Building a high-performance, scalable ML & NLP platform with Python

Sheer El Showk

CTO, Lore Ai

www.lore.ai

LORE AI

Lore is a small startup focused on developing and applying machine-learning techniques to solve a wide array of business problems.

We are product-focused but we also provide consulting services (because building and selling products is hard!)

We are self-funded so we need to be cost-effective but also flexible and scalable.

- Need to be ready to pivot if product is not working
- Need to support both consulting work and product development

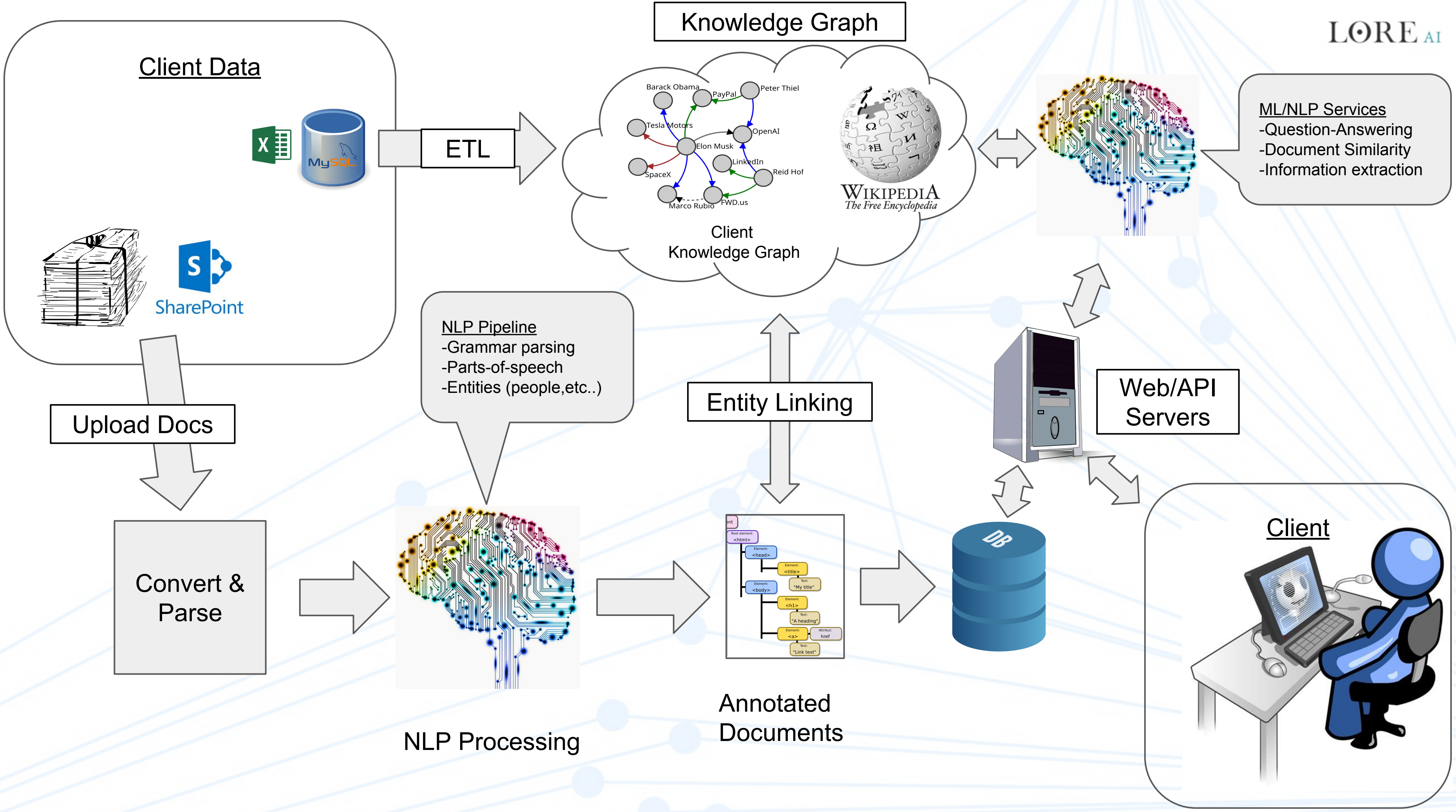
Over time developed a complex but modular stack:

