KALEVA.fi

How we replaced 10 years of legacy with Django & AWS

Contents

- Background stuff -- who are we and why are we doing what we do?
- Technology bits -- what are we doing?
- Processes -- how are we doing it?

Who are we?

- Kaleva Oy
- Founded in 1899
- 520 employees
- Kaleva -- the printed newspaper
  - 192k readers every day
  - 4th largest 7-days/week newspaper in Finland
- commercial print

Okay, but who are we?

- Kaleva Digital Business
  - separated as its own unit a couple of years ago
  - 4 developers, 1 graphic designer
  - bunch of sales people and managers
  - “startup” within the company
- kaleva.fi
  - ~190k unique visitors/week
  - ~10M pageviews per month
  - 75% of users visit at least once a day
  - 85% of uses come from Oulu Province

*) Metrix 2011

Not just kaleva.fi

- our unit has huge growth goals
  - not possible with just the news portal
- completely new services in the works
  - rapid prototyping!
- openness (data, services, code, ideas)

Replacing legacy kaleva.fi

What did we replace?

- 10 years of legacy ColdFusion code
- Development practices had remained almost the same the whole time
  - mostly everything was done ad hoc
  - no version control
  - no automated testing
  - no test environments and/or DBs --development directly in production
  - no documentation
- A lot of much silent information concentrated on a few people

Kaleva.fi - rich platform

- news, mobile site, blogs, reviews, comments, events calendar, discussion forums, reader photos, videos, photo galleries, cartoons, mini sites, polls, competitions, weather...
- sister publications: Kiekko-Kaleva, Hightech Forum
- loads of customized tools for journalists and other content providers
- a plethora of integrations: content feeds from print newspaper, Kaleva's photo journalists, STT-Lehtikuva, European Photo Agency, SMS/MMS gateways plus numerous other partners

What did we want?

- Agility, reusability and scalability
  - Processes
  - Tools
  - Infrastructure
  - Code
  - People
- From “thoughtland” to working prototype in one spike
- Metrics vs. opinions

Choosing the core

- why not an existing CMS?
  - independence from vendors' roadmaps
  - not having to change the way journalists do their work
- we're not just building a news portal
  - new services that have very little in common with Kaleva.fi
  - reusability, common practices (development, packaging, deployment, ops...)
- how not to build every feature from scratch?
- How to avoid feature creep?

Why we chose Django?

