Advanced Celery Tricks

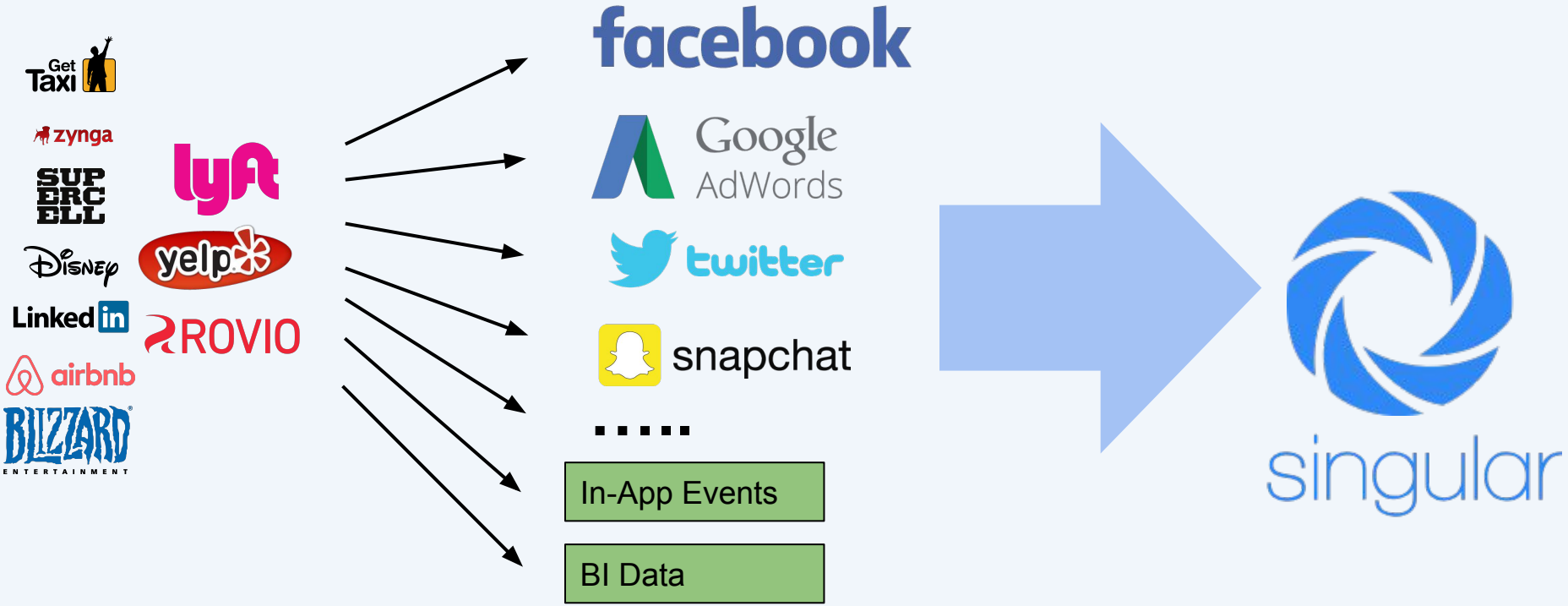
How we adapted and extended Celery to fit our data pipeline

Itamar Hartstein
PyCon IL 2018

singular

A bit about Singular

A bit about Singular



Agenda

- Celery 101
  - What is Celery?
  - Quick start
  - General Architechure
  - Workflows
- Integration and Challenges in Singular
  - Our use case
  - Challenges we had and how we solved them