- ML & NLP tools/services
- Standard web-stack: DB, caching, API servers, web servers
- Fully scalable (all components can be parallelized)

This has allowed us to develop & maintain several products:

- A chatbot
- A content-based recommender engine
- Our flagship product, Salient, a powerful document analysis engine

The Task



LORE AI
Client Data
MySQL
SharePoint
ETL
Knowledge Graph
Barack Obama
PayPal
Peter Thiel
Tesla Motors
OpenAI
Elon Musk
LinkedIn
SpaceX
Reid Hof
Marco Rubio
FWD.us
Client
Knowledge Graph
WIKIPEDIA
The Free Encyclopedia
ML/NLP Services
-Question-Answering
-Document Similarity
-Information extraction
Upload Docs
NLP Pipeline
-Grammar parsing
-Parts-of-speech
-Entities (people,etc..)
Entity Linking
Web/API
Servers
Convert &
Parse
DB
Client
NLP Processing
Annotated
Documents

Applications

- Contract Management: automate labelling database of contracts (PDFs, word, OCR)
    - Contract type, expiration date, other parties, important clauses, etc..
- Analyze news
    - Build a database of product releases or funding rounds from press releases
    - Find companies fitting a profile -- competitors, acquisition, investors, etc..
- Patent and Policy Document Analysis
    - Build an ontology and network linking concepts and content

The Challenges

Lore's stack has all the normal challenges of developing a large web/business application in python.

The addition of ML/NLP brings additional challenges whose solutions are less well known.

LORE AI

Scalability

- Some use cases require processing *millions* of documents
    - E.g. we have a news DB with 4 million articles in it
- Python considered “slow” and not scalable (hard to parallelize)
    - Java preferred language of enterprise
- ML brings new problems:
    - Need models to be very performant
    - Parallelizing harder: need to share models/data between servers

Maintainability

- Mixing many different technologies
    - ML/compute stack: theano, gensim, spaCy, etc..
    - Web stack: django, celery, mysql, etc..
- Multiple servers/services can mean expensive/difficult devops.
- Large code base with different kinds of code (JS/web vs ML/NLP) make it hard to coordinate between developers.

Flexibility

- Business requirements change very quickly
    - Need to provide a wide range of services from same platform
    - Need to be able to easily upgrade/improve parts of system
- Agility often requires integrating existing off-the-shelf solutions to solve non-core tasks.
- Easy to deploy or scale a deployment.

