Case Study: Twitter

Kevin Scannell, David Ferry
CSCI 5030 – Principles of Software Development
Saint Louis University
St. Louis, MO 63103

SAINT LOUIS UNIVERSITY

Twitter

- Created in 2006 by Jack Dorsey (from St. Louis!)
- Major shipper of open source software (200+ projects)
- "Innovator's Patent Agreement" – company has promised to only use patents defensively
- 500 million tweets per day
  - Average rate of ~5,700 tweets per second
  - Peak reported rate of 143,199 tweets per second https://blog.twitter.com/2013/new-tweets-per-second-record-and-how
  - Critical to a company that wants to be a real-time meeting place for world events

Big Data / Big Infrastructure Company

Where is the value in a company like Twitter?

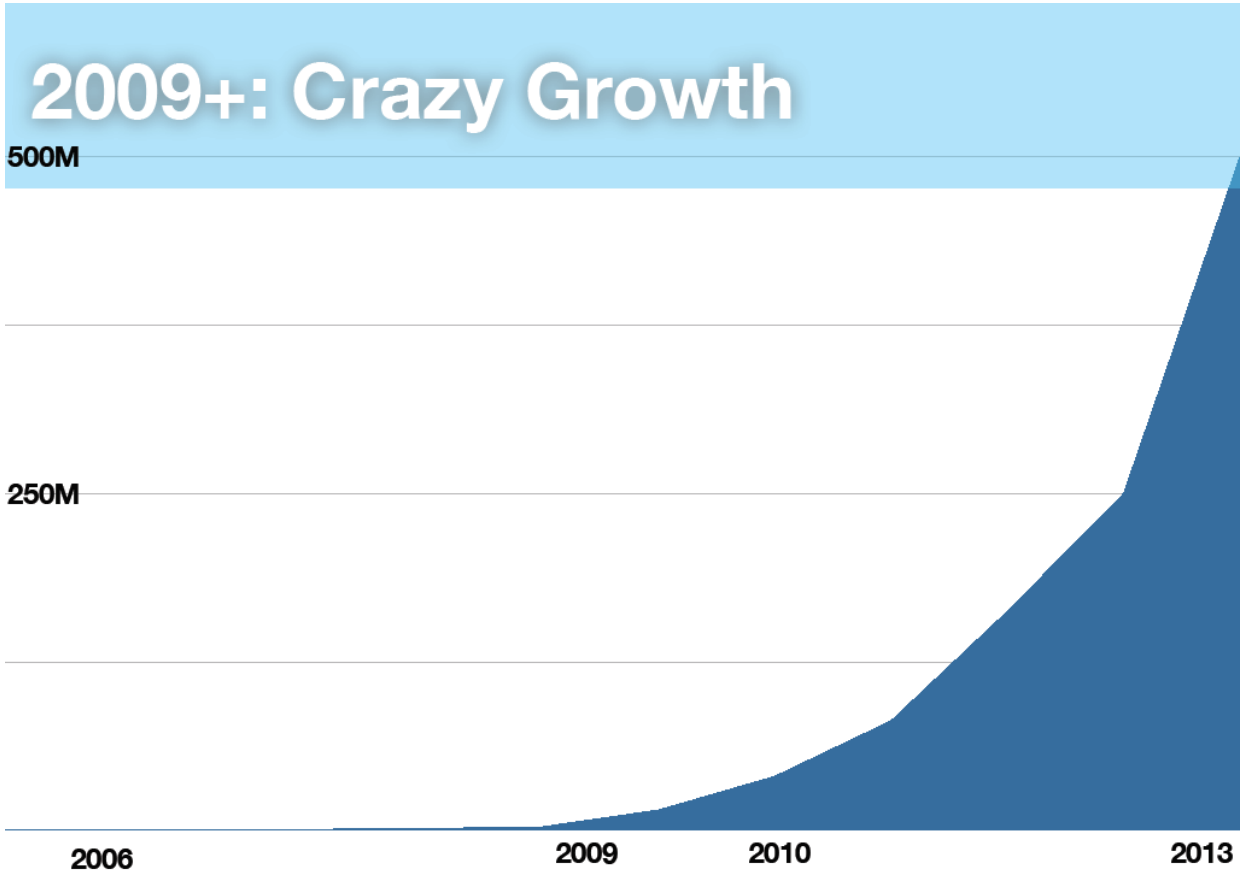
- Users (user data => ad dollars)
- Engineering infrastructure

What would you need to build a Twitter clone?

