Aalto University
School of Electrical
Engineering

Some more models of organizing microservices

28.2.2018

Santeri Paavolainen

So far …

- **We've covered mostly request-oriented architecture models**
  - Primarily from synchronous assumption: request-process-respond
- **How to process …**
  - Long process time – minutes to hours to days
  - Multi-step operations across many systems
  - Large amounts of data (terabytes+)
  - Responding to events – not "requests" – f. ex. perturbations
  - Continuous streams of data (dataflow)

Overview

- **Workflow systems**
  - Multi-step processes, potentially with retries
  - Across heterogenous systems
- **Batch processing**
  - “Need to re-encode all of video files”
- **Data streaming**
  - Continuous data streams with real-time processing (not batch!)

Workflow systems

Aalto University
School of Electrical
Engineering

“Workflow system”

- **System that orchestrates a flow of work**
  - Potentially across different systems (e.g. always in microservice architectures)

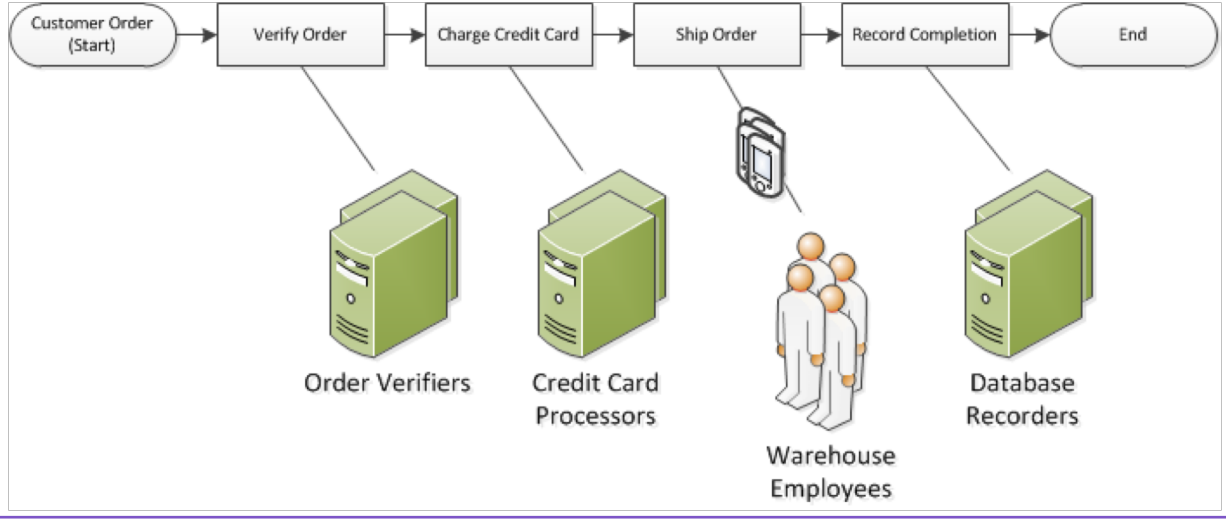


Image: AWS

Overall

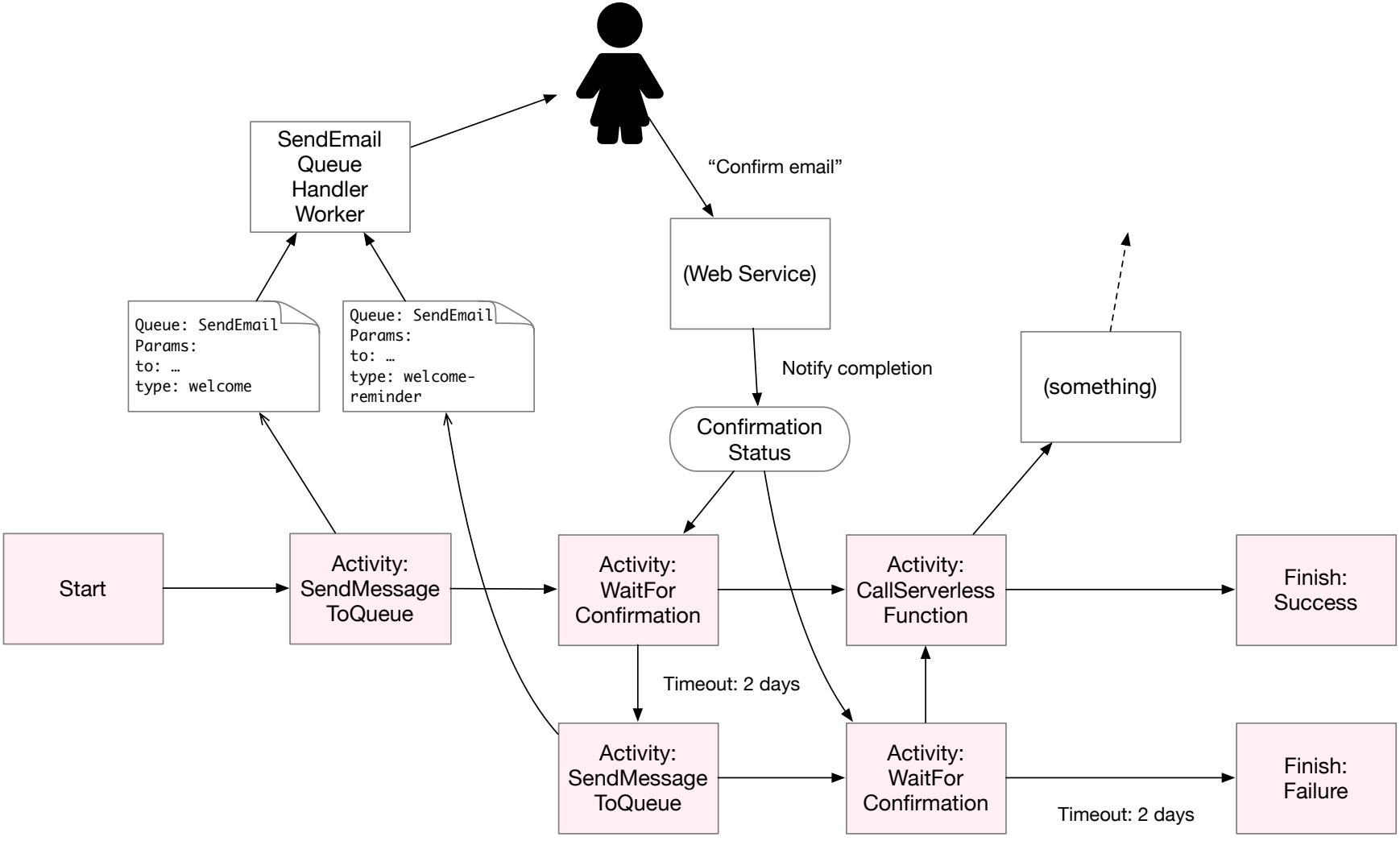
- **Workflow systems execute “trivial” programs**
  - Linear scripting
    - *if CONDITION then ACTION1 endif;*
      ACTION2;
      (ACTION3a; ACTION3b);
  - State graphs (most common)
    - *STATE1 { ACTION: …, NEXT-STATE: { CONDITION1: STATE2, … } }*
  - JSON, XML, graphical UI construction …

Overall

- **Focus isn't on programmability (of orchestration), but**
  - Management of state over long periods of time, reliably
  - Integrability across different services and activity types
  - Scripting is easy – state management is not!
  - "Workflow management system" (WfMS) vs. "Workflow system" (WfS)
- **Boundary between WfMS and WfS vague**
  - Apache Airflow … is it workflow system?
  - Task queues – are they WfS? (e.g. one task dispatches further tasks etc.- implicit task workflow)
    - *Orchestration vs. choreography – former usually better approach for Wf problems*