Where we Started

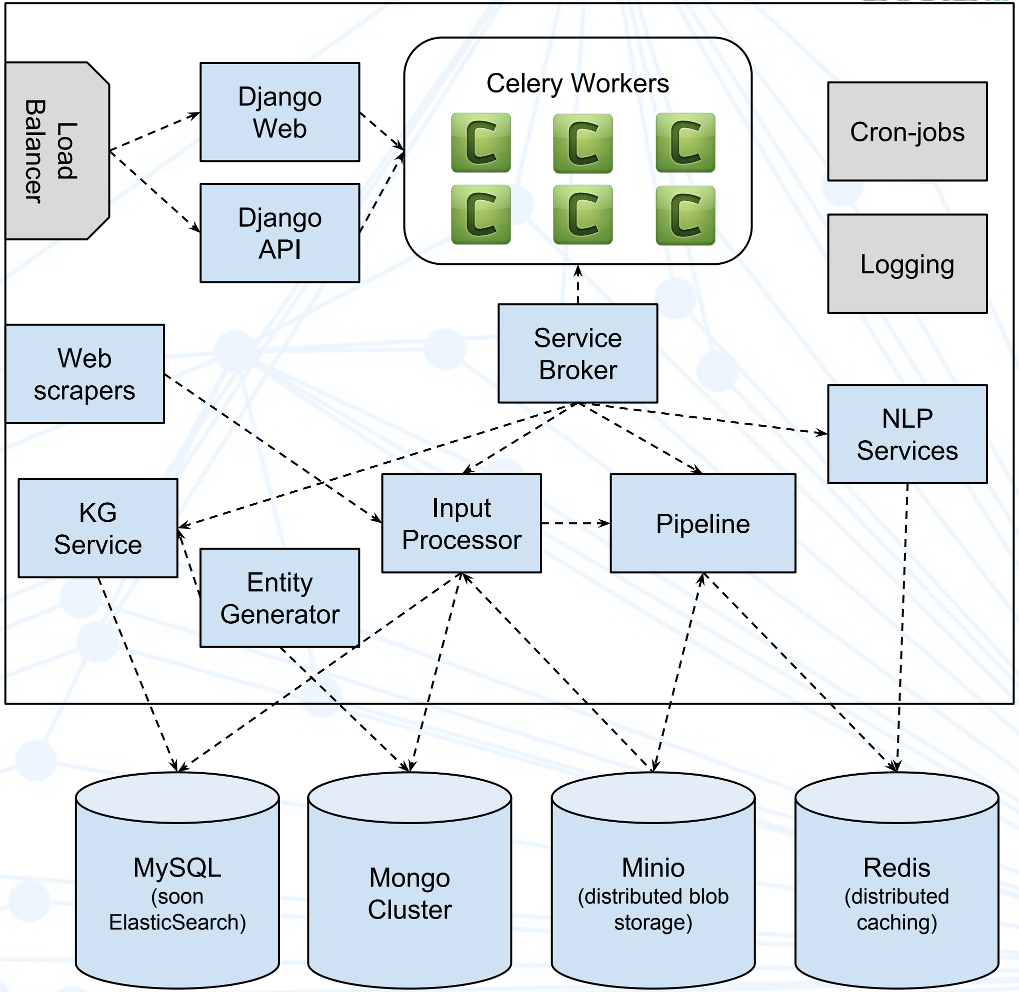
Where we started...

- ➔ Monolithic python-based server + PHP UI
- ➔ MySQL DB backend
- ➔ Hard coded dependence on target document set.
- ➔ Basic off-the-shelf NLP tools
    - ◆ NLTK, etc..
- ➔ Serial pipeline
- ➔ No redundancy or scalability
- ➔ Hard to install/maintain (all manual)

A few pivots and consulting gigs later...

The new stack...

- ➔ Kubernetes & docker for devops
- ➔ Celery for parallelization
- ➔ Micro-services Architecture
- ➔ In-house NLP engine
- ➔ Distributed data stores:
    - ◆ Mongo, Minio, Redis
- ➔ Very modular, re-usable
- ➔ Rapid deployment (even cluster)



What we learned along the way...

Lessons

- Devops: kubernetes, docker, docker-compose
- Parallelization: celery, dask, joblib,...
- Modularity: (stateless) microservices architecture
- Persistence: scalable distributed caching/persistence (redis, mongo, ES,...)
- Future-proofing: wrapper patterns
- Performance: cython or pythran for bottleneck code