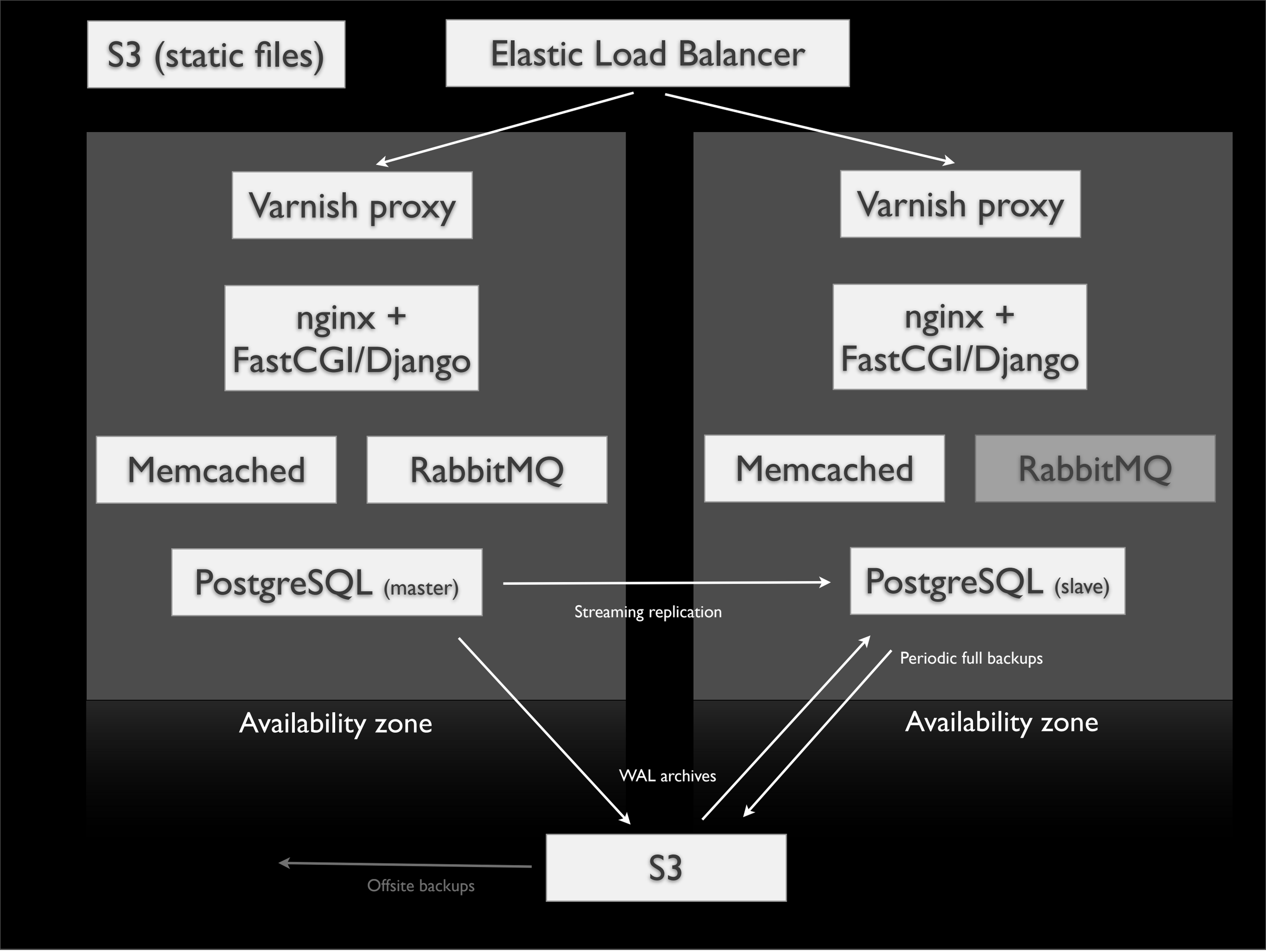
- Has CMS-like functionality built-in that can we could use
- Design principles guide towards a consistent SW architecture
- Fast to develop
- Proven & scalable
  - Chicago Tribune, The Onion, New York Times...
  - Disqus, Instagram, Pinterest, Google
- Strong community
- Python <3

Technology bits

Kaleva.fi v2 - what did we build?

AWS

Region	Geographical location of resources/services: Ireland, US east coast...
Availability Zone	Each region is divided into logical data centers
ELB	Elastic Load Balancer distributes load on attached instances and can perform SSL offloading
EC2	Elastic Compute Cloud service -- the virtual machines instances you use to build your services
EBS	Elastic Block Storage -- the network storage that you can mount on your EC2 instances
S3	Simple Storage Service -- highly durable (99.999999999%) storage service provided by Amazon


S3 (static files)
Elastic Load Balancer
Varnish proxy
Varnish proxy
nginx + FastCGI/Django
nginx + FastCGI/Django
Memcached
RabbitMQ
Memcached
RabbitMQ
PostgreSQL (master)
PostgreSQL (slave)
Streaming replication
Periodic full backups
Availability zone
Availability zone
WAL archives
S3
Offsite backups

S3 (static files)

Elastic Load Balancer

Varnish proxy

nginx + FastCGI/Django

Memcached

RabbitMQ

Varnish proxy

nginx + FastCGI/Django

Memcached

RabbitMQ

PostgreSQL (master)

Streaming replication

PostgreSQL (slave)

