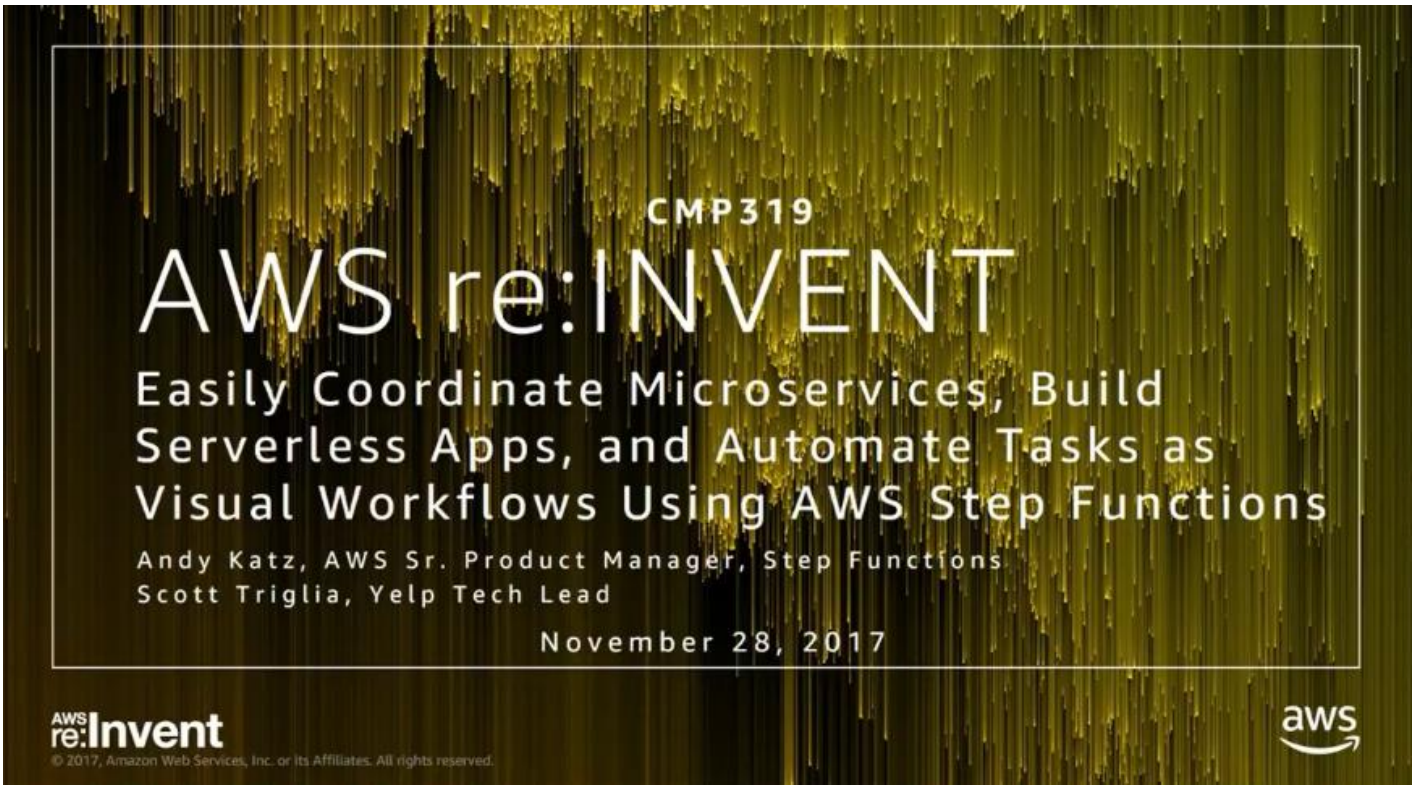
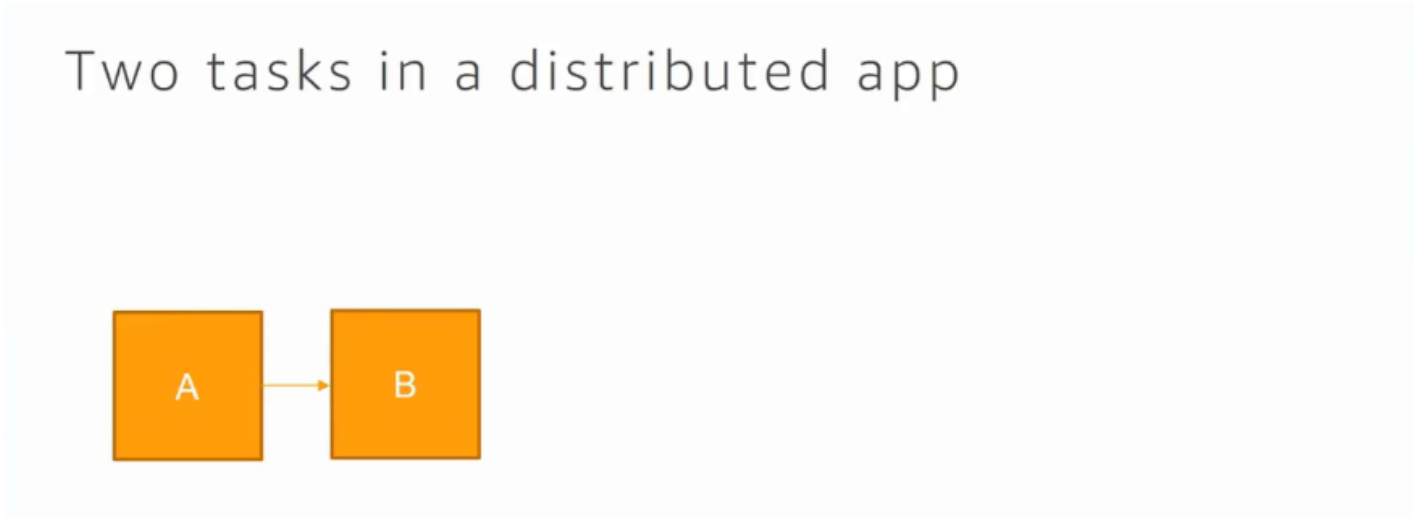
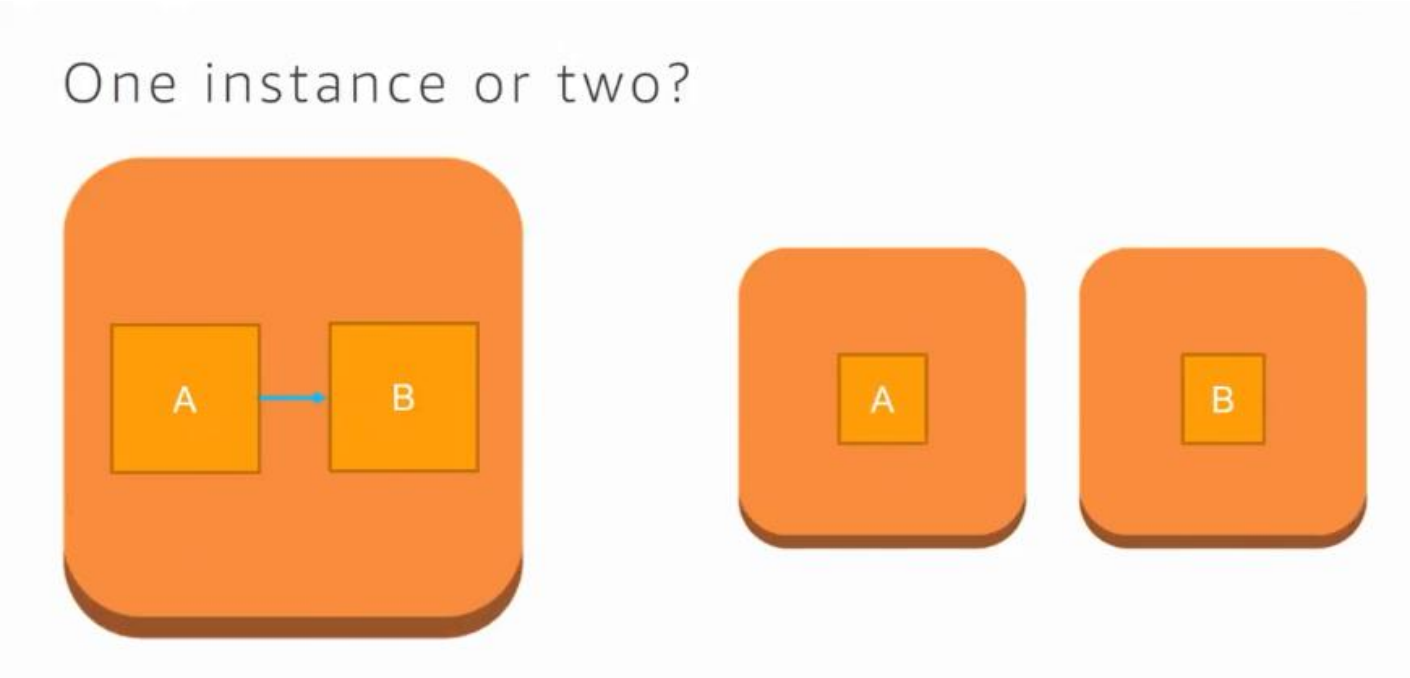


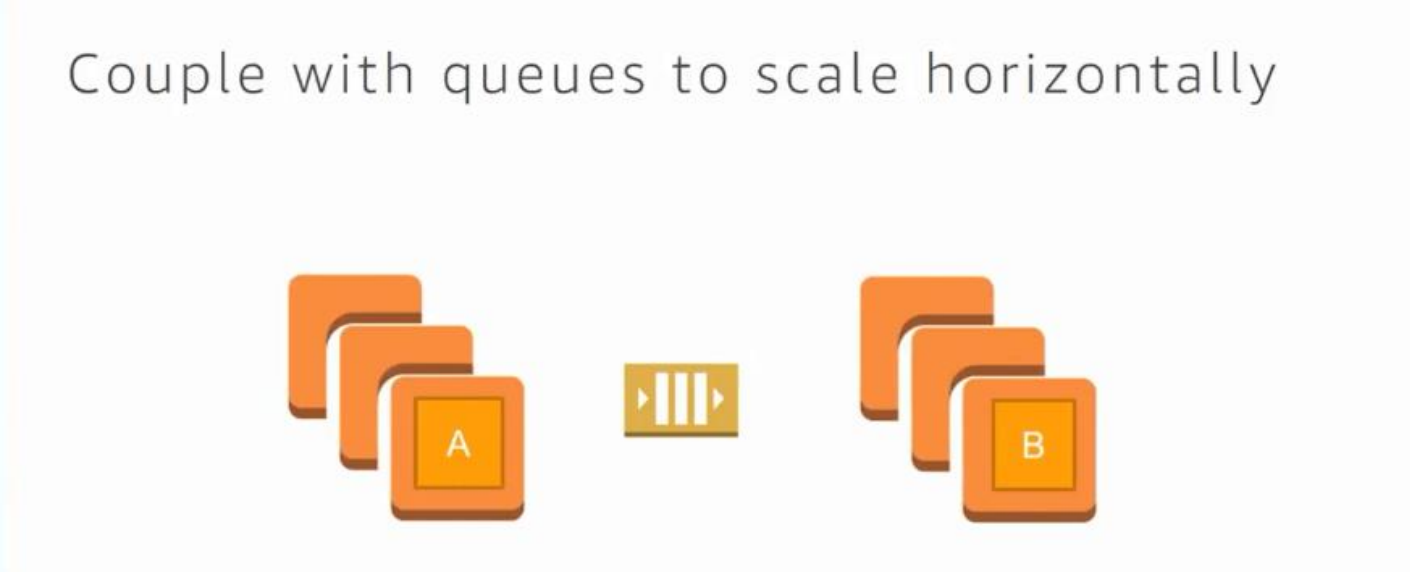
We will talk about the benefit of using orchestration in a distributed environment.

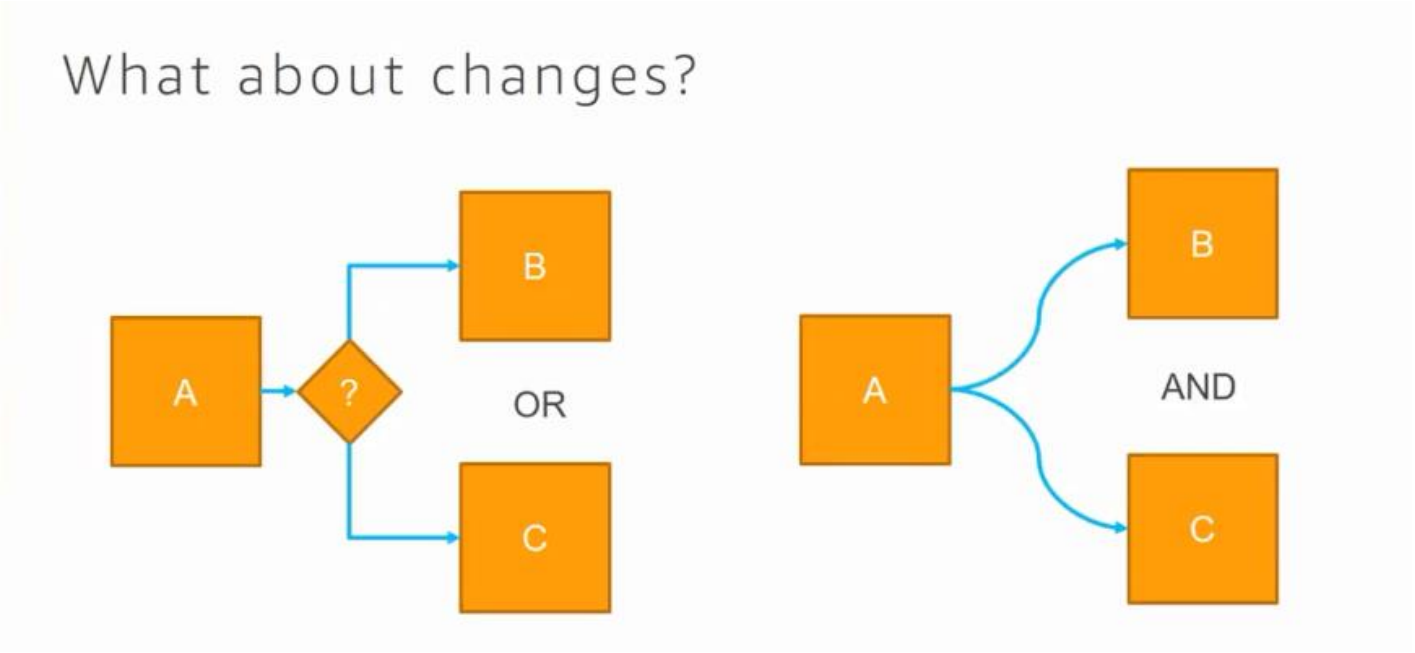


Workflows always describe a series of steps, like checking for a restaurant before making a reservation