Celery 101

Celery - Distributed task management in Python

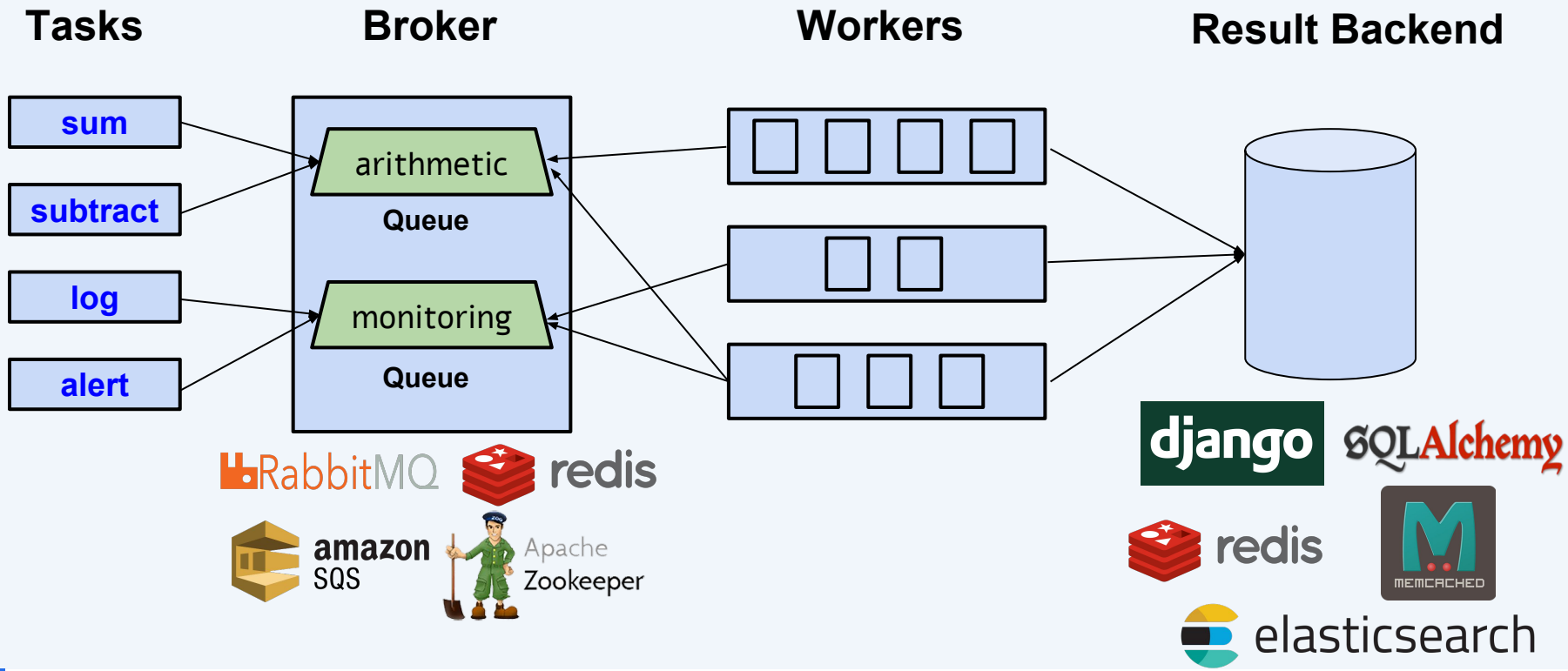
```python
from celery import Celery

app = Celery('tasks', broker='redis://localhost')

@app.task
def add(x, y):
    return x + y
```

```
celery worker -A tasks --loglevel=info
```

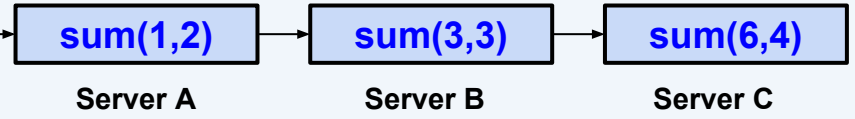
Celery - Architecture



Celery - Workflows

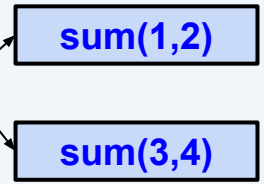
Chains

```
celery.chain(my_sum.s(1, 2),
             my_sum.s(3),
             my_sum.s(4)).apply_async()
```



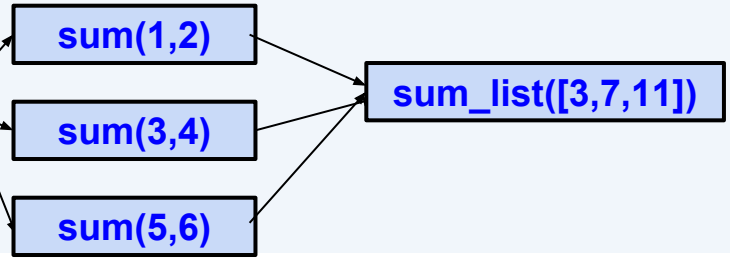
Groups

```
celery.group(my_sum.s(1, 2),
             my_sum.s(3, 4)).apply_async()
```