Some gotchas

- **WfMS history contains a lot of enterprise'y things**
  - BPM, BPML, WS-BPEL – very much alive over there
  - Large focus on visuals – for people who are not really programmers
- **No established non-BP* way for workflow definitions**
  - AWS has SWF and Step Functions (why have just one way?)
  - Google Compose (e.g. Apache Airflow)
  - Azure Logic Apps
  - List of OSS WfS(M): https://github.com/meirwah/awesome-workflow-engines


SendEmail Queue Handler Worker
"Confirm email"
(Web Service)
Queue: SendEmail
Params:
to: …
type: welcome
Queue: SendEmail
Params:
to: …
type: welcome-reminder
Notify completion
(something)
Confirmation Status
Start
Activity: SendMessage ToQueue
Activity: WaitFor Confirmation
Activity: CallServerless Function
Finish: Success
Timeout: 2 days
Activity: SendMessage ToQueue
Activity: WaitFor Confirmation
Timeout: 2 days
Finish: Failure

Concepts

- **Activities**
  - Internal vs. external
    - *Waiting vs. spawning a task*
  - Asynchronous vs. synchronous
- **Workers**
  - Open-ended integrations
  - “Pulls” pending activity tasks
  - “Pushes” completed states

- **States and transitions**
  - Conditional transitions
  - Parallel state execution
  - Failures and retries

