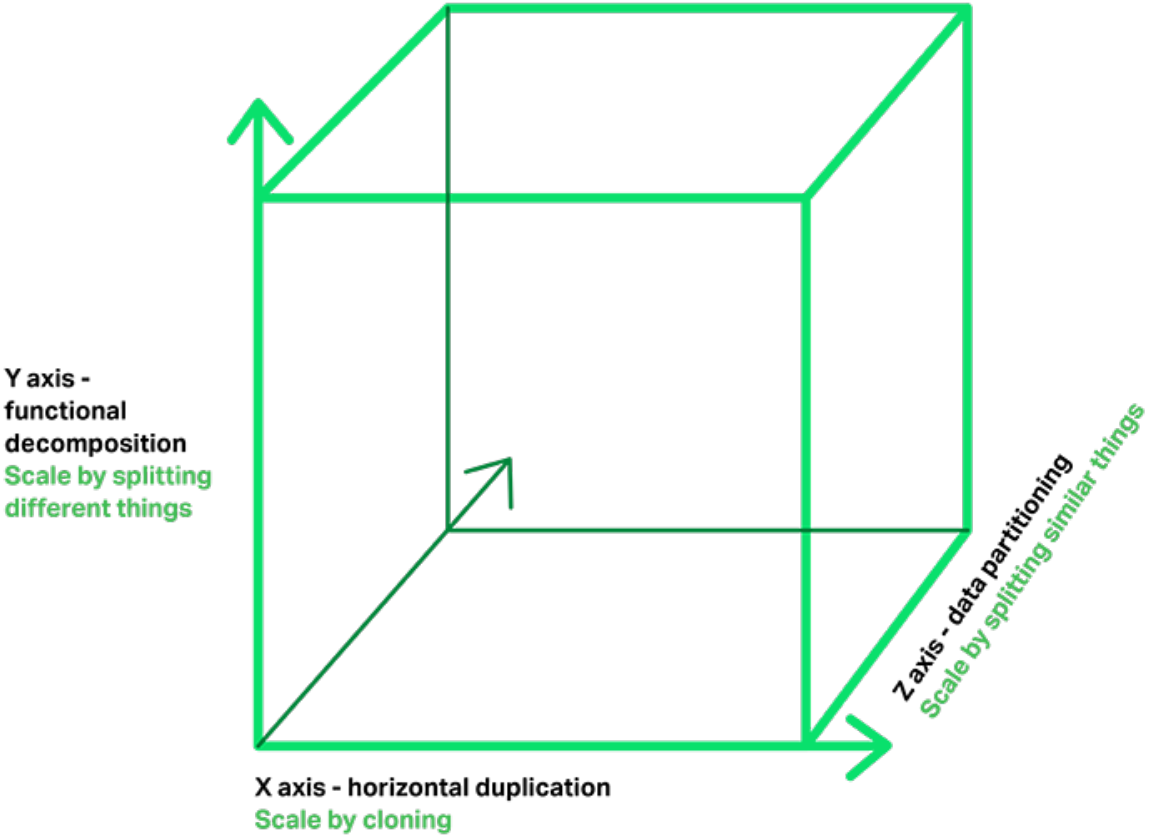
E6156 : Topics in Software Engineering Cloud and Microservice Applications