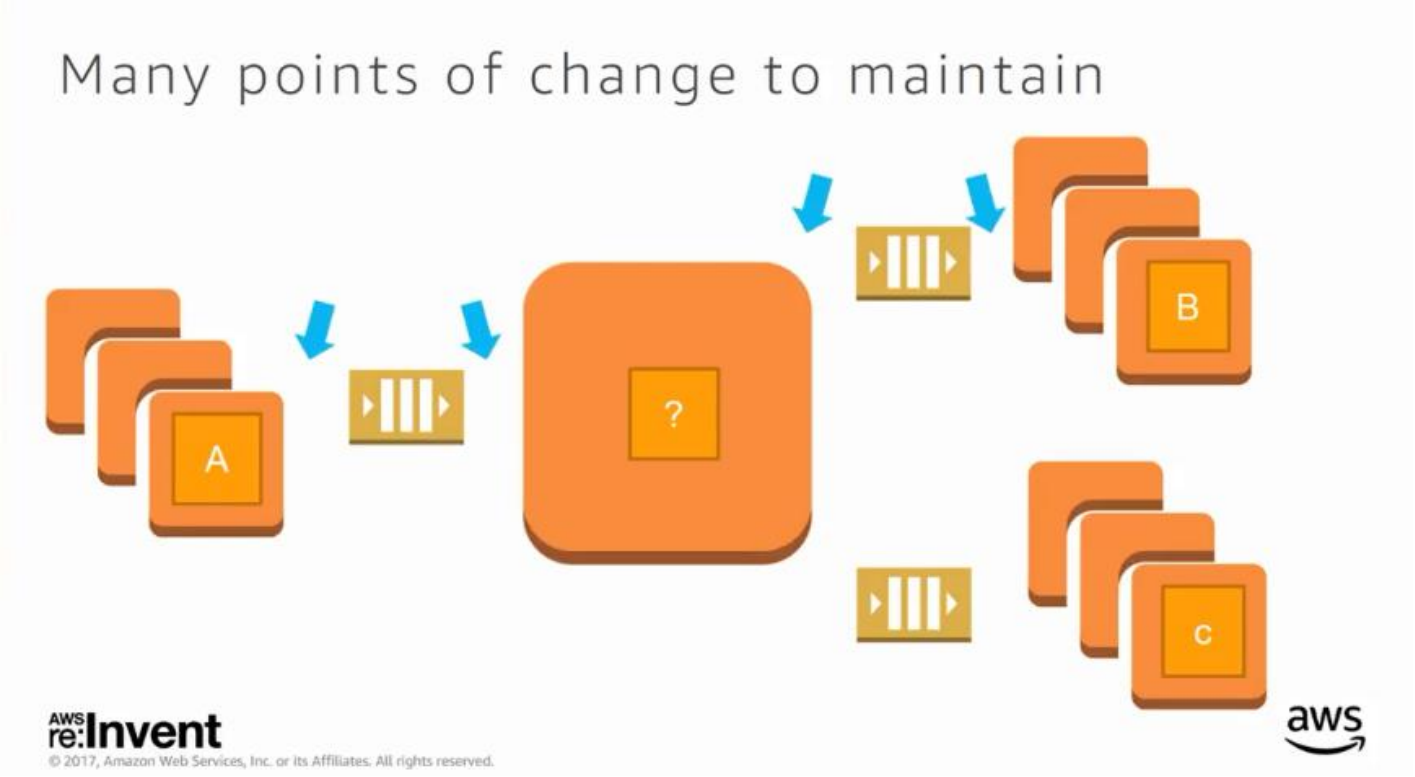


Are you building this as a monolithic app or a series of smaller apps for easy scaling

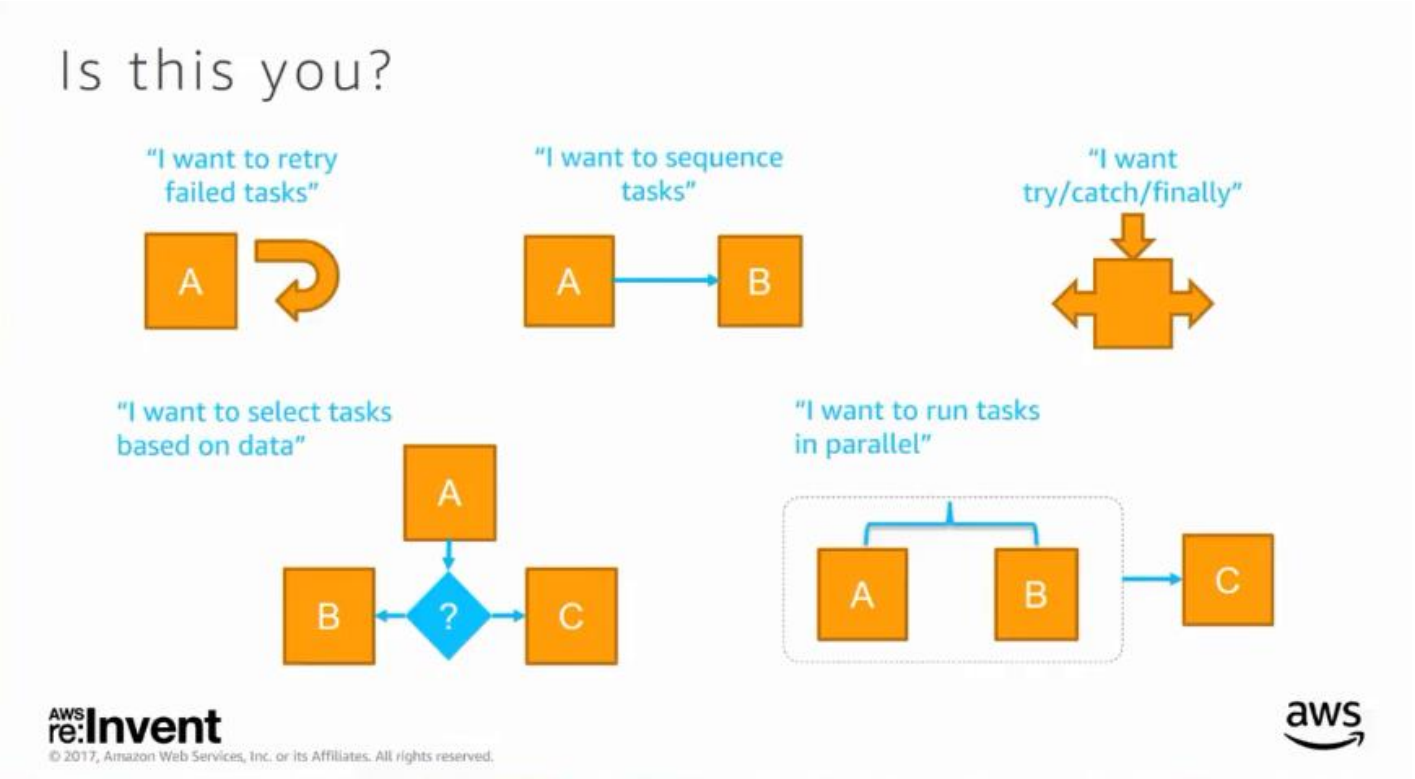


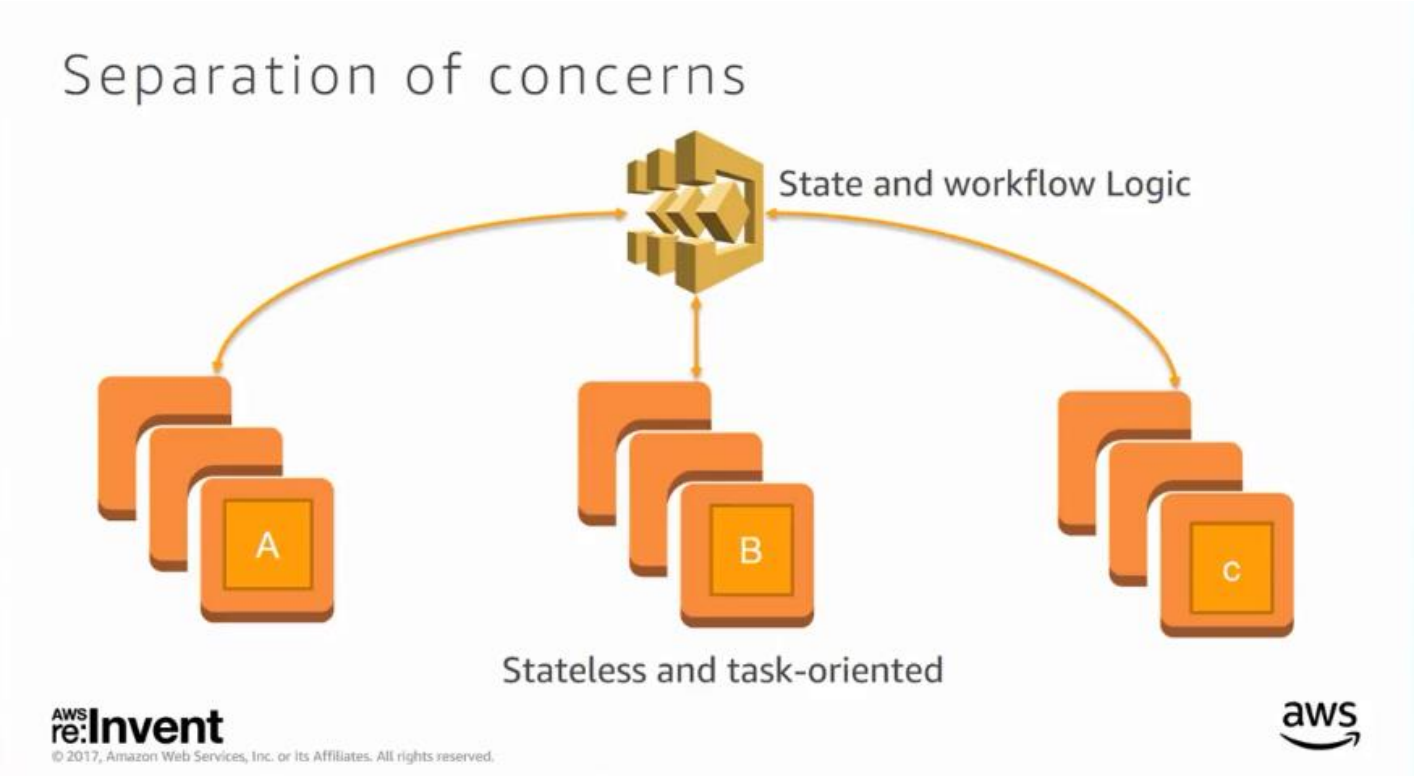


You can also arrange the tasks in their logical order

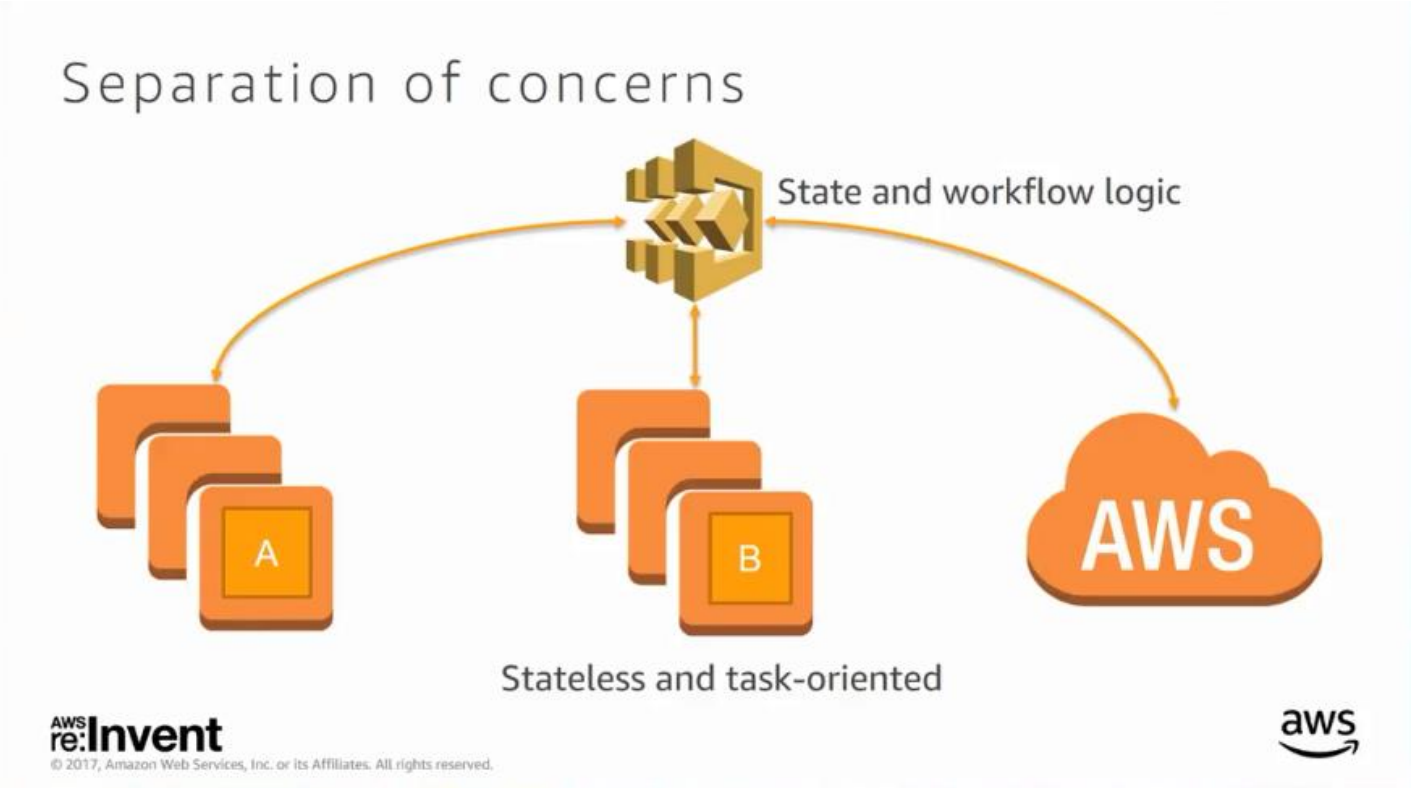


You can do this with queues but then you end up with many points of contacts to maintain as your app changes. Soon, your workflow logic gets intermingled with your application logic.