- **Note: Terminology wildly varying across different WfS**

Workflow systems

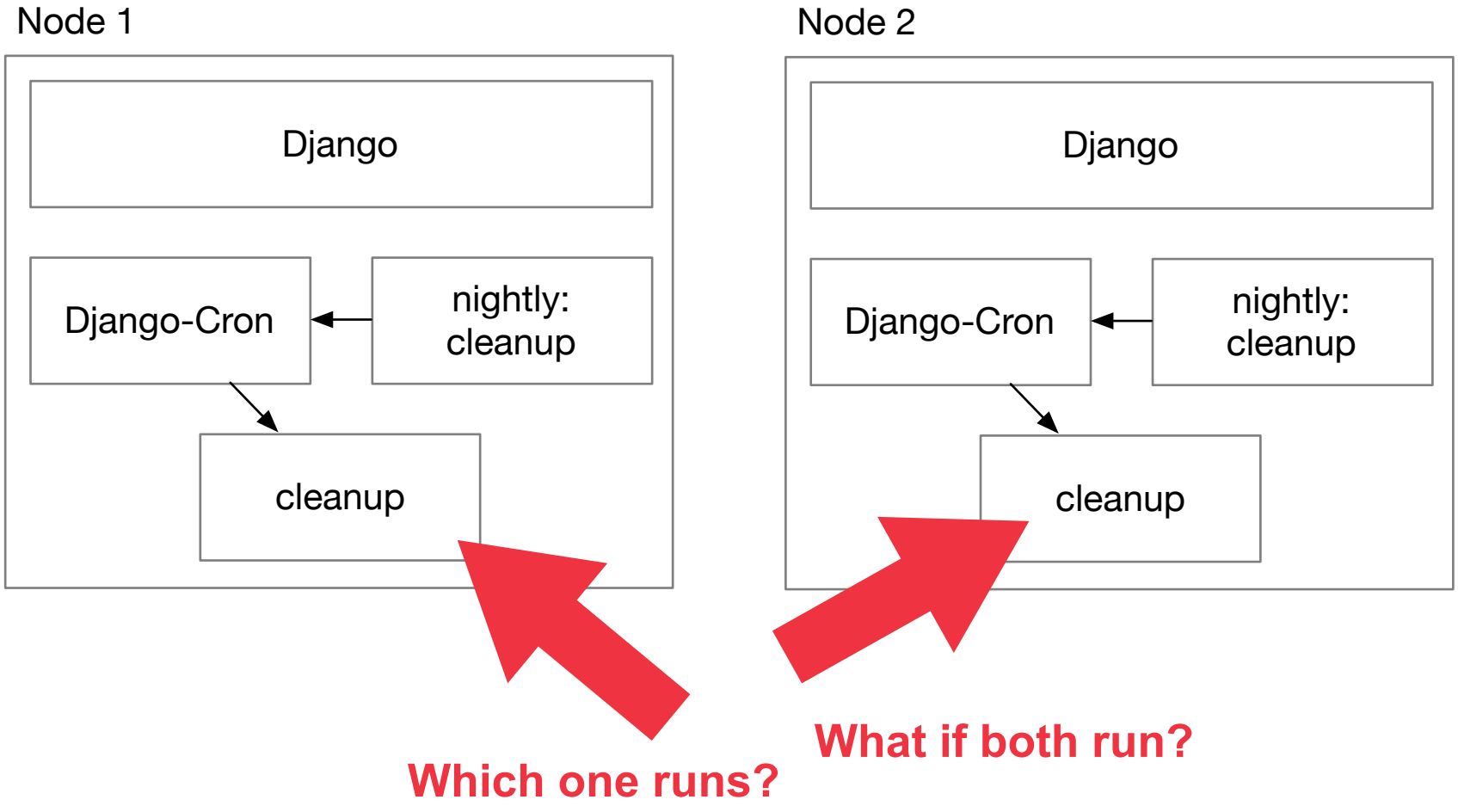
Pros

- **Microservices often natural fit as parts of WfS**
  - Small, well-defined boundaries functionalities and interfaces
  - Makes workflows explicit and easier to develop and understand
- **Common operations well-tested**