- Database
- Web server
- Network programming
- Application logic

The basic components are not that complicated…

The idea is simple, the scale is hard



Scaling Twitter with Open Source
Chris Aniszczyk

Data Scale at Twitter

1 tweet ~ 200 bytes

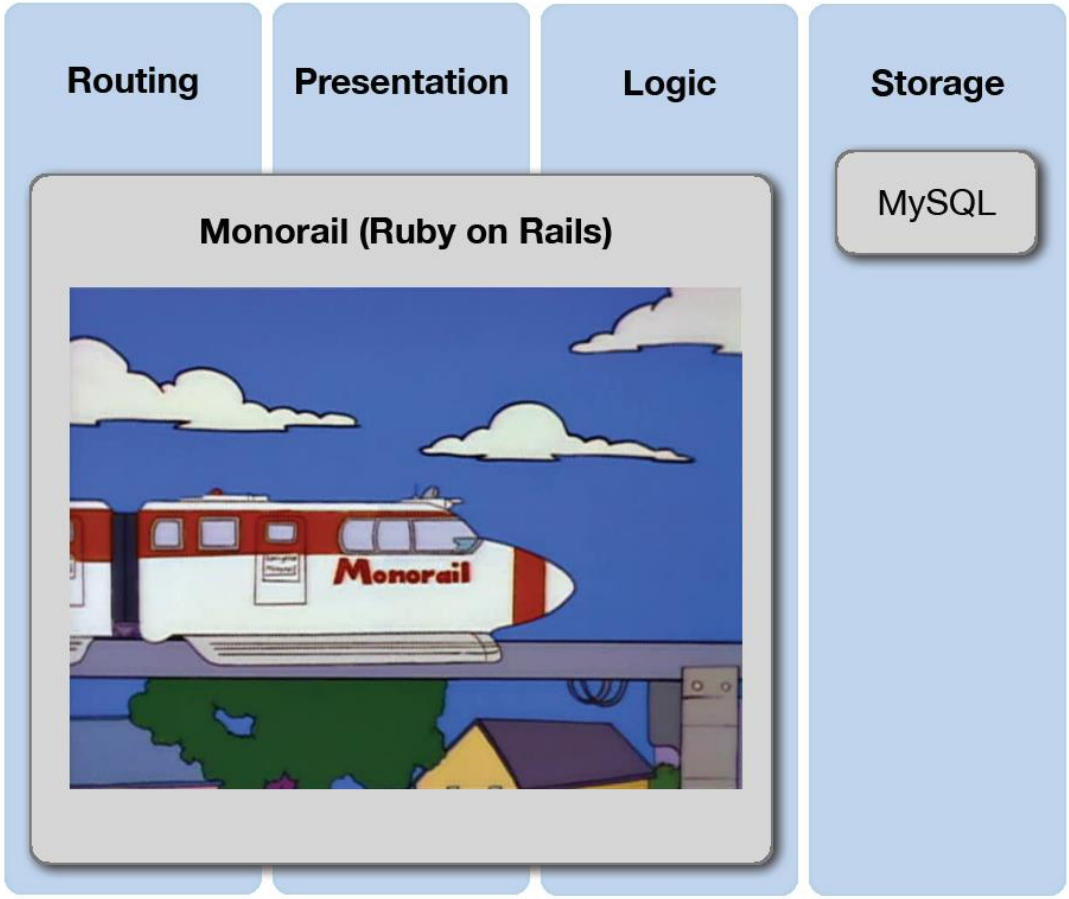
Is that a lot?