Periodic full backups

Availability zone

Availability zone

WAL archives

S3

Offsite backups

Elastic Load Balancer

- Balances load to Varnish proxies in different AZs
- One virtual instance, scales automatically depending on the load
- Capable of SSL offloading
- There's a catch:
  - you cannot point to an IP so A records are out of the picture
  - If you want naked domains (e.g. domain.com), you have some choices:
    - Route53
    - Redirects

S3 (static files)

Elastic Load Balancer

Varnish proxy

Varnish proxy

nginx +
FastCGI/Django

nginx +
FastCGI/Django

Memcached

RabbitMQ

Memcached

RabbitMQ

PostgreSQL (master)

Streaming replication

PostgreSQL (slave)

Periodic full backups

Availability zone

Availability zone

WAL archives

S3

Offsite backups

Varnish

- A must-have for read-heavy sites
- Most challenges in news portals are related the giant user peaks when something extreme has happened
  - shootouts, natural disasters etc.
- Ideal for Varnish: few unique pages, loads of users
- Even in normal day, most users hit the front page and a reasonable number of “hottest news” of the moment

Varnish

- We're seeing an average cache hit rate ~80%
- CPU load insignificant
- Works very well with default configurations
  - Do not store your cache files in EBS, instead use RAM :)
- Caching non-personalized content is easy
- E.g. cookies (think of CSRF tokens) may need some extra work
- While caching is easy, invalidation is harder

Varnish

- Currently, in most scenarios we use a somewhat naive TTL-based cache strategy
  - Origin servers return `max_age` headers, which Varnish obeys
  - Can be overriden by Varnish
  - Key is to also use the built-in cache infrastructure in internet: allow any intermediate proxies and the User Agent to cache the content
  - Grace period allows Varnish to serve stale content when fetching updated versions from backends

Varnish

- Caching and static files
  - version x of site:
    ```
    <link rel="stylesheet" type="text/css" href="kalevafi-min.css" />
    ```
  - Response headers:
    ```
    [...]
    Cache-Control: max-age=2592000
    [...]
    ```
    - Changes to the CSS will be visible to users after 30 days
- File versioning to the rescue!
  - On deployment, append MD5 sum to the names of static files:
    ```
    <link rel="stylesheet" type="text/css" href="kalevafi-min--beac47.css" />
    ```
  - Nginx ignores the MD5 sum
  - Checksum only changes when the static file has changed
  - Hide the complexity to e.g. a custom templatetag
  - We're still using Django 1.3.x, might be able to move from custom implementation to Django's static file versioning in 1.4

S3 (static files)

Elastic Load Balancer

Varnish proxy

nginx + FastCGI/Django

Memcached

RabbitMQ

Varnish proxy

nginx + FastCGI/Django

Memcached

RabbitMQ

PostgreSQL (master)

Streaming replication

PostgreSQL (slave)

Periodic full backups

Availability zone

Availability zone

WAL archives

S3

Offsite backups

nginx + Django

- nginx -> FastCGI -> Django
- Simple throwaway work horses
- Easy to scale (Autoscaling, yay!)
- CPU-heavy
  - Currently “fast enough”
  - intuition says should be faster, we’ve got profiling to do
  - we want to see how uWSGI, Gunicorn or perhaps Jinja2 would perform under live load
    - easy because new instances can be launched within minutes
- Nginx handles compression of content
  - gzip_vary makes it work with Varnish (`Vary: Accept-Encoding`)

S3 (static files)

Elastic Load Balancer

Varnish proxy

Varnish proxy

nginx + FastCGI/Django

nginx + FastCGI/Django

Memcached

RabbitMQ

Memcached

RabbitMQ

PostgreSQL (master)

Streaming replication

PostgreSQL (slave)