  - State transitions, retries etc.
- **Monitoring usually built-in**

Cons

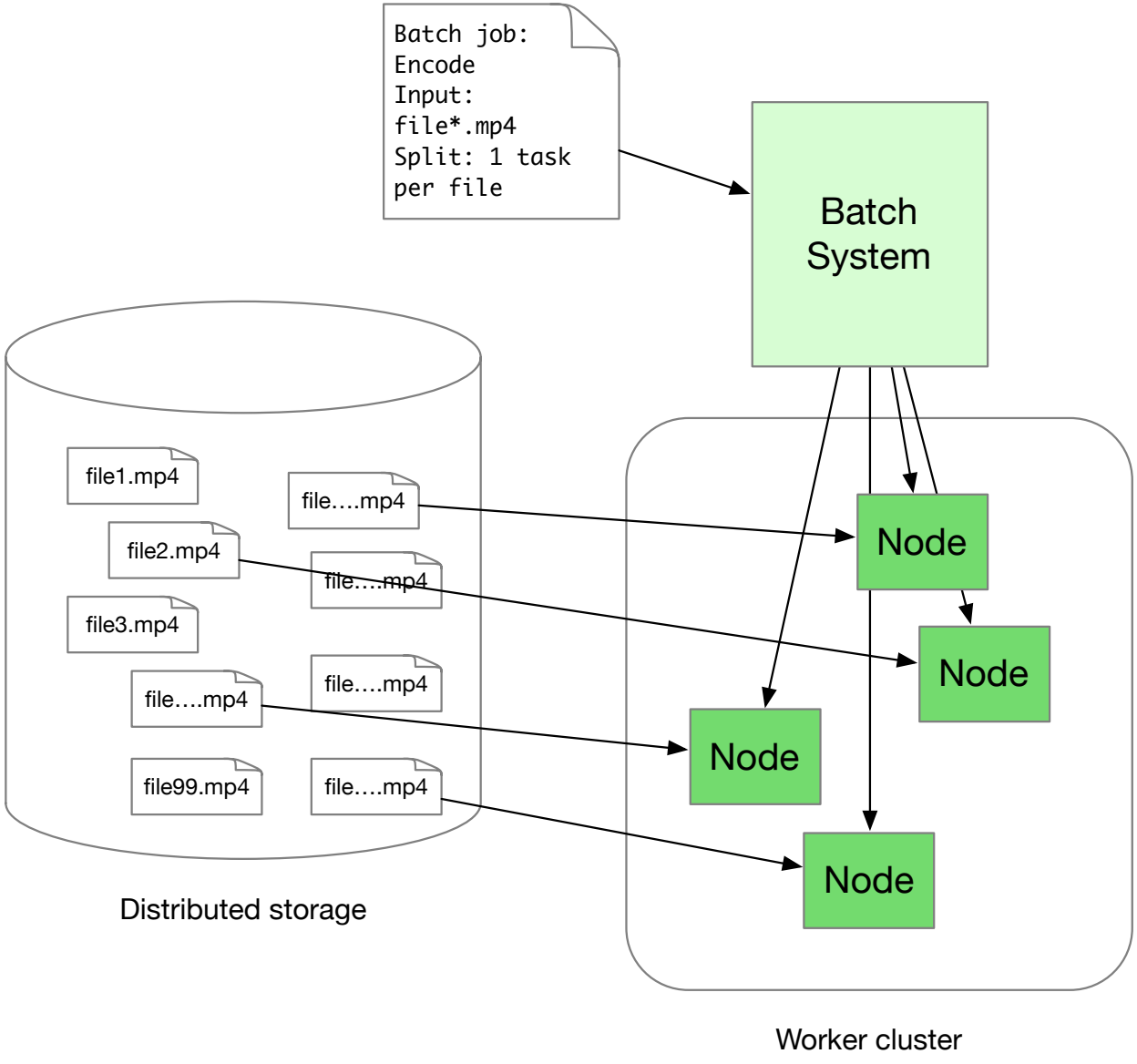
- **Centralized**
  - May hinder development
  - Creates centralized dependency on interfaces
- **Workflow "language" often very restricted**
  - Need external logic for complex decisions → complexity
- **Asynchronous workers**