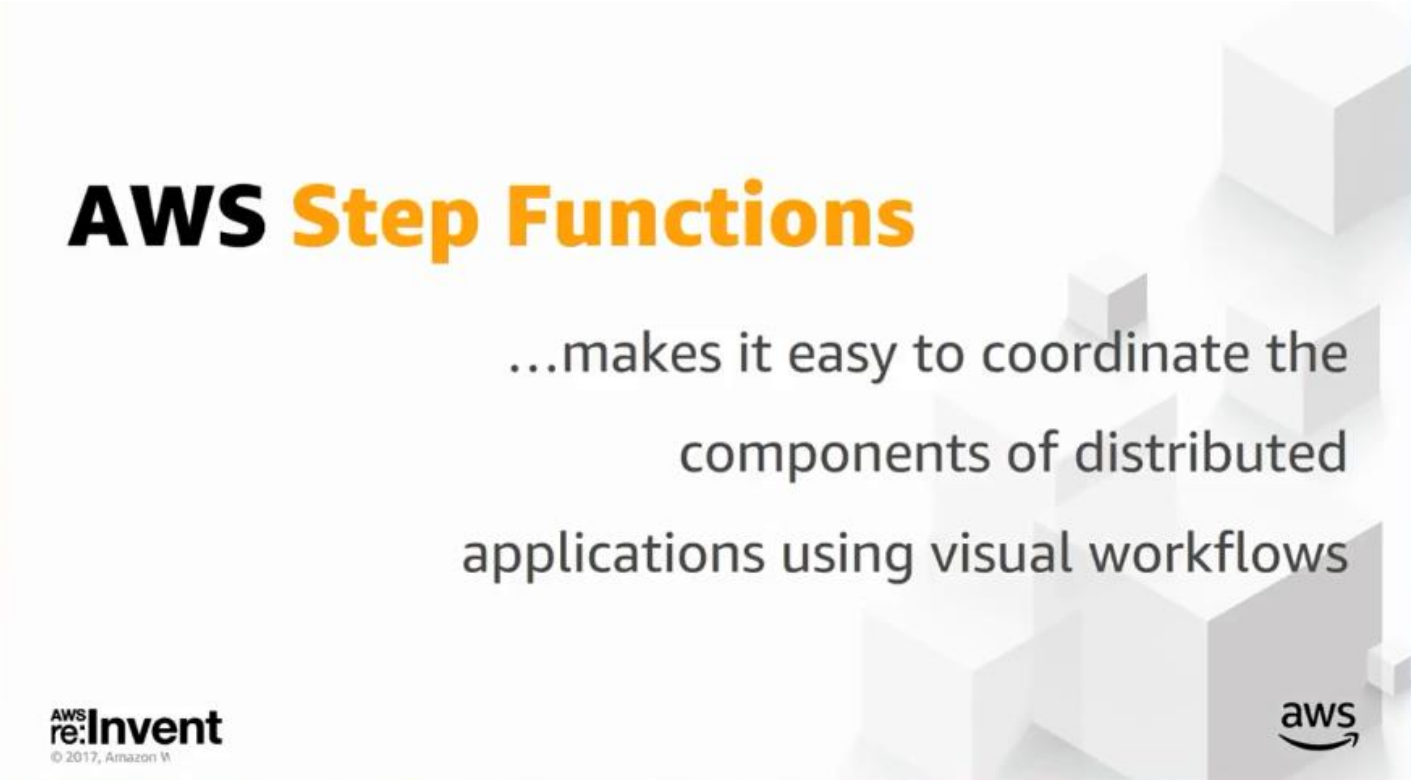


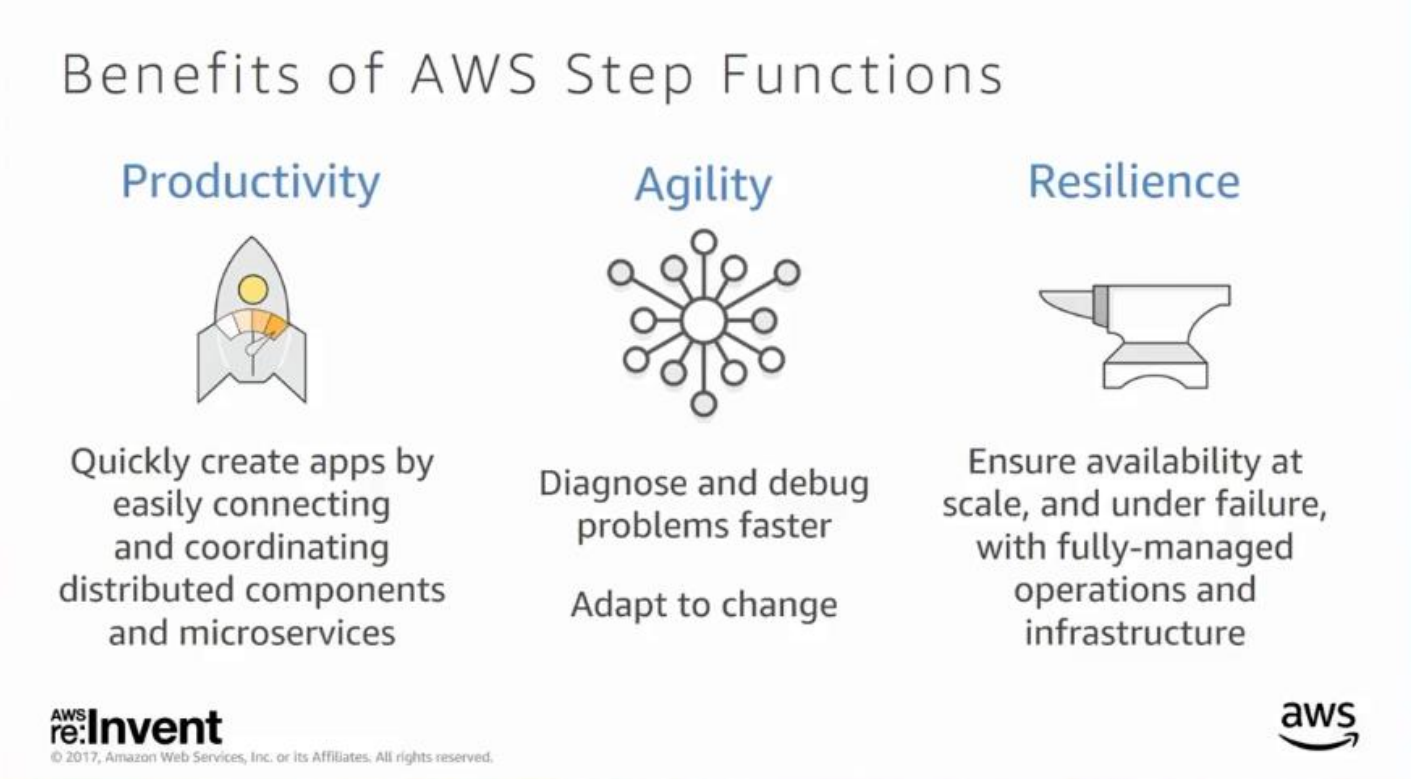


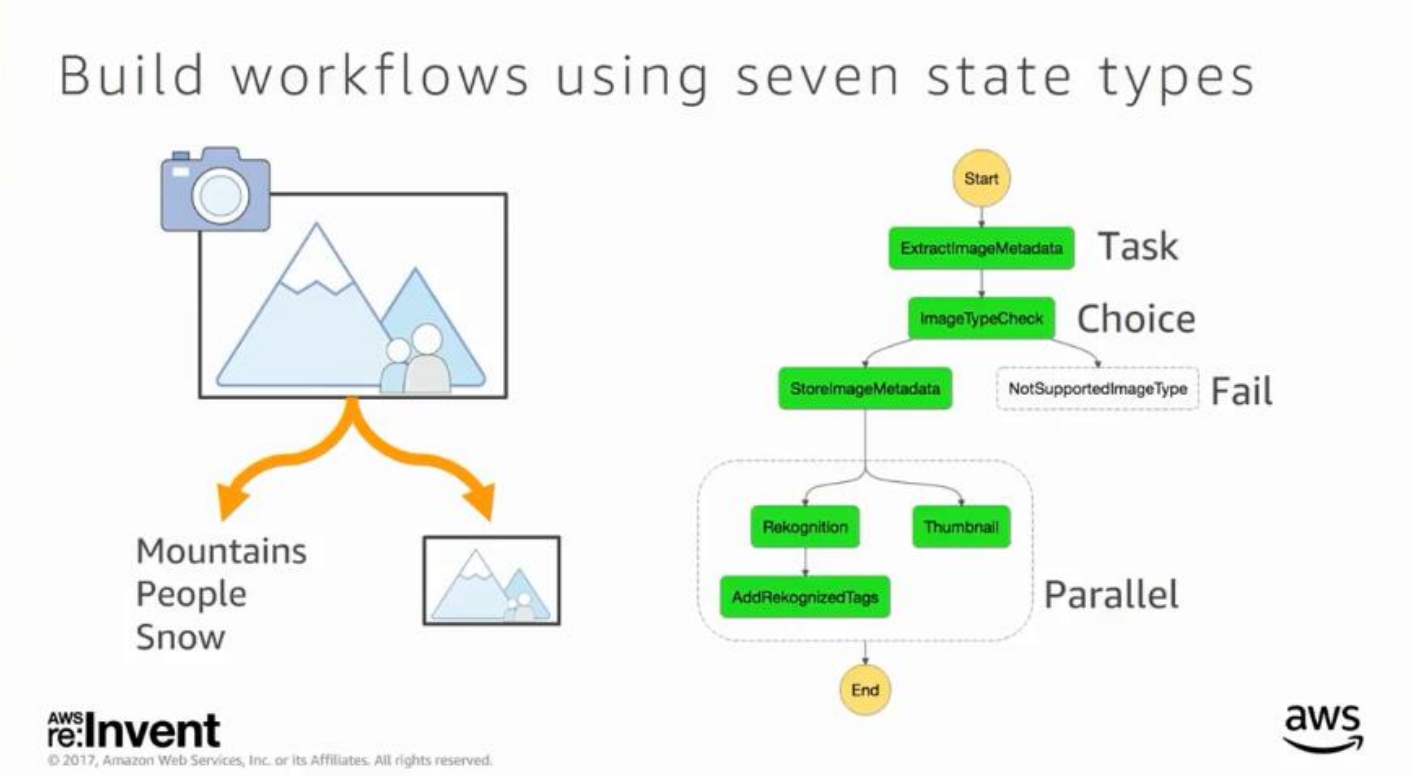
you should separate your business and workflow logic, you can make your tasks stateless and focus them on doing one thing well as single, deterministic tasks. Then you can track the state of your workflow in a separate layer of your application, this layer should take care of the overheads of retries, timeouts, error handling, conditional branches, and parallel sets.



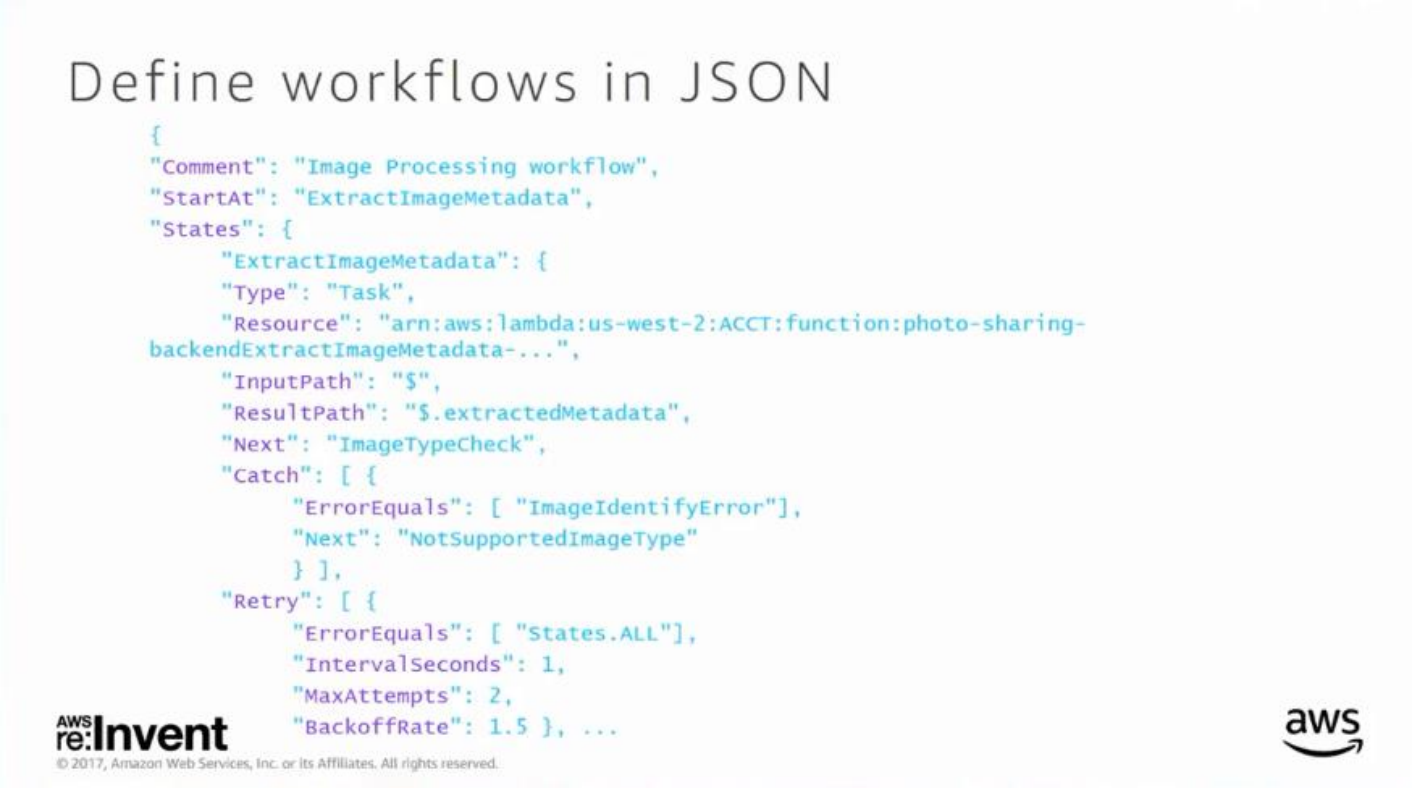
You then need to work exclusively with your compute resources like AWS resources as your tasks grow in number.