Periodic full backups

Availability zone

Availability zone

WAL archives

S3

Offsite backups

Memcached

- Each availability zone has two Memcached nodes (failure/memory corruption within AZ only affects 1/2 of the keyspace local to the zone)
  - We're discussing about moving to Amazon ElastiCache (protocol-compatible with Memcached)
- Precalculated things (e.g. counts, which are very slow with PostgreSQL)
- Generally do whatever you can to reduce amount of queries hitting the database
- Again, we mostly use a naive TTL-based eviction policy
  - not trivial to co-ordinate TTLs between Varnish and Memcached optimally
- The way to handle peak traffic if requests do not hit Varnish cache
- Effective for contents that are common to several different pages

S3 (static files)

Elastic Load Balancer

Varnish proxy

nginx + FastCGI/Django

Memcached

RabbitMQ

Varnish proxy

nginx + FastCGI/Django

Memcached

RabbitMQ

PostgreSQL (master)

Streaming replication

PostgreSQL (slave)

Periodic full backups

Availability zone

Availability zone

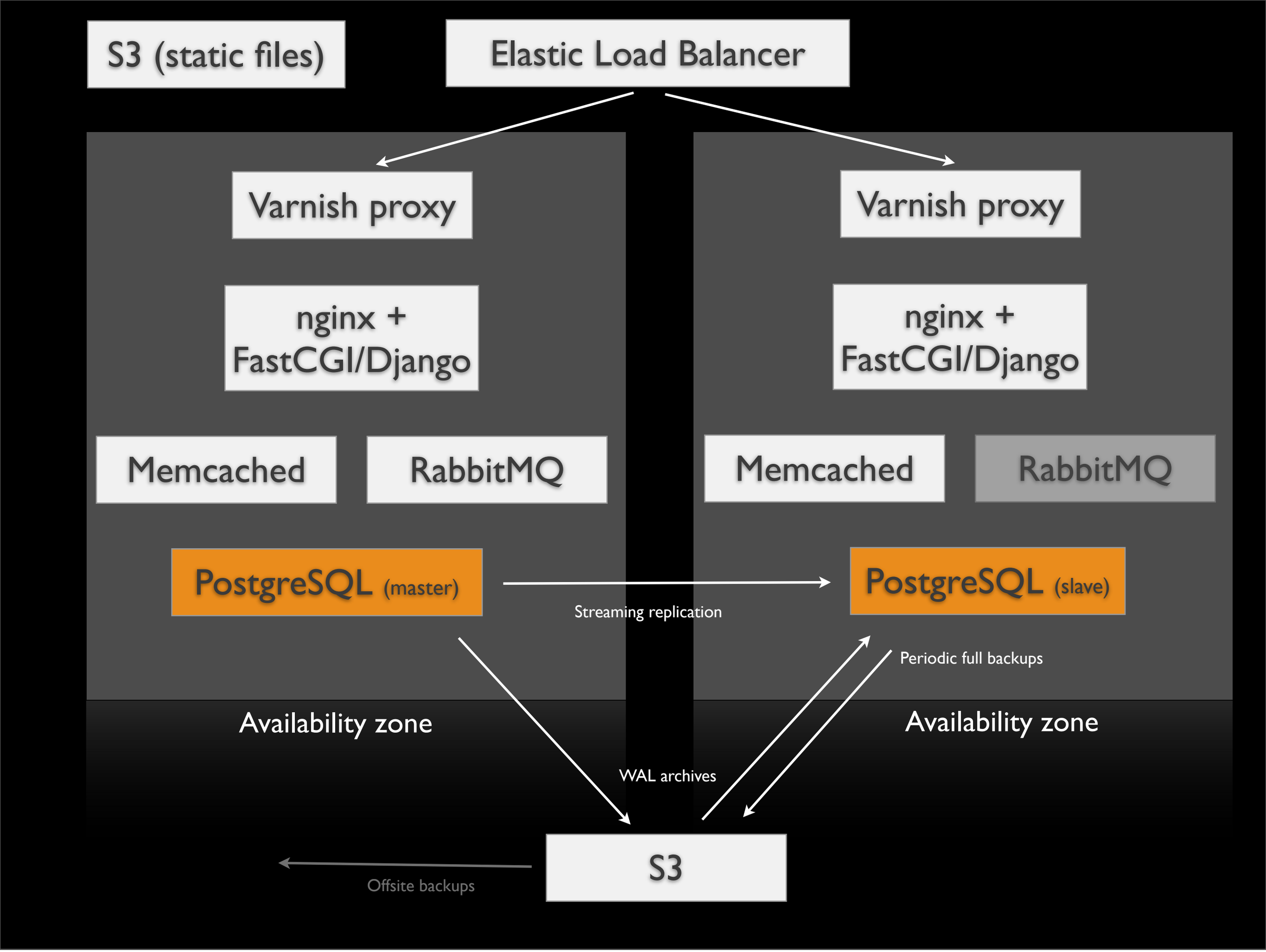
WAL archives

S3

Offsite backups

RabbitMQ

- Celery + RabbitMQ for asynchronous processing
- For example, we use it for automatically scaling user uploaded images
  - file is uploaded and the resize job is put to the queue
  - once we have a spare worker, the job is carried out
- Scales well, integrates nicely with Django
- Can be used as a buffer e.g. against latency spikes in DB
  - perform DB writes in batches -> peak commit frequency drops to a fraction
  - your DB doesn't have to scale 1:1 for write ops from users


S3 (static files)
Elastic Load Balancer
Varnish proxy
nginx + FastCGI/Django
Memcached
RabbitMQ
Varnish proxy
nginx + FastCGI/Django
Memcached
RabbitMQ
PostgreSQL (master)
Streaming replication
PostgreSQL (slave)
Periodic full backups
Availability zone
Availability zone
WAL archives
S3
Offsite backups

PostgreSQL

- Reliable, proper RDBMS for our core features
- Open source, not depending on corporate strategies