Docker & Kubernetes

LORE AI

Dockerize early, dockerize often

1. Uniform environment between devs
   a. Use ipython to develop in stack
2. Easy to add services
   a. **Solve dev problems using devops!**
3. Fast, consistent deploy to production
   a. Deploying our stack is rate-limited by download speed :-)
4. Cut costs by running on bare metal
   a. Run your own “AWS” with k8ns

```
sheer@core:docker$ sudo ./run_stack.sh start xynnweb
** USING DEV MODE
Creating docker_mongo-nascent-svc_1
Creating docker_spacy-svc_1
Creating docker_mysql-sia-svc_1
Creating docker_redis-svc_1
Creating docker_mysql-kg-svc_1
Creating docker_minio-svc-1_1
Creating docker_mysql-xynnweb-svc_1
Creating docker_celery-worker-svc_1
Creating docker_sphinx-svc_1
Creating docker_xynnweb_1
sheer@core:docker$
```

type	AWS (reserved)	Hetzner (bare metal)
16 threads 64 gb ram	$360/mo (no storage)	$70/mo (1tb ssd)
GPU server	$450/mo (K80)	$120/mo (1080)

Celery for Parallelization

LORE AI

[10 minute parallelization]()

- Many interesting options for parallelization:
  - Dask.distributed, joblib, celery, ipython
- Celery "complicated" -- requires Redis/RabbitMQ, etc..
  - Very easy with docker/kubernetes
  - NOTE: Redis has much lower latency
- Transparent parallelization
  - Wrap "entry-point" functions in a task and Bob's your uncle.
- Lots of fancy features but don't need to use them until you need them.

```python
@app.task()
def do_something_parallel(func_args, func_kwargs):
    t0=time.time()
    res=do_something(*func_args, **func_kwargs)
    t1=time.time()
    return res,t1-t0
```

Stateless Microservices

Unlimited Scalability and Flexibility

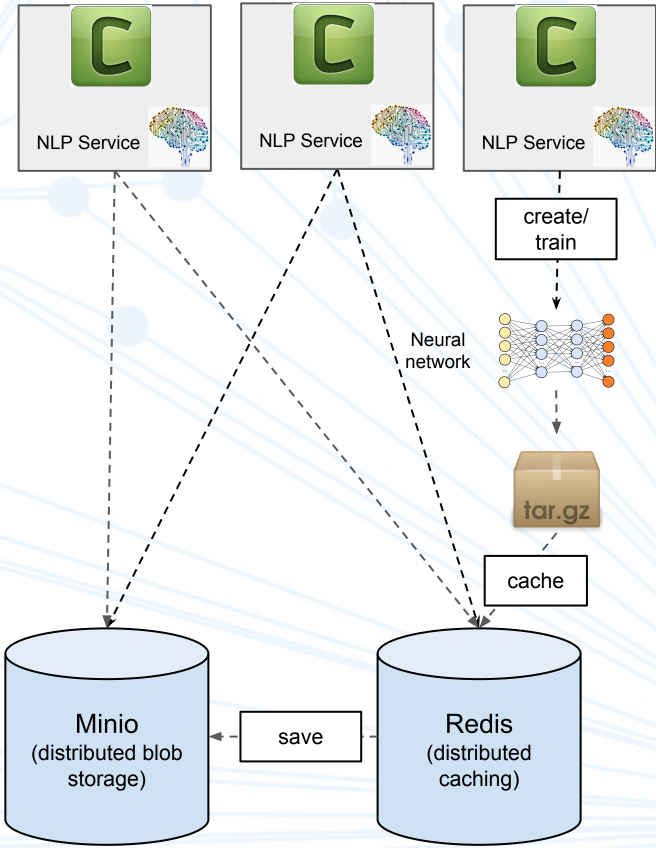
- High level analog of an Abstract Base Class
- Split codebase into *independent* microservices
    - Each microservices handles one kind of thing: NLP, logging, DB persistence, etc…
    - Wrap underlying service *abstractly*: redis→ kv_cache, minio → kv_store, mysql → sql_db
- Applications can include multiple microservices talking to each other
- Microservices should be stateless
    - All state (caching, persistence, etc..) should be handled by external services (DB, redis, etc..)
- Microservices encapsulate underlying implementation of a service
- Microservices are *not* micro-servers: services are just APIs within an API.
    - Should not be tied to an interface (REST, JSON, etc..)