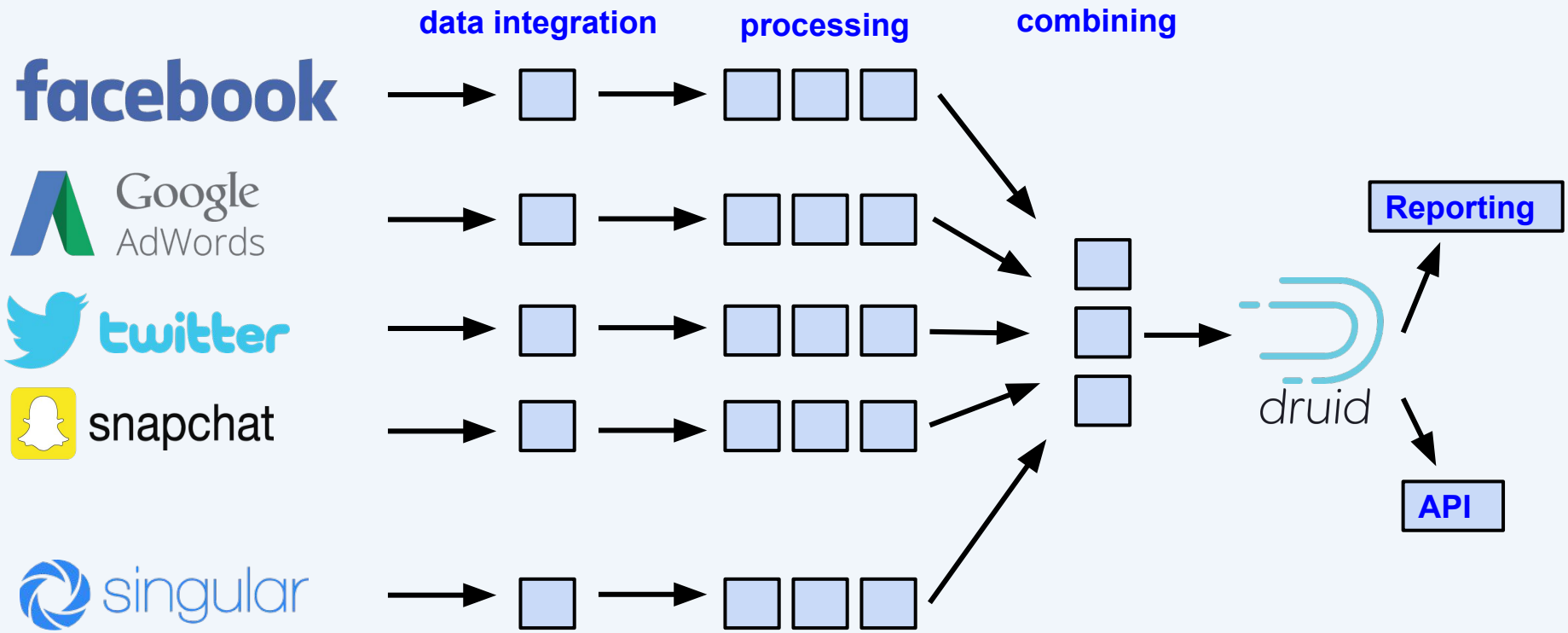
Chords

```
celery.chord([my_sum.s(1, 2),
              my_sum.s(3, 4),
              my_sum.s(5, 6)
             ],
             my_sum_list.s()).apply_async()
```



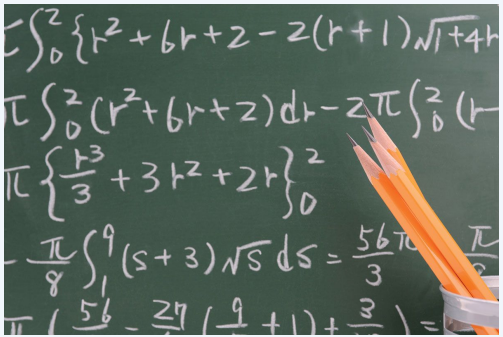
Our integration

Our data pipeline in high level



Some numbers

- **Tasks per day:** ~1M.
- **Task types:** ~100.
- **Queues:** 44
- **Workers:** 130
- **Task durations:** ranging from 0.5 seconds to 20 hours
- **Task memory:** ranging from 100MB to 45GB



Our use case is a bit different

The common use case

- A lot of small tasks
  - Event processing
  - Web tasks offloading
- Code deploys will typically involve worker restarts
- Workflows are usually simple.

Our use case

- Great variation in duration
  - Tasks can take hours
  - Restarting workers often is problematic
- Unpredictability
  - We depend on third parties
  - Tasks can fail / hang / etc
- A lot of dependencies between tasks.