- PostGIS+GeoDjango for location-aware features

- NoSQL could be beneficial in certain write-heavy components
  - in our scale, PostgreSQL (+queues) still does its job well enough
  - more technology == more things that can break == more time spent doing ops stuff instead of developing

PostgreSQL and AWS

- Very challenging to build reliable and reasonably performing DB in AWS
- EBS is slow
  - not just slow but performance varies a lot + shares NIC with your instance...
  - assume max. 100 iops per volume
  - get the largest instance type you can afford
  - our solution: RAID 10 (8 volumes mirrored and striped + 1 spare) + trying to fit as much in RAM as possible
- Everything can fail -- instances, EBS platform...
  - *usually* local to an availability zone
  - multifails have happened (EBS fails in one AZ -> cannot launch new instances to other AZs)
- Master-slave configuration a must
  - Streaming replication between multiple availability zones
  - WAL archiving to S3

Amazon Web Services

- Initially seems easy
- Soon, you'll notice it's actually very hard
  - With elasticity comes the need to take control of the ever-changing configuration
    - Amazon provides building blocks for your infrastructure, how you manage it is up to you
    - Solve this as soon as you can
    - Learn from others' mistakes
- Once you've solved how you manage it, (most) things become easy again

Processes

How are we doing it?

DevOps culture

- How to make zero downtime deployments?
  - During peak hours?
- Think how your features can be deployed when you design/develop them
- When you feel the pain of having to fix production problems, you do your best to try to prevent problems
  - Fear of a phone call in the middle of the night is an effective motivator ;)

DevOps culture

- It grows from simple things
- We have a rolling “Janitor” role -- each developer gets to be the Janitor for one sprint at a time
  - primary person to handle deployments, monitoring etc.
  - gives time for the rest of the team to focus on their backlog items