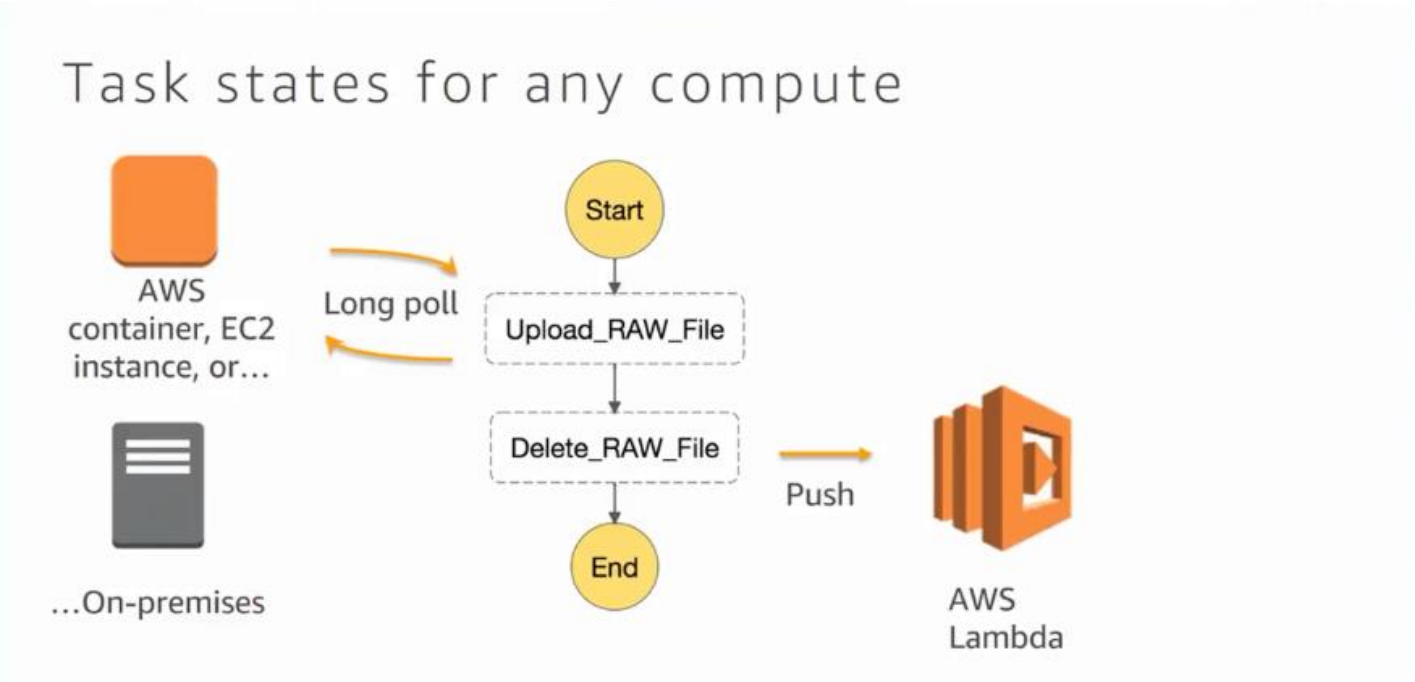




Above is an example of a Photo Processing Workflow. You take a photo, upload it, then in parallel get a thumbnail back and use AWS Rekognition service to identify features within the photo like people. The flow chart on the right is the step function graph, each stage in the workflow is represented by a state and there are different types of states. ***Task states*** are work horses that do the work, ***Choice states*** have branching logic. You can go to a ***Fail state*** to exit with an error condition you choose, ***Parallel states*** do work in decoupled fashion to run your tasks simultaneously. This diagram is a Finite State Machine.

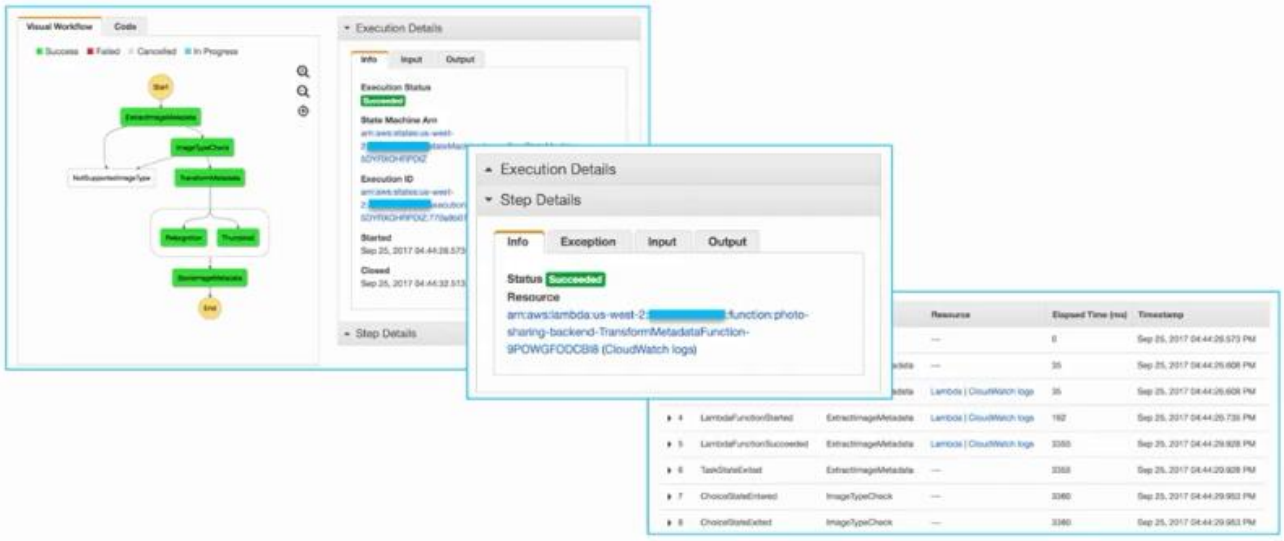


You define your workflow declaratively in JSON in the Amazon States Language ***ASL***, above is what the previous workflow looks like in ASL. Each state is connected to the next state by a state called ***Next***.



Step functions are supports 2 types of tasks, Lambda tasks and Activity tasks. Lambda tasks push work to a lambda for you, function for you. When a step function workflow lands in a lambda task state, it would give the states data as input data into a lambda function you specify, it then waits for the lambda function to return the result back to it so that it can take that payload and transition to the next state in the workflow. Activity tasks states work with all sorts of compute like EC2, ECS, your laptop, etc. an activity task state can remain open for up to 1 year, so it can be used for very long running jobs for bioinformatics.

Monitor from the console, CLI or API



Use cases

- Automate daily, weekly monthly tasks
- Coordinate components of distributed applications
- Build microservices

home24
theguardian
SKYCATCH
yelp
outsystems
EllieMae

AWS Step Functions use cases:

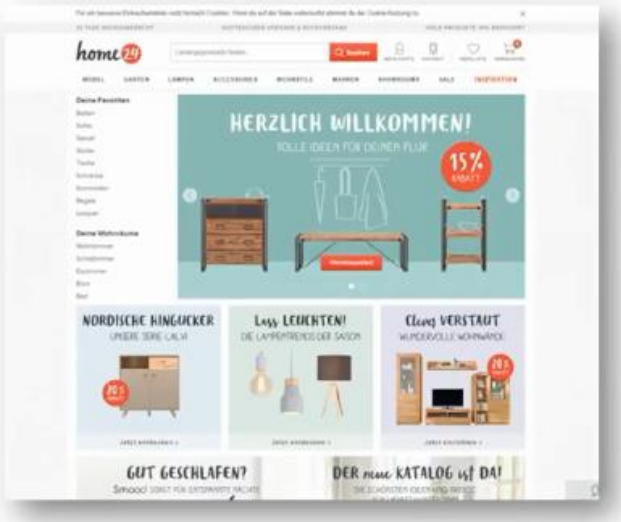
Automating daily, weekly, and monthly tasks

Coordinating daily imports of marketing data

home24

European market leader
in online shopping for home and living products, operating in
eight countries and two continents

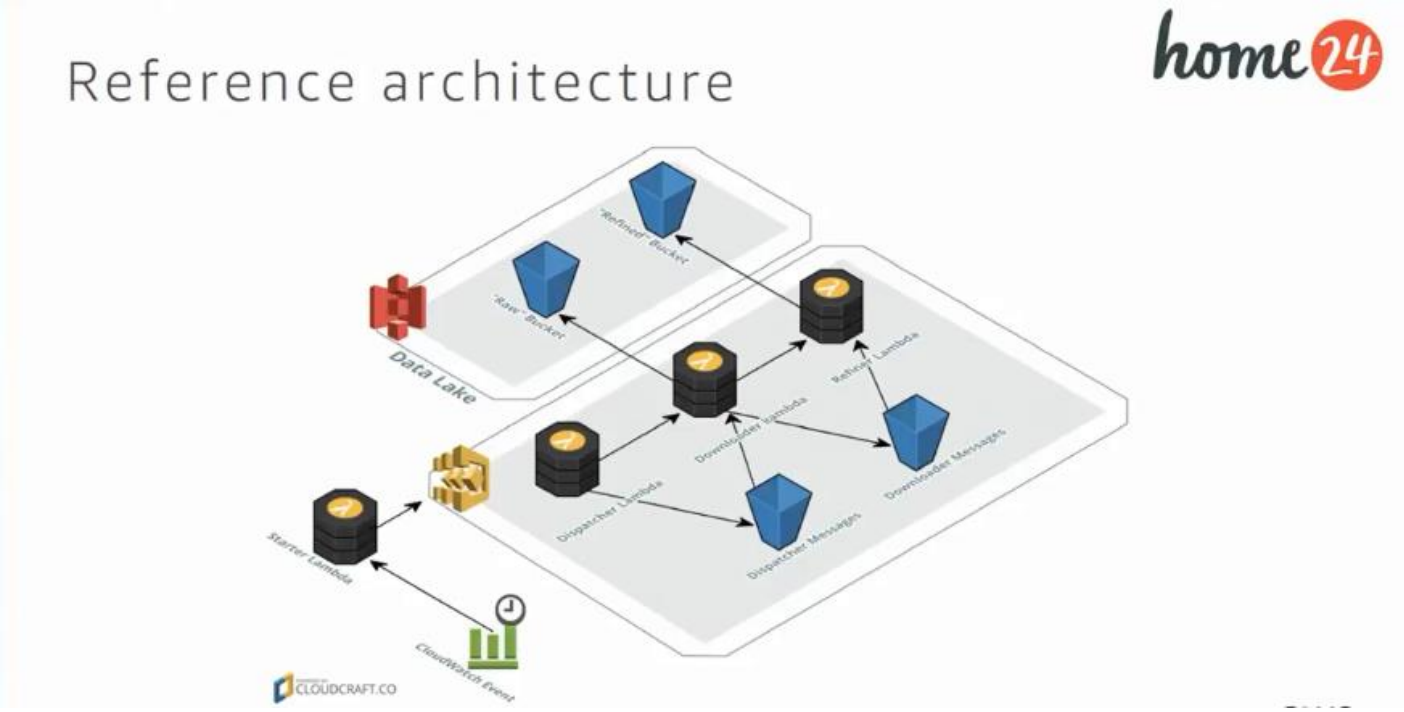
Built a reliable, long-running serverless application to coordinate the daily import of gigabytes of social media data from 20+ sources for use in business analytics



How home24 l lowered costs by 99%

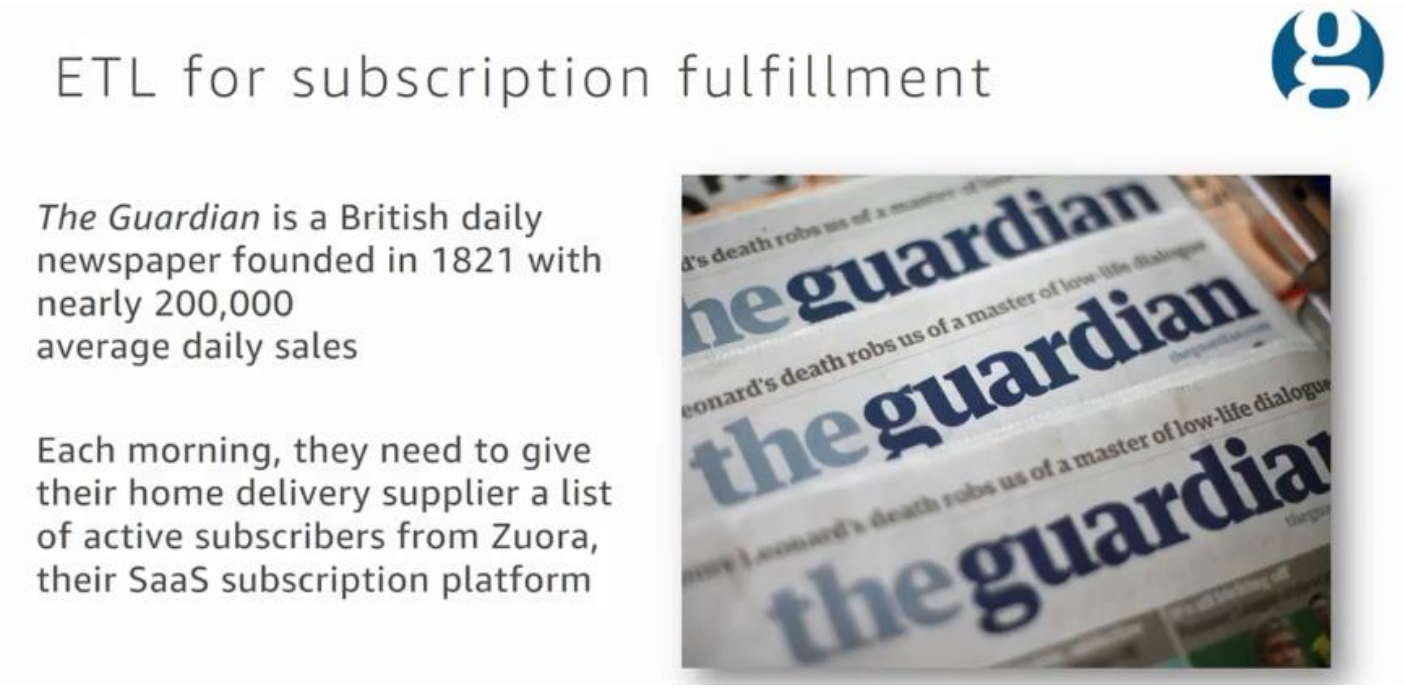
Home24 sequenced a varying number of AWS Lambda functions with AWS Step Functions to download and refine files from multiple sources.

