# CRANGRAPH

James, Jason, Surya

# Roadmap

- Core Requirements
  - Problem space: R package dependencies
  - Architecture overview
    - Live Demo
  - Functionality overview
- Extra Requirements
  - How to scale Crangraph
  - How to evolve the project

# Problem Space: R Package Dependencies

**R is a popular open-source software suite used for statistics.**

**Management of interdependencies in the R ecosystem is a non-trivial concern for data science shops. Several alternative package management solutions have been proposed for those doing heavy-duty development in R.**

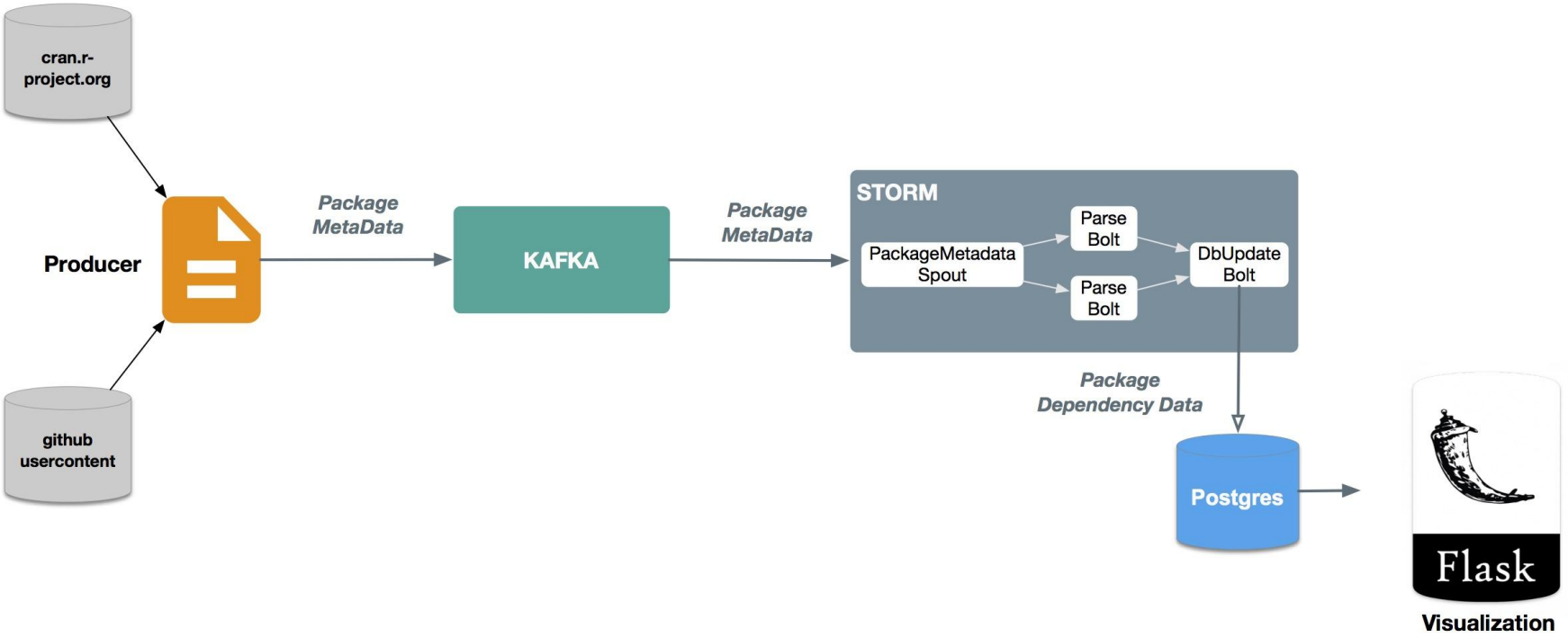
**Breaking changes do happen in the open source world (sometimes unintentionally) and each new external dependency you take on raises your risk of disruption from these changes.**

# Project Summary: Implementation Details

**We propose building a real-time dependency graph for R so developers can make informed decisions about the dependencies they take on.**

- Model new package releases as a stream
- Store package metadata changes in Kafka
- Analyze the data using Storm
- Front-end visualizations
    - Current-state network graph (in D3)

# Architecture Overview



# Architecture - Kafka Producer

**To get package metadata into our application, we use a Kafka producer written in Python using the python-kafka library. This producer runs the following simple algorithm:**