- 500 million **new** tweets => writes 100GB/day
- 350 billion tweets **served** => reads 70TB/day
- 4 million tweets per sec. **served** => 800MB/sec

But also in the context of:

- Followers, #hashtags, @mentions, lists, and direct messages (DMs)

Early Architecture



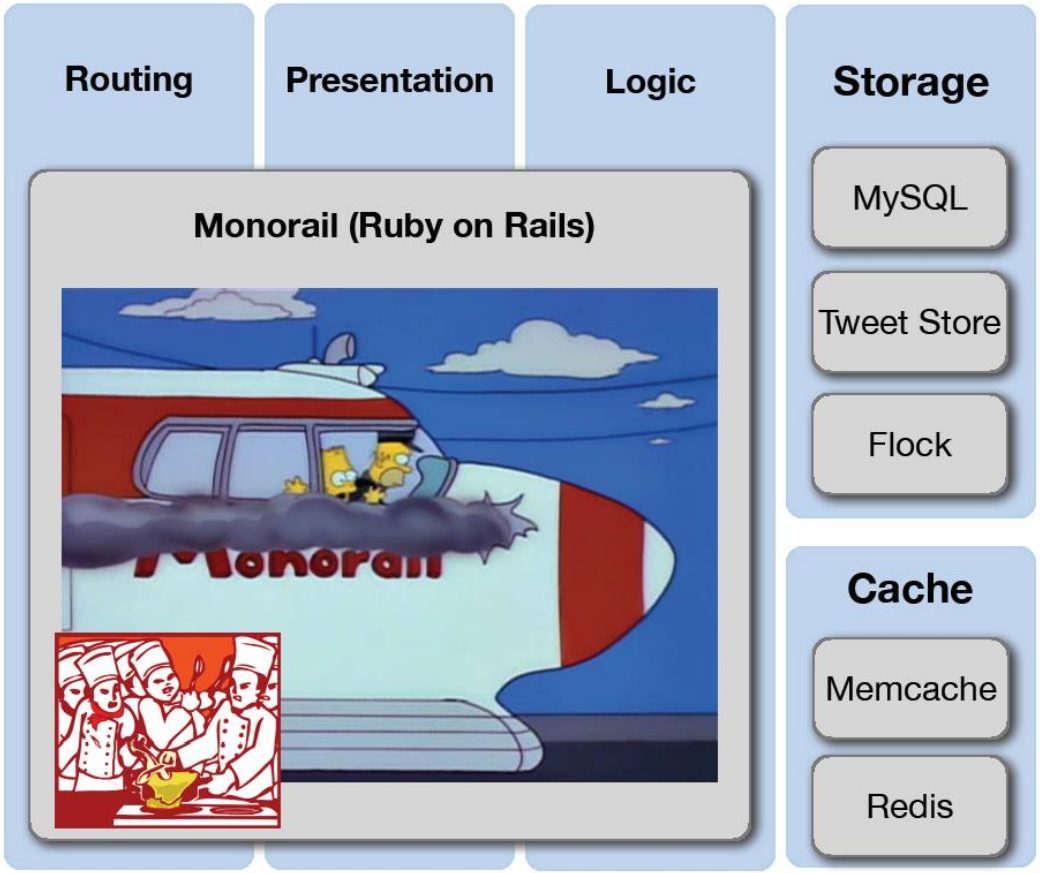
- Monolithic application
- Written in Ruby on Rails
- A highly customized but general purpose "web stack" that you get with a framework
- Business logic mixed into a full stack application

Scaling Twitter with Open Source
Chris Aniszczyk

Monorail Architecture Problems

- Performance
  - Storage layer bottlenecks
  - Single threaded/poor concurrency
  - Single MySQL storage system couldn't keep up
- Brittle
  - All software components part of the same runtime
  - E.g. Japanese translation error brought down entire site
  - Failure "blast radius" is infinite
  - Large data centers reduce mean time between hardware failures to hours, not weeks or months
- Hard to modify
  - Hundreds of developers working on the same codebase
  - Merge conflict chaos with hundreds of feature branches
  - Lack of ownership over functionality
  - Tight coupling between services