- **Cost savings:** saved 99% with Step Functions as opposed to their previous SaaS solution
- **Productivity and agility:** engineers can easily coordinate various microservices to build ELT pipelines
- **Increased resilience:** automatic retries when service APIs are unavailable



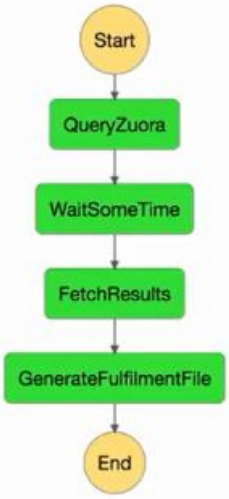
They have a data lake in S3 and they use this data lake to store both raw and refined/processed data. The Step Functions workflow is completely serverless, they chunk the data from big files into byte sized pieces that can be processed by a single lambda function since lambdas can only run for a maximum time of 5 mins at a time, this is a process of chunking the big file, iterate and hand off the work to individual lambdas that do their work and store the results into files in S3 buckets.

They start the workflow on a schedule using CloudWatch events, download the data from 24 different sources, break the files up into chunks, iterate over the individual chunks to refine the data into a form that they can consume, then store the results into their Resources bucket for use in their data infrastructure and consumer apps.



They have several work flows running on step functions, this example is about managing a home delivery list. Every morning, they need to get their home delivery fulfilment vendors a list of active subscribers for their website.

How The Guardian improved reliability and developer scalability

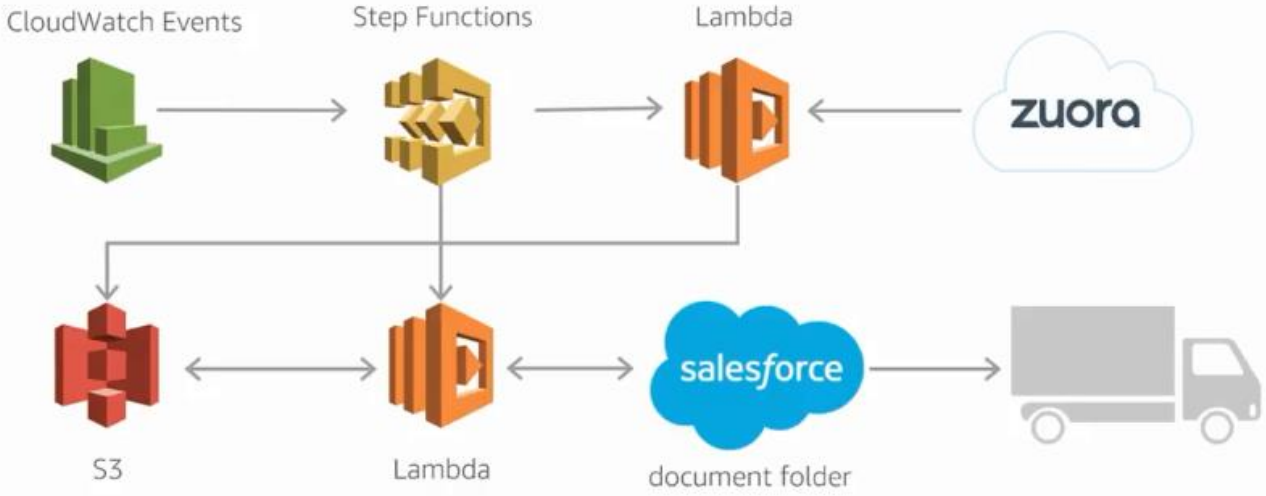


The Guardian rewrote their process into an ETL paradigm.

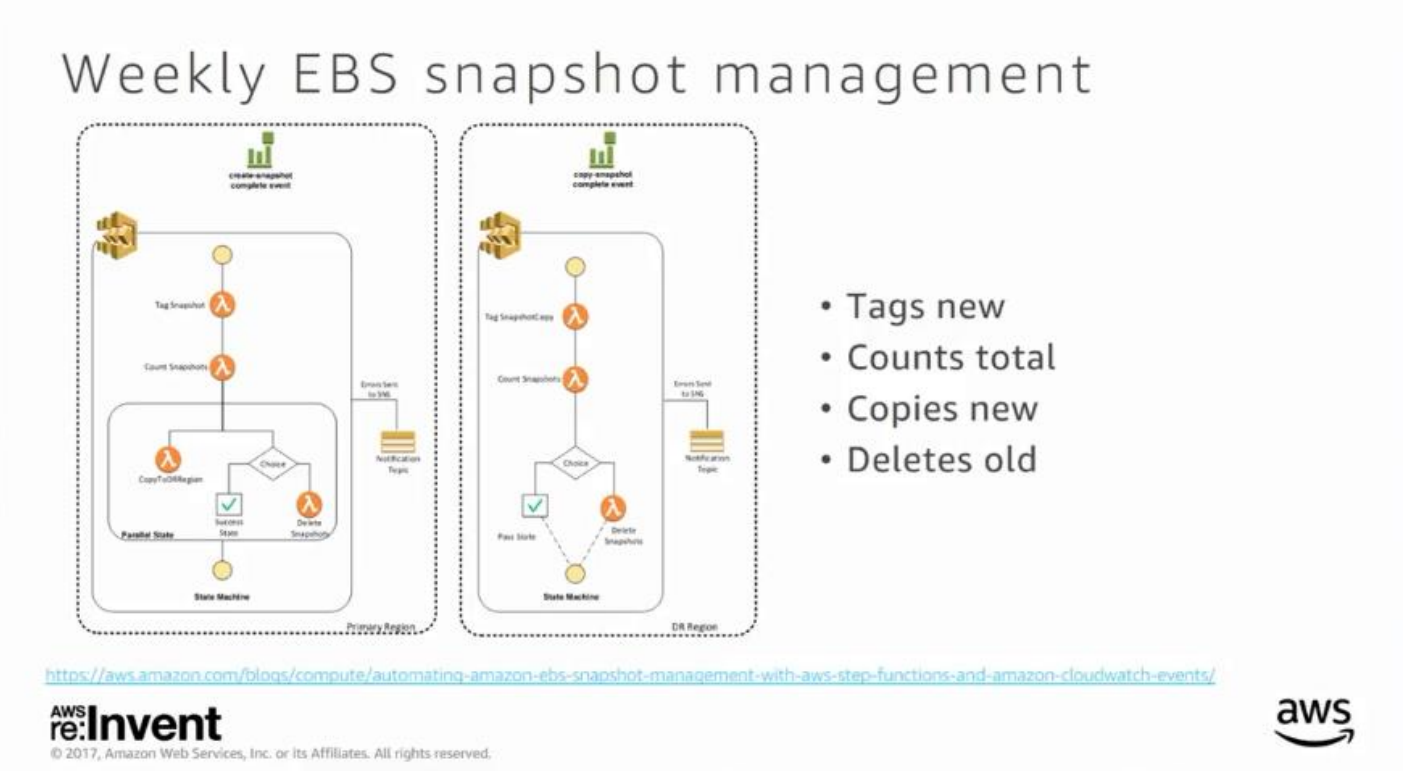
- **Improved reliability:** Amazon Simple Storage Service (Amazon S3) versioning makes the process idempotent so it's easy to recover from failure
- **Developer scalability:** less dependencies on contract developers by moving to a familiar technical landscape
- **Supplier efficiencies:** fulfillment files are pre-generated and faster to download

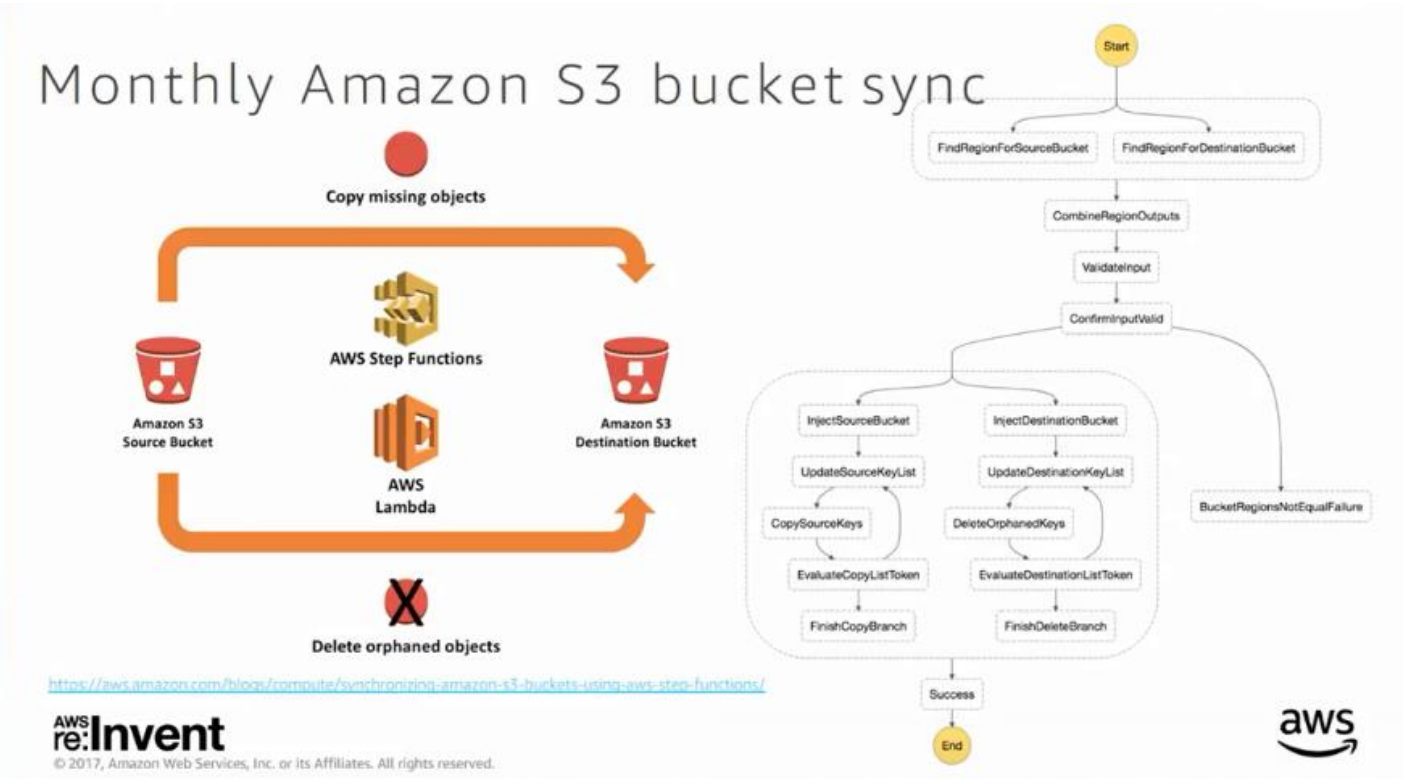
Reference architecture

See: https://github.com/guardian/fulfilment-lambdas

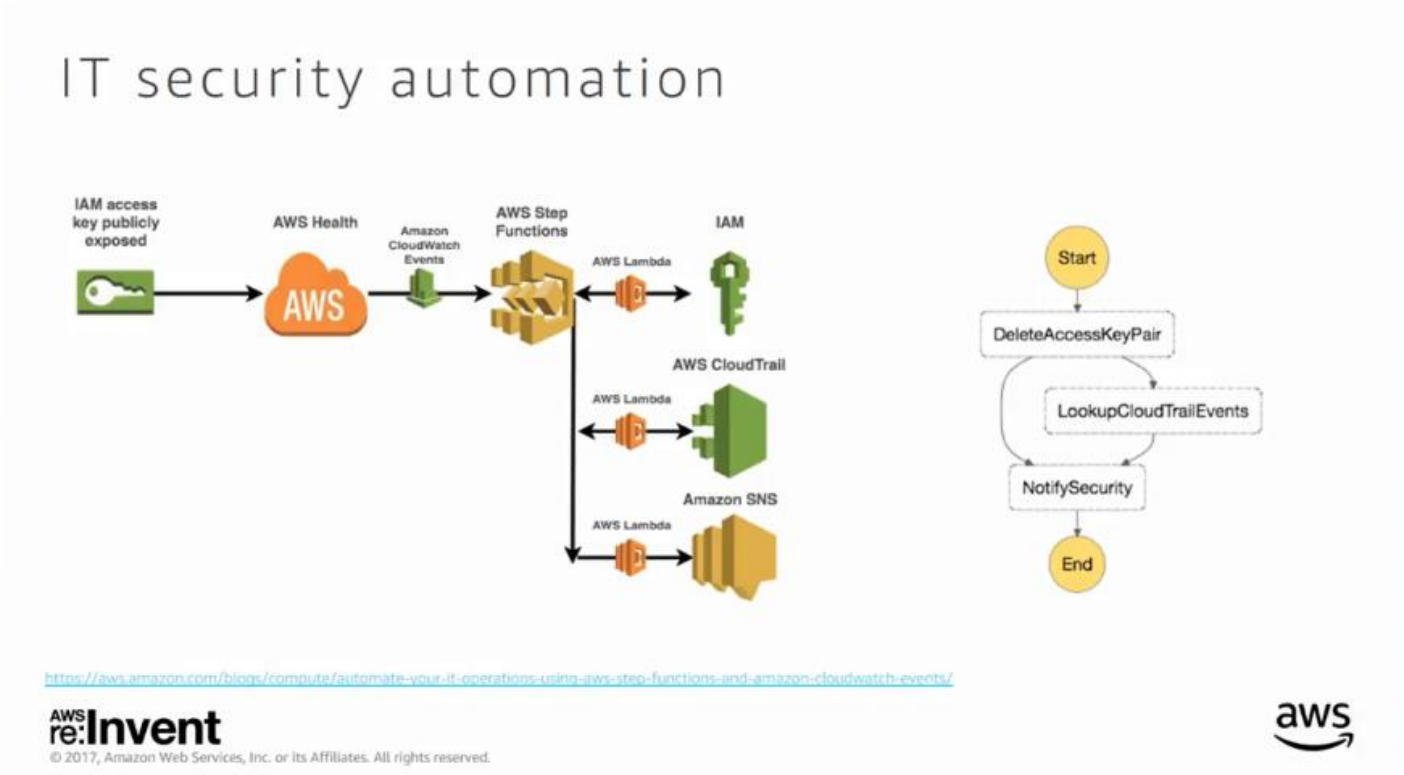


This is what their workflow looks like, Zuora hosts the 3rd party data.