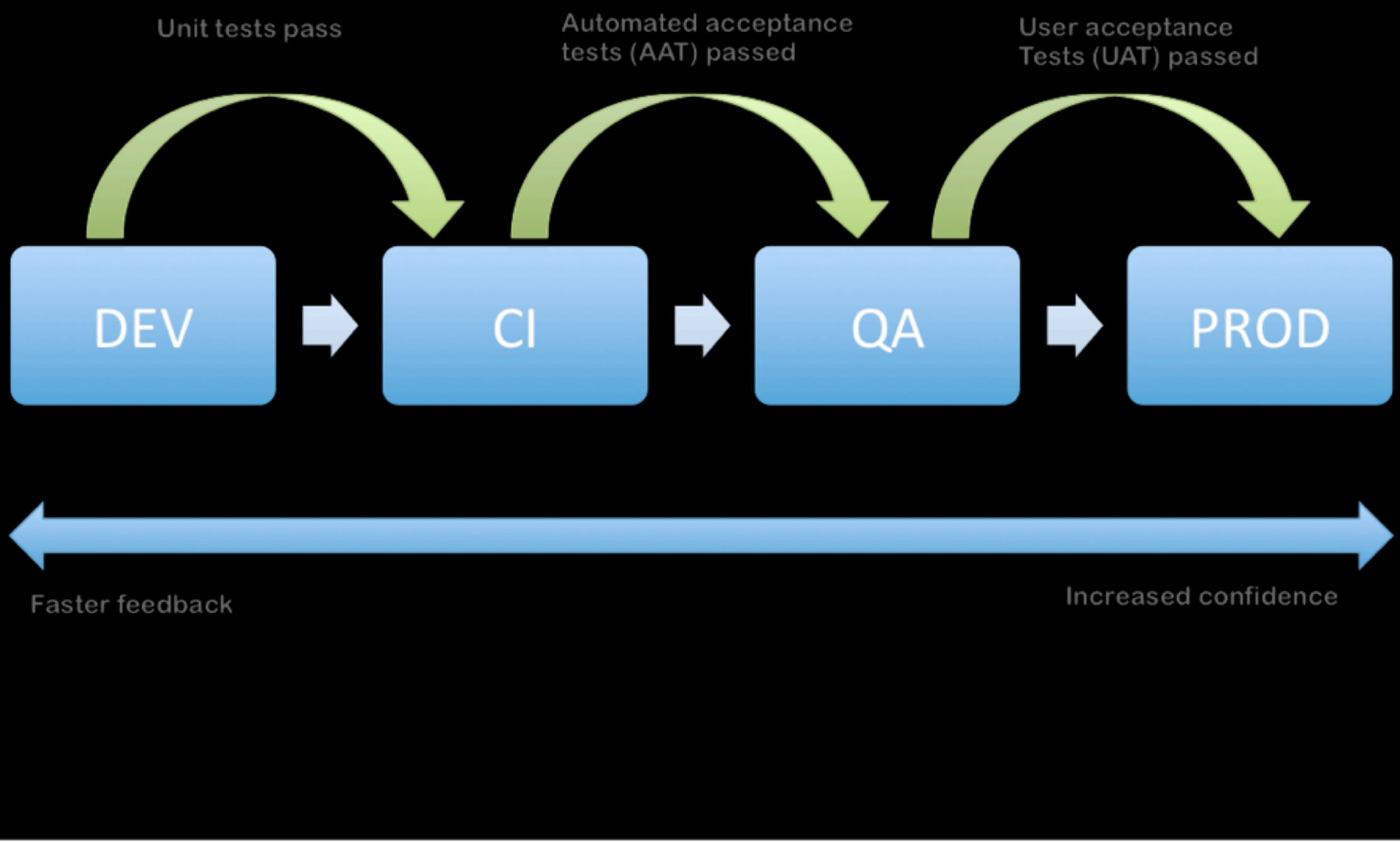
Infrastructure as a code

- The best way to document how your servers can be rebuilt is to have it in your version control
- Server configurations are part of your application and they do change in time
- We tried to use Fabric for managing configuration
  - didn't believe when everyone on the web said it's a mistake
  - Fabric is a great for SSH automation but not for configuration management