Distributed Persistence

LORE AI

Let others do the hard work

- All persistence handled by “layers” of persistence services.
  - Layering helps performance
- Persistent services accessed via “wrappers” (see next slide) for easy replacement.
- “State” of services managed by distributed cache
  - ML models can be shared by workers using e.g. Redis (caching) and Minio (persistence)
- Different types of persistence for different problems:
  - Mongo stores JSON, Minio stores blobs, mysql stores tables



Design Patterns

LORE AI

Wrapper Pattern

- Wrap access to all external services (db, logging, etc..)
- Easy to swap in new versions
  - MySQL vs Postgres, etc.
- Easily add logging, performance, etc..
- Insert custom logic to modify the behavior, eg
  - Manually shard/replicate DBs
  - Add new logging destinations
- This pattern has saved us countless hours of refactoring!

Examples

- Local → Centralized logging & config with just a few lines of code.
- Migrating from NLTK to better (in-house) NLP
- Restricting user access to DB by transparently replacing tables with views.

LORE AI

Performance

- Use native types correctly:
  - set vs list, iteritems vs items
- Pythonic code is almost always faster.
- Use %timeit to test code snippets everywhere
- Beware of hidden memory allocation
- For critical bottlenecks use:
  - Pythran (very easy but limited coverage)
  - Cython (harder, but more flexible)
- For ML use generators to stream data from disk/db (see gensim).
- Know your times

```
In [31]: strdict=dict( (str(k), str(k)) for k in range(10**6))

In [32]: %timeit '10' in strdict
10000000 loops, best of 3: 36.9 ns per loop

In [33]: %timeit '10' in strdict.keys()
10 loops, best of 3: 43.9 ms per loop
```

```
In [24]: t=np.random.normal(0,1, (10000,1,400))

In [25]: %timeit x=np.sum(t, axis=1)
100 loops, best of 3: 5.58 ms per loop

In [26]: %timeit x=t.reshape((t.shape[0], t.shape[2]))
The slowest run took 14.48 times longer than the fastest.
1000000 loops, best of 3: 544 ns per loop
```

```
In [82]: s1=set(str(x) for x in np.random.randint(0,10**6, (10**6,)))

In [83]: s2=set(str(x) for x in np.random.randint(0,10**6, (10**6,)))

In [84]: %timeit len(s1.intersection(s2))
10 loops, best of 3: 67.9 ms per loop
```

Performance (II)

Example

Classifying (short) documents

- Initial rate: 2k docs/sec
- Initial Profiling:
  - Db read rate: 250k docs/s
  - Feature generation 2k docs/s
  - Classification 10k docs/s
- Feature generation involves for-loops & complex logic.

Fixes

- Refactored feature generation:
  - Extract features in list comprehensions
  - Convert to vectors in Pythran code
- New times
  - Python part: 10k docs/s
  - Pythran: 40k docs/s
- Fix memory allocation in classifier: 50k docs/s
- New times: ~10k docs/s
- Going forward: cache features in Mongo?

Further Reading...

- Radim Řehůřek (gensim):

  [“Does Python Stand a Chance in Today’s World of Data Science”](https://youtu.be/jfbgt3KjWFQ)
  (https://youtu.be/jfbgt3KjWFQ)

- High Performance Python (O’Reilly)
  - “Lessons from the Field” (chapter 12)
- Fluent Python (O’Reilly)
- Latency Numbers Every Programmer Should Know

  https://gist.github.com/jboner/2841832

Bye!

Open Problems

- Cores vs RAM
  - Many models require lots of RAM (models can be Gbs in size).
  - Models read-only but because of python GIL hard to share memory between “processes/threads”
  - Use “sharedmem” module?
- Generating models from terabytes of data?
  - “Embedding models” can capture interesting information from huge amounts of data
  - Train very quickly (millions words/sec per server)
  - How can we distribute parameters/data between large number of workers?