Suppose you are managing a website, and you need a way to update the website with all the local changes in a reliable and fast way. You don’t want to delete the contents of the entire bucket and then re0copy the files back again. this is on Github



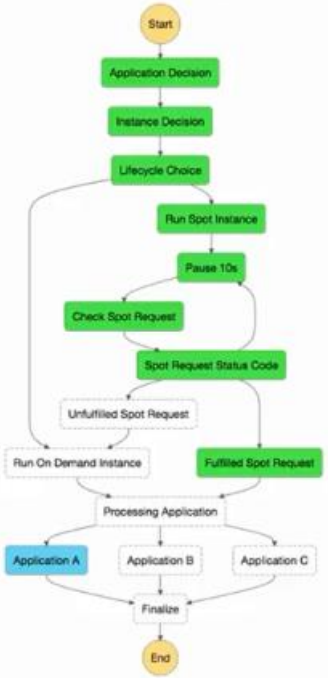
This automates the response to an accidental exposure of your IAM keys, it monitors Github continuously to see if some has accidentally exposed your IAM keys, this triggers as event that does 3 things to remedy the situation.

Let us see some examples of distributed applications that are built using step functions.

How SKYCATCH saved 60% and improved SLAs

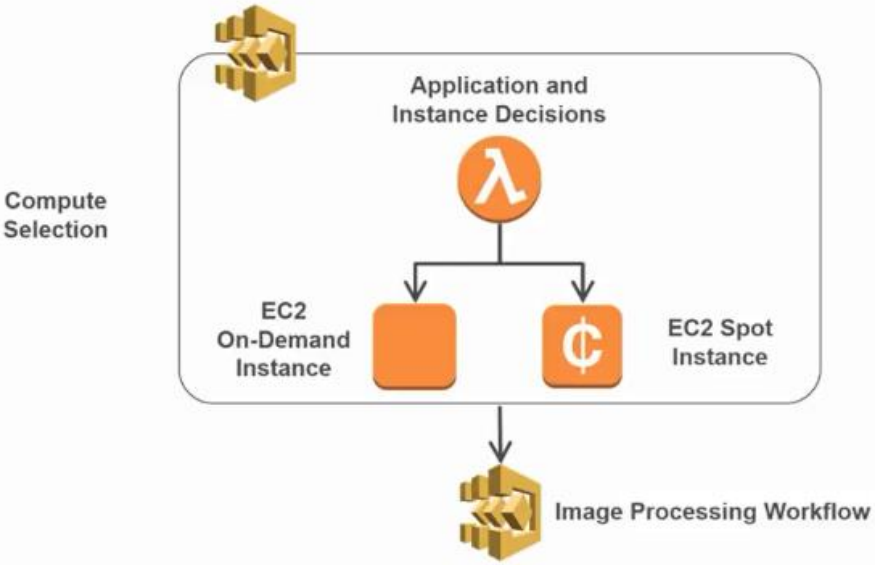
Skycatch built a solution to select the most cost-effective AWS resource available for each image processing workflow

- **Approach:** select Spot or on-demand Amazon EC2 instances based on service turnaround time, and automate restarts of failed image processing workflows
- **Cost savings:** reduced compute spend 60% and allowed the business to scale independently of headcount
- **Lessons learned:** a central audit trail simplifies error analysis, and state machines promote consistency in operations



SKYCATCH

Reference architecture



AWS Step Functions use cases:

Subscription billing at Yelp

Moving a 10-year-old codebase from tangled spaghetti toward serverless

yelp

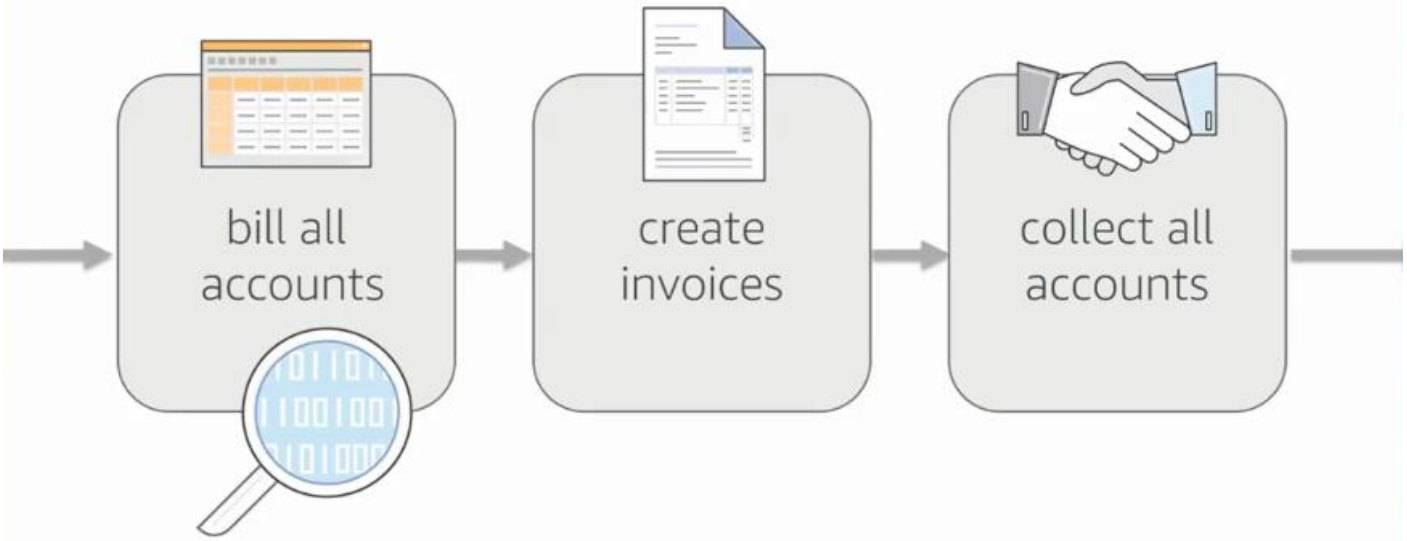
Monthly invoicing for advertising accounts

Business critical, happens at scale (>100k accts)

Older code, evolved over ~10 years

Subscription billing at Yelp

yelp



This is a nightly job that runs for about 7 hours on a schedule, it is very important to processing accounts and accounts collections to determine what each of 100,000 advertisers owe to Yelp, then it creates an invoice to send out to each customer.

Migration challenges

Not great observability

Older code, evolved over ~10 years

We are going to target just a segment of the job at first and put in a lot of comments to determine what went wrong

Migration challenges

No good API existed for the business process

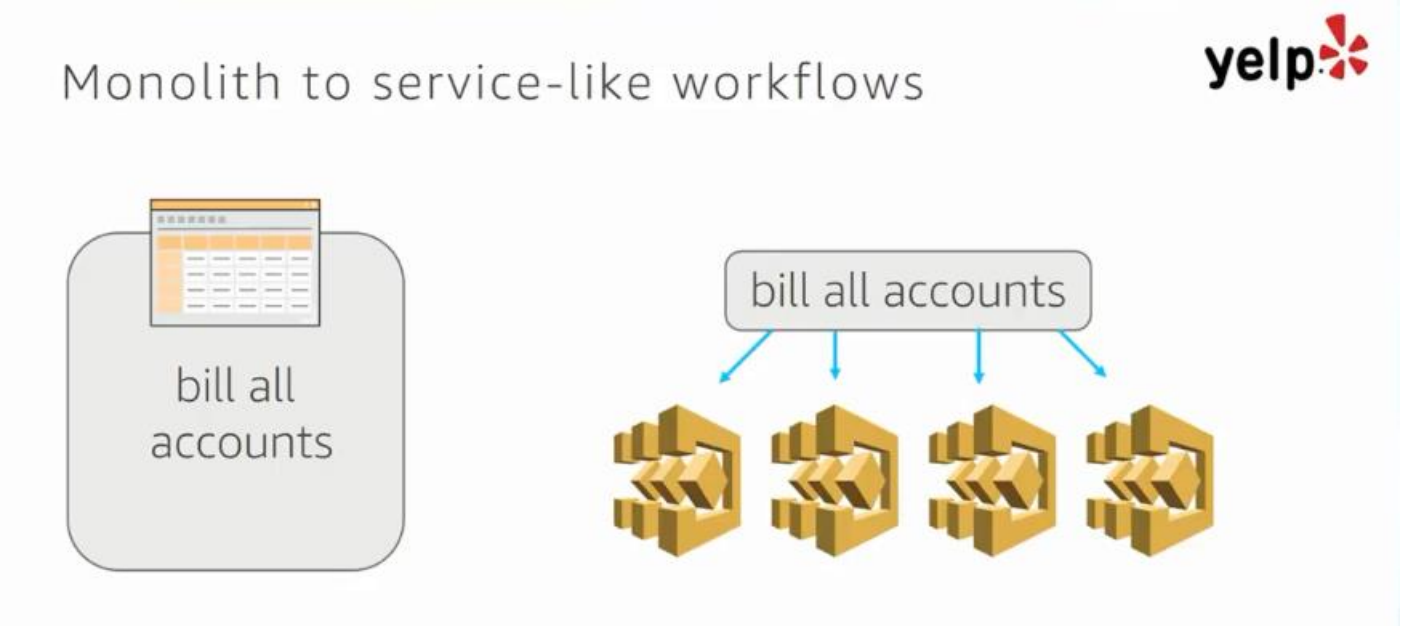
Very far from being Lambda function compatible

Not nicely factored into small pieces

Old logic


```
for chunk in all_accounts:
    for account in chunk:
        bill_account(account)
```

This is the process for billing every customer account that needs to be analyzed.

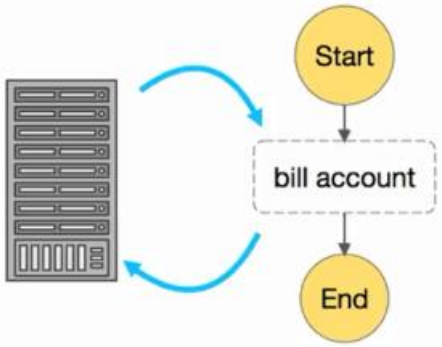


We had to migrate that logic to step functions by running each account through a step function. This allowed us to parallelize the entire batch of accounts

We simply pass the data needed by the step function at the start of each execution in the format above. We simply specify what the step function needs to start calculating the billing for a single account like the clientId, date, account type.

Activity tasks where AWS Lambda functions aren't yet practical

```
"Bill Account": {
    "Type": "Task",
    "Resource":
    "arn:aws:states:us-west-2:
     1234:activity:bill-account",
},
```

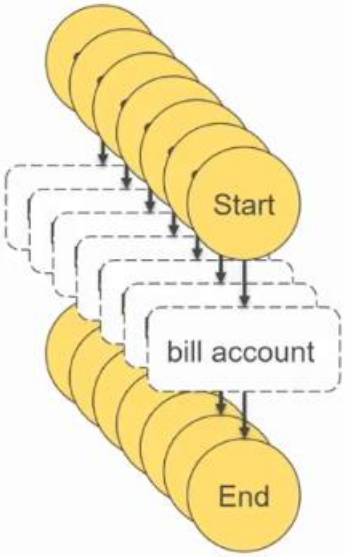


There are several ways that you can do this implementation, you can push data into the lambdas or you can be pulling work for lambdas using activity tasks. We only need to add a little bit of glue code to wrap up our different pieces of logic to get this working

Benefits: concurrency

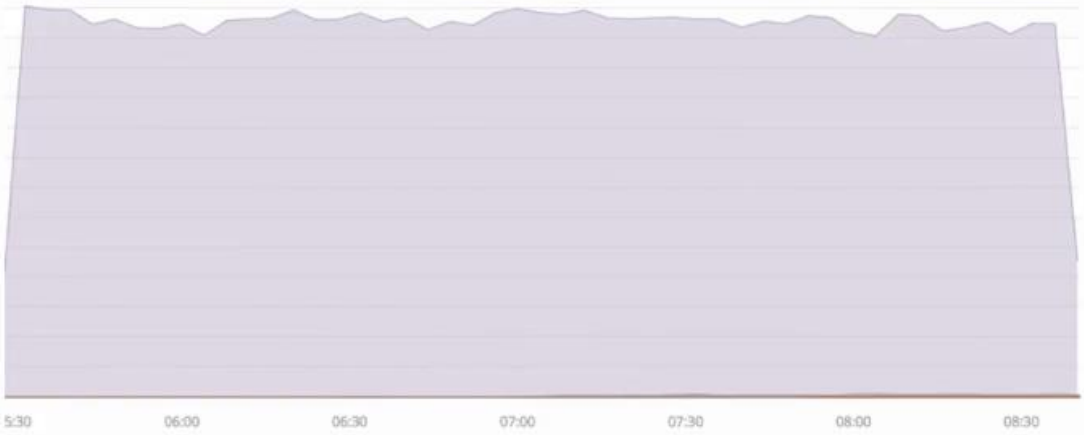
Easily modify parallelism factor with concurrent executions

Concurrent executions offers significant speedup



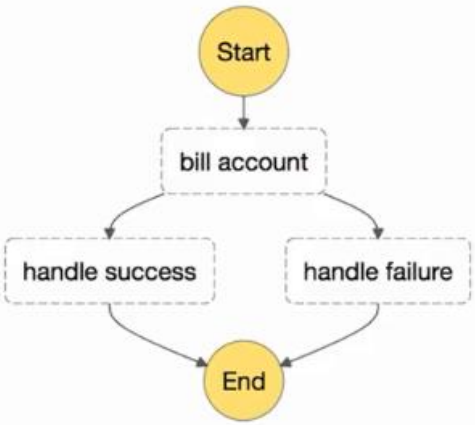
We can now kick of simultaneous step function executions or even decide to run them in serial order.