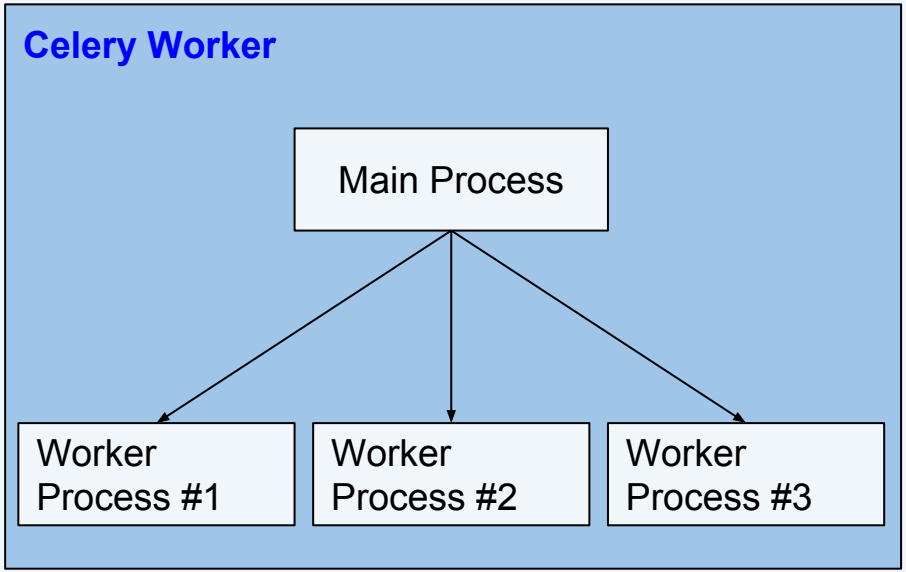
Challenges

1. Updating Code
2. Customizations
3. Chords at scale
4. Brutally killed workers
5. Prefetching behaviour

Updating code

- Tasks in Celery by default run in processes forked from a main process.
- **All the main celery files are imported in the main process.**

imported modules:
tasks.py
celeryconfig.py
(django?)



Updating code

- We moved the implementation of all the tasks to an “inner” module that is imported **only when the task actually runs**:

tasks.py (master)

```python
@app.task(name="my_task")
def my_task(*args, **kwargs):
    from dashboard.celery import tasks_inner
    return tasks_inner.i_my_task(*args, **kwargs)
```

tasks_inner.py (slave)

```python
from singular.utils import amazing_sum

def i_my_task(a, b):
    return amazing_sum(a, b)
```

- We defined:
  - worker_max_tasks_per_child=1
- We stopped using a django backend.

Customizations - Serialization

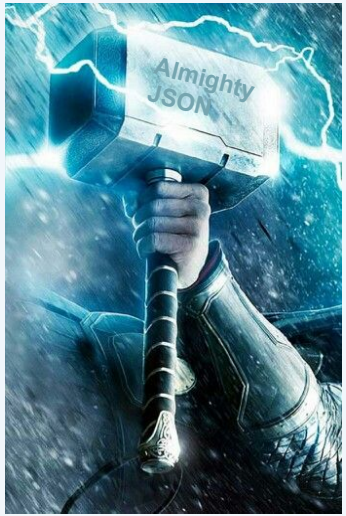
- Task calls and results must be serialized in various stages:

- Celery supports multiple serialization methods:
  - **json** (default since Celery 3.2)
  - **pickle** (default up to Celery 3.1)
  - **msgpack**
  - **yaml**
- pickle is the most flexible
  - … but insecure.

Customizations - Serialization

```
result_serializer = 'almighty-json'
accept_content = ['almighty-json']
task_serializer = 'almighty-json'

register('almighty-json', almighty_dumps, almighty_loads,
         content_type='application/x-almighty-json',
         content_encoding='utf-8')
```



- Custom JSON encoder / decoder that supports:
  - **Sets**
  - **Date / Datetime objects**
  - **Exceptions**
  - **Django Models**

Customizations - Result Backend

- We wanted to have more information in our result backend:
  - **Task name**
  - **args + kwargs**
  - **Custom information:**
    - Customer
    - Ad Network
    - Scrape ID
- We created a custom database table that contains all the fields we need.