Batch processing


Node 1
Django
Django-Cron
nightly:
cleanup
cleanup
Node 2
Django
Django-Cron
nightly:
cleanup
cleanup
Which one runs?
What if both run?

Batch systems

- **Compared to workflow systems**
  - Instead of orchestrating complex flow of a task …
  - Perform a single operation (maybe for a large number of tasks)
- **Big data comes to play too**
- **Operating-level solutions, OSS and IaaS ones**
  - AWS Batch, Azure Batch, Google DataFlow
  - HTCondor, Slurm, …
- **Batch systems handle “jobs”**
  - Which may contain multiple stages and parallel execution


Batch job:
Encode
Input:
file*.mp4
Split: 1 task
per file
Batch System
file1.mp4
file2.mp4
file3.mp4
file….mp4
file99.mp4
file….mp4
file….mp4
file….mp4
file….mp4
Distributed storage
Node
Node
Node
Node
Worker cluster

Use cases

- **Big Data analytics**
  - Specialized batch job systems: Hadoop, Spark, …
  - Data storage, transport and job design often critical
- **Parallelization of simple task on large data set**
  - Re-encode all videos with new codec?
  - Reprocess all archived log files for ingestion into a new system?
- **Scientific computing (see Aalto Triton for an example)**
- **Scheduled batch jobs (aka cronjobs)**
- **Recurring Extract-Transform-Load jobs across data stores**

Batch systems

Pros

- **Automation on task placement and distribution**
- **Management, monitoring and failure handling (retries)**
- **Conceptually relatively simple**
  - But for data intensive, devil is in the details …
- **Scheduling easier**
  - Potential for resource usage optimization across org

Cons

- **Centralized**
  - Conflicting resource needs across jobs?
- **Rather large hammer for many problems**
- **Long running times**
  - What if daily job gets stuck and runs >24 hours?
  - Time-consuming to develop and modify

Data streaming

Aalto University
School of Electrical
Engineering

What if …

- You need results NOW instead of tomorrow?
- Nightly job cannot run to completion in 6 hours?
- You can not store all of the data?
- Data is coming in continuously?
- Data rate is highly variable?
- You need to pass the data raw or after pre-processing to different data consumers?