Concurrent executions offers significant speedup



Time is on the x-axis and y-axis is how many executions that we are starting over the batch process period. This improves the execution pace of the over 100,000 accounts

Enhanced observability



Explicitly represent and handle failure modes

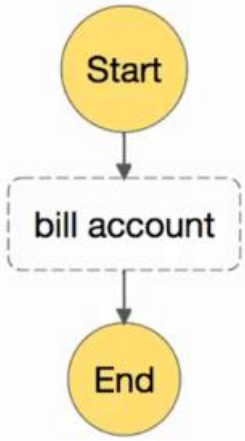
Easily observed

Execution-level failures become a huge red flag

We also saw a lot of improvement around observability like knowing if we have successfully billed a single account or all the accounts during the job run. This gives insight at the granularity of a single account or a few accounts.

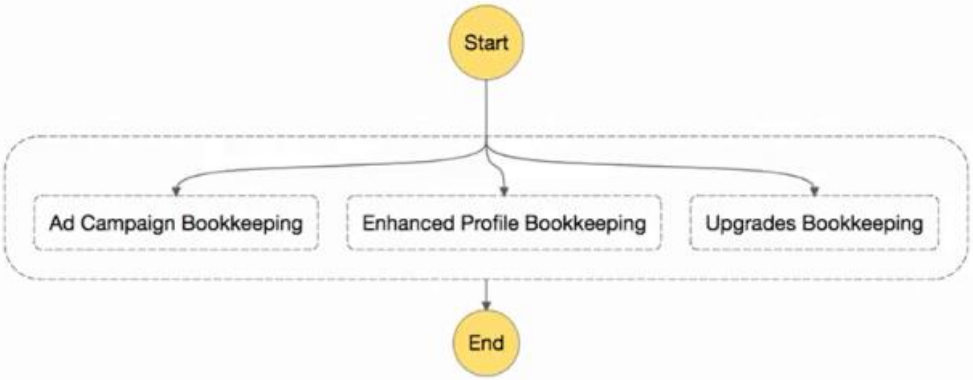
Flexible timeouts

```
"Bill Account": {
        …
        "TimeoutSeconds": 1200,
        "HeartbeatSeconds": 60,
        …
}
```



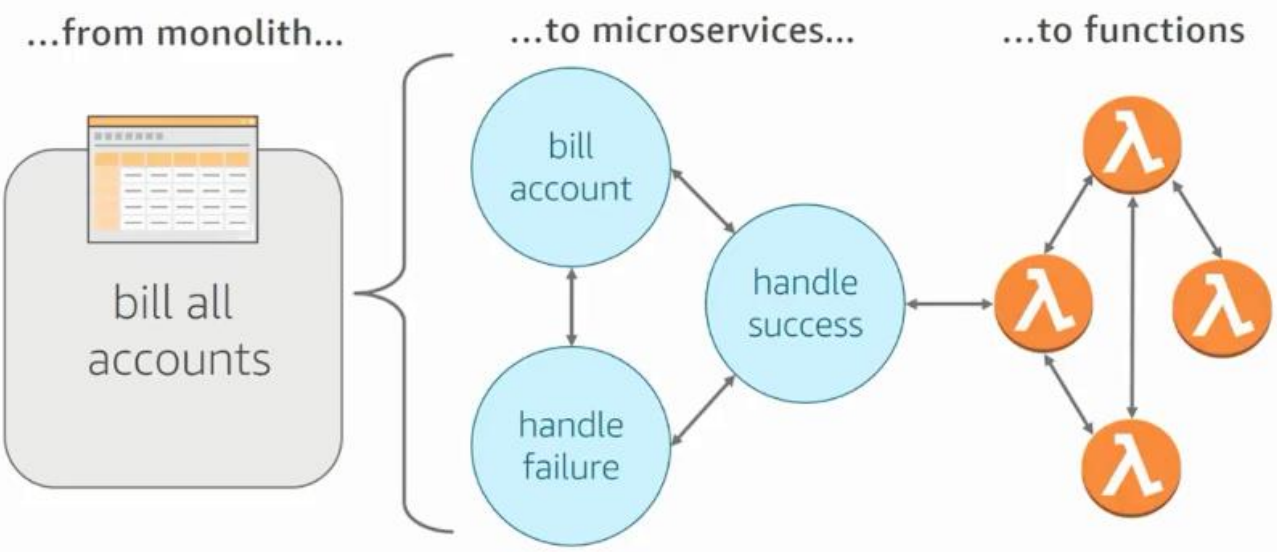
We now have a concept of timeouts in this step function approach, it allows us to put in logic for retries in case something goes wrong in a step function execution

Performance gains: internal parallelism



We can now start making the real refactoring of our previous code by breaking up the logic into separate step function flows as above

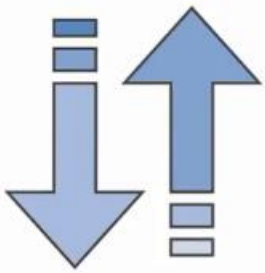
Flexible execution model evolves...



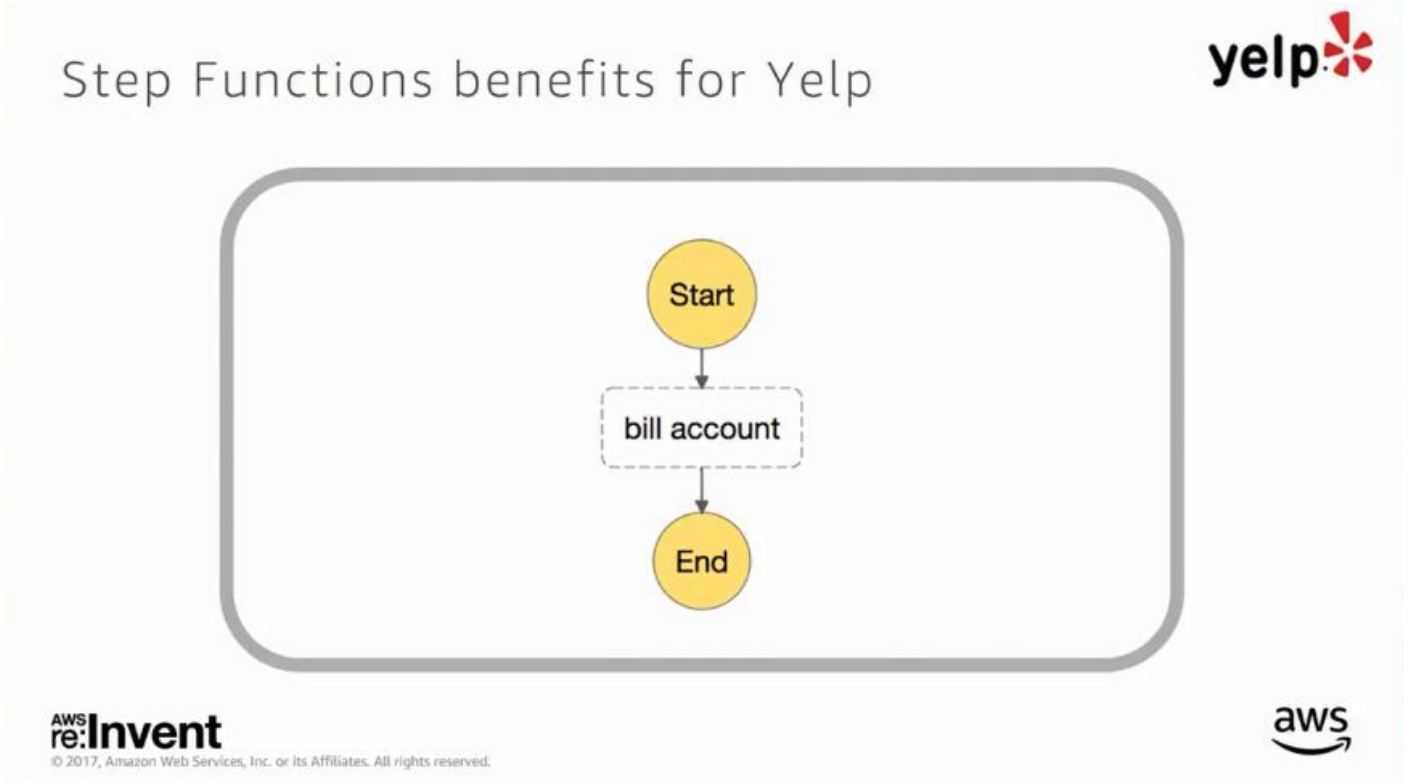
Step Functions benefits for Yelp

Enforced decoupling

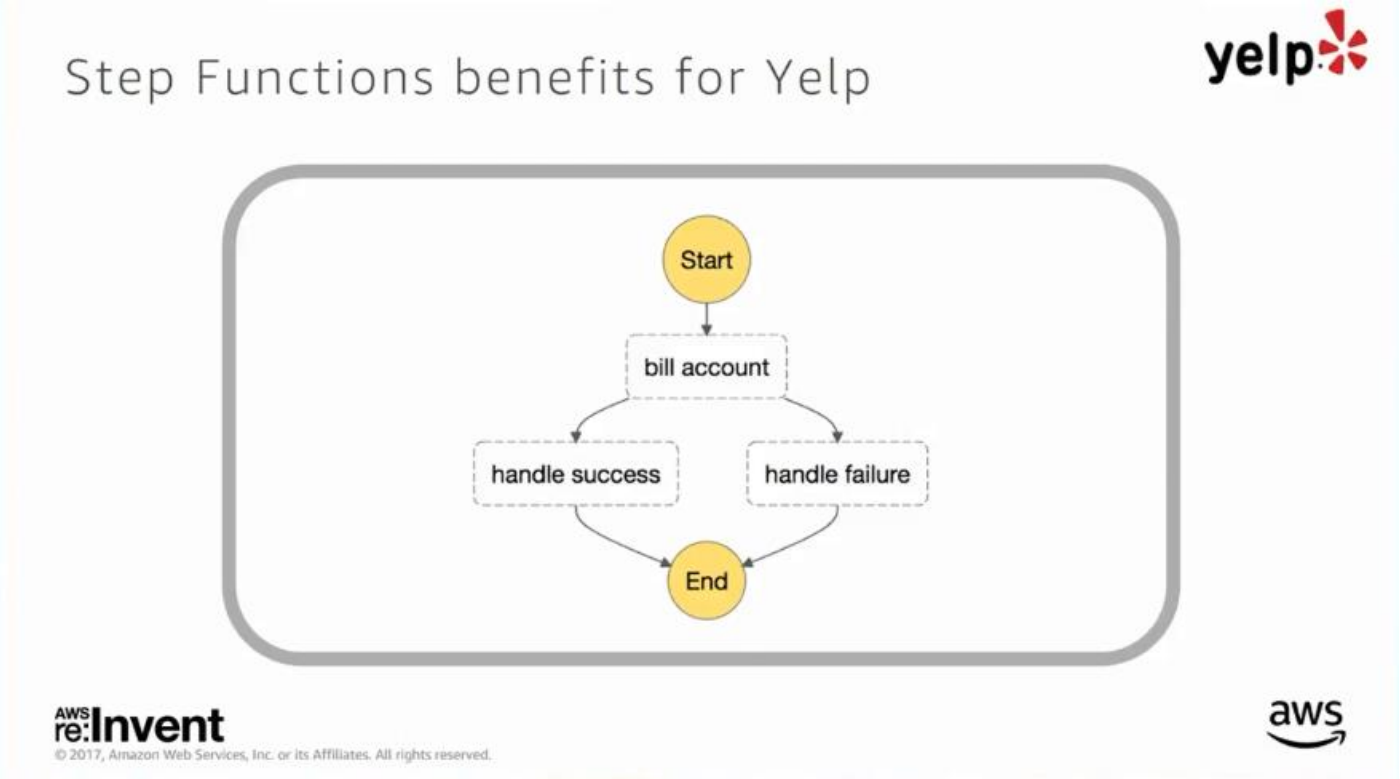
Native parallelism



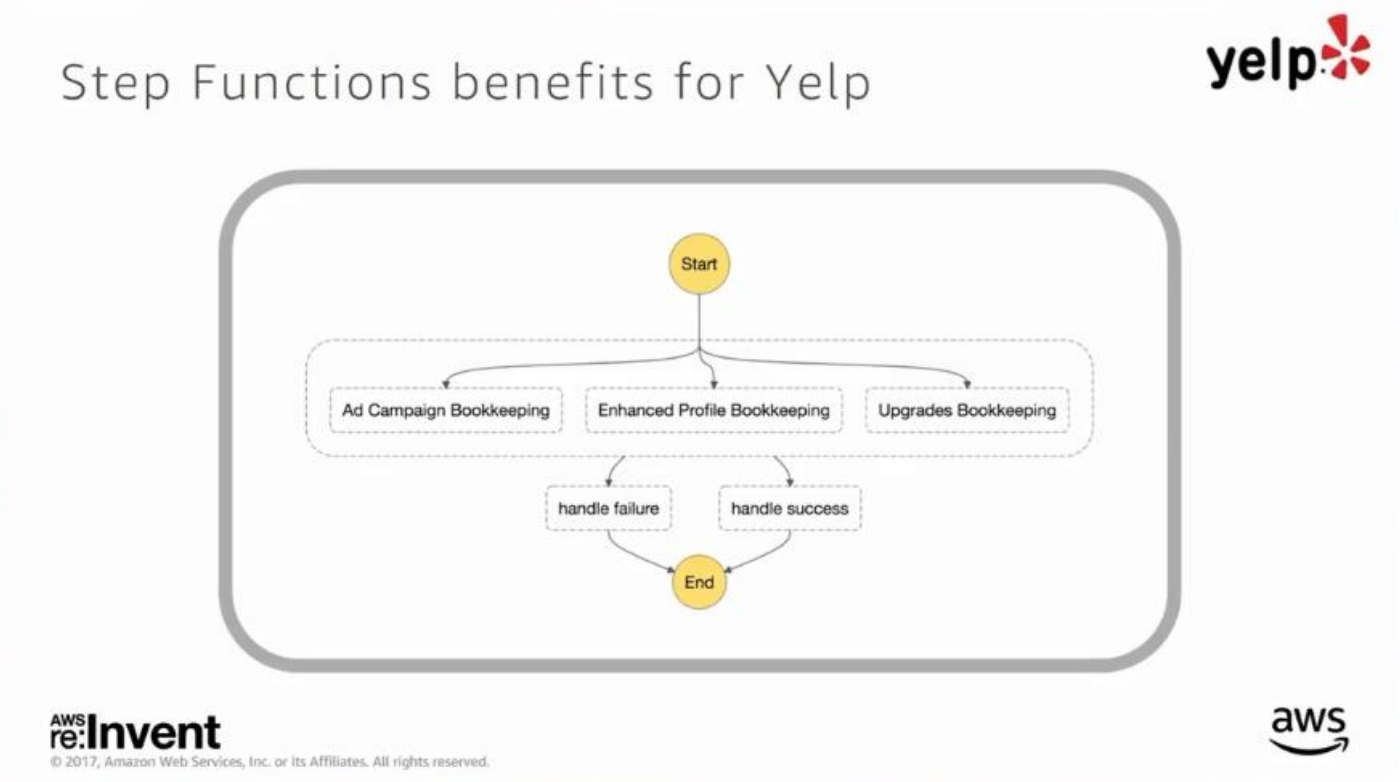
Free observability



We started off with one very simple workflow. We immediately get the concurrency and observability benefits.

