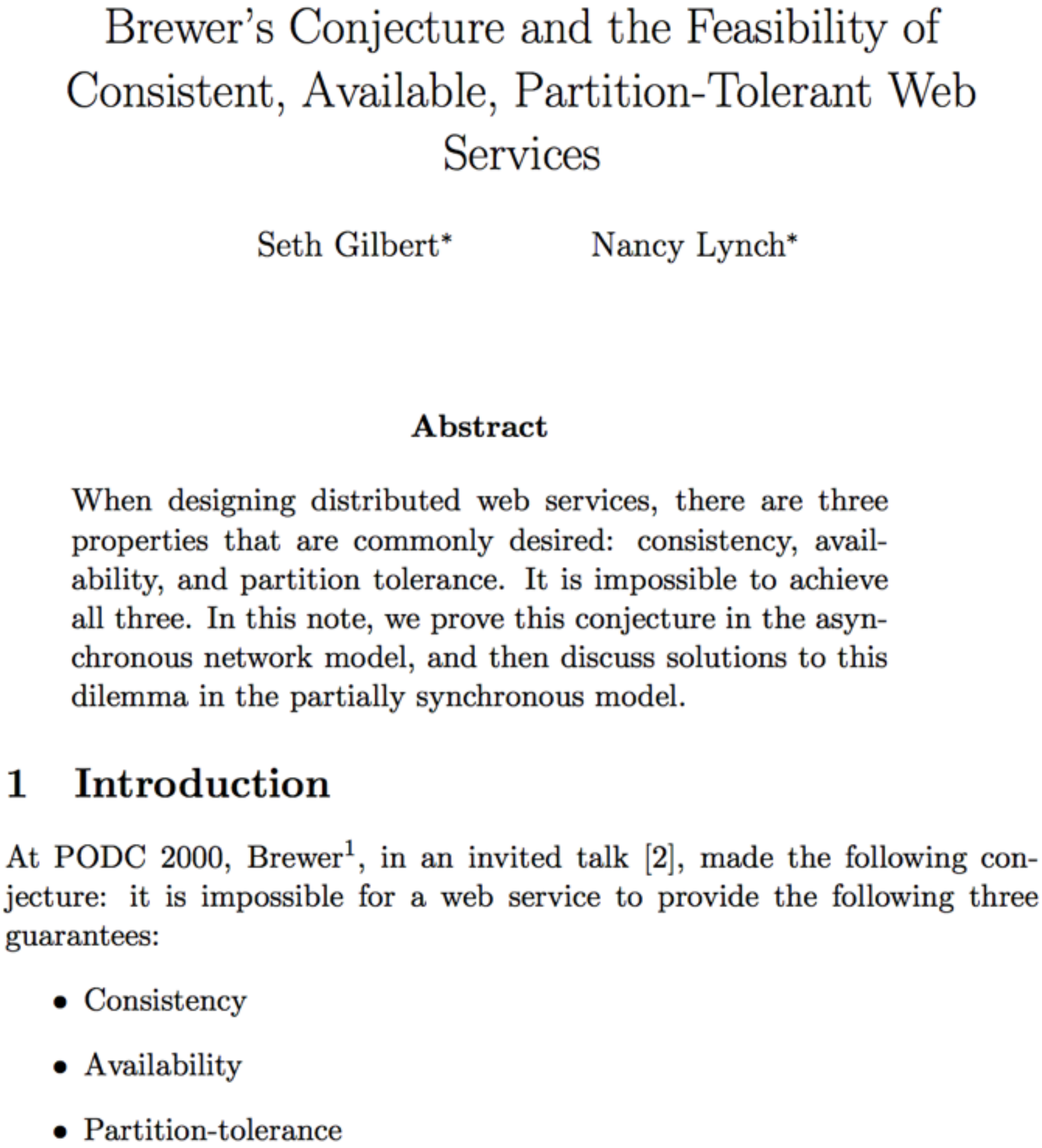
On 11 Dec, 2012 By Christopher Woo

Yesterday, the internet experienced a moment of apocalypse angst when Gmail users around the world (including C2) experienced a variety of issues getting email. Lasting roughly 40 minutes, users experienced complete outages, slowness and, if they were using Chrome with browser syncing enabled, outright application crashes. It turns out, rather than being able to blame ancient prophecies, Google fingered one of their own as the root source of the problem.

What this means for you:

Cloud nay-sayers may have had a brief moment in the sun while Gmail was on the ropes, but the fact remains that it's still a very reliable service. Several lessons may be learned from the experience, all of them common sense:

BREAKING EVERYTHING



Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services

Seth Gilbert[*] Nancy Lynch[*]

Abstract

When designing distributed web services, there are three properties that are commonly desired: consistency, availability, and partition tolerance. It is impossible to achieve all three. In this note, we prove this conjecture in the asynchronous network model, and then discuss solutions to this dilemma in the partially synchronous model.

1 Introduction

At PODC 2000, Brewer[1], in an invited talk [2], made the following conjecture: it is impossible for a web service to provide the following three guarantees:

- Consistency
- Availability
- Partition-tolerance

HAPPY USERS

HAPPY USERS

BREAKING EVERYTHING

THANKS