Monorail Pushed to the Limit



Scaling Twitter with Open Source
Chris Aniszczyk

- Monorail was a single-threaded program with host-level duplication
- Peak performance was 200-300 requests/sec/host
- Reconsider the growth curve from before and the linear nature of improvement from throwing hardware at a problem
- You want to be at 4 million requests/sec average
- What is the peak-case here?
- "Optimization corner" – team started to trade off readability and flexibility for performance

Intuition: Service Oriented Architecture

Twitter had grown beyond a single web stack

- Separate services into their own components

Desires:

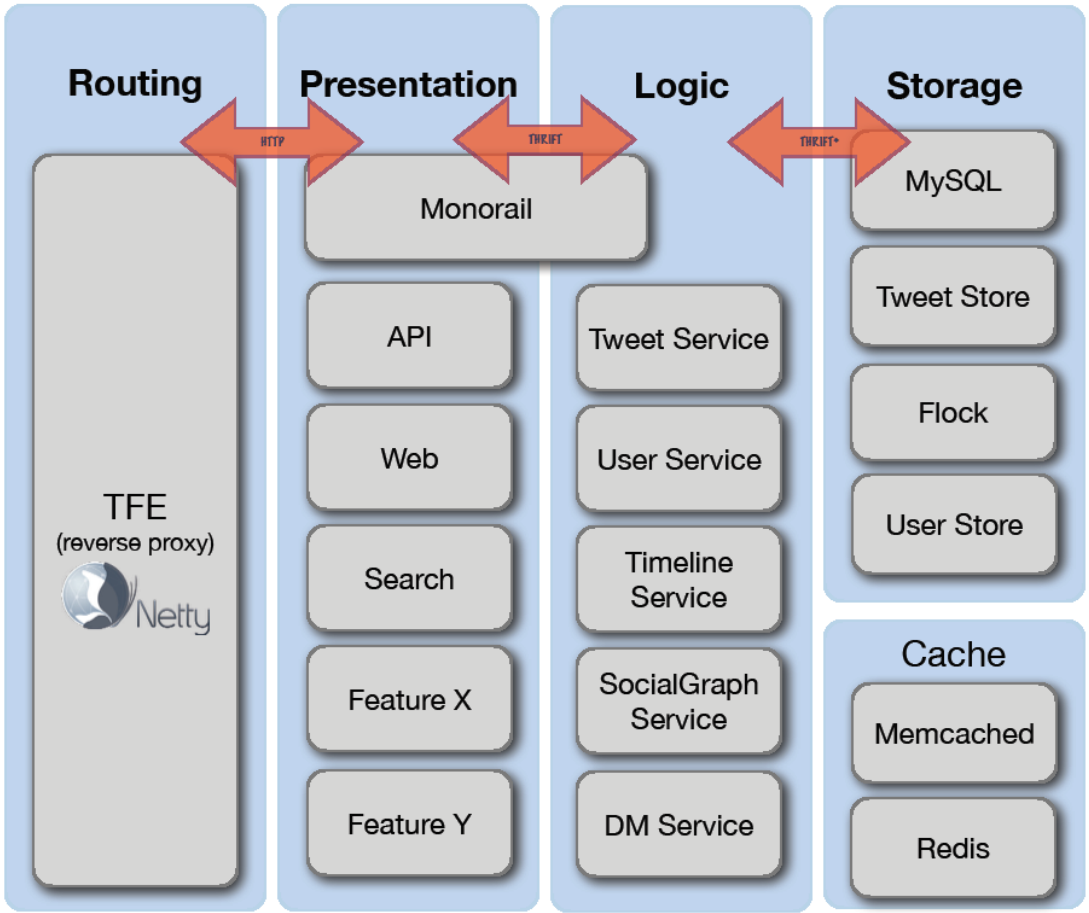
- Site speed/performance
- Isolate responsibilities into components
- Isolate failures / improve reliability
- Improve developer productivity
  - As expensive as machines are, people are worse
  - Clear ownership of services and features
  - Components were a mixed bag – intra-team productivity improved, cross cutting inter-team work got slower