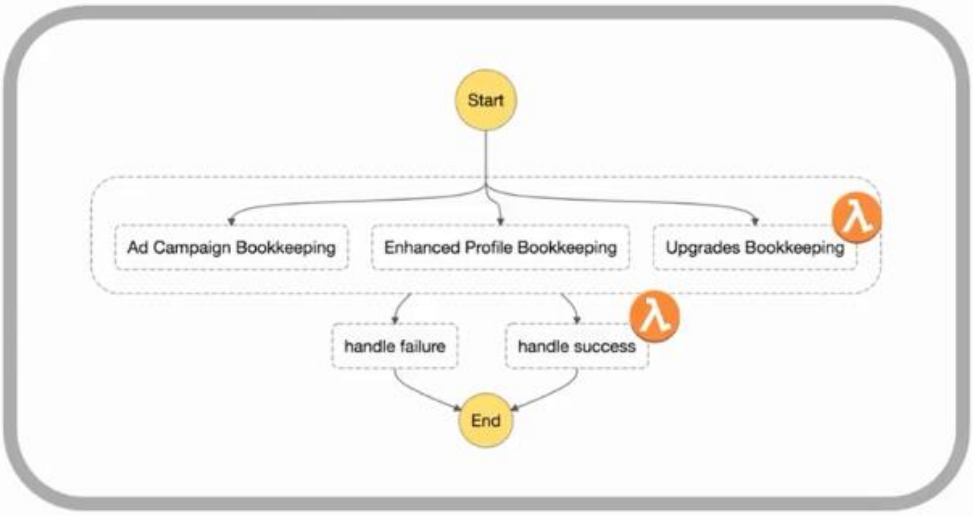


We can then start to instrument it for better observability and start to decouple that workflow further for better error handling, logging, metrics, failure modes, etc as above.



We then start to break down larger functions into smaller sub-steps and run the sub-steps in parallel for efficiency purposes.

Step Functions benefits for Yelp

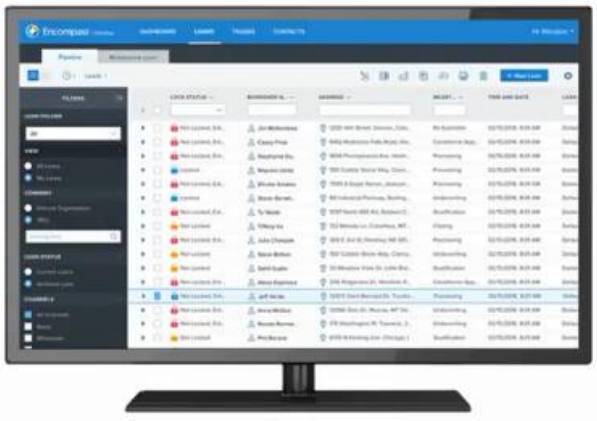


We can also ***selectively move some or all of these step function tasks over to a push model using AWS Lambda*** and completely transform our monolith into a distributed application as above

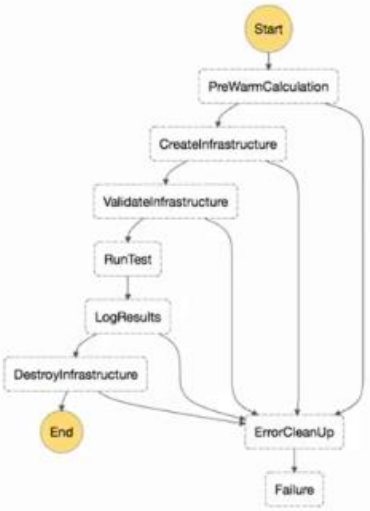
Self-service performance testing and infrastructure management

Ellie Mae is a cloud-based platform provider for the mortgage finance industry that processes nearly 40% of all US mortgage applications

They created an internal self-service portal/API to run performance tests and manage infrastructure efficiently in order to reduce costs



How EllieMae saved $10,000 a month

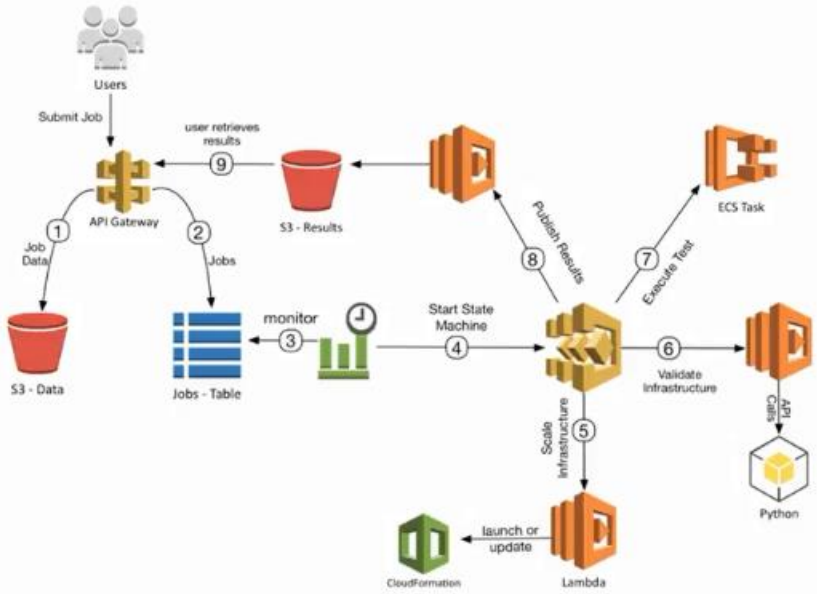


Ellie Mae created POP! (Performance Orchestration Platform) to reduce costs

- **Better developer experience**: less code, "fantastic sprint demos," and CI/CD ready for automated test results
- **Less infrastructure:** reduced costs by at least $10,000 a month
- **Lessons learned**: break down tasks until each is deterministic, Lambda tasks are easier to debug, store larger artifacts on Amazon S3 and pass object keys

Reference architecture

EllieMae



The job they want to run is put into S3, they add the job to a DynamoDB table which gets picked up on a schedule on CloudWatch events. The jobs are picked from DynamoDB by the State Machine, runs the workflow by calling CloudFormation to scale up an ECS cluster, validate the new infrastructure, runs the tests, publishes results, user gets the results back through the API Gateway, then the infrastructure is taken down.

Delivering a better customer experience while reducing costs

outsystems

Outsystems offers a low-code visual development platform for rapid application development.

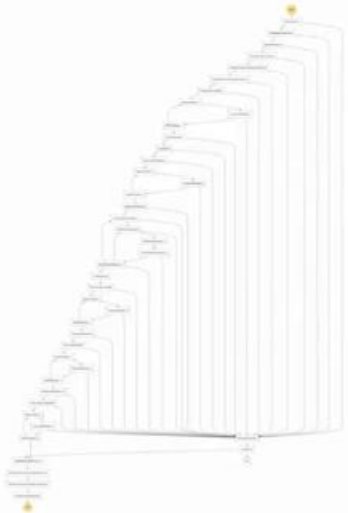
They wanted to ensure that their customers wouldn't run out of disk space—without having to over-provision storage allocated to RDS.

aws re:Invent

aws

They built a microservice that manages migrating off of 1 RDS instance to another RDS instance with data moved over to S3 and back to a new instance when a customer outgrows their original schedule RDS instance.

How outsystems reduced costs 60%



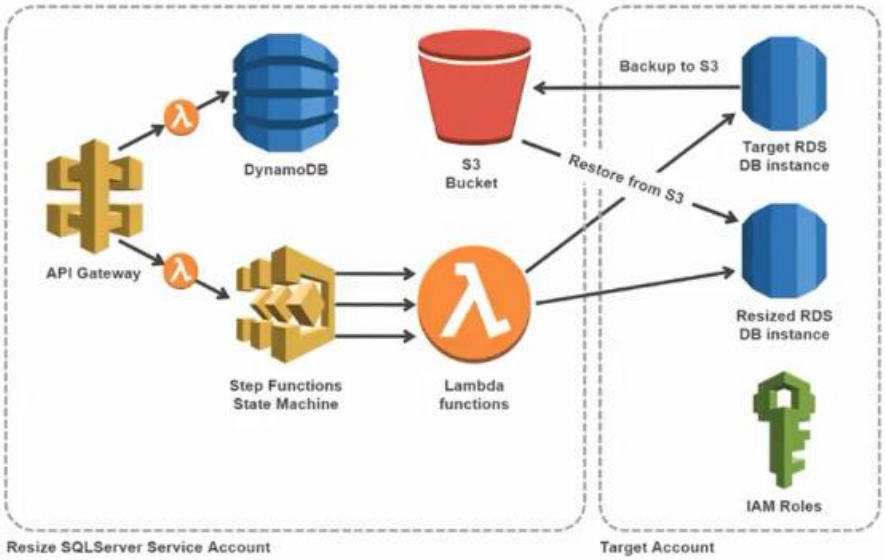
Outsystems developed an automatic resizing process which they exposed as a microservice behind Amazon API Gateway.

- **Native backup/restore:** RDS SQLServer to/from AmazonS3
- **Better customer experience:** automatic resizing ensures customers won't run out of disk space
- **Cost savings:** reduced storage costs up to 60% per database by changing default prod SQL server size from 1,024 GB to 400 GB, and then resizing storage as needed by customers

aws re:Invent

aws

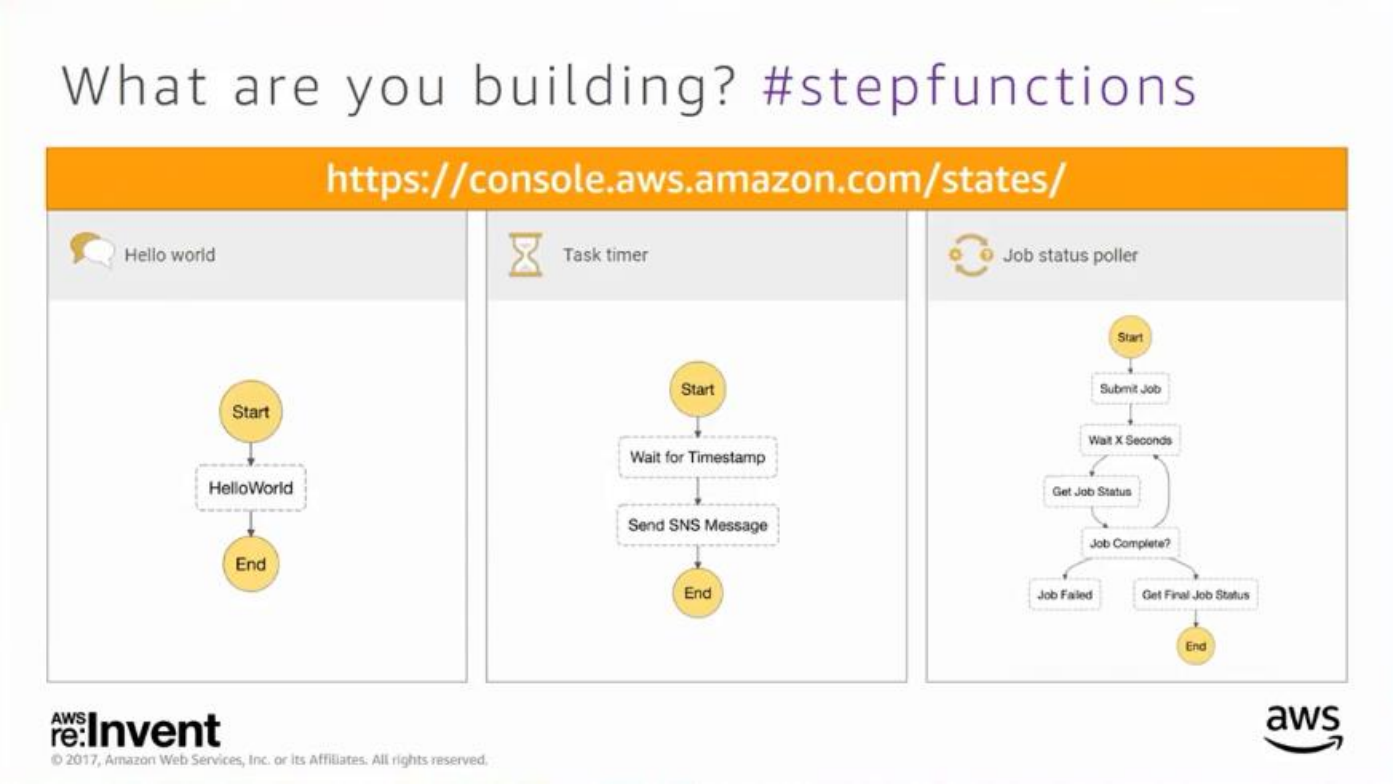
Reference architecture



Learn more

Tue Nov 28 1:45 – 2:45	**GPSCT305**: Thinking Microservices as Workflows: AWS Lambda and AWS Step Functions
Wed Nov 29 11:30 – 2:00	**SRV333:** Designing and Implementing a Serverless Media Processing Workflow Using AWS Step Functions (Workshop)
Thu Nov 30 5:30 – 6:30	**SRV306:** State Machines in the Wild! How Customers use AWS Step Functions

Meet experts at the serverless kiosk in the Expo
Resources: https://aws.amazon.com/step-functions/reinvent



You can build these 3 State Machines as a start to explore step functions using the Step Functions Developer Guide