The solution to you problem is:

Data streaming!!

Batch vs. stream processing

	Batch processing	Stream processing
Data scope	Queries or processing over all or most of the data in the dataset.	Queries or processing over data within a rolling time window, or on just the most recent data record.
Data size	Large batches of data.	Individual records or micro batches consisting of a few records.
Performance	Latencies in minutes to hours.	Requires latency in the order of seconds or milliseconds.
Analyses	Complex analytics.	Simple response functions, aggregates, and rolling metrics.

Source: AWS

Examples:

- Log ingestion
- Device sensors
- User interactions (game, website, mobile app, ...)
- News / social media feeds

Data streaming

- **Commercial and open source solutions**
  - AWS Kinesis and its variants, Azure Stream Analytics, Google Cloud DataFlow
  - Apache Kafka, Apache Spark Streaming, …
- **Concepts**
  - Data producer, consumer & stream
  - Streams, records, partition keys, … differences between solutions, also pricing units and resource allocations
  - ”Streaming” is not a continuous flow of bytes – instead: large number of small records (kilobytes)

Other comments

- **Many systems are internally streaming architectures or appear so functionally**
  - Log and metrics collection (Elasticsearch, Logstash, ...)
- **Bugs in data consumers**
  - Debugging ...
  - What if need to reprocess data?
  - What if data retention is short in the stream?

- **Generally: The shorter latency in processing, more difficult to develop and maintain (batch vs. workflow vs. streaming)**

Why?

Why asynchronous models?

- **Splitting a big task to smaller, sequential pieces**
  - Easier to develop and debug each in isolation
  - Natural for microservice architectures to create service boundaries
- **Less prone to failures, easier to recover**
  - Management can be made HA and resilient
  - State transitions ~idempotent → no (big) problem re-running
- **Less sensitive to processing delays and load variations**
  - Not in path of synchronous processing (order fulfilment ~ days!)
  - Buffering, capacity scaling
- **Many business processes <u>are</u> workflow processes!**

Messaging

Messaging

- **Messaging is exchange of asynchronous messages via a 3rd party**
  - Message queues: unordered / FIFOs, single message (1-1)
  - Publish/Subscribe (PubSub): Message fanout 1-N
  - Message bus: PubSub, but goes much into ESBs …
  - Specialized systems (Celery – task queue, e.g. asynchronous RPC, message priorities, …)
- **Lots of OSS and commercial solutions**
  - AWS SQS (FIFO) & SNS (PubSub), Apache ActiveMQ, RabbitMQ, … (lots and lots), also can use databases

Sender and receiver

```
queue = sqs.get_queue_by_name(
    QueueName='request-queue')

@app.route("/")
def hello():
  queue.send_message(
    MessageBody="got request")
  return "Big bro knows now!"
```

```
queue = …
while True:
  for message in \
    queue.receive_messages():
    print("Got message: {}".format(
        message.body))
    message.delete()
```

Why?

Kubernetes example

- Uses Apache ActiveMQ
- Sender: User “registration” web page
- Receiver: Receives registration, demonstrates choreographed workflow
- All containers run in same pod, can see localhost:<port> of others

Why?

- **Standard way to decouple systems**
  - Assuming that MQ system itself is reliable
  - "Fire-and-forget" solution
  - Naturally suitable for bursty traffic
  - Many other solutions build on top of messaging systems!!
    - *Integrations in workflow systems*
- **Ease of changes (just one example)**
  - Originally: A → queue 1 → B
    With filtering: A → queue 1 → C → queue 2 → B
    (need only to change B's source queue configuration)

Problems

- **“Fire-and-forget” does not guarantee a receiver**
  - Received crashed? Incorrect destination? Badly formatted message?
  - Dead Letter Queues – one more resource to monitor
- **“Enterpriseyness”**
  - Problems if too much logic encoded into messaging system (ESB!)
- **Centralization**
- **At-most-once vs. at-least-once delivery**
  - Always will be either one! Think about Brewer’s theorem too ☺