Decomposing Monorail

Monorail carved into a layered architecture based on technical concerns and business groups/objectives within each layer

- Routing Layer
- Presentation Layer
  - Desktop web interface
  - API interface
  - Search, other public-facing features
- Business Logic Layer
  (separation based on core business nouns)
  - **Tweet** service
  - **User** service
  - **Social Graph** service
  - **Timeline** service
  - **Direct Message** Service
- Everything is connected via remote procedure call (RPC)

Modern Twitter



Scaling Twitter with Open Source
Chris Aniszczyk

Decomposition Challenges

You're asked to componentize part of a monolithic application… what do you do? Copy-paste!

Except now your architecture is:

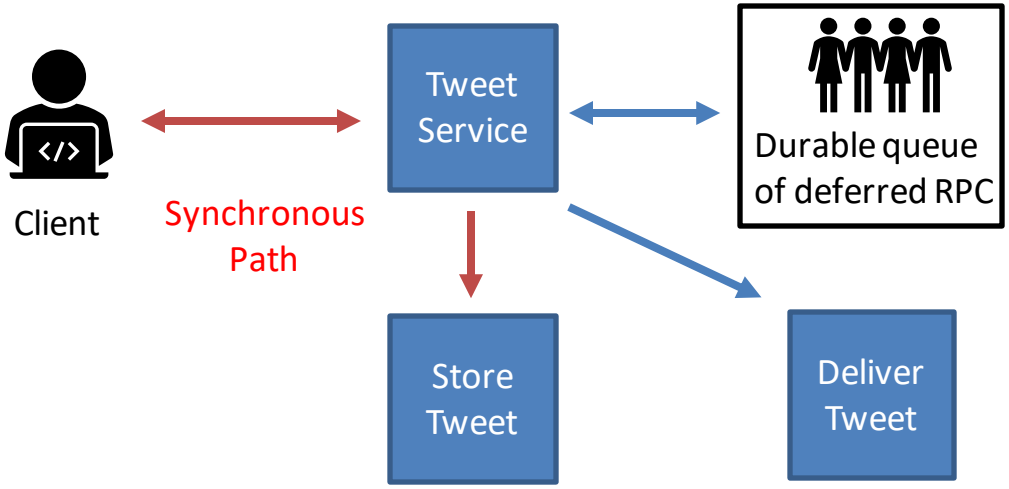
- Distributed
- Networked
- Many function calls are now RPC

…and most of your engineers haven't done that before.

Each team handles problems in slightly different ways:

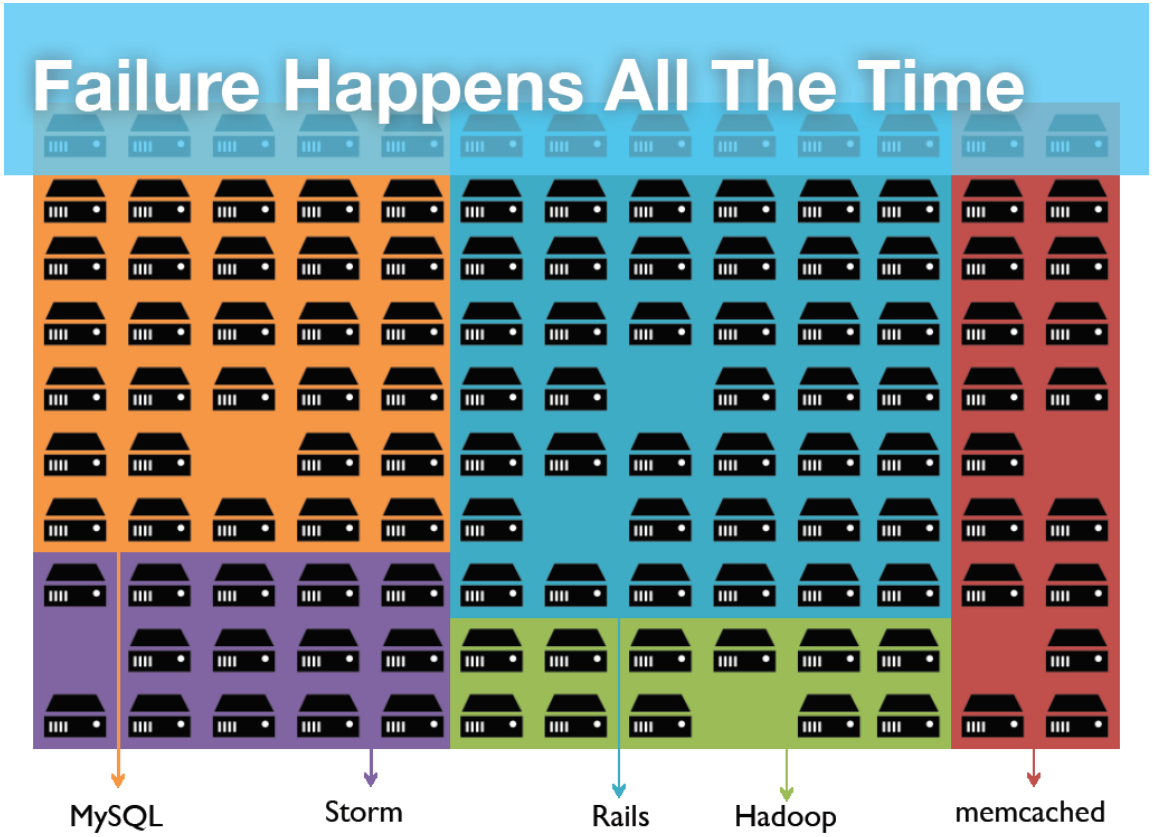
- Error handling/network issues/retrying
- Load balancing
- Service discovery
- Stats collection and distributed tracing

Leveraging Concurrency



- Storing tweets is handled synchronously so that service can return success to the client, but making the whole process synchronous would be too heavyweight (suppose if you have millions of followers)
- Handling delivery of those tweets can be deferred
- Delivery actions placed in a durable queue that survives crashes
- Delivery actions are idempotent- retries do not cause error
- Client submits tweets with low latency, and if they get an OK response then delivery is guaranteed- or even be farmed out to different services

Collection of Hosts => Datacenter



- Static partitioning is brittle
- Manual oversight is slow
- Scaling issues cause load imbalance
- Failures require human intervention

Scaling Twitter with Open Source
Chris Aniszczyk

Results

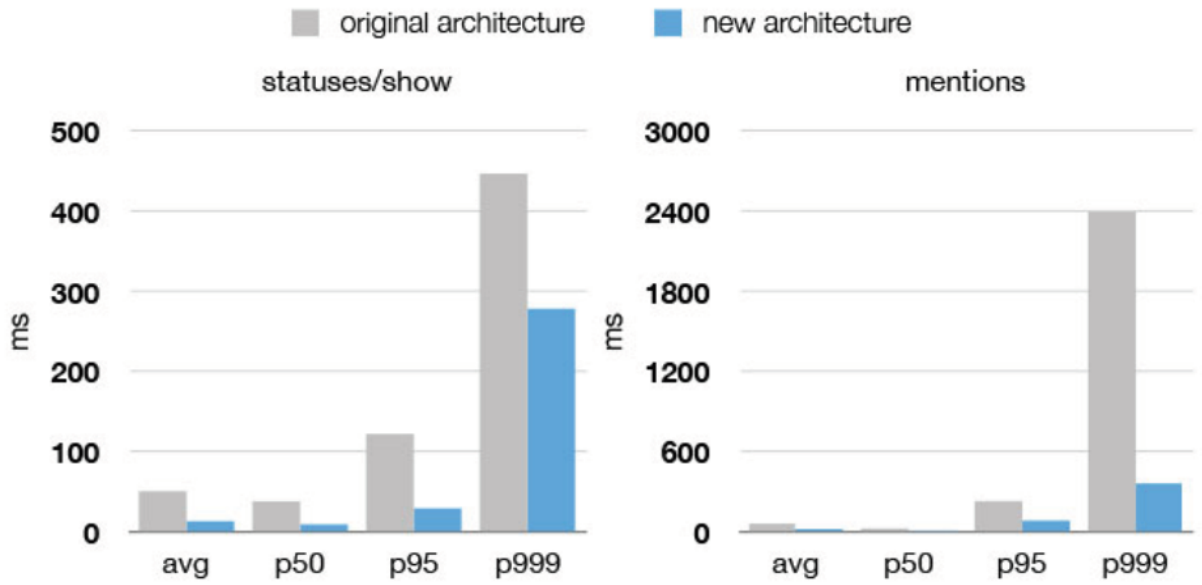
Many other things could be said architecturally, but we have to stop somewhere