1. **Pull the source of the CRAN package index**
2. **Scrape that page for a list of package names**
3. **Iterate over the list:**
    a. **For each package, pull DESCRIPTION file and put it directly onto a Kafka topic**
4. **Once all packages have been pulled and put on the topic, re-pull the package index and repeat**

# Architecture - Kafka Producer

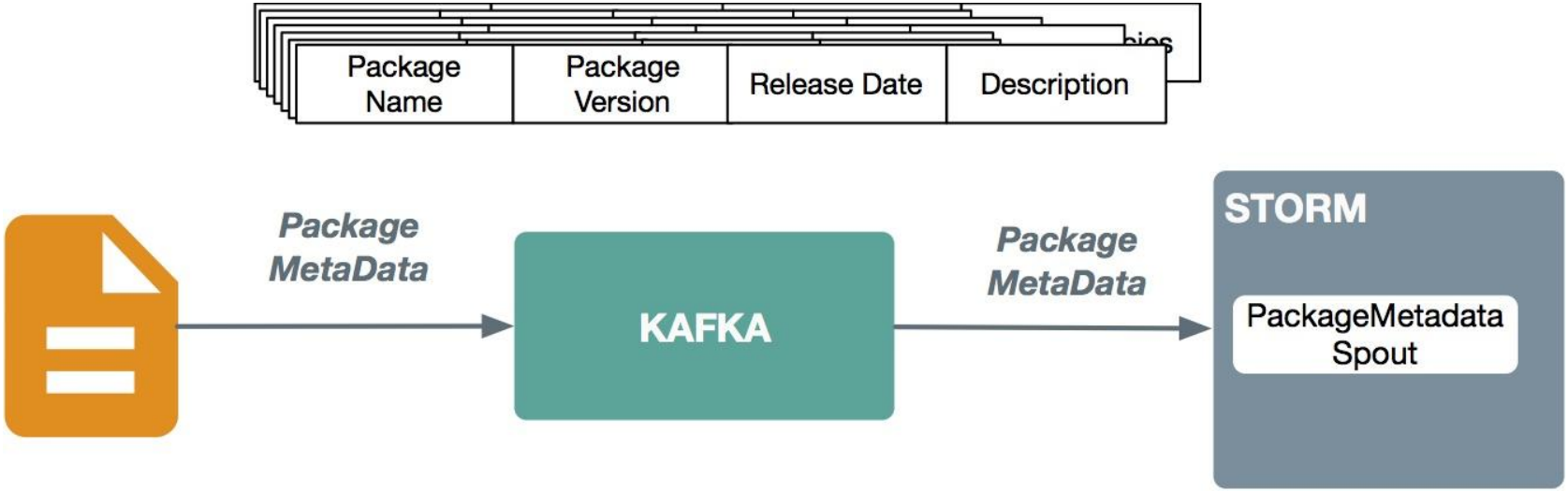
- Takes about 1 hour to cycle through the entire package list, pull DESCRIPTION files, and get them on the topic
- Implies that each package could be checked for updates 24 times per day. This is more than sufficient coverage for users, and could even be slowed if the application became overworked.

# Architecture - Data Storage, Retrieval & Processing

**Crangraph is both compute and storage intensive. The amount of incoming data can be bursty in nature so it needs a scalable solution. To address this we have used a combination of Kafka, Apache Storm and Postgres.**

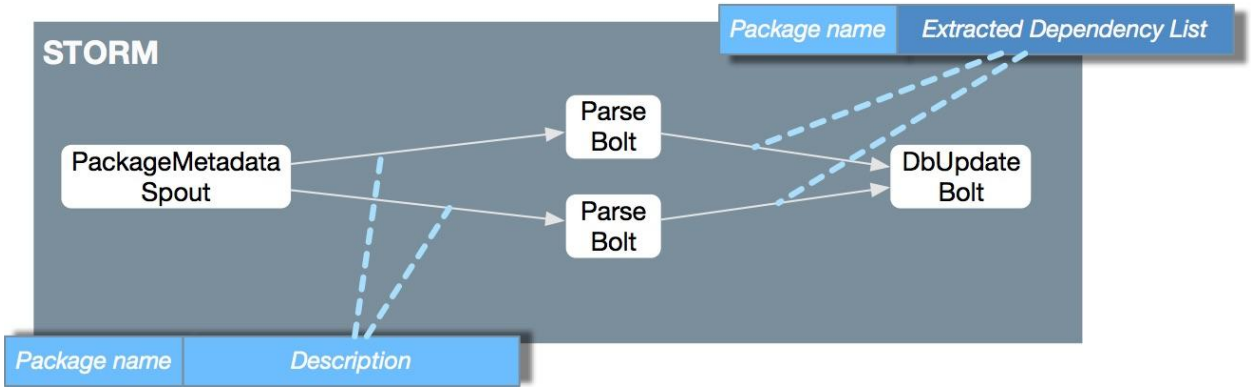
**The producer sends the R package metadata as logs to Kafka server. At the other end, we connected a PackageMetadata spout that is configured as a Kafka consumer and implemented to fetch the package metadata from the Kafka server.**