Infrastructure as a code

- We've started replacing the Fabric-managed nodes with Chef + CloudFormation managed nodes
  - Painful to get started, but gets better
    - Amazon provides a sample CloudFormation template to setup Chef server + infra for validation.pem provisioning
- Ansible - a promising model-driven, push-based newcomer to keep an eye on: http://ansible.github.com/

Continuous Delivery



Deployments

- Goal: repeatable, low-risk, zero downtime releases during office hours
- Decouple deployments from launching new features
- Use feature toggles to enable/disable features
    - For Django, Gargoyle by Disqus does the trick very well
- Canary releases -- deploy only to a certain group of users (e.g. % of user base, users coming from a certain IP block etc) and monitor results
    - monitor also also business metrics: did the feature have the effect we wanted?
- If a feature appears to cause issues, disable it and have it fixed -- no need to worry about rollbacks

Dark launches

- For example -- add a new high-risk (not yet tested thoroughly enough) widget on front page
- Add it to front page but do not yet **show** it to users (just make it do whatever it is supposed to do)
- Measure impact (e.g. performance) without the users noticing something might be wrong
- You can combine canary releasing with this pattern, too
- Once you've made sure everything works smoothly, unhide the widget

The tricky parts?

Packaging Django

- Deploying Django is a pain
- Pip + virtualenv is great in theory
- In practice, we've experienced too many issues with it:
  - Pypi down or slow (you'll need your own Pypi mirror)
  - 10 minutes/server for building virtualenv -- deployments take too long even when doing multiple servers in parallel
  - Each server needs to have C compiler + bunch of `-devel` packages -- stuff that doesn't belong to production machines!

Packaging Django

- We prepare a RPM package on CI build server
  - Includes virtualenv (not yet using `--relocatable` due to issues with it)
  - RPM packaging easy to automate with fpm (https://github.com/jordansissel/fpm)
  - The RPM package is installed in all environments from an internal Yum repository
  - Easy to manage -- just like any other application
- Once installed, update symlinks & restart FastCGI and we're live

Packaging Django

- The whole deployment process is currently automated with Fabric
- One command can do everything:
  - rolling update, one availability zone at a time
  - attaches/detaches nodes from Elastic Load Balancer
  - automatically handle scheduled downtime in Nagios
- The whole stack is updated in a few minutes

Data model changes

- Simple solution: avoid them
- Backwards compatibility critical
- Use patterns such as expand/contract:
  - Add a new (NULLable) field
  - Lazy migration: when data is requested, try to read from new field. If data is not present, fill it
  - Once most data is lazily migrated, it may no longer be a problem to update the rest in a batch
  - Finally, enforce constraints if needed (NOT NULL,s FKs etc)
  - http://exortech.com/blog/2009/02/01/weekly-release-blog-11-zero-downtime-database-deployment/

Data model changes

- Decoupling database changes and SW difficult with ORM + South
- Locking issues
  - PostgreSQL requires exclusive locks when it creates foreign keys
  - Example: we use Generic Foreign Keys and `django_content_type` extensively for linking content to other content
  - Adding new fields with FK to `django_content_type` are not possible if there are long-running transactions having any locks on the `django_content_type_id`

Django ORM

- Sometimes you may need to use raw SQL to trivialize your problems
- Trees are traditionally nasty for relational DBs
- news sections are a tree structure and we often need to fetch data from current section and all its children and parent recursively
    - Pretty much the bread and butter kind of query type that occurs in several parts of the site for various types of content
- We did it first with Django ORM
    - In worst cases, it could take several seconds to execute per query
- Replaced it with raw SQL that uses PostgreSQL's `WITH RECURSIVE` and the query does its job in milliseconds
- It's great to know you always have the choice of raw SQL

Summary

- Django was critical for our success and we strongly believe it will enable us to do great things also in the future
- Deploying Django is hard
- Doing things right in the cloud is hard
  - Not saying we're doing everything right, but we're going there
- But luckily we like challenges!

Questions?

Thank you!

- @raveli
- http://www.linkedin.com/in/markussyrjanen