Customizations - Result Backend

Implementation

```
class SingularBackend(BaseBackend):
    @classmethod
    def _store_result(cls, task_id, result, status, traceback=None, **kwargs):
        pass

    @classmethod
    def _get_task_meta_for(cls, task_id):
        pass
```

Configuration

```
result_backend = 'dashboard.celery.singular_backend:SingularBackend'
```

Customizations - Base Task

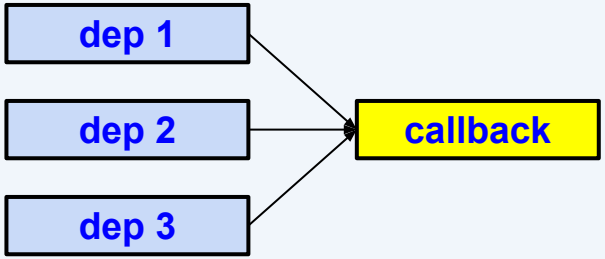
- You can have all your tasks implement a base task

```
@app.task(name="add", base=SingularTask)
def add(x, y):
    return x + y
```

- We used that to add logging/monitoring capabilities to all our tasks:
  - **DataDog** - time & memory performance
  - **Sentry** - task exceptions
  - **ELK** - additonal logging parameters

Chords at scale

- When we started using chords heavily we saw they turned out to be really inefficient :(
- We started with a worker that runs 20 processes and **around 10-12 would actually run in parallel.**



Chords at scale

Name	UUID	State	Started	Worker
celery.chord_unlock	e2efacf9-88d5-46ff-b224-e271bd2ae27f	RETRY	2018-05-30 09:57:07.560	celery@itamarmac.lan
celery.chord_unlock	91d354e6-2fec-42ed-aa14-52b08ccaa596	RETRY	2018-05-30 09:57:07.489	celery@itamarmac.lan
celery.chord_unlock	877378ec-512e-4200-8a37-be808f554957	RETRY	2018-05-30 09:57:06.507	celery@itamarmac.lan
celery.chord_unlock	bc199218-0b0c-4da0-b249-e3f6646c385f	RETRY	2018-05-30 09:57:06.449	celery@itamarmac.lan
celery.chord_unlock	cce8f66f-1f82-494e-9ec0-168970c054cd	RETRY	2018-05-30 09:57:05.425	celery@itamarmac.lan
celery.chord_unlock	e01e8dcc-b088-415c-b54c-3c34bb640f05	RETRY	2018-05-30 09:57:04.035	celery@itamarmac.lan

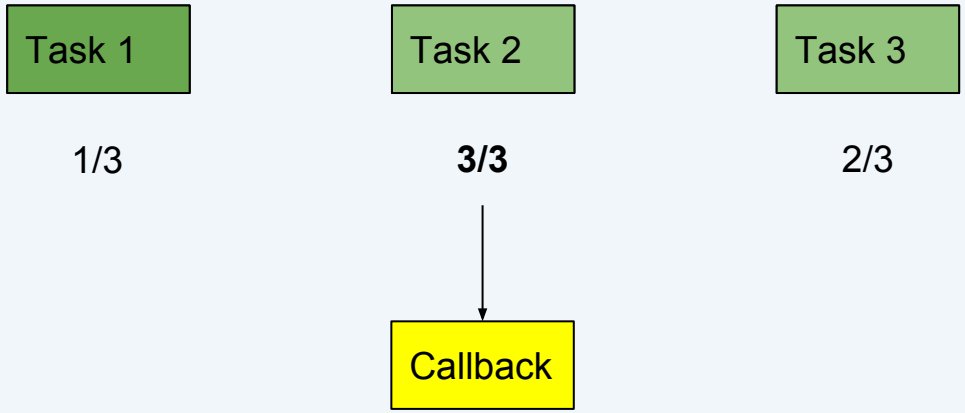
Chords at scale

- The depending tasks are polled every second by a task called “chord_unlock”
- **Problematic for long tasks!**

```
try:
    ready = deps.ready()
except Exception as exc:
    raise self.retry(
            exc=exc, countdown=interval, max_retries=max_retries,
        )
```

Chords at scale

- But, turns out something else happens when using a **Redis backend**
- Instead of chord_unlock, a simple counter is used:



Identifying brutally killed tasks

- Workers might be killed brutally.
- A common reason for this can be the **OOM Killer**.
- Celery can handle this well when a child dies
  - The master process is notified and marks the failure.
- More problematic when a main process dies
  - You can reduce the chances of that by using **Early OOM**
  - It may also help to adjust the OOM scores:

```
echo 15 > /proc/{child_pid}/oom_adj
echo -15 > /proc/{main_pid}/oom_adj
```

Prefetching is not always good

- Celery workers by default prefetch as many tasks as they can
  - Good performance for environments with lots of small tasks.
  - Can cause major delays when you have long / inconsistent tasks!
- Can be avoided with configuration:
  - **-Ofair**
    - Guarantees child processes will only be allocated tasks when they are actually available.
  - **prefetch_multiplier=1**
    - Guarantees parent processes will prefetch as little as possible.

Questions?