# Architecture - Data Storage, Retrieval & Processing



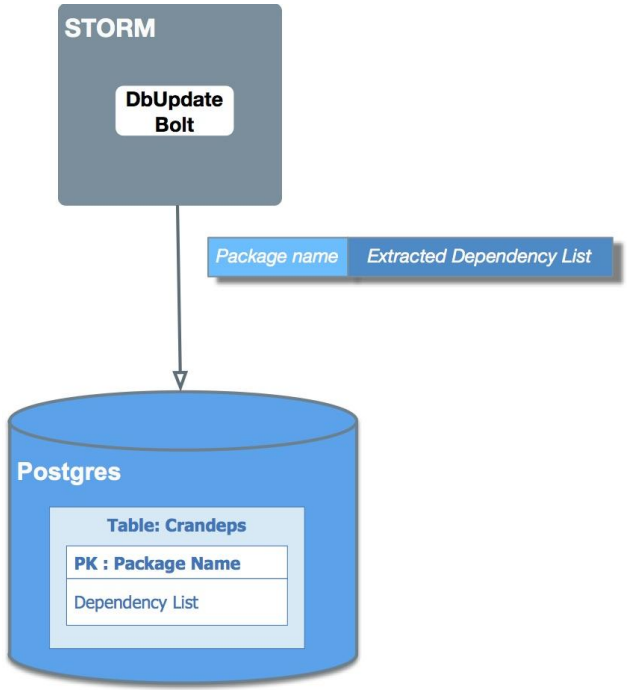
# Architecture - Data Storage, Retrieval & Processing

**The PackageMetadata spout then emits the package name and description to the parse bolt. The parse bolt scrapes through the description and extracts dependency list that is then given to the database update bolt.**



# Architecture - Data Storage, Retrieval & Processing

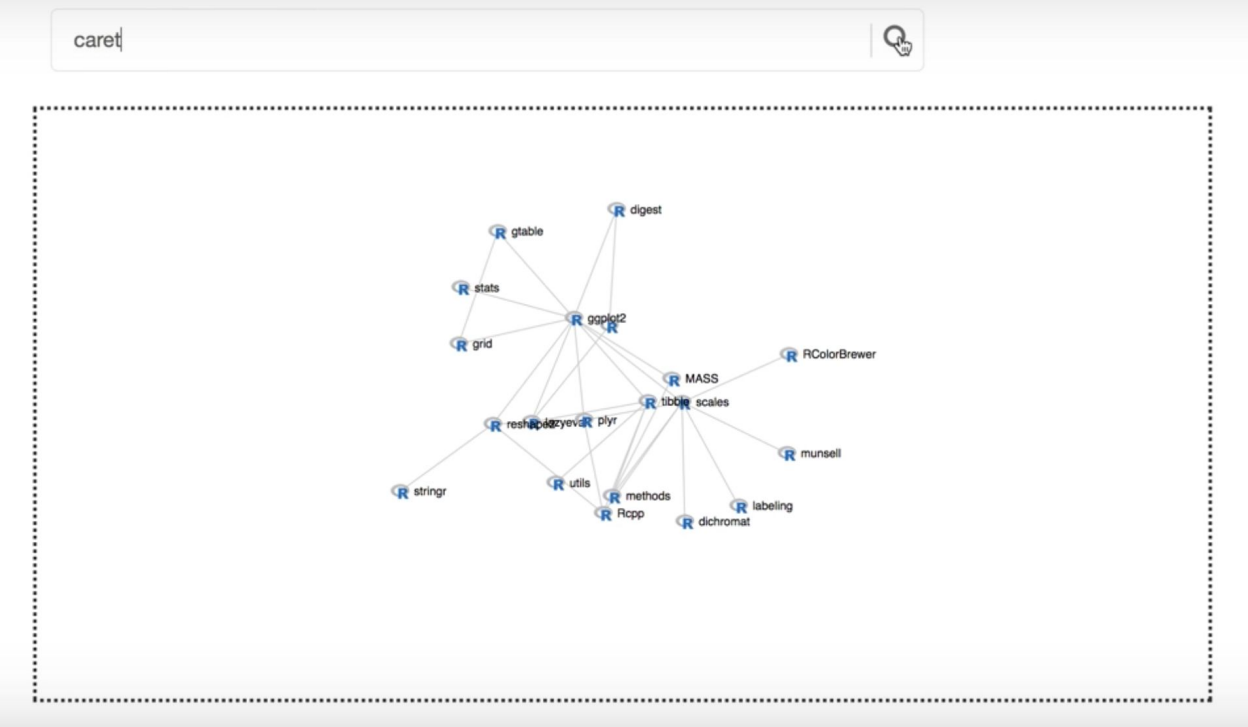
- The database update bolt stores the parsed information in a postgres database named crangraph in a table named crandeps.
- The table stores the package name as primary key and the dependency list in JSON.
- To interface with postgres, psycopg2 is used within the database update bolt.