Technical team hoped that each hardware host could handle 10x as many requests

- Started with 200-300 requests/sec per host
- Achieved 10K-20K requests/sec per host
- Better architecture led to 100x better performance with the same hardware

Latency Improvement

Performance Today :)

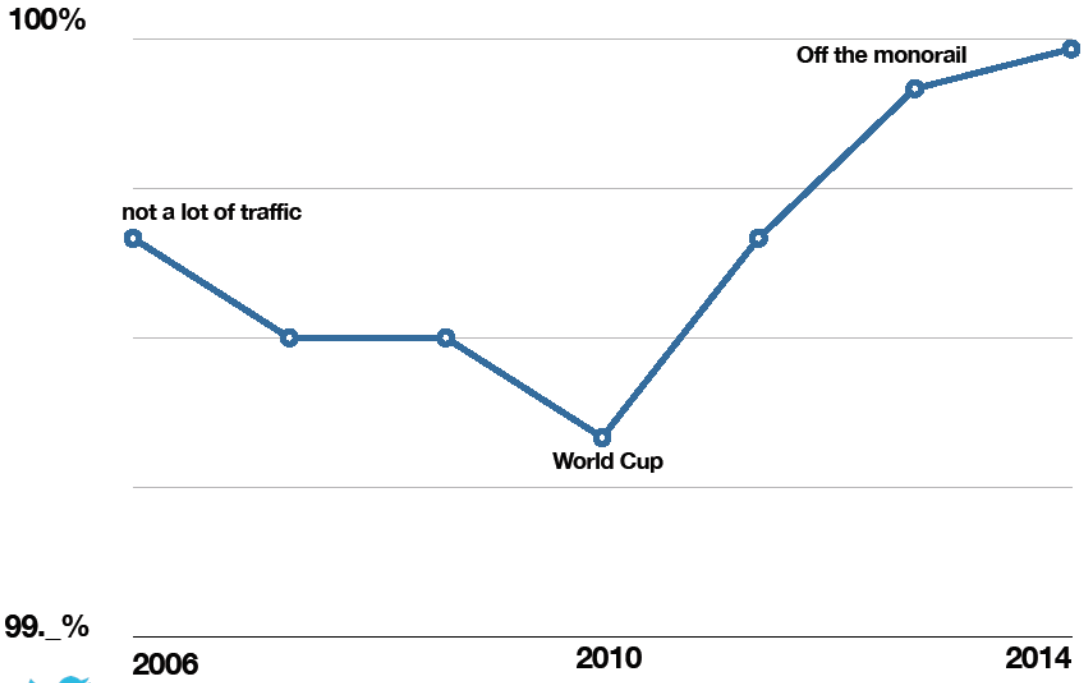


- Average case latency improved
- Some of the biggest gains in the long tail distribution

Scaling Twitter with Open Source
Chris Aniszczyk

Reliability Improvement

Site Success Rate Today :)



Scaling Twitter with Open Source
Chris Aniszczyk

- Reliability improved, even better than when they had almost no traffic

Lessons Learned

The software architecture did many things:

- Separated business logic from technical challenges
- Focused efforts of developers and clarified responsibility
- Provided a concurrency-heavy programming model
  - Allowed much higher throughput
- Minimized the impact of software/hardware failures