Donald F. Ferguson, dff@cs.columbia.edu

Microservice Scalability

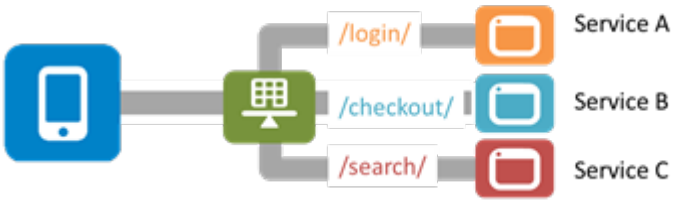
Scaling Microservices



Microservice Scaling

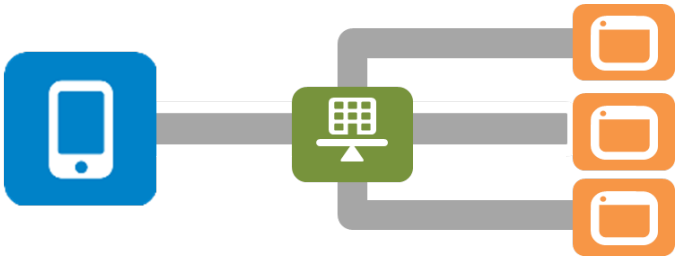
Y-AXIS SCALING

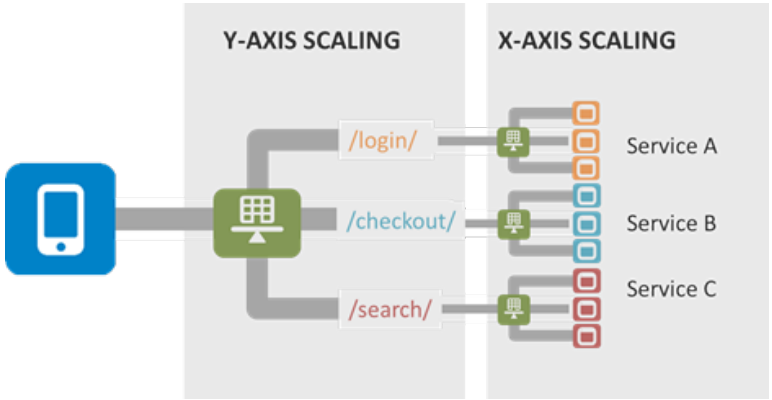
Network name: Layer 7 Load Balancing, Content switching, HTTP Message Steering



X-AXIS SCALING

Network name: Horizontal scaling, scale out





Z-AXIS SCALING

Network name: Layer 7 Load Balancing, Content switching, HTTP Message Steering

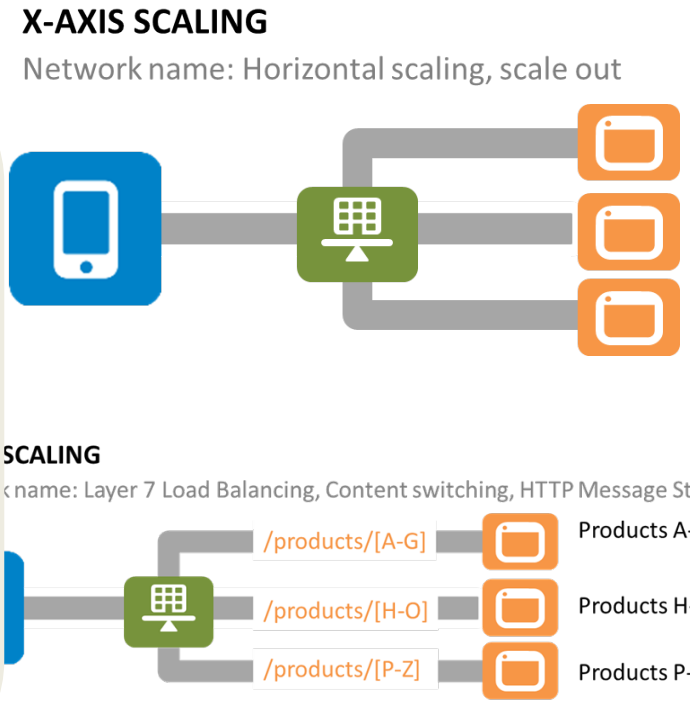
https://devcentral.f5.com/articles/the-art-of-scale-microservices-the-scale-cube-and-load-balancing

Microservice Scaling

X-Axis (Horizontal Scaling) assumes
- Each instance has equal access to data ➔
- Highly scalable database in some scenarios.

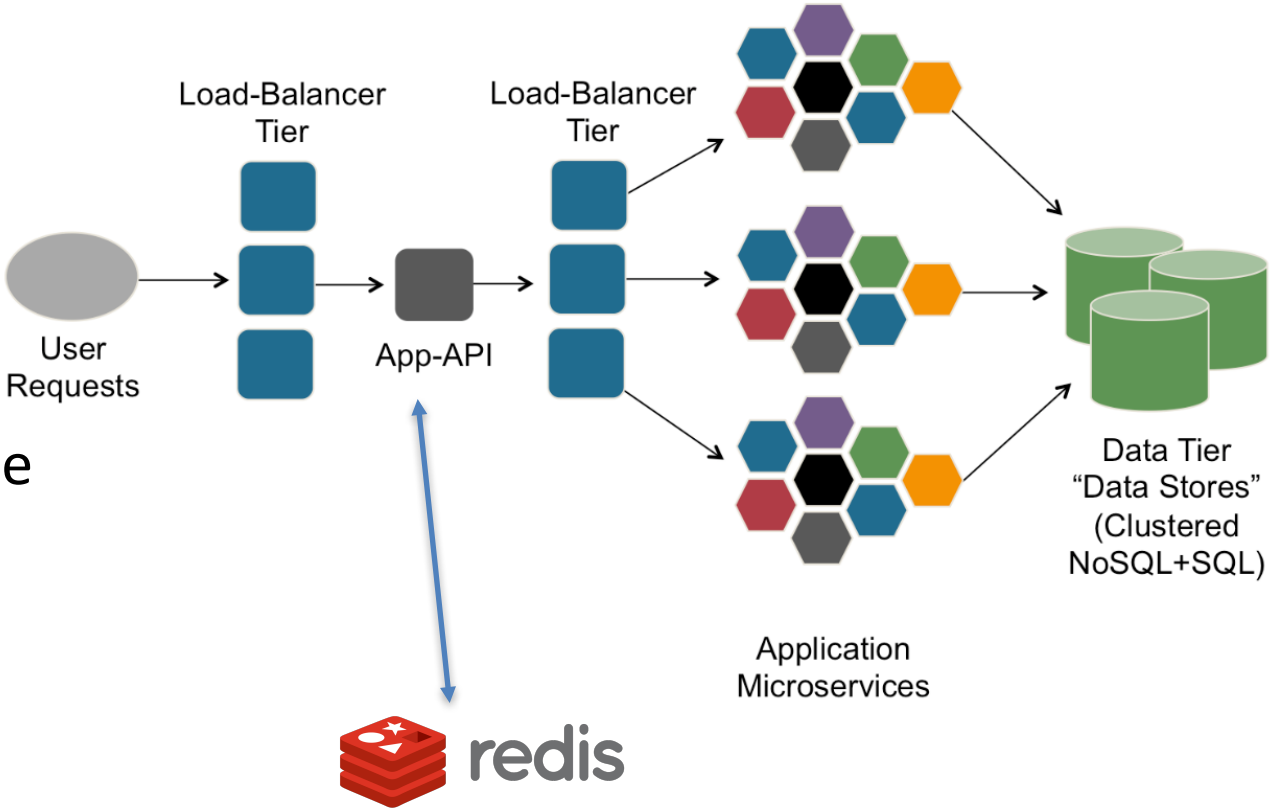
And is the assumed model for
- Lambda functions
- Elastic BeanStalk

This is hard to achieve for scenarios like
- ETag
- Idempotentcy tokens



https://devcentral.f5.com/articles/the-art-of-scale-microservices-the-scale-cube-and-load-balancing

Memcache/Redis and Scaling



Check Redis/Memcache

- API results.
- ETag
- Idempotency

Similar to express-redis-cache

www.npmjs.com/package/express-redis-cache

Just use it as a middleware in the stack of the route you want to cache.

```
var app = express();
var cache = require('express-redis-cache')();

// replace
app.get('/',
  function (req, res)  { ... });

// by
app.get('/',
  cache.route(),
  function (req, res)  { ... });
```

This will check if there is a cache entry for this route. If not. it will cache it and serve the cache next time route is called.

You can configure caching middleware to:

- Bind to Redis for basic testing.
- ElastiCache when deployed in AWS Cloud

Twelve Factor Applications

The 12 Factors

I. Codebase
One codebase tracked in revision control, many deploys

II. Dependencies
Explicitly declare and isolate dependencies

III. Config
Store config in the environment

IV. Backing Services
Treat backing services as attached resources

V. Build, release, run
Strictly separate build and run stages

VI. Processes
Execute the app as one or more stateless processes

VII. Port binding
Export services via port binding

VIII. Concurrency
Scale out via the process model

IX. Disposability
Maximize robustness with fast startup and graceful shutdown

X. Dev/prod parity
Keep development, staging, and production as similar as possible

XI. LogsTreat logs as event streams

XII. Admin processes
Run admin/management tasks as one-off processes

Twelve Factor Application

1. Codebase

A twelve-factor app is always tracked in a version control system.

A codebase is any single repo (in a centralized revision control system like Subversion), or any set of repos who share a root commit (in a decentralized revision control system like Git).

If there are multiple codebases, it's not an app – it's a distributed system.

Multiple apps sharing the same code is a violation of twelve-factor. The solution here is to factor shared code into libraries which can be included through the dependency manager.

Twelve Factor Application

2. Dependencies

A twelve-factor app **never** relies on implicit existence of system-wide packages.

An app declares all dependencies, completely and exactly, via a dependency declaration manifest.

An app uses a **dependency isolation** tool.

Apps do not rely on the implicit existence of any system tools. Examples include shelling out to curl.

If an app needs to shell out to a system tool, that tool should be vendored into the app

Twelve Factor Application

3. Configuration

An app's config is **everything** that is likely to vary between deploys

An app stores its config in **environment variables**

Env vars are easy to change between deploys without changing any code.

Env vars are granular controls, each fully **orthogonal** to other env vars

Twelve Factor Application

4. Backing services

A backing service is any service the app consumes over the network as part of its normal operation (DB, broker, object store, etc)

The code for a twelve-factor app makes no distinction between local and third party services, both are attached resources, accessed via a URL or other locator/credentials stored in the config. The services should be **swappable** without code changes

Each distinct backing service is a resource.

Resources can be attached and detached to deploys at will.

Twelve Factor Application

5. Build, release, run

A codebase is transformed into a (non-development) deploy:

- The build stage converts a code repo into an executable bundle known as a build. Using a version of the code at a commit specified by the deployment process, the build stage fetches vendors dependencies and compiles binaries and assets.
- The release stage takes the build and combines it with the deploy's current config. The resulting release contains both the build and the config and is ready for immediate execution.
- The run stage (also known as "runtime") runs the app in the execution environment, by launching some set of the app's processes against a selected release.

Twelve Factor Application

6. Processes

The app is executed in the execution environment as one or more processes.

Twelve-factor processes are **stateless** and share-nothing. Any data that needs to persist must be stored in a stateful backing service, typically a database.

The twelve-factor app **never** assumes that anything cached in memory or on disk will be available on a future request or job

An app prefers to do this compiling during the **build** stage, rather than at runtime

Sticky sessions are a **violation** of twelve-factor

Twelve Factor Application

7. Port binding

An app is completely **self-contained**

A web app exports HTTP as a service by binding to a port, and listening to requests coming in on that port

The **contract** with the execution environment is binding to a port to serve requests

Twelve Factor Application

8. Concurrency

Processes are a first class citizen

The share-nothing, horizontally partitionable nature of twelve-factor app processes means that adding more concurrency is a simple and reliable operation

App processes should **never daemonize or write PID files**. Instead, rely on the operating system's process manager to manage output streams, respond to crashed processes, and handle user-initiated restarts and shutdowns

Twelve Factor Application

9. Disposability

App's processes are disposable, meaning they can be started or stopped at a moment's notice

Processes should strive to minimize startup time.

Processes shut down gracefully when they receive a SIGTERM.

Processes should also be robust against sudden death

An app is architected to handle unexpected, non-graceful terminations.

Twelve Factor Application

10. Dev/prod parity

An app is designed for continuous deployment by keeping the gap between development and production small

Time between deploys take only hours

Code authors and code deployers can be the same people

Dev vs production environments should be as similar as possible

Developers resists the urge to use different backing services between development and production

Twelve Factor Application

11. Logs

An app never concerns itself with routing or storage of its output stream.

It should **not** attempt to write to or manage logfiles. Instead, each running process writes its event stream, unbuffered, to stdout.

During local development, the developer will view this stream in the foreground of their terminal to observe the app's behavior.

Production deploys, each process' stream will be captured by the execution environment and routed to one or more final destinations for viewing and long-term archival. These archival destinations are not visible to or configurable by the app, and instead are managed by the **execution** environment

Twelve Factor Application

12. Admin processes

One-off admin processes should be run in an identical environment as the regular long-running processes of the app.

They should run against a release, using the same codebase and config as any process run against that release.

Admin code must ship with application code to avoid synchronization issues.

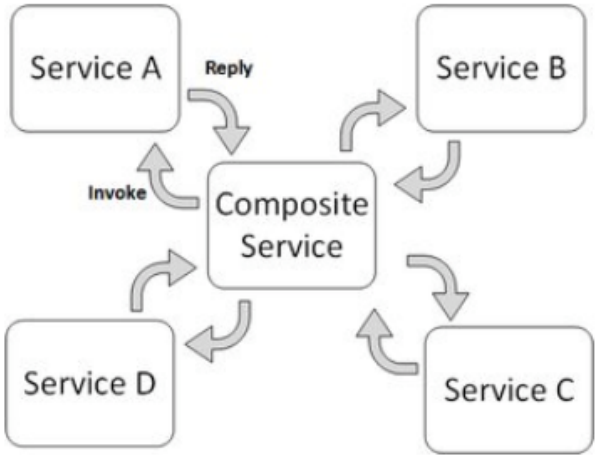
Orchestration Pattern

Service Orchestration

Service orchestration

Service orchestration represents a single centralized executable business process (the orchestrator) that coordinates the interaction among different services. The orchestrator is responsible for invoking and combining the services.

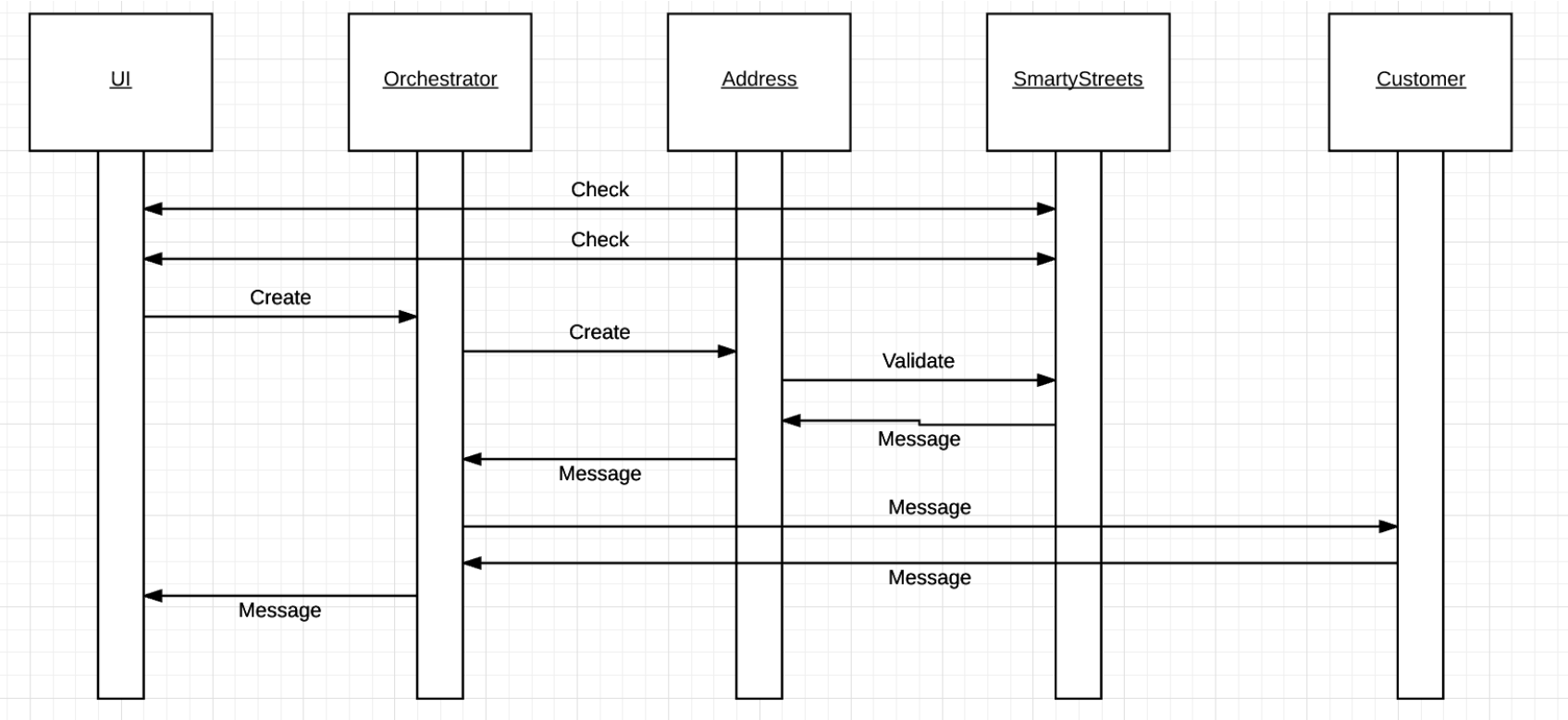
The relationship between all the participating services are described by a single endpoint (i.e., the composite service). The orchestration includes the management of transactions between individual services. Orchestration employs a centralized approach for service composition.



This can get tricky in code:

- "Slow" if all calls are synchronous.
- Hold threads/processes during waits.
- Going asynchronous and event oriented introduces it's own problems:
  - "Callback Hell"
  - Some things happen serially while others can happen in parallel.

Simple Example We Saw



Orchestration Pattern – Deeper Example

Example Use Case of "Submit a Shopping Cart"

Let's look at a typical "submit shopping cart" business process which has five main steps:

1. Validate shipping address
2. Reserve payment
3. Allocate inventory
4. Place order
5. Notify customer

https://dzone.com/articles/event-driven-orchestration-an-effective-microservi

The business process will execute the five steps in sequence. In case of exception during the workflow, rollback should be executed to maintain the transaction's ACID (atomic, consistent, isolated, and durable).

How can this "submit shopping cart" business process be implemented as a group of interactions between the microservices?

Orchestration Pattern – Event Driven

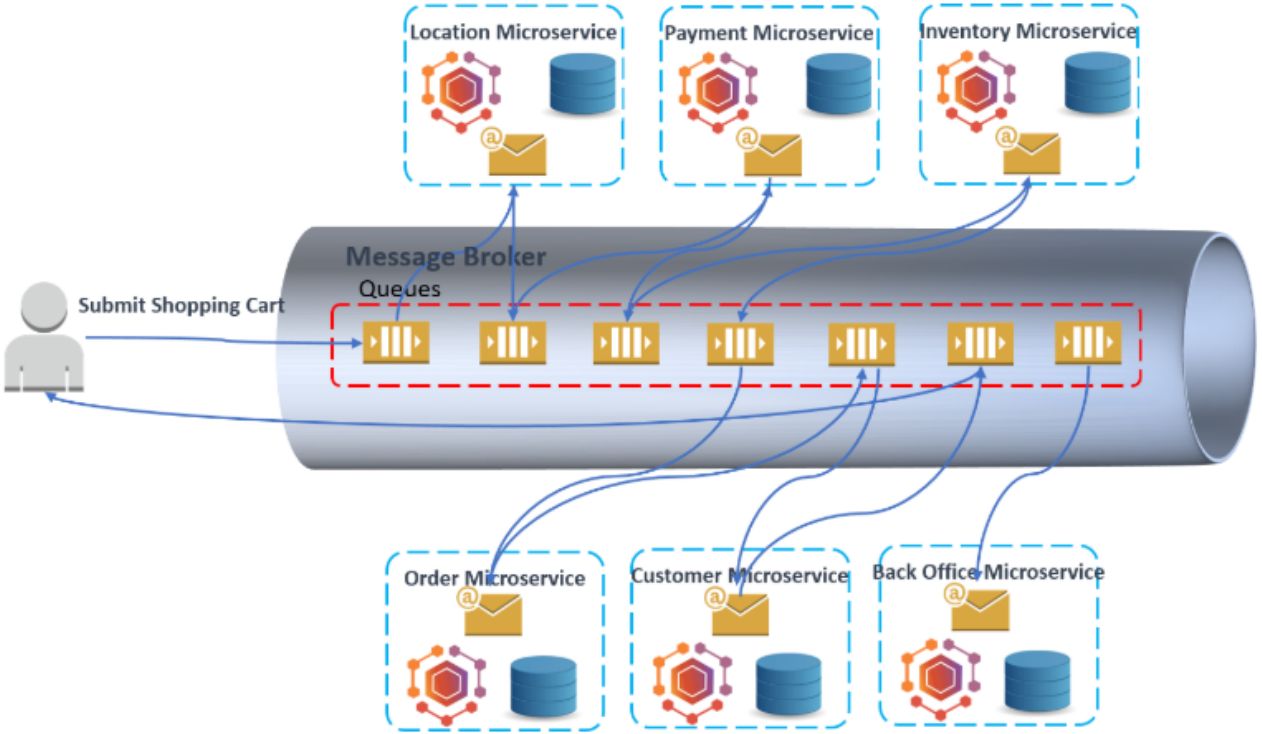


Figure 1. Service Choreography using Message Broker.

SQS or SNS

Orchestration Pattern – Event Driven

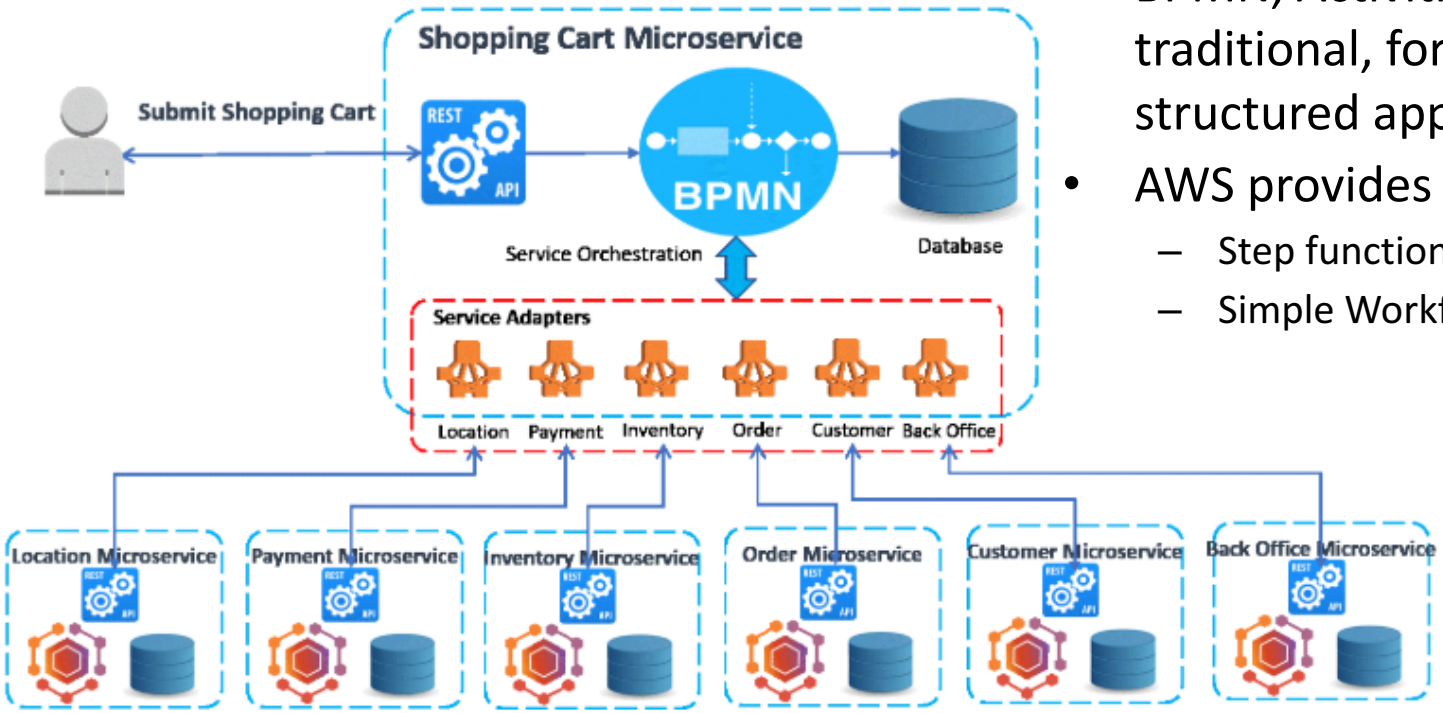


Figure 2. Service Orchestration using BPMN and REST.

- BPMN, Activiti, … are more traditional, formal and structured approaches.
- AWS provides less formal:
  - Step functions.
  - Simple Workflow Service

AWS Step Functions

Productivity:

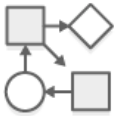
Build Applications Quickly

AWS Step Functions includes a visual console and blueprints for commonly-used workflows that make it easy to coordinate the components of distributed applications into parallel and/or sequential steps. You can build applications in a matter of minutes, and then visualize and track the execution of each step to help ensure the application is operating as intended.

Resilience:

Scale and Recover Reliably

AWS Step Functions automatically triggers each step so your application executes in order and as expected. It can handle millions of steps simultaneously to help ensure your application is available as demand increases. Step Functions tracks the state of each step and handles errors with built-in retry and fallback, whether the step takes seconds or months to complete.



Agility:

Evolve Applications Easily

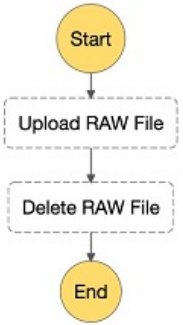
AWS Step Functions makes it easy to change workflows and edit the sequence of steps without revising the entire application. You can re-use components and steps without even changing their code to experiment and innovate faster. Your workflow can support thousands of individual components and steps, so you can freely build increasingly complex applications.

Concepts

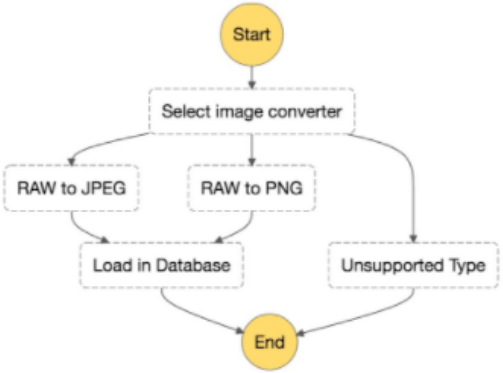
1. Define your Application Visually as a Series of Steps

Define your application workflow as a series of steps. The visual console automatically graphs each step in the order of execution, making it easy to design complex workflows for multi-step applications. The following diagrams provide examples of the flow of steps – including sequential, branching and parallel steps, for a photo sharing application.

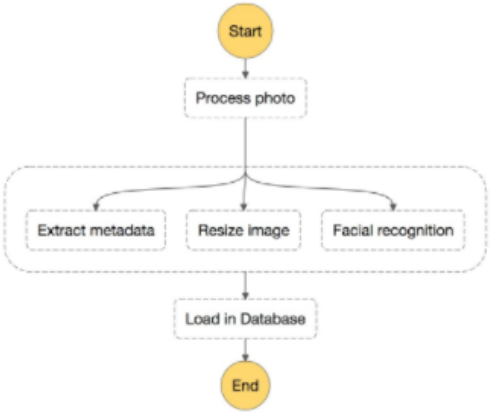
Sequential Steps



Branching Steps (Choice of Path)



Parallel Steps



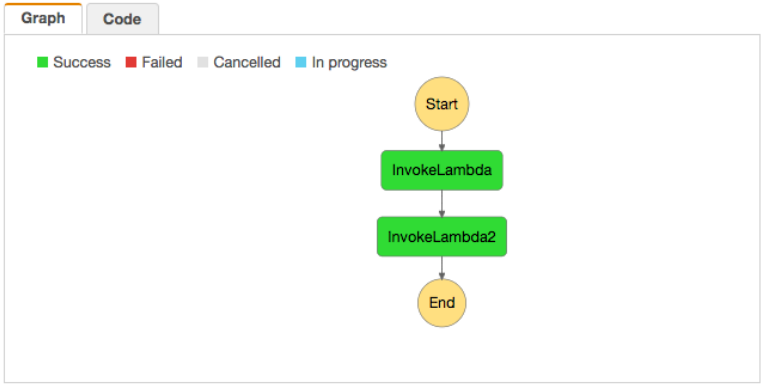
Execute and Monitor

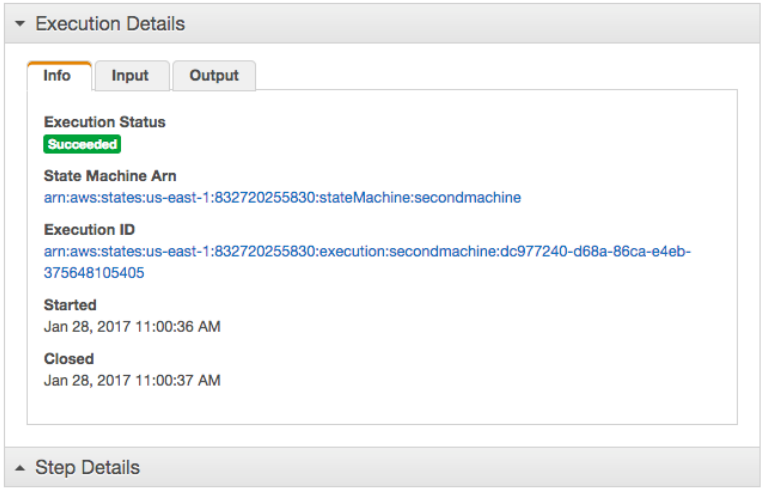
2. Verify Everything is Operating as Intended

Start an execution to visualize and verify the steps of your application are operating as intended. The console highlights the real-time status of each step and provides a detailed history of every execution.

AWS Step Functions

Simple Example





ID	Type	Timestamp
1	ExecutionStarted	Jan 28, 2017 11:00:36 AM
2	TaskStateEntered	Jan 28, 2017 11:00:36 AM
3	LambdaFunctionScheduled	Jan 28, 2017 11:00:36 AM
4	LambdaFunctionStarted	Jan 28, 2017 11:00:36 AM
5	LambdaFunctionSucceeded	Jan 28, 2017 11:00:36 AM
6	TaskStateExited	Jan 28, 2017 11:00:36 AM
7	TaskStateEntered	Jan 28, 2017 11:00:36 AM

Simple Example

Graph | Code

```
{
  "Comment": "A Hello World example of the Amazon States Language using an AWS Lambda Function",
  "StartAt": "InvokeLambda",
  "States": {
    "InvokeLambda": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:us-east-1:832720255830:function:helloworld",
      "Next": "InvokeLambda2"
    },
    "InvokeLambda2": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:us-east-1:832720255830:function:columbiaecho",
      "End": true
    }
  }
}
```

- AWS Step Functions Concepts
  - Amazon States Language
  - States
  - Tasks
  - Transitions
  - State Machine Data
  - Executions
  - Error Handling
  - Creating IAM Roles for Use with AWS Step Functions

```
{
  "Comment": "An Amazon States Language example using a Choice state.",
  "StartAt": "FirstState",
  "States": {
    "FirstState": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:us-east-1:123456789012:function:FUNCTION_NAME",
      "Next": "ChoiceState"
    },
    "ChoiceState": {
      "Type" : "Choice",
      "Choices": [
        {
          "Variable": "$.foo",
          "NumericEquals": 1,
          "Next": "FirstMatchState"
        },
        {
          "Variable": "$.foo",
          "NumericEquals": 2,
          "Next": "SecondMatchState"
        }
      ],
      "Default": "DefaultState"
    },

    "FirstMatchState": {
      "Type" : "Task",
      "Resource": "arn:aws:lambda:us-east-1:123456789012:function:OnFirstMatch",
      "Next": "NextState"
    },

    "SecondMatchState": {
      "Type" : "Task",
      "Resource": "arn:aws:lambda:us-east-1:123456789012:function:OnSecondMatch",
      "Next": "NextState"
    },
```

Tasks and Activities

Tasks

All work in your state machine is done by *tasks*. A task can be:

- An Activity, which can consist of any code in any language. Activities can be hosted on EC2, ECS, mobile devices—basically anywhere. Activities must poll AWS Step Functions using the GetActivityTask and SendTask* API calls. (Ultimately, an activity can even be a human task—a task that waits for a human to perform some action and then continues.)
- A Lambda function, which is a completely cloud-based task that runs on the Lambda service. Lambda functions can be written in JavaScript (which you can write using the AWS Management Console or upload to Lambda), or in Java or Python (uploaded to Lambda).

Tasks are represented in Amazon States Language by setting a state's type to `Task` and providing it with the ARN of the created activity or Lambda function. For details about how to specify different task types, see Task in the Amazon States Language Overview.