# Architecture - Visualization

- Crangraph's user interface is a simple web app which shows a force-directed graph of a package's 1st-degree and 2nd-degree dependencies.
- Frontend: D3 force-directed graph
- Backend: Flask + NetworkX
- Algorithm: The whole Crangraph data is first loaded into a NetworkX graph object. Then we create a subgraph of a package, its neighbors and its neighbors' neighbors in the flask server.

# Live Demo

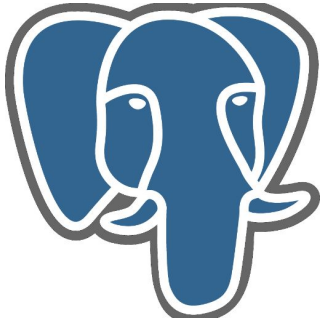

caret
digest
gtable
stats
ggplot2
grid
RColorBrewer
MASS
scales
plyr
munsell
utils
stringr
methods
Rcpp
labeling
dichromat

# Kafka - Streaming Message Queue

- We model R package updates as a stream.
- Crangraph places raw package metadata onto a Kafka topic and uses a Kafka consumer as the spout for its main Storm topology.
- We chose Kafka for this application because it is built to handle bursty streams and multiple data consumers.

# Storm - Stream Processing Framework

- Once package metadata have been written to Kafka, they must be parsed into the format required by the UI and stored in our PostgreSQL serving database.
- Apache Storm is used to organize this processing and to distribute multiple tasks in the processing DAG across physical resources available to the application's EC2 instance.

# Flask, D3, Nginx - Simple Web App

- Flask - Serves the webpage and graph of dependencies in JSON format
- D3 - Interactive visualization of graph dependencies between packages
- Nginx - Communication between external clients and internal server

# How to Scale Crangraph

- Monitoring the resource usage and failure points
- Scaling out our tech stack with microservices
  - A large topology of Kafka and Storm nodes
  - Distributed Graph database
  - Graph-parallel visualizations (e.g. Spark GraphX)
- Move to an image with more vCPUs then add more parsing bolts
- Add better dedupe logic so we don’t waste CPU cycles
- Measure growth of logs and tune verbosity + persistence strategy
- Give more resources (memory and compute) to webserver

# How to Evolve the Project

1) Support arbitrary snapshots
    - E.g. Crangraph on Feb 2017
2) Network metrics (link density, centrality, “influencer” packages)
3) Identify development communities
4) Detecting removals from CRAN
    - We currently only allow additions of dependencies to the graph

# How to Evolve the Project

4) Animation of the evolution of Crangraph over time
5) Better app practices
    a) Security practices
    b) Error handling
    c) Strategy for deleting/archiving logs so they don't grow indefinitely
    d) Strategy for tuning how frequently we check for updates

# Thank you!