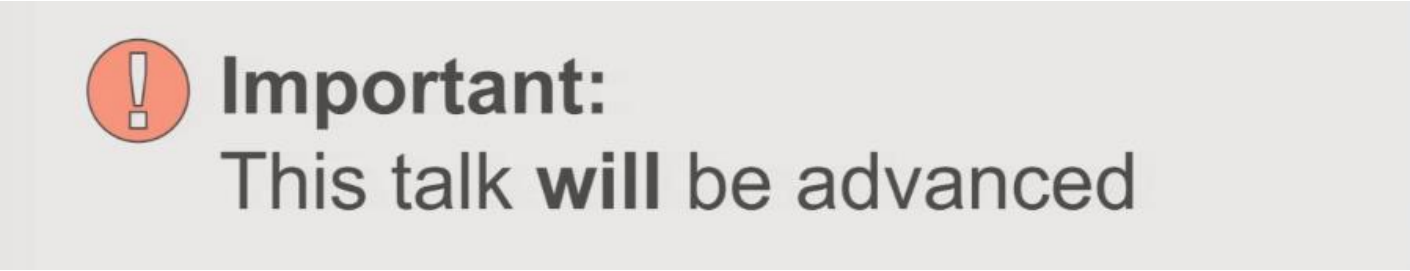
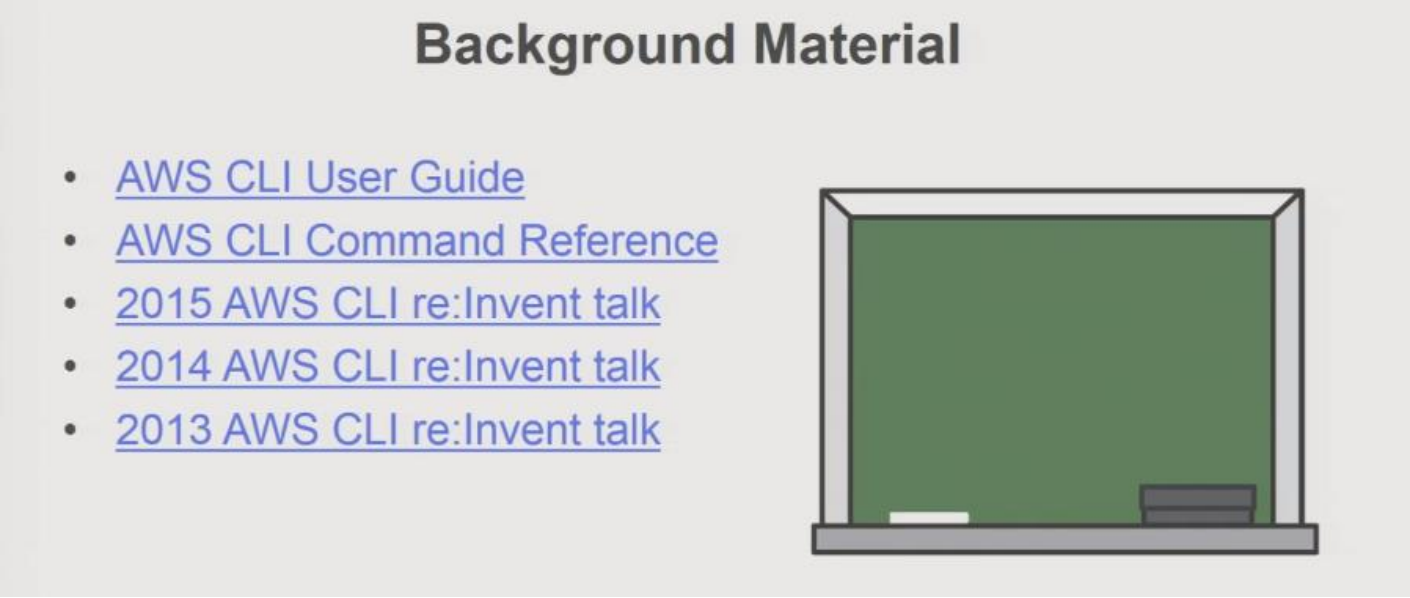
Understanding the internals of the AWS CLI will make you a more effective user. This talk provides a deep dive into the architecture, debugging techniques, advanced usage patterns, and some of its more advanced features. We recommend this talk for people who are already familiar with the AWS CLI because the topics will be advanced. By the end of the talk, audience members will have a deeper understanding of the AWS CLI, allowing them to tackle current or future applications from a different perspective.





AWS Command Line Interface

Unified tool to manage AWS services

The Effective AWS CLI User Tenets

The effective AWS CLI user:

- Uses an iterative workflow
- Troubleshoots well
- Is resourceful with tooling
- Understands performance implications

A Sample Application

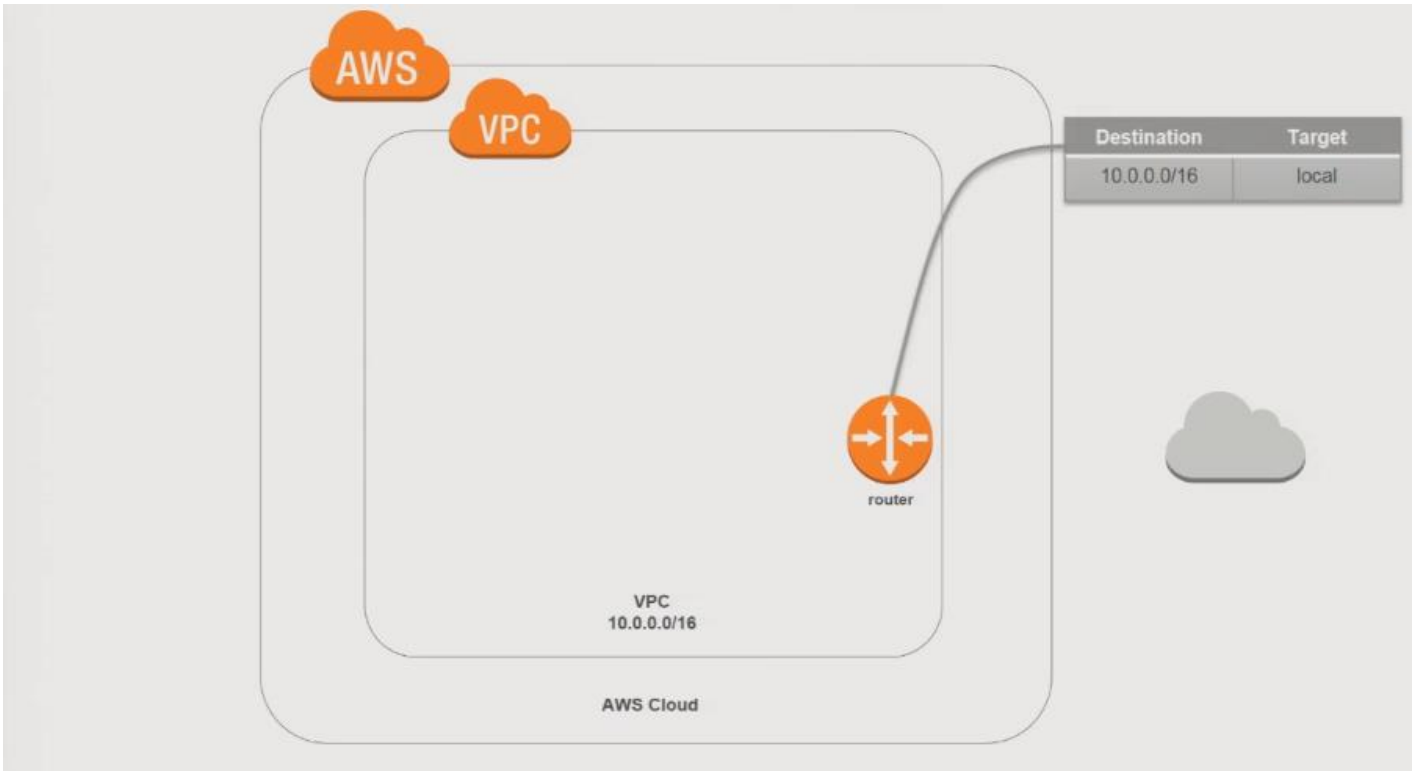
Amazon VPC

Amazon EC2

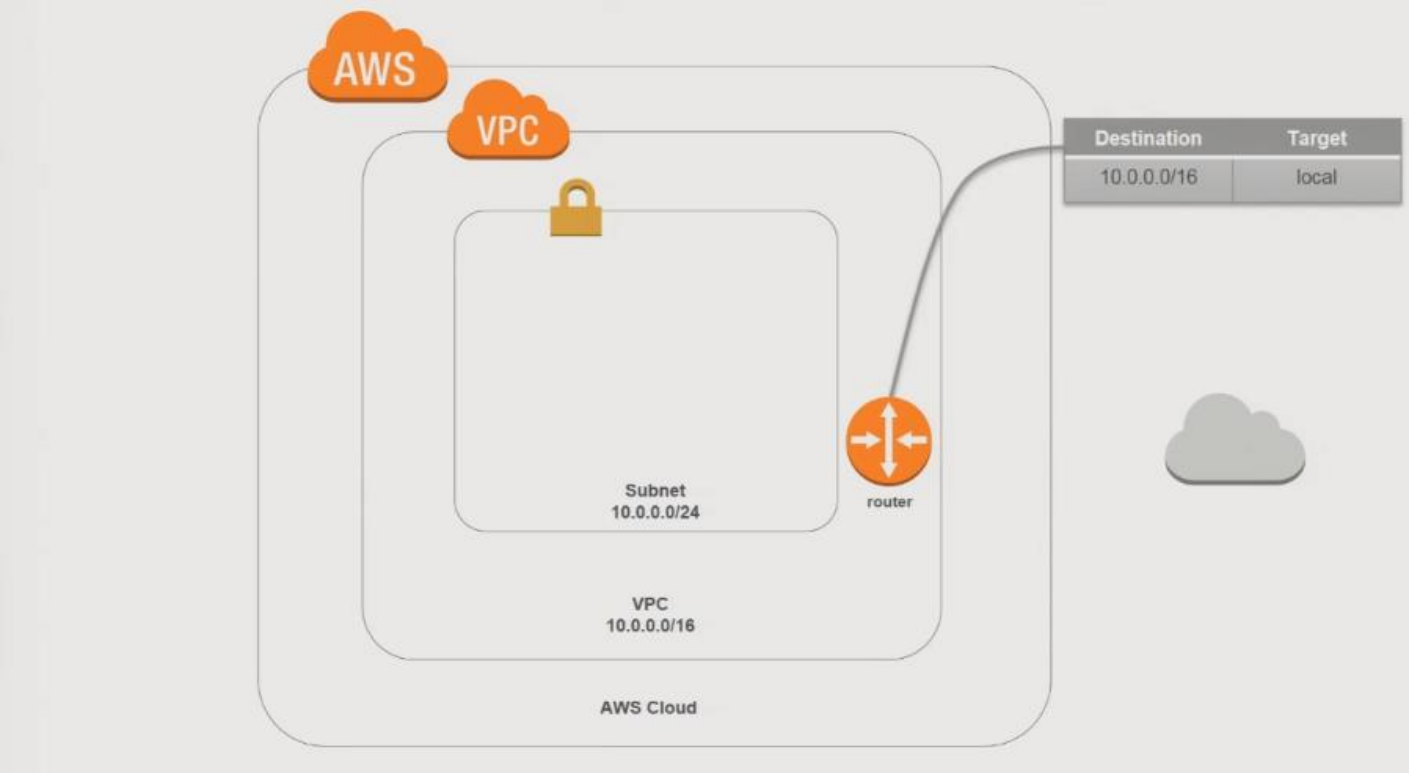
IAM

Amazon S3

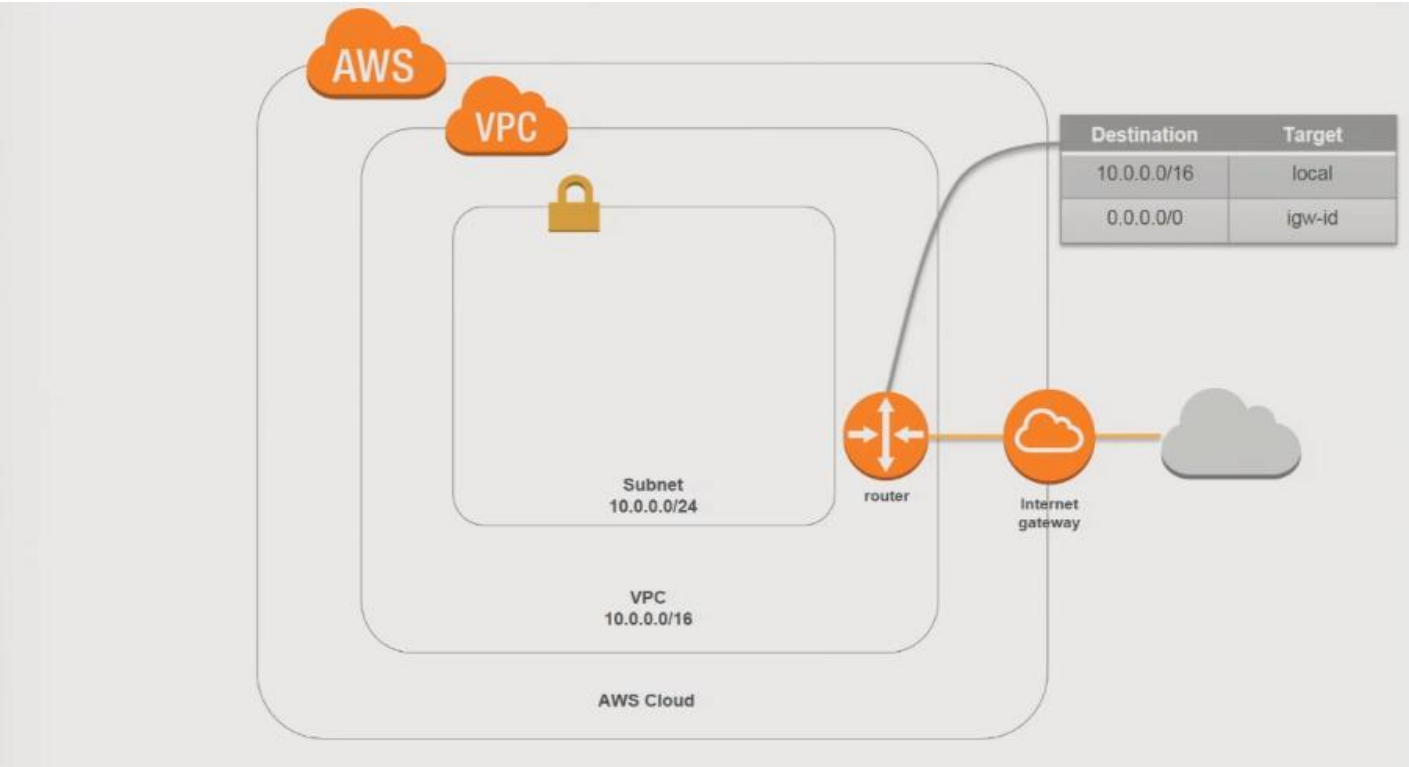
For the sample application we will be using the Amazon VPC service to create a virtual private cloud, the EC2 service to launch EC2 instances into the VPC, we will use IAM to create role for the EC2 instances that will allow the EC2 instances to get temporary credentials to upload files to S3.



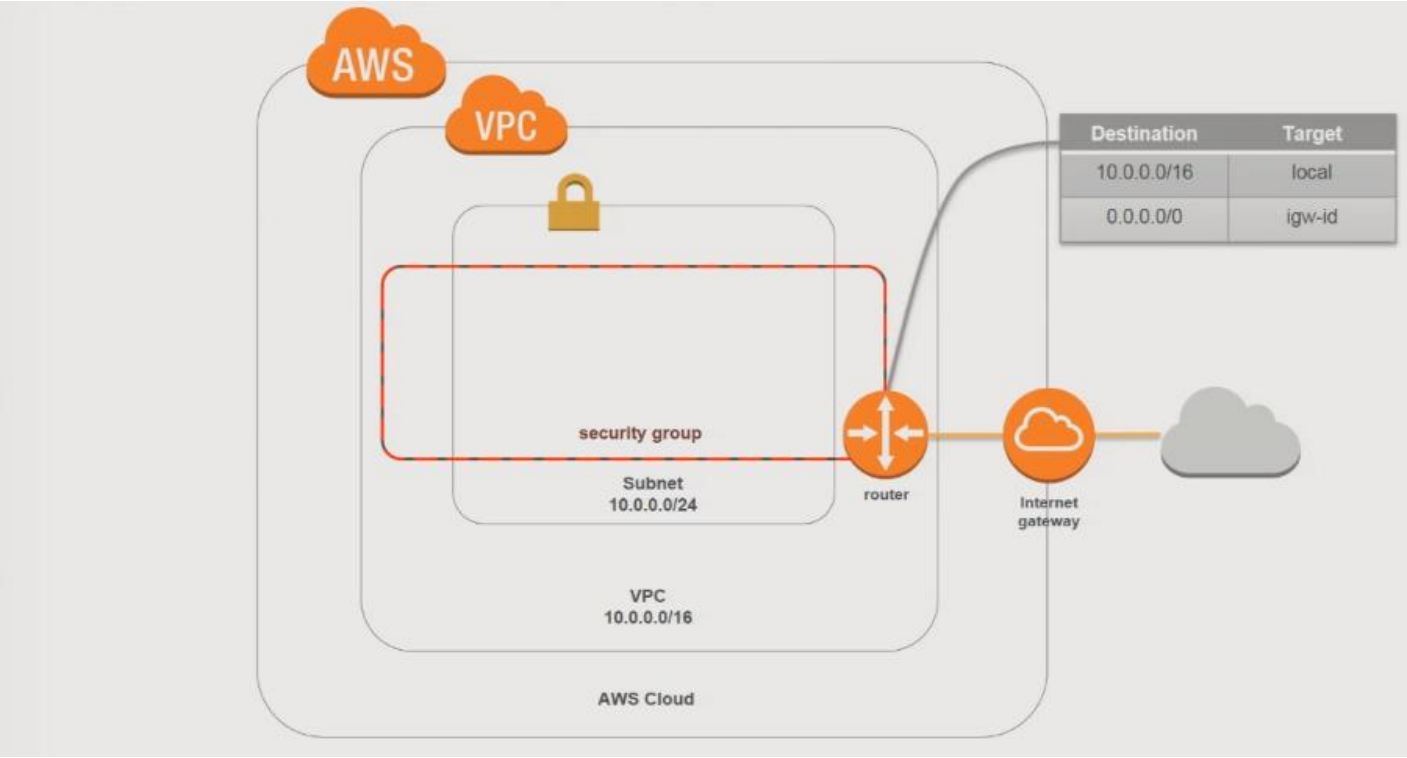
We will start by creating a VPC in the AWS cloud environment, each VPC comes with a built-in route table



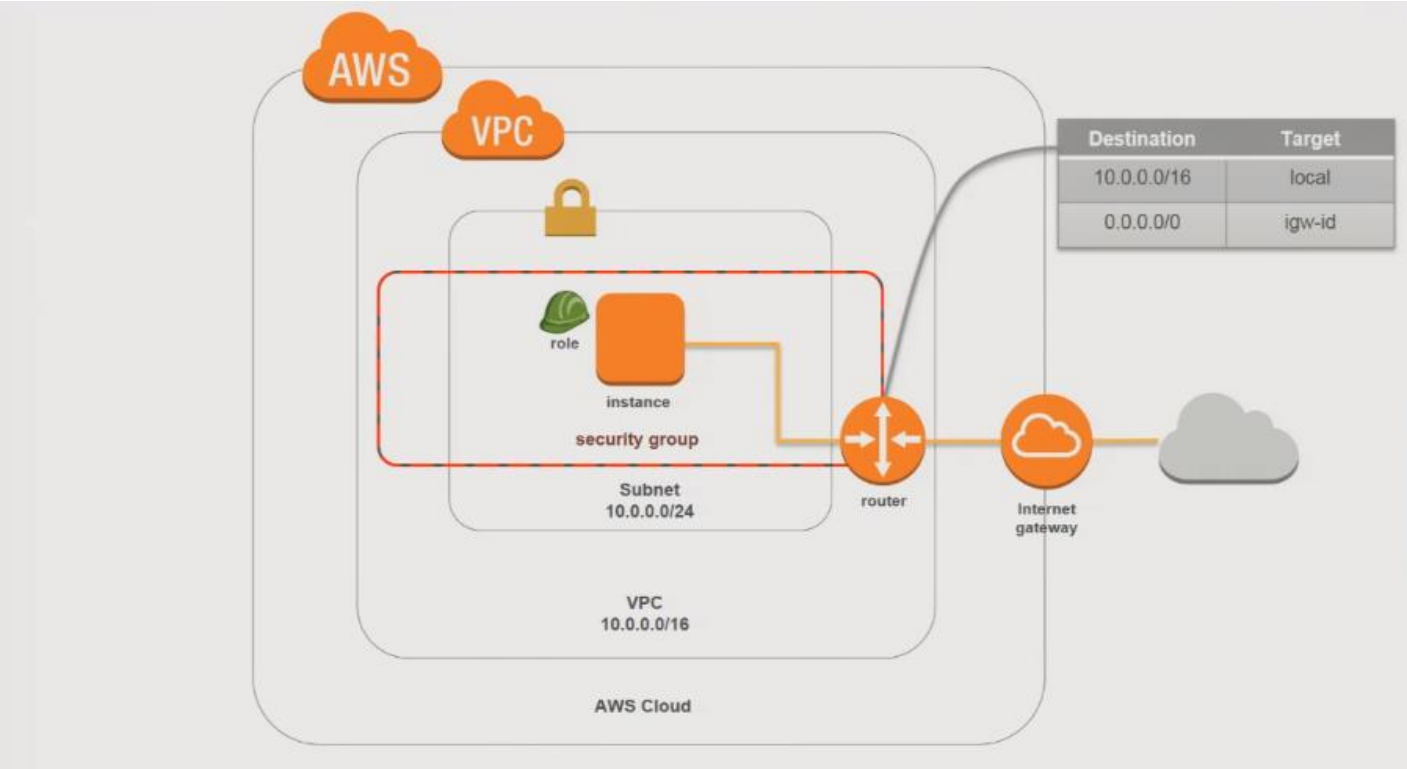
We will then create a subnet inside the VPC



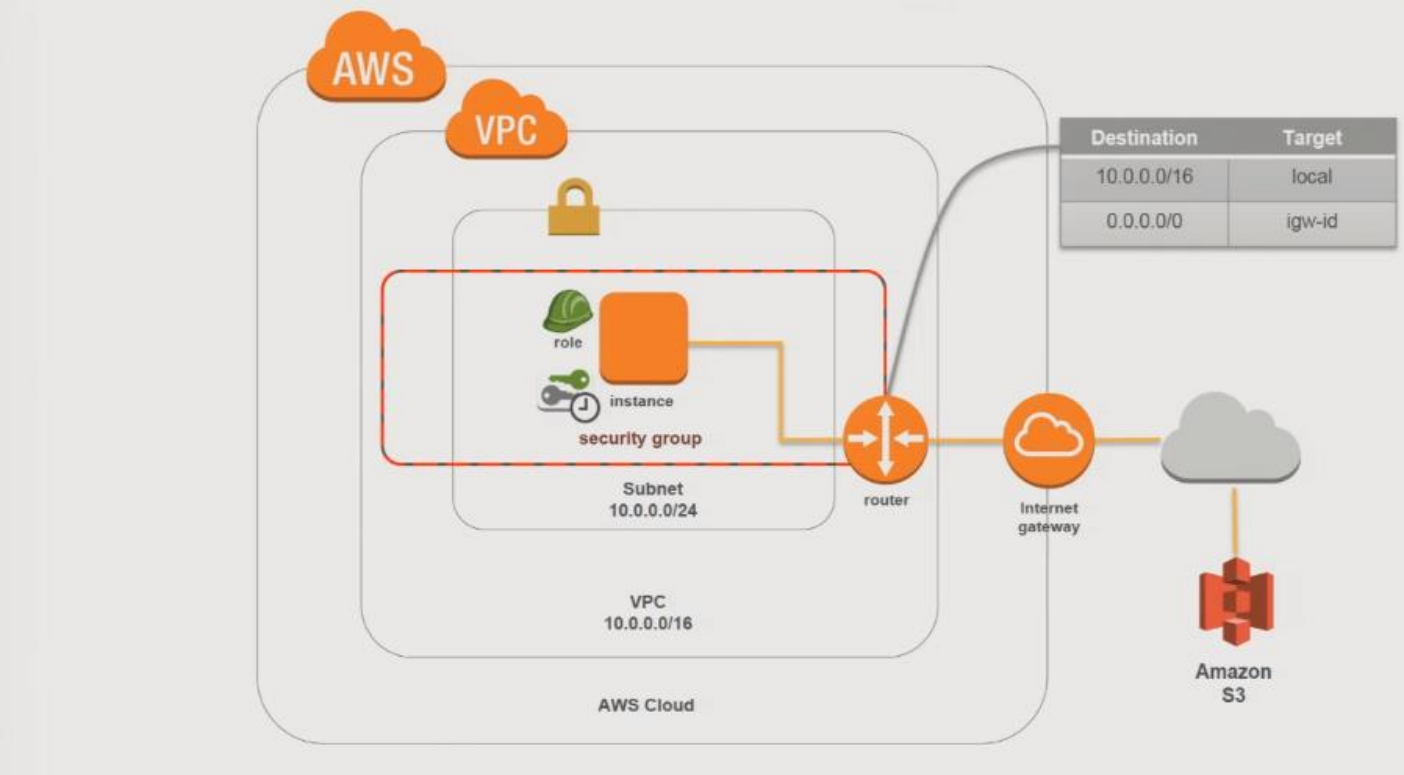
Then we create an internet gateway and attach it to our VPC so that the VPC can communicate with the outside world.



We then create a security group



Then we launch EC2 instance with an IAM role



Using that IAM role, the EC2 instance can then upload files to S3.

The Effective AWS CLI User Tenets

The effective AWS CLI user:

- Uses an iterative workflow
- Troubleshoots well
- Is resourceful with tooling
- Understands performance implications

```
$ aws ec2 create-vpc --cidr-block 10.0.0.0/16
```

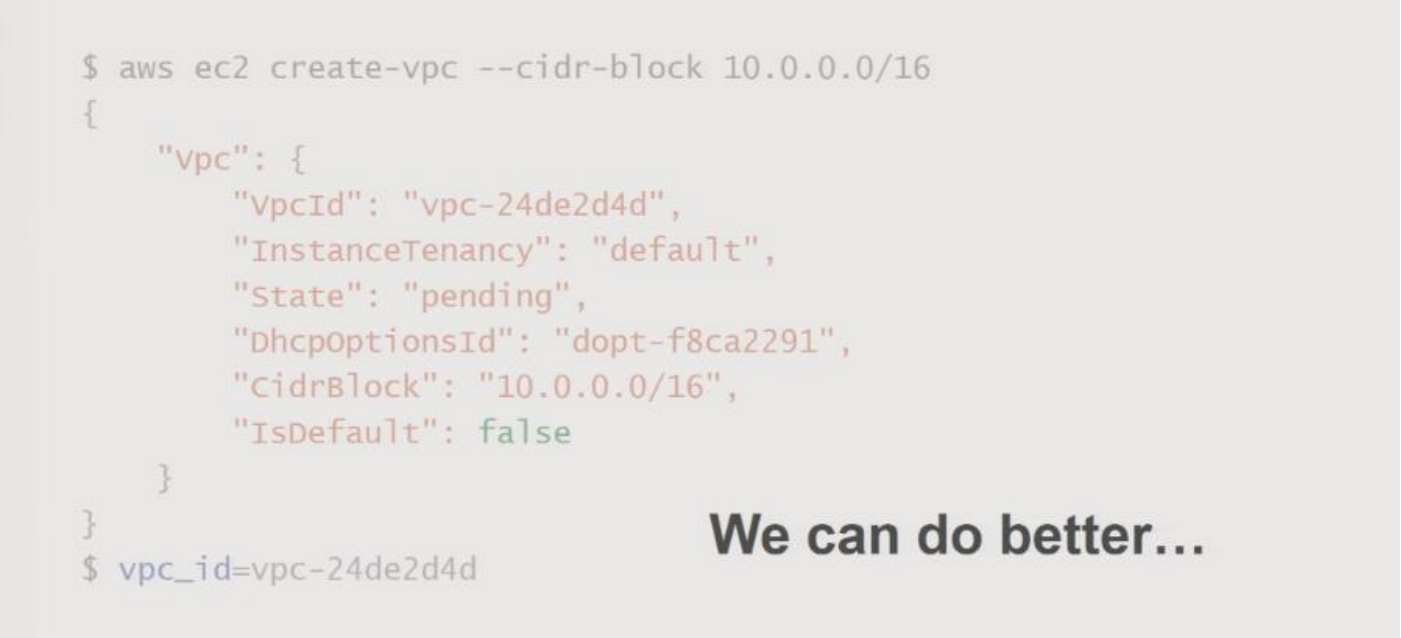
This command is a sample command that a general CLI user can use to create a VPC, they specify the service as ec2, the operation as create-vpc, and the set of parameters that they want to pass as - -cidr-block 10.0.0./16.

```
$ aws ec2 create-vpc --cidr-block 10.0.0.0/16
{
    "Vpc": {
        "VpcId": "vpc-24de2d4d",
        "InstanceTenancy": "default",
        "State": "pending",
        "DhcpOptionsId": "dopt-f8ca2291",
        "CidrBlock": "10.0.0.0/16",
        "IsDefault": false
    }
}
```

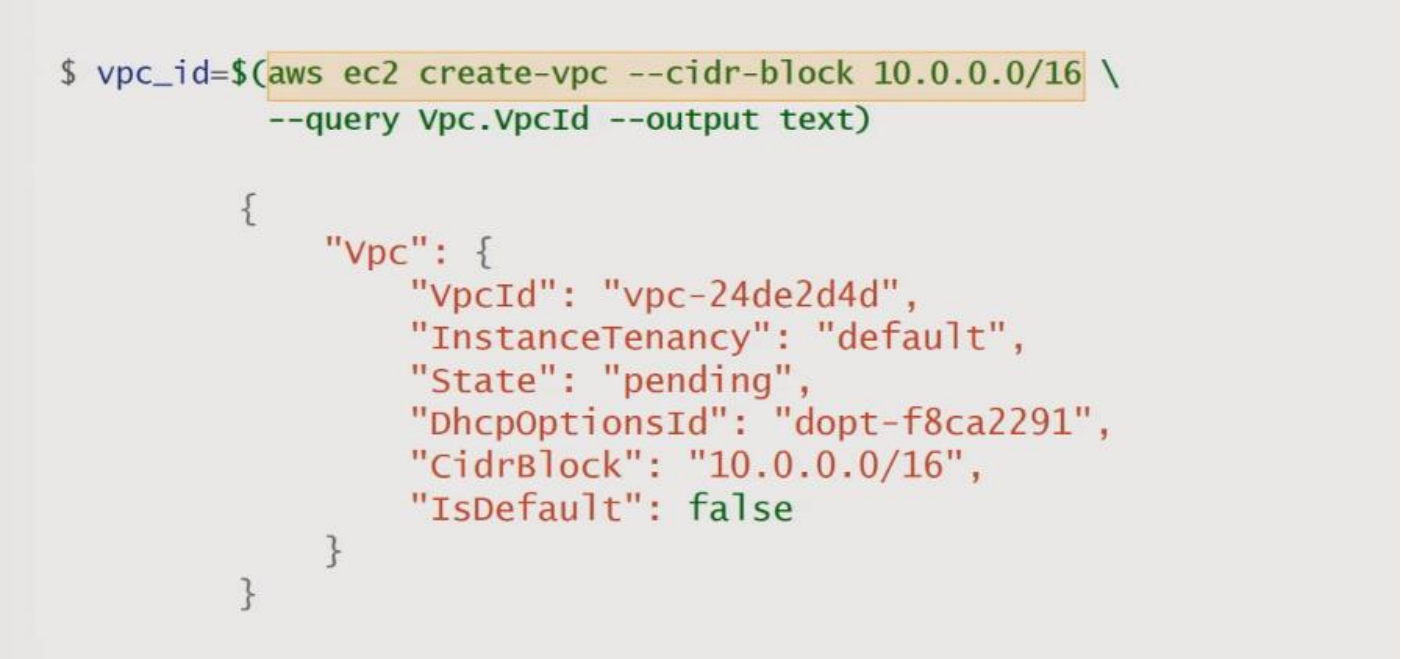
The command then returns back a JSON response from the service as above, the user can then use details in the response to do further things further downstream. We are going to be using the ***VpcId*** value a lot because a lot of the resources depend on that VPC value.

```
$ aws ec2 create-vpc --cidr-block 10.0.0.0/16
{
    "Vpc": {
        "VpcId": "vpc-24de2d4d",
        "InstanceTenancy": "default",
        "State": "pending",
        "DhcpOptionsId": "dopt-f8ca2291",
        "CidrBlock": "10.0.0.0/16",
        "IsDefault": false
    }
}
$ vpc_id=vpc-24de2d4d
```

We can go ahead and copy that ***VpcId*** value to a file or copy and paste that value into another command, or even save the value to a ***bash variable*** as above.



If we are trying to build shell scripts off of the commands we run, we need to do this in a programmatic way with no manual steps like copy and pasting values in between.



The ***$ vpc_id=$(aws ec2 create-vpc - -cidr-block 10.0.0.0/16 - -query Vpc.VpcId - -output text)*** is a better way of writing the same command to create a VPC.

```
$ vpc_id=$(aws ec2 create-vpc --cidr-block 10.0.0.0/16 \
          --query Vpc.VpcId --output text)
```

```
{
    "Vpc": {
        "VpcId": "vpc-24de2d4d",
        "InstanceTenancy": "default",
        "State": "pending",
        "DhcpOptionsId": "dopt-f8ca2291",
        "CidrBlock": "10.0.0.0/16",
        "IsDefault": false
    }
}
```

Now we are using parameters like the - -query to use a JMESPath expression to query down for specific values that we may need as below

```
$ vpc_id=$(aws ec2 create-vpc --cidr-block 10.0.0.0/16 \
          --query Vpc.VpcId --output text)
```

```
{
    "Vpc": {
        "VpcId": "vpc-24de2d4d",
        "InstanceTenancy": "default",
        "State": "pending",
        "DhcpOptionsId": "dopt-f8ca2291",
        "CidrBlock": "10.0.0.0/16",
        "IsDefault": false
    }
}
```

```
$ vpc_id=$(aws ec2 create-vpc --cidr-block 10.0.0.0/16 \
          --query Vpc.VpcId --output text)

        {
            "Vpc": {
                "VpcId": "vpc-24de2d4d",
                "InstanceTenancy": "default",
                "State": "pending",
                "DhcpOptionsId": "dopt-f8ca2291",
                "CidrBlock": "10.0.0.0/16",
                "IsDefault": false
            }
        }
```

```
$ vpc_id=$(aws ec2 create-vpc --cidr-block 10.0.0.0/16 \
          --query Vpc.VpcId --output text)

        {
            "Vpc": {
                "VpcId": "vpc-24de2d4d",
                "InstanceTenancy": "default",
                "State": "pending",
                "DhcpOptionsId": "dopt-f8ca2291",
                "CidrBlock": "10.0.0.0/16",
                "IsDefault": false
            }
        }
```

We then use the ***- -output text*** flag to strip out the double quotes in the JSON response

```
$ vpc_id=$(aws ec2 create-vpc --cidr-block 10.0.0.0/16 \
          --query Vpc.VpcId --output text)
$ echo "$vpc_id"
vpc-24de2d4d
```

```
$ vpc_id=$(aws ec2 create-vpc --cidr-block 10.0.0.0/16 \
          --query Vpc.VpcId --output text)
$ echo "$vpc_id"
vpc-24de2d4d

$ create_vpc_output=$(aws ec2 create-vpc --cidr-block 10.0.0.0/16)
$ vpc_id=$(jp -u  Vpc.VpcId <<< "$create_vpc_output")
$ echo "$vpc_id"
vpc-24de2d4d
```

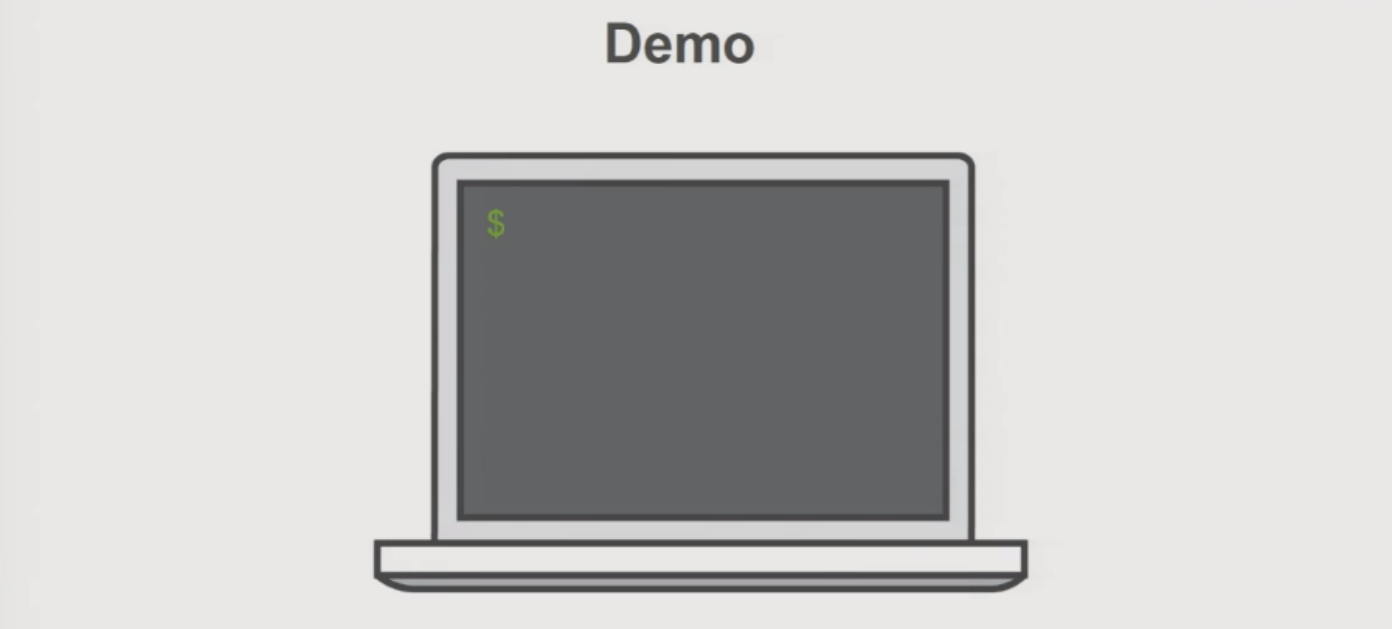
Another way of doing the same thing is by running the entire command and then saving the entire output into a variable like ***create_vpc_output*** as above. We then use a command line tool like ***jp*** to get the same functionality as the - -query and the - -output commands used earlier.

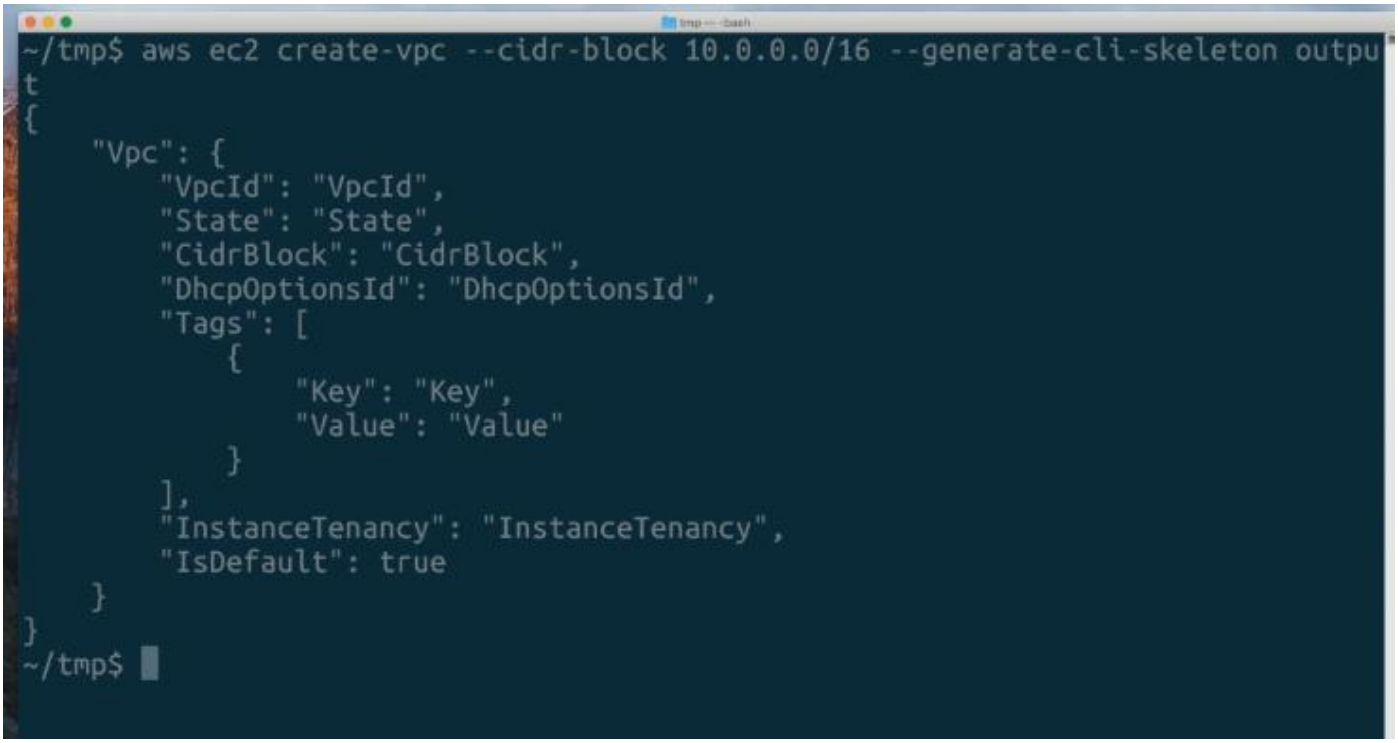

```
$ vpc_id=$(aws ec2 create-vpc --cidr-block 10.0.0.0/16 \
          --query Vpc.VpcId --output text)
$ echo "$vpc_id"
vpc-24de2d4d

$ create_vpc_output=$(aws ec2 create-vpc --cidr-block 10.0.0.0/16)
$ vpc_id=$(jp -u  Vpc.VpcId <<< "$create_vpc_output")
$ echo "$vpc_id"
vpc-24de2d4d
```


And we can make this even better!

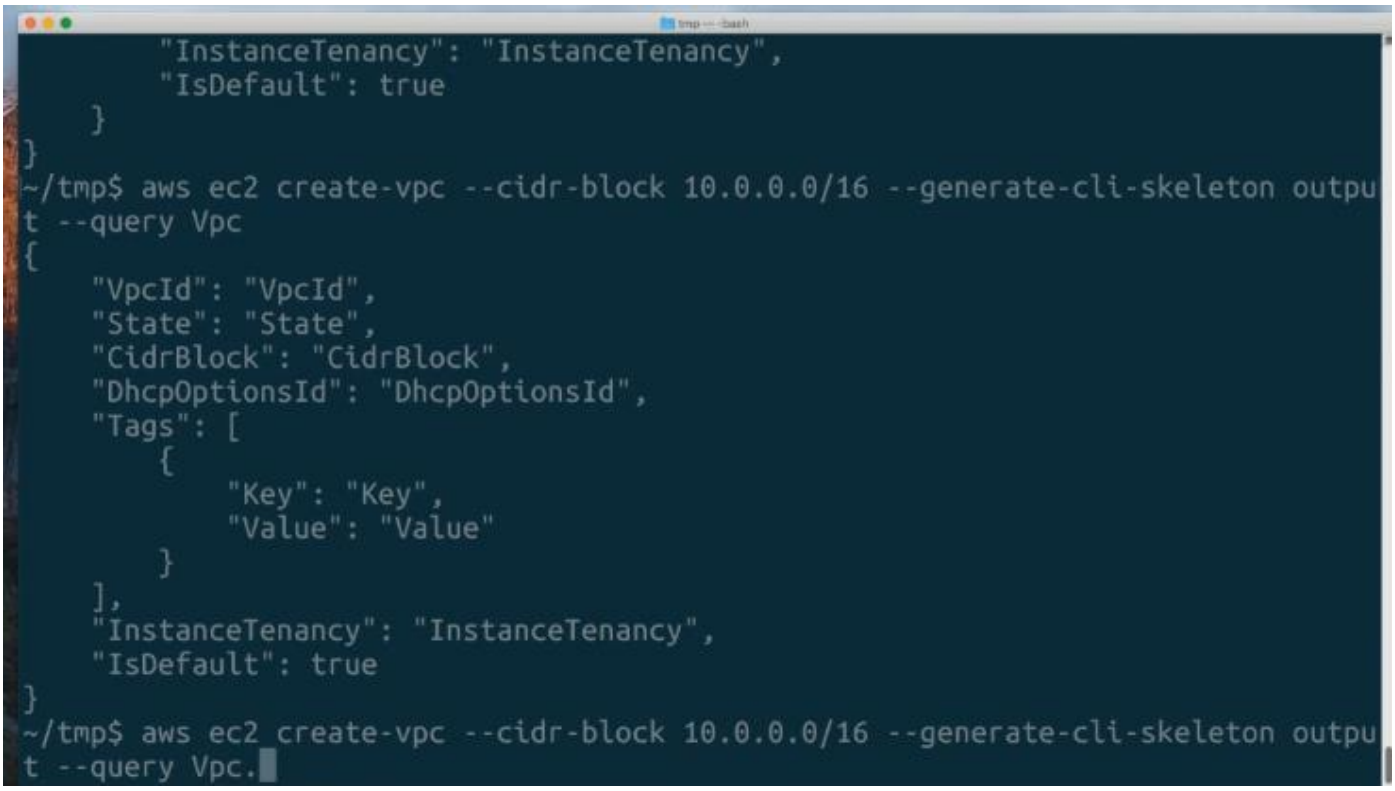
Problem Statement

The problem is that we don't know what the output of the command is going to be ahead of time, it would be better if we can get a sample output to work without having to make an API request

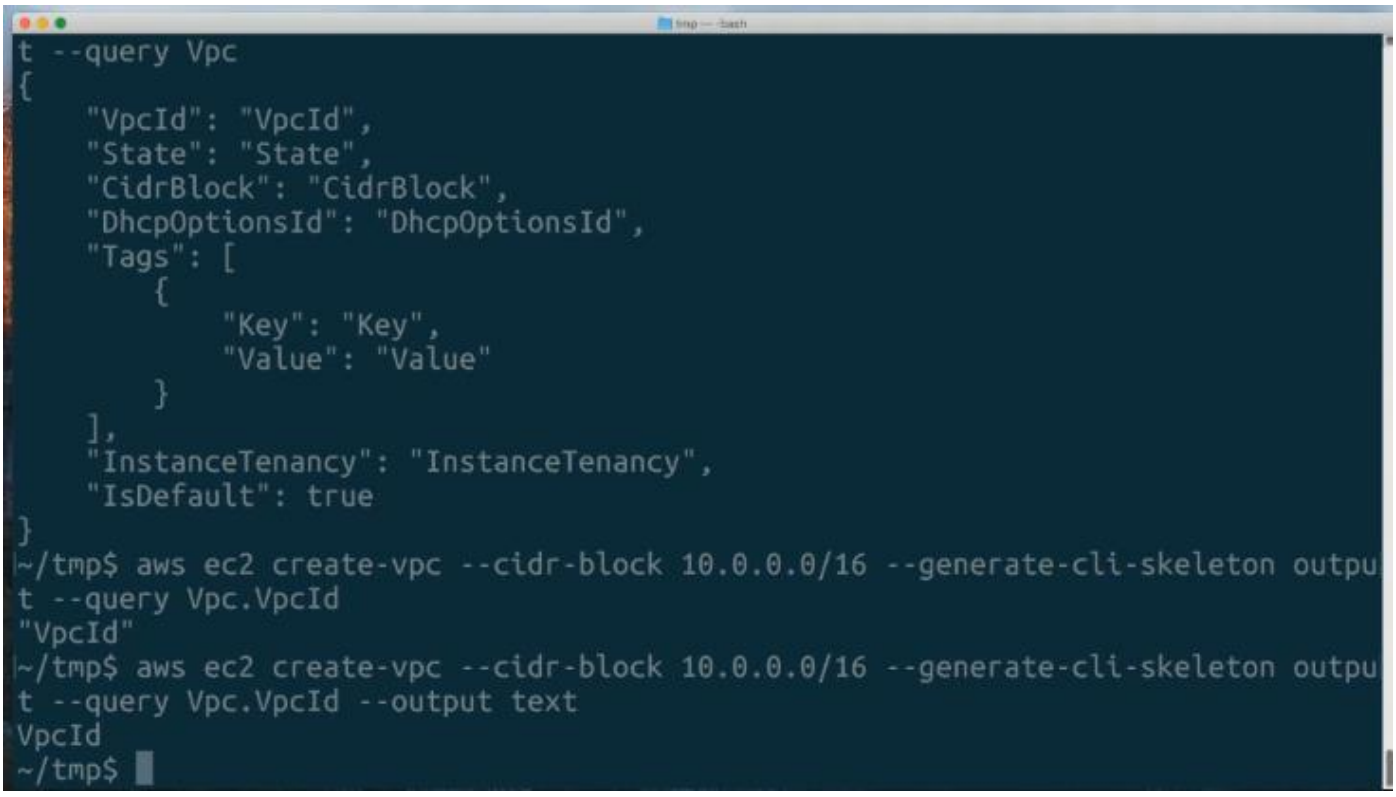




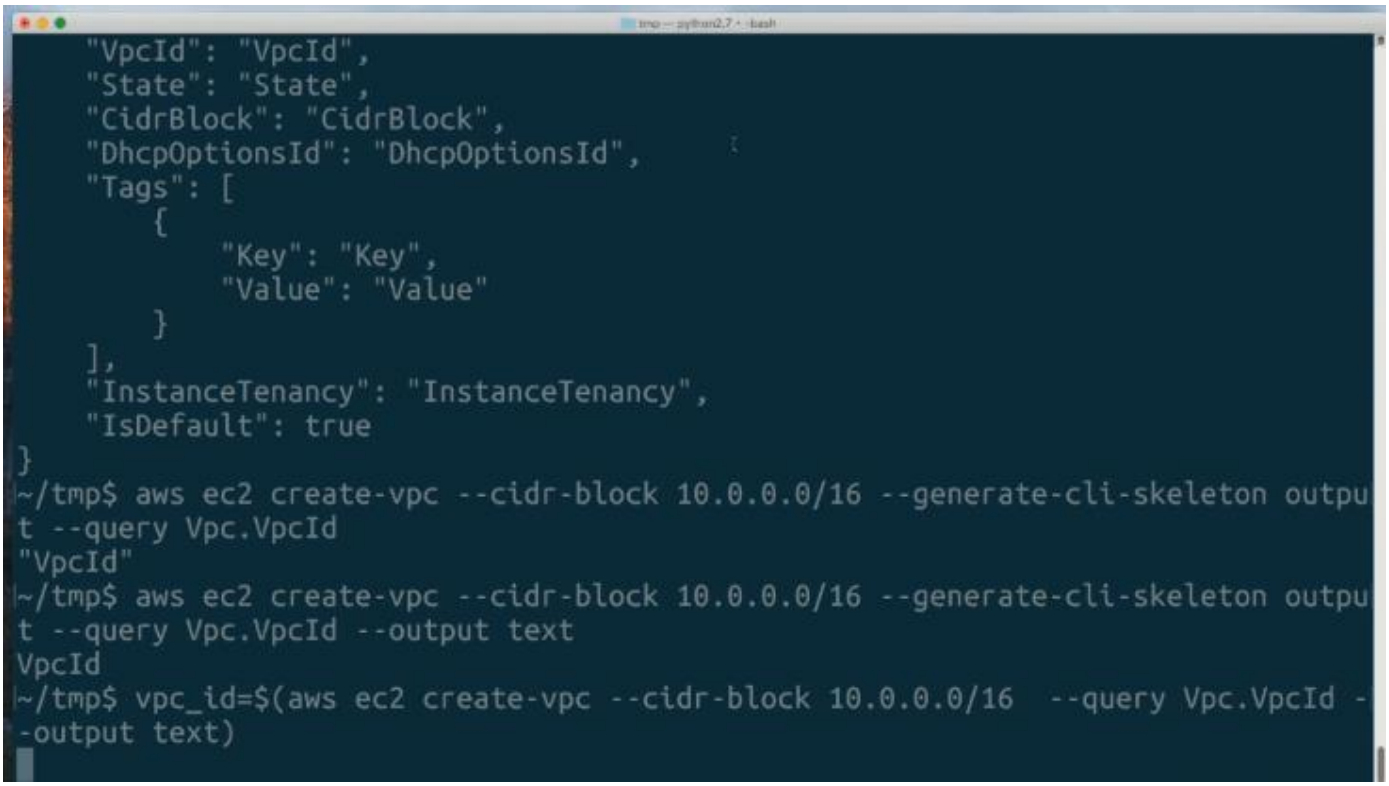
We use the command ***$ aws ec2 create-vpc - -cidr-block 10.0.0.0/16 - -generate-cli-skeleton output***, the - -generate-cli-skeleton parameter will print out a sample template that we can then use for CLI input JSON parameters. Using the ***output*** allows us to see what a sample response looks like as above



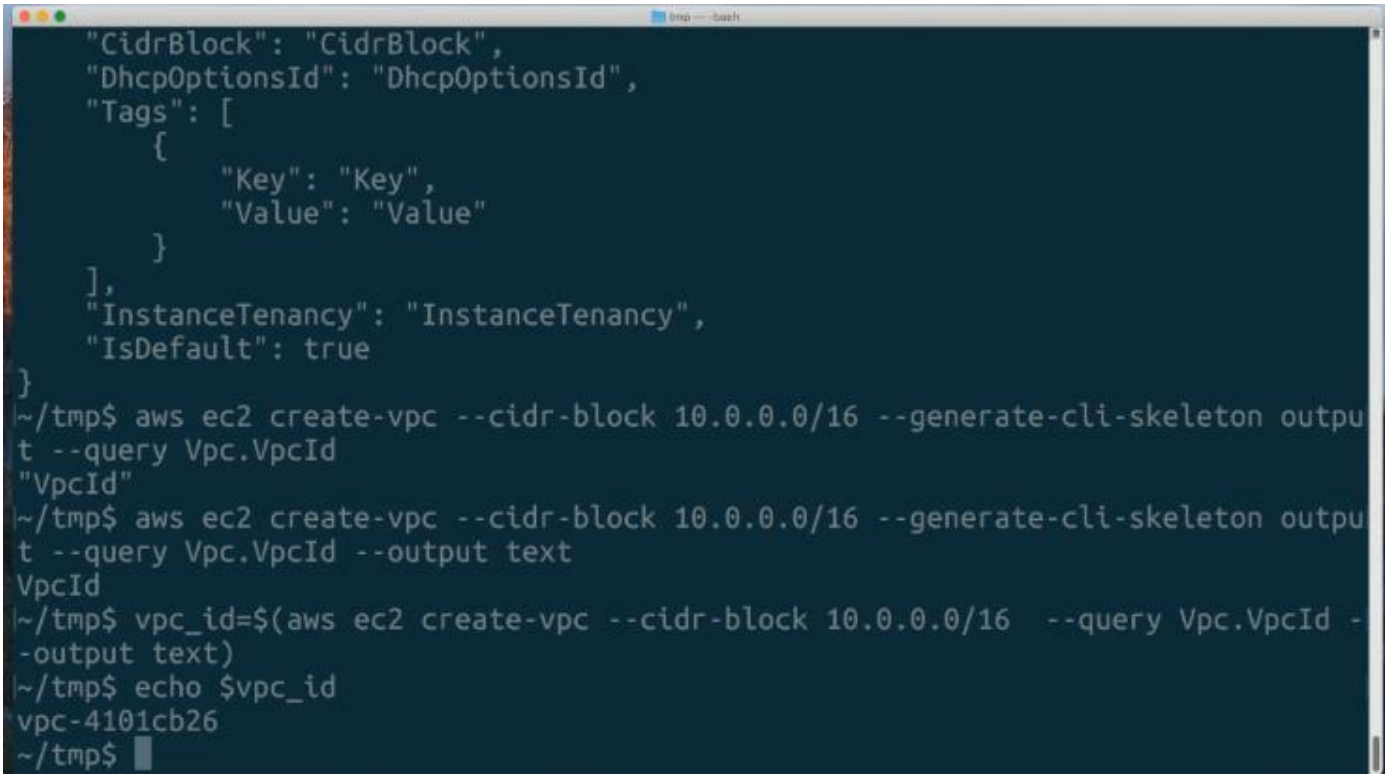
We can now apply the - -query parameters that we want to actually figure out the command we need as above. We query for the Vpc using the command ***$ aws ec2 create-vpc - -cidr-block 10.0.0.0/16 - -generate-cli-skeleton output - -query Vpc***



To get back the created Vpc's VpcId value, we edit the query further as ***$ aws ec2 create-vpc - -cidr-block 10.0.0.0/16 - -generate-cli-skeleton output - -query Vpc.VpcId***



We can now remove the - -generate-cli-skeleton command and make the actual API call ***$ vpc_id=$(aws ec2 create-vpc - -cidr-block 10.0.0.0/16 - -query Vpc.VpcId - -output text)*** as above



Using ***echo***, we can then see the ***VpcId*** (***vpc_id***) of the VPC that was created for us with the command.

Let's talk about how

```
$ aws ec2 create-vpc --cidr-block 10.0.0.0/16 --generate-cli-skeleton output
{
    "Vpc": {
        "VpcId": "VpcId",
        "State": "State",
        "CidrBlock": "CidrBlock",
        "DhcpOptionsId": "DhcpOptionsId",
        "Tags": [
            {
                "Key": "Key",
                "Value": "Value"
            }
        ],
        "InstanceTenancy": "InstanceTenancy",
        "IsDefault": true
    }
}
```

We are able to do this command because the CLI relies on the underlying ***Botocore library*** that powers the AWS SDK for Python and the AWS CLI. Botocore is completely data driven such that all its methods and models are driven off API models representing the different AWS APIs.

```
$ aws ec2 create-vpc --cidr-block 10.0.0.0/16 --generate-cli-skeleton output
{
    "Vpc": {
        "VpcId": "VpcId",
        "State": "State",
        "CidrBlock": "CidrBlock",
        "DhcpOptionsId": "DhcpOptionsId",
        "Tags": [
            {
                "Key": "Key",
                "Value": "Value"
            }
        ],
        "InstanceTenancy": "InstanceTenancy",
        "IsDefault": true
    }
}
```

```
"Vpc":{
  "type":"structure",
  "members":{
    "VpcId":{
      "shape":"String"
    },
    "State":{
      "shape":"VpcState"
    },
    "CidrBlock":{
      "shape":"String"
    },
    "DhcpOptionsId":{
      "shape":"String"
    },
    "Tags":{
      "shape":"TagList"
    },
    "InstanceTenancy":{
      "shape":"Tenancy"
    },
    "IsDefault":{
      "shape":"Boolean"
    }
  }
}
```

botocore/botocore/data/ec2/2016-09-15/service-2.json

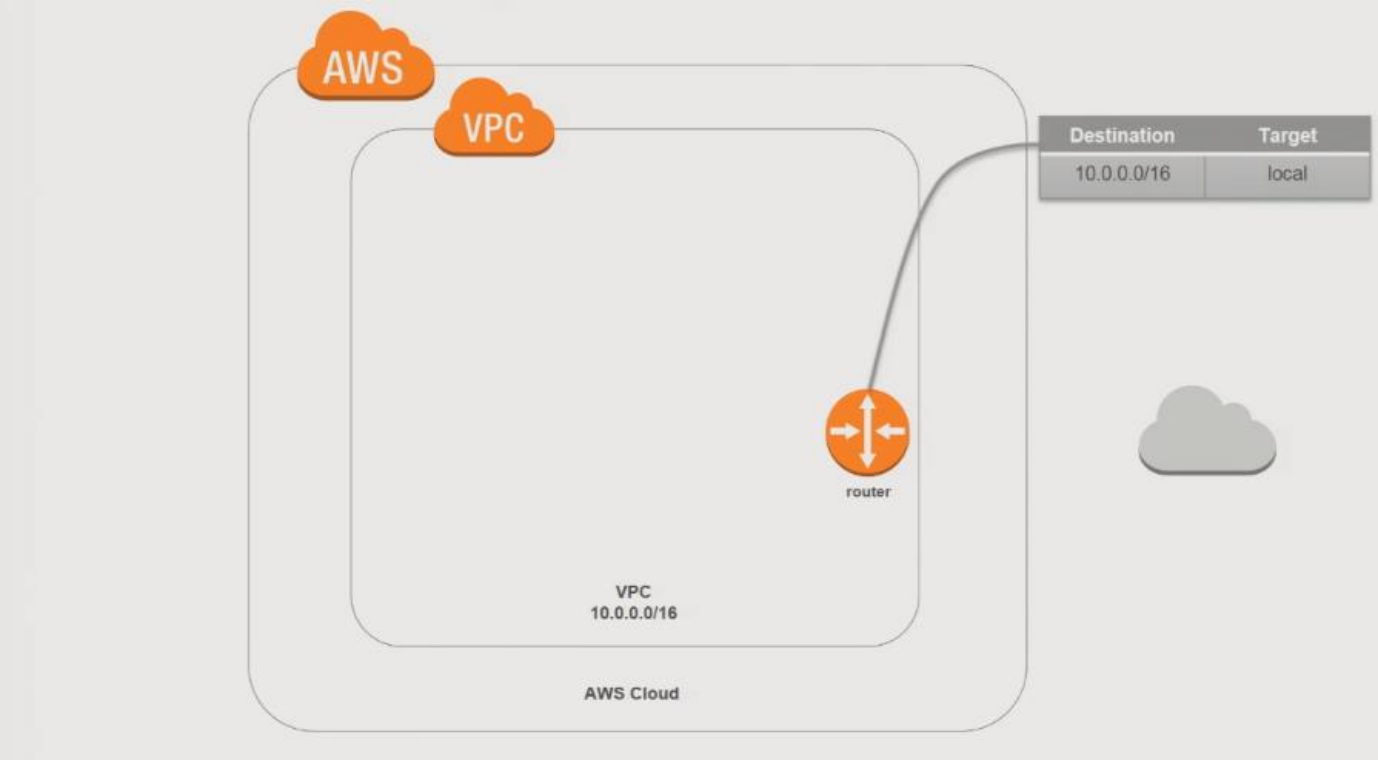
On the right is an example of one of the APIs,

```
$ aws ec2 create-vpc --cidr-block 10.0.0.0/16 --generate-cli-skeleton output
{
    "Vpc": {
        "VpcId": "VpcId",
        "State": "State",
        "CidrBlock": "CidrBlock",
        "DhcpOptionsId": "DhcpOptionsId",
        "Tags": [
            {
                "Key": "Key",
                "Value": "Value"
            }
        ],
        "InstanceTenancy": "InstanceTenancy",
        "IsDefault": true
    }
}
```

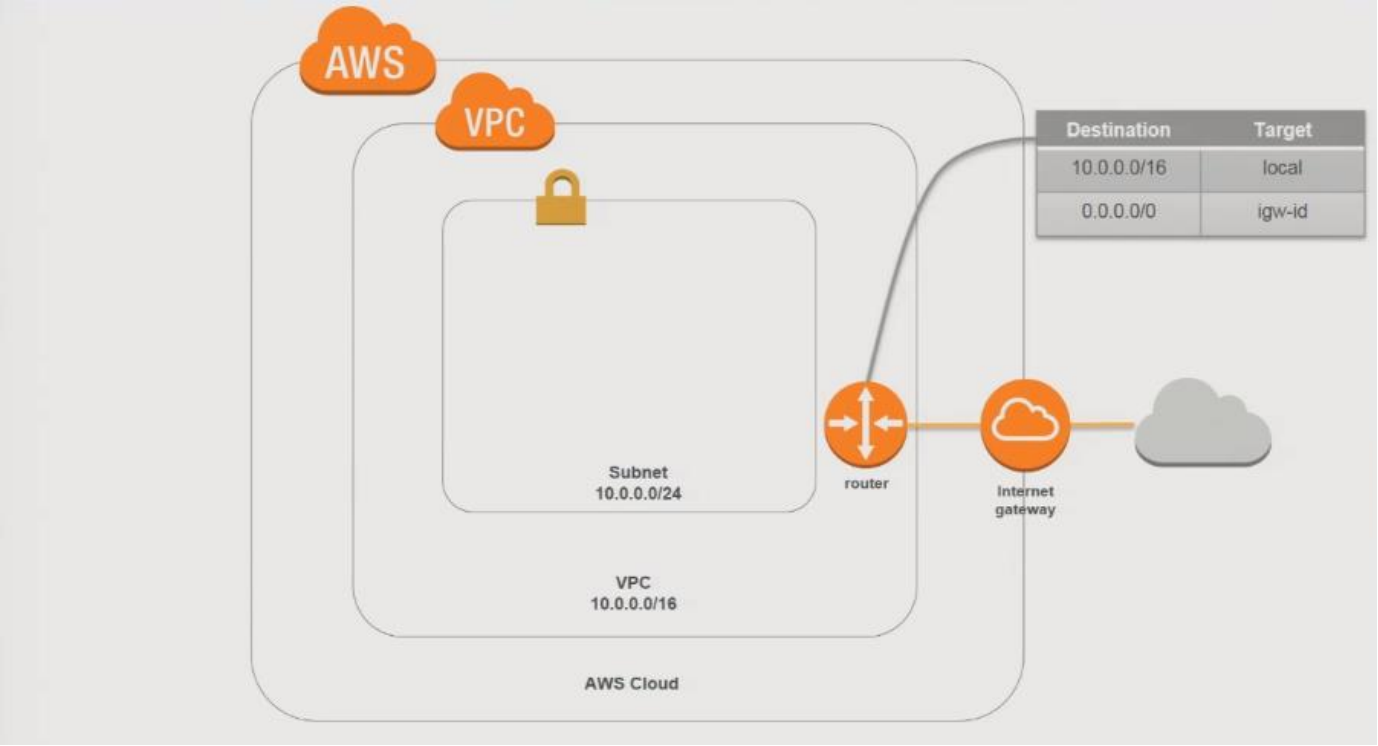
```
"Vpc":{
  "type":"structure",
  "members":{
    "VpcId":{
      "shape":"String"
    },
    "State":{
      "shape":"VpcState"
    },
    "CidrBlock":{
      "shape":"String"
    },
    "DhcpOptionsId":{
      "shape":"String"
    },
    "Tags":{
      "shape":"TagList"
    },
    "InstanceTenancy":{
      "shape":"Tenancy"
    },
    "IsDefault":{
      "shape":"Boolean"
    }
  }
}
```

botocore/botocore/data/ec2/2016-09-15/service-2.json

The outer and inner members match up with the API response



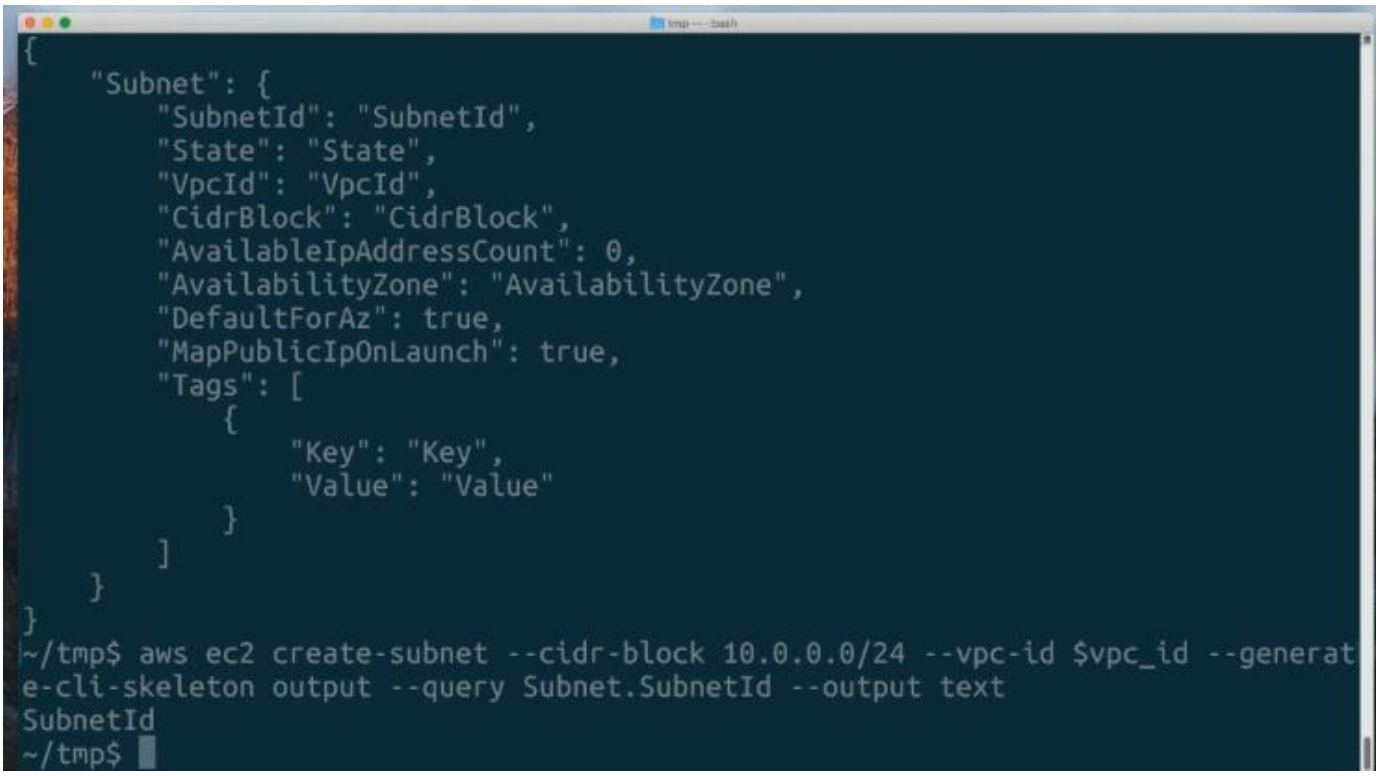
Now that we have our VPC, we can continue with our sample application



We now need to build up a few more resources like the IGW and Subnet.

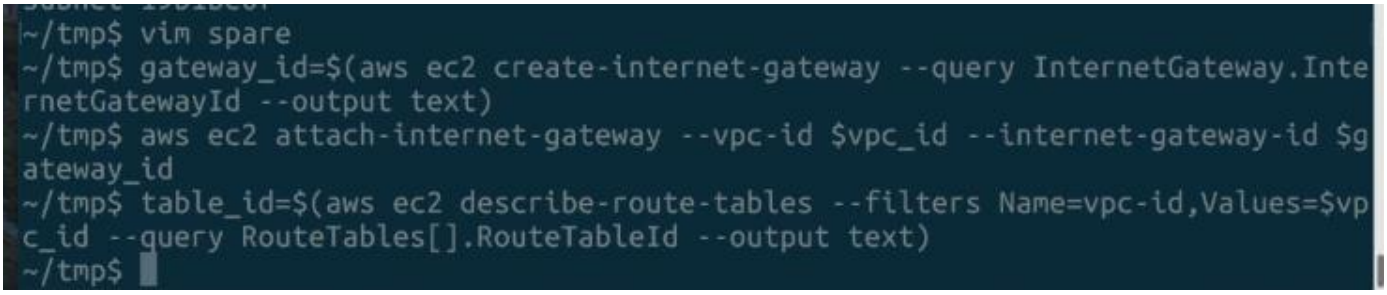
```
vpc-4101cb26
~/tmp$ aws ec2 create-subnet --cidr-block 10.0.0.0/24 --vpc-id $vpc_id --generat
e-cli-skeleton output
{
    "Subnet": {
        "SubnetId": "SubnetId",
        "State": "State",
        "VpcId": "VpcId",
        "CidrBlock": "CidrBlock",
        "AvailableIpAddressCount": 0,
        "AvailabilityZone": "AvailabilityZone",
        "DefaultForAz": true,
        "MapPublicIpOnLaunch": true,
        "Tags": [
            {
                "Key": "Key",
                "Value": "Value"
            }
        ]
    }
}
~/tmp$
```

To create the subnet within the VPC, we use the ***- -generate-cli-skeleton output*** command with the main ***$ aws ec2 create-subnet - -cidr-block 10.0.0.0/24 - -vpc-id $vpc_id - -generate-cli-skeleton output*** command to see what our output will look like as above.



We then create our JMESPath query to get the SubnetId value out using the command ***$ aws ec2 create-subnet - -cidr-block 10.0.0.0/24 - -vpc-id $vpc_id - -generate-cli-skeleton output - -query Subnet.SubnetId - -output text***

We can now make the actual CLI call to create the subnet within the VPC using the command ***$ subnet_id=$(aws ec2 create-subnet - -cidr-block 10.0.0.0/24 - -vpc-id $vpc_id - -query Subnet.SubnetId - -output text).*** We also save the SubnetId value in the response to another variable called ***subnet_id***.



Then, we create an IGW using the ***gateway_id=$(aws ec2 create-internet-gateway - -query InternetGateway.InternetGatewayId - -output text)*** command,

Then we attach the IGW to the VPC using the ***aws ec2 attach-internet-gateway - -vpc-id $vpc_id - -internet-gateway-id $gateway_id*** command.

Then create a Route table for the VPC using ***table_id=$(aws ec2 describe-route-tables - -filters Name=vpc.id,Values=$vpc_id - -query RouteTables[].RouteTableId - -output text)*** command

```
~/tmp$ echo aws ec2 create-route --route-table-id $table_id --destination-cidr-b
lock 0.0.0.0/0 --gateway-id $gateway_id
aws ec2 create-route --route-table-id rtb-fa8e179d --destination-cidr-block 0.0.
0.0/0 --gateway-id igw-429ff826
~/tmp$
```

We then echo out the result of creating a Route using the ***echo aws ec2 create-route - -route-table-id $table_id - -destination-cidr-block 0.0.0.0/0 - -gateway-id $gateway_id*** command as above. We can confirm that the ***$table_id*** is ***rtb-fa8e179d*** and the $gateway_id ***igw-429ff826*** variables are correctly substituted in the query to be sent in the CLI call

```
~/tmp$ echo aws ec2 create-route --route-table-id $table_id --destination-cidr-b
lock 0.0.0.0/0 --gateway-id $gateway_id
aws ec2 create-route --route-table-id rtb-fa8e179d --destination-cidr-block 0.0.
0.0/0 --gateway-id igw-429ff826
~/tmp$ aws ec2 create-route --route-table-id $table_id --destination-cidr-block
0.0.0.0/0 --gateway-id $gateway_id
{
    "Return": true
}
~/tmp$
```

We now run the actual ***aws ec2 create-route - -route-table-id $table_id - -destination-cidr-block 0.0.0.0/0 - -gateway-id $gateway_id*** command against the CLI and get the response above. We can now create shell script for doing all the steps at once

```
~/tmp$ aws ec2 create-route --route-table-id $table_id --destination-cidr-block
0.0.0.0/0 --gateway-id $gateway_id
{
    "Return": true
}
~/tmp$ history 20 | cut -c 8- > vpc
~/tmp$ vim vpc
```

We can use the ***$ history 20 | cut -c 8- > vpc*** command to see that last 20 commands that we last ran

```
  1 clear
  2 aws ec2 create-vpc --cidr-block 10.0.0.0/16 --generate-cli-skeleton output
  3 aws ec2 create-vpc --cidr-block 10.0.0.0/16 --generate-cli-skeleton output -
    -query Vpc
  4 aws ec2 create-vpc --cidr-block 10.0.0.0/16 --generate-cli-skeleton output -
    -query Vpc.VpcId
  5 aws ec2 create-vpc --cidr-block 10.0.0.0/16  --query Vpc.VpcId --output text
    )
  6 vpc_id=$(aws ec2 create-vpc --cidr-block 10.0.0.0/16  --query Vpc.VpcId --ou
    tput text)
  7 echo $vpc_id
  8 aws ec2 create-subnet --cidr-block 10.0.0.0/24 --vpc-id $vpc_id --generate-c
    li-skeleton output
  9 aws ec2 create-subnet --cidr-block 10.0.0.0/24 --vpc-id $vpc_id  --query Sub
    net.SubnetId --output text)
 10 subnet_id=$(aws ec2 create-subnet --cidr-block 10.0.0.0/24 --vpc-id $vpc_id
     --query Subnet.SubnetId --output text)
 11 echo $subnet_id
 12 vim spare
 13 gateway_id=$(aws ec2 create-internet-gateway --query InternetGateway.Interne
    tGatewayId --output text)
"vpc" 18L, 1343C                                          1,1           Top
```

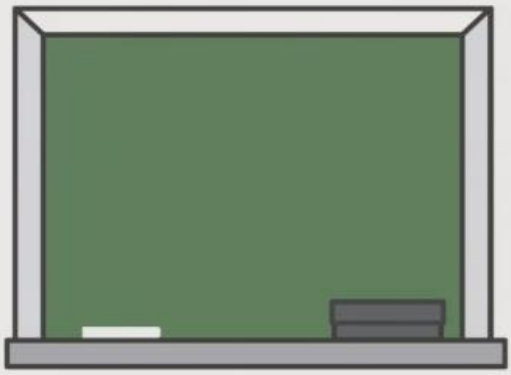
We can then remove the commands that we do not need and leave only the set of commands we need to have the script work as below

```
  1 vpc_id=$(aws ec2 create-vpc --cidr-block 10.0.0.0/16  --query Vpc.VpcId --ou
    tput text)
  2 subnet_id=$(aws ec2 create-subnet --cidr-block 10.0.0.0/24 --vpc-id $vpc_id
     --query Subnet.SubnetId --output text)
  3 gateway_id=$(aws ec2 create-internet-gateway --query InternetGateway.Interne
    tGatewayId --output text)
  4 aws ec2 attach-internet-gateway --vpc-id $vpc_id --internet-gateway-id $gate
    way_id
  5 table_id=$(aws ec2 describe-route-tables --filters Name=vpc-id,Values=$vpc_i
    d --query RouteTables[].RouteTableId --output text)
  6 aws ec2 create-route --route-table-id $table_id --destination-cidr-block 0.0
    .0.0/0 --gateway-id $gateway_id
  7 echo "$vpc_id"
~
~
```

This is a very simple way to go from using CLI commands to using shell scripts

Important Takeaways

- `--generate-cli-skeleton output`
- Leverage UNIX commands (e.g. `echo`, `history`)



The Effective AWS CLI User Tenets

The effective AWS CLI user:

- Uses an iterative workflow
- Troubleshoots well
- Is resourceful with tooling
- Understands performance implications

```
$ aws ec2 create-security-group --group-name re:Invent \
    --vpc-id VpcId
aws: error: argument --description is required


$ aws ec2 create-security-group --group-name re:Invent \
    --description 're:Invent demo' --vpc-id VpcId

An error occurred (InvalidVpcID.NotFound) when calling the
CreateSecurityGroup operation: The vpc ID 'VpcId' does not exist
```

CLI errors/exceptions are mostly self-explanatory when you make a mistake, but you might still need some deeper insight into why you are getting the errors

--debug

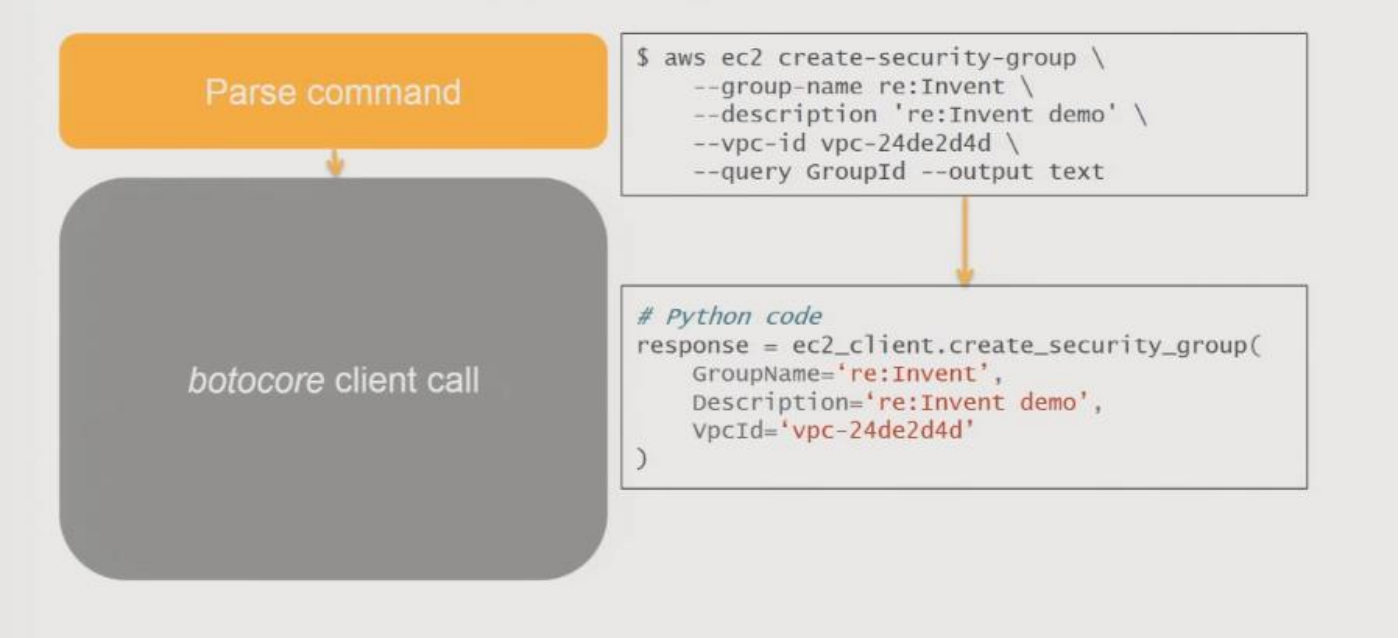
```
$ aws ec2 create-security-group --group-name re:Invent \
     --description 're:Invent demo' --vpc-id "$vpc_id" --debug

2016-11-04 10:30:33,532 - MainThread - awscli.clidriver - DEBUG - CLI version: aws-
cli/1.11.10 Python/2.7.10 Darwin/15.6.0 botocore/1.4.65
2016-11-04 10:30:33,533 - MainThread - awscli.clidriver - DEBUG - Arguments entered to
CLI: ['ec2', 'create-security-group', '--group-name', 're:Invent', '--description',
're:Invent demo', '--vpc-id', 'vpc-43cb382a', '--debug']
2016-11-04 10:30:33,533 - MainThread - botocore.hooks - DEBUG - Event session-
initialized: calling handler <function add_scalar_parsers at 0x10eefc488>
2016-11-04 10:30:33,533 - MainThread - botocore.hooks - DEBUG - Event session-
initialized: calling handler <function inject_assume_role_provider_cache at
0x10ecde578>
2016-11-04 10:30:33,533 - MainThread - botocore.credentials - DEBUG - Skipping
environment variable credential check because profile name was explicitly set.
2016-11-04 10:30:33,537 - MainThread - botocore.loaders - DEBUG - Loading JSON file:
/Users/kyleknap/GitHub/botocore/botocore/data/ec2/2016-09-15/service-2.json
2016-11-04 10:30:33,622 - MainThread - botocore.hooks - DEBUG - Event service-data-
loaded.ec2: calling handler <function register_retries_for_service at 0x10e904500>
```

The **- -*debug*** helps to print out a set of log messages that represents how the CLI is processing the command you entered.

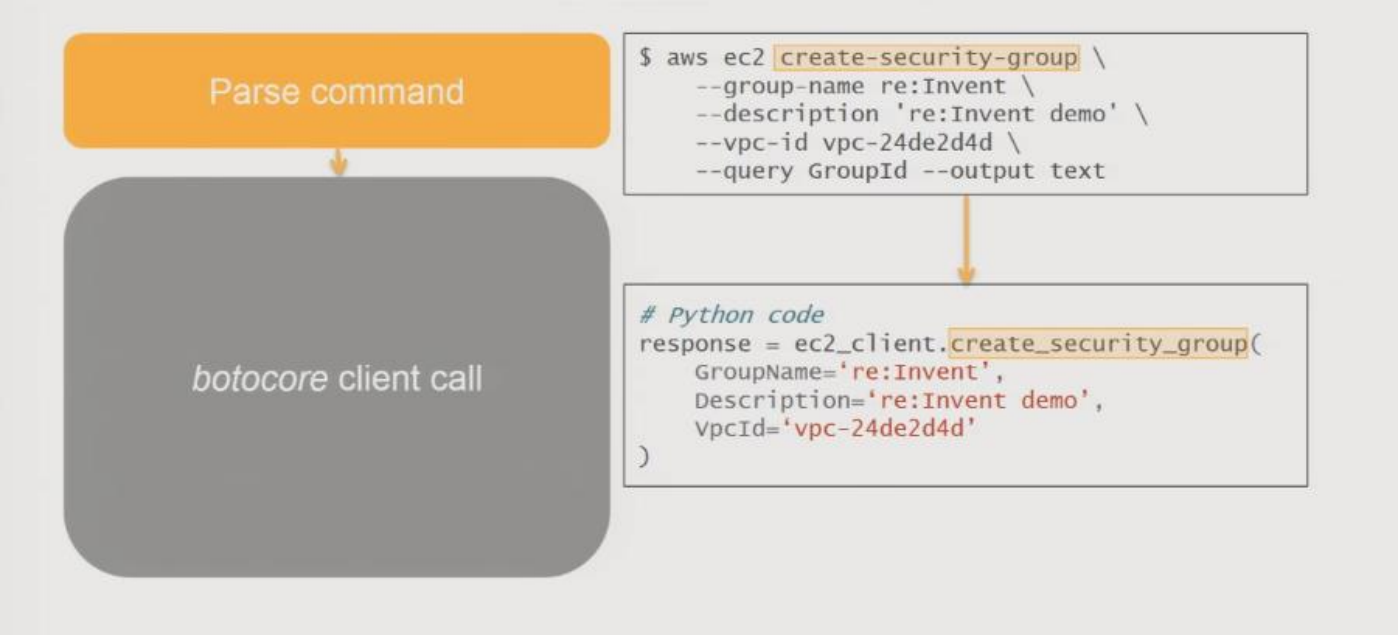
```
2016-11-04 10:30:33,623 - MainThread - botocore.handlers - DEBUG - Registering retry
handlers for service: ec2
2016-11-04 10:30:33,628 - MainThread - botocore.hooks - DEBUG - Event building-command-
table.ec2: calling handler <functools.partial object at 0x10ef02940>
2016-11-04 10:30:33,628 - MainThread - awscli.customizations.removals - DEBUG -
Removing operation: import-instance
2016-11-04 10:30:33,628 - MainThread - awscli.customizations.removals - DEBUG -
Removing operation: import-volume
2016-11-04 10:30:33,628 - MainThread - botocore.hooks - DEBUG - Event building-command-
table.ec2: calling handler <function add_waiters at 0x10ef03578>
2016-11-04 10:30:33,631 - MainThread - botocore.loaders - DEBUG - Loading JSON file:
/Users/kyleknap/GitHub/botocore/botocore/data/ec2/2016-09-15/waiters-2.json
2016-11-04 10:30:33,635 - MainThread - awscli.clidriver - DEBUG - OrderedDict([(u'dry-
run', <awscli.arguments.BooleanArgument object at 0x10febe950>), (u'no-dry-run',
<awscli.arguments.BooleanArgument object at 0x10febe990>), (u'group-name',
<awscli.arguments.CLIArgument object at 0x10febe9d0>), (u'description',
<awscli.arguments.CLIArgument object at 0x10febea10>), (u'vpc-id',
<awscli.arguments.CLIArgument object at 0x10febea50>)])
```

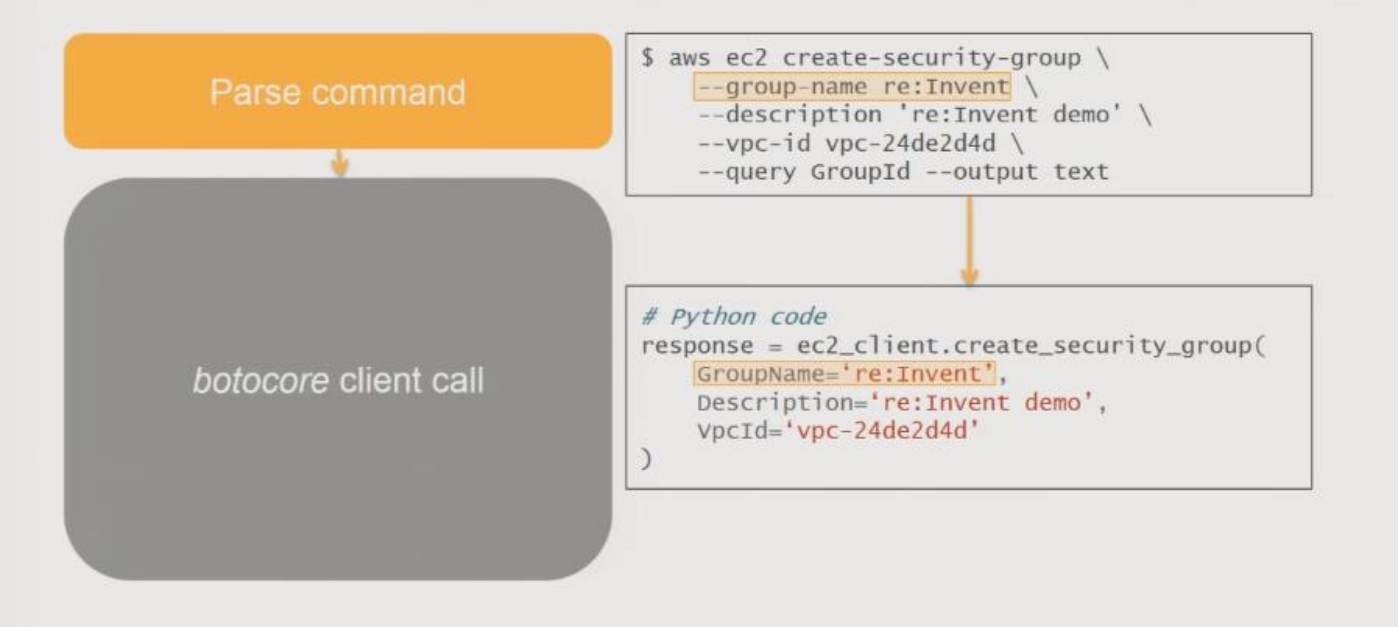
And this continues on for a while…

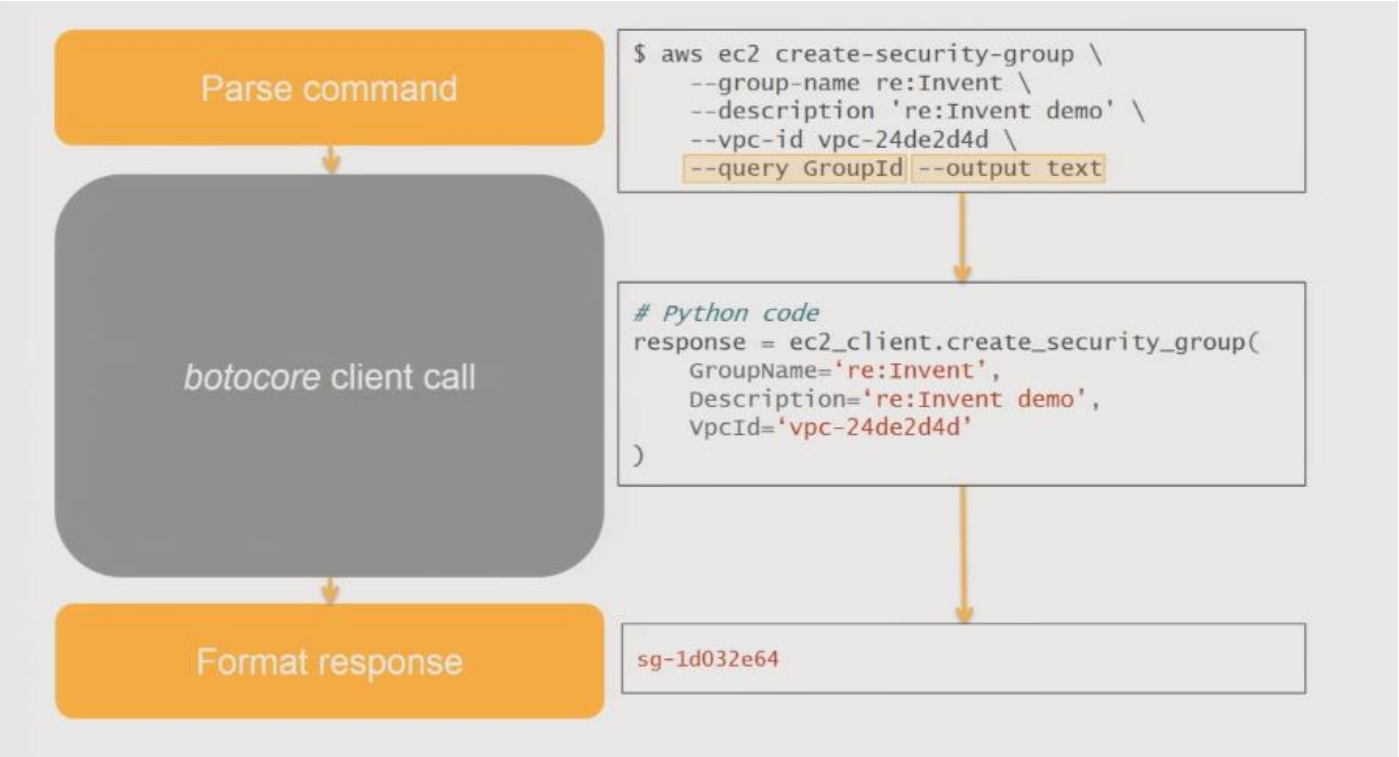
You don’t need to understand all the debug messages, there are only a handful of the debug messages you need to understand to be able to make meaning of the debug logs. You need some understanding of how the CLI stack works



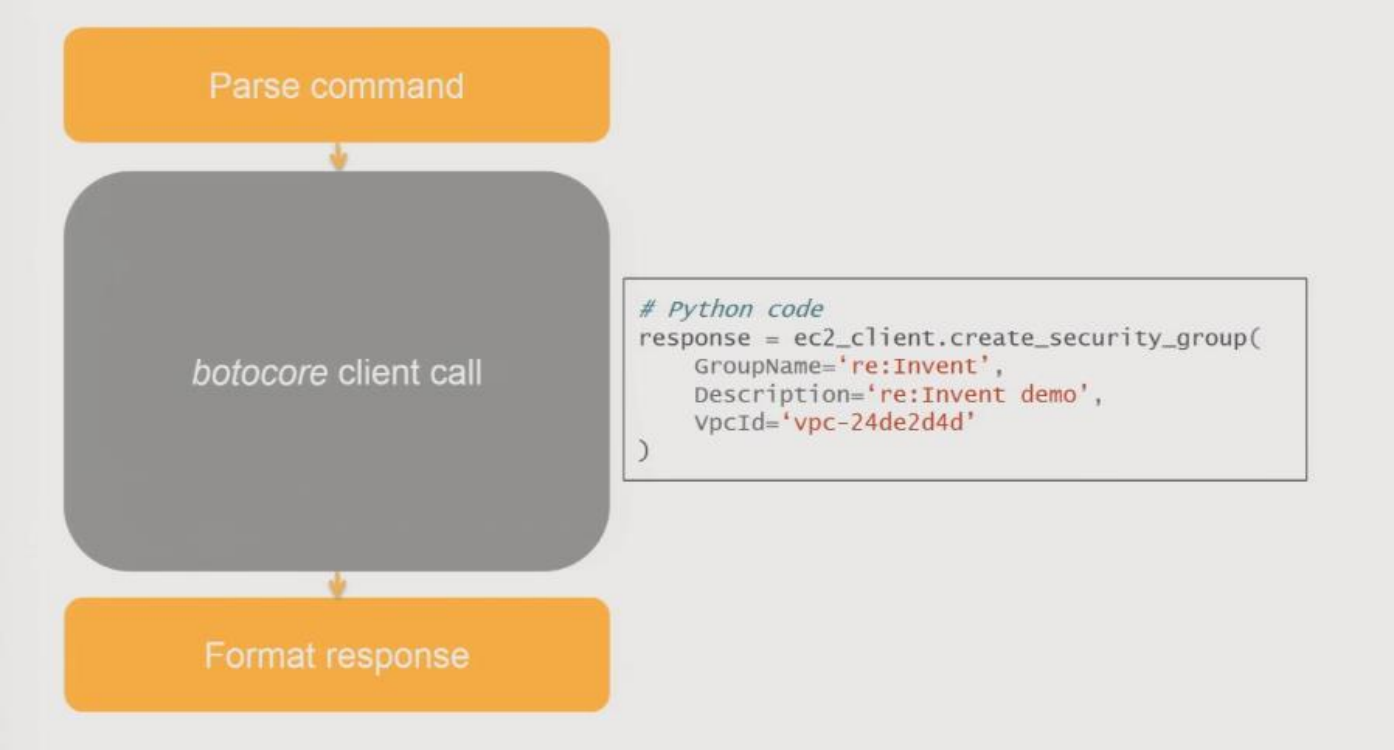
The first step of the CLI command is the parsing part where the CLI wants to know what operation and parameters got passed in.







After the call has been made, you get a response back and the output format you specified will be applied to the response

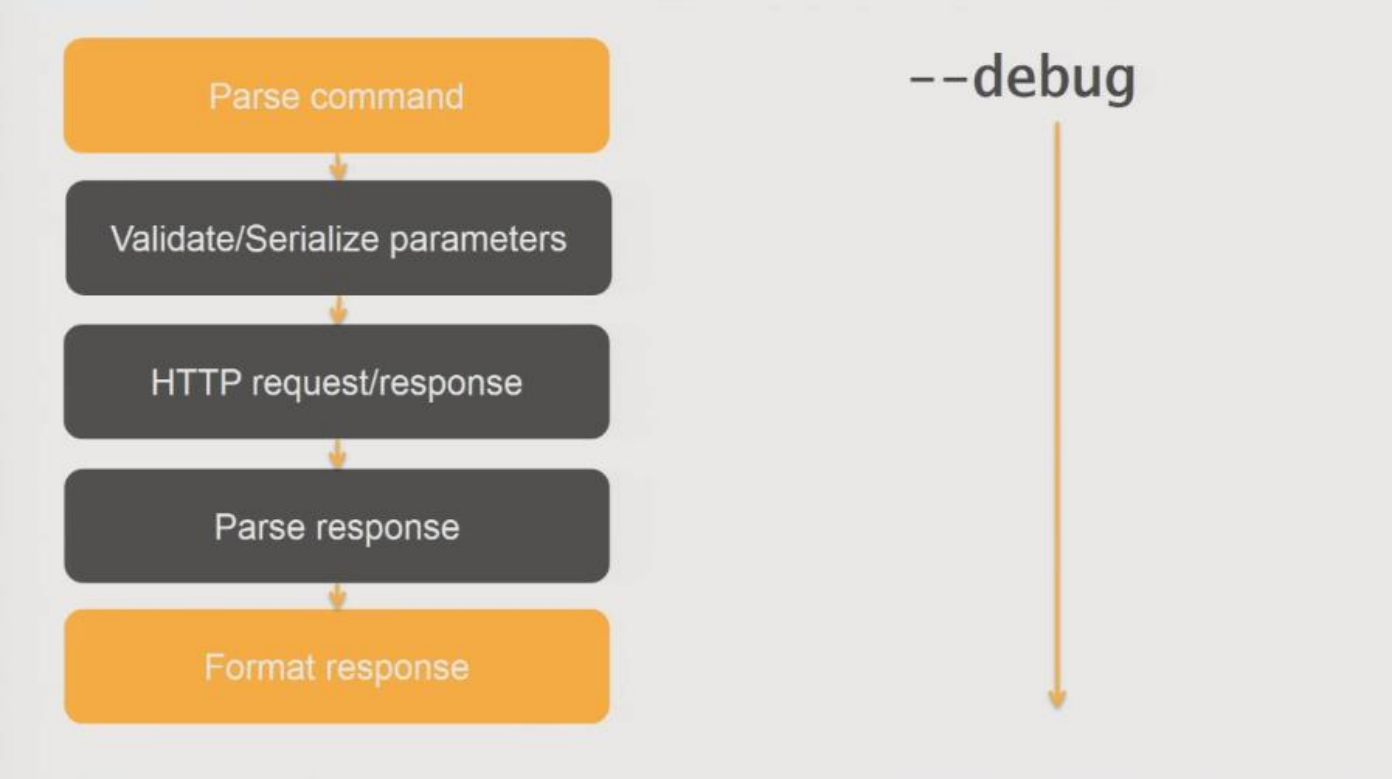


Let us now see the steps made inside the botocore client call part

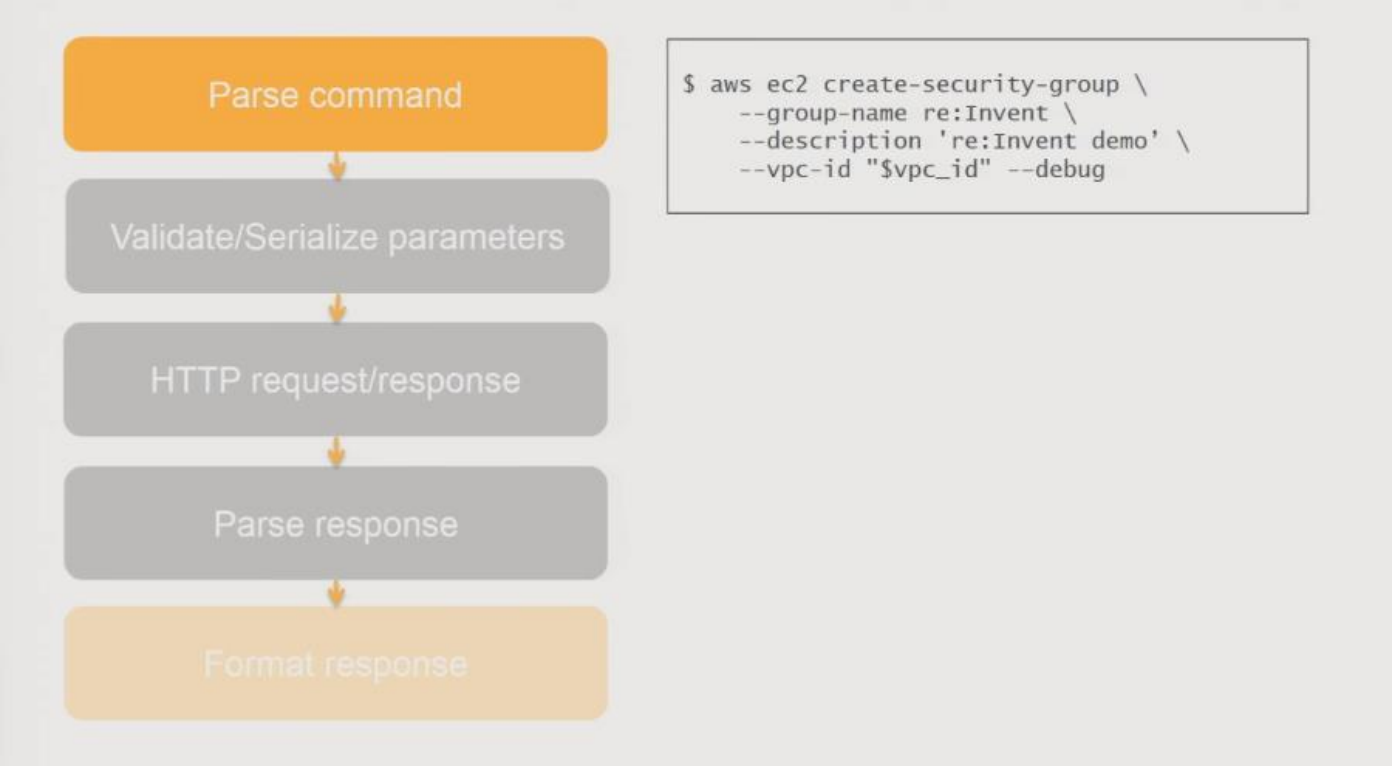
Botocore will validate and serialize the parameters passed to the method call using the sample model for the CreateSecurityGroupRequest command with the possible parameters


```
"CreateSecurityGroupResult":{
  "type":"structure",
  "members":{
    "GroupId":{
      "shape":"String"
    }
  }
}
```

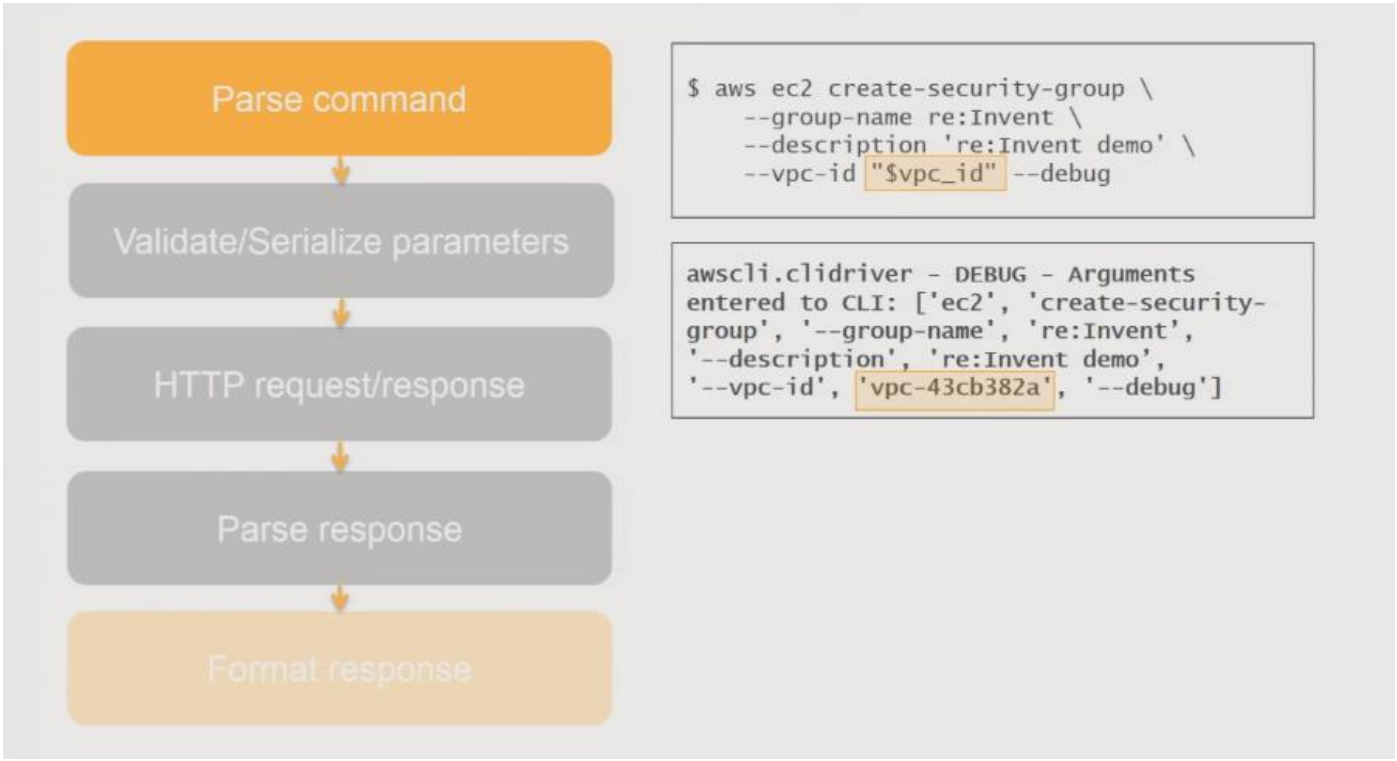
botocore/botocore/data/ec2/2016-09-15/service-2.json

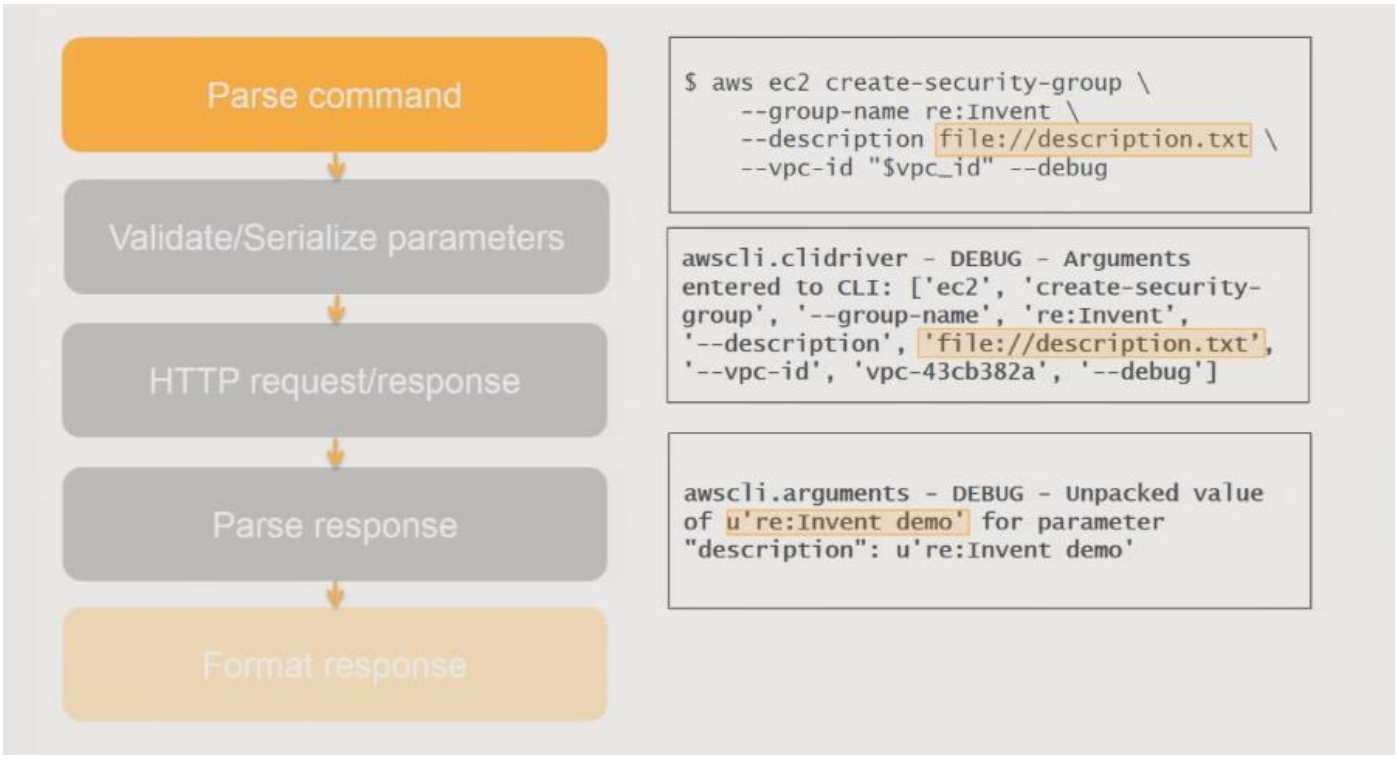


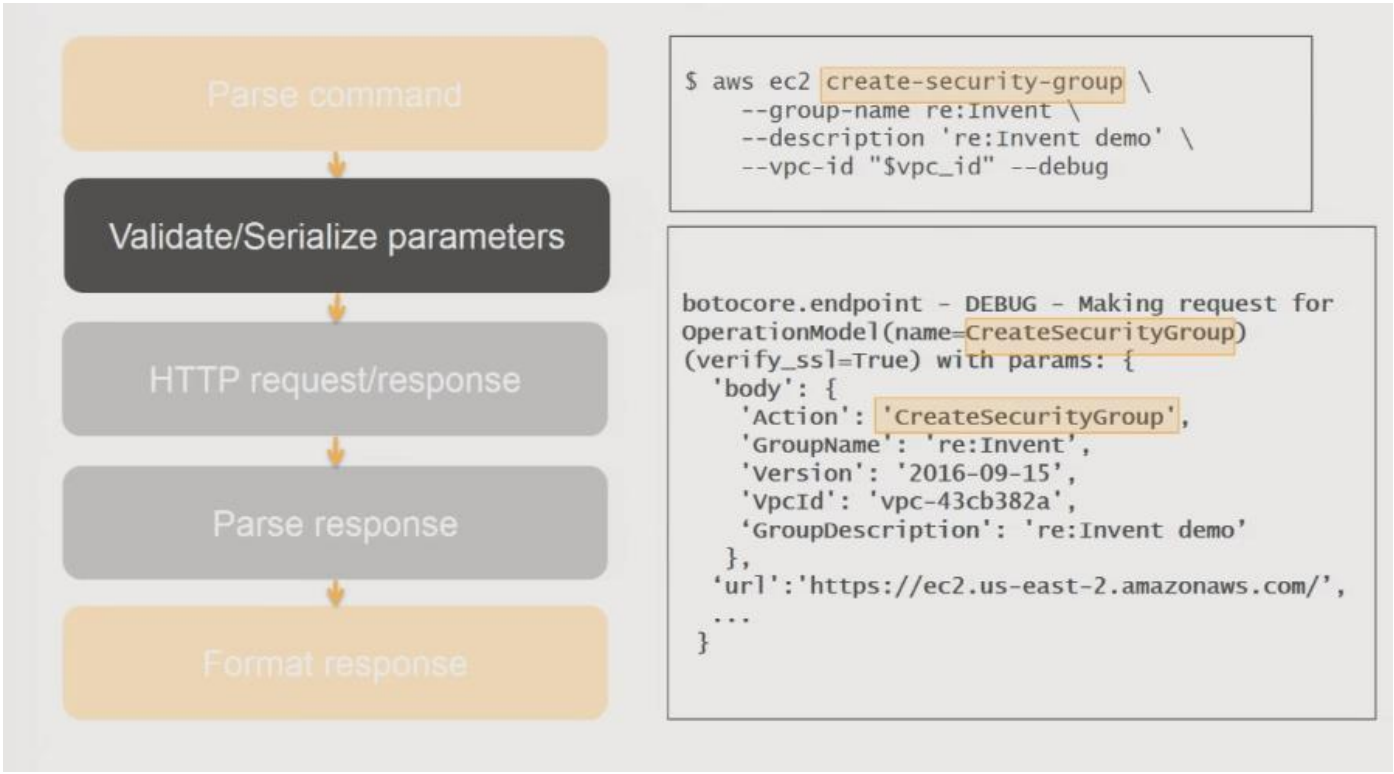
The debug logs follow the same direction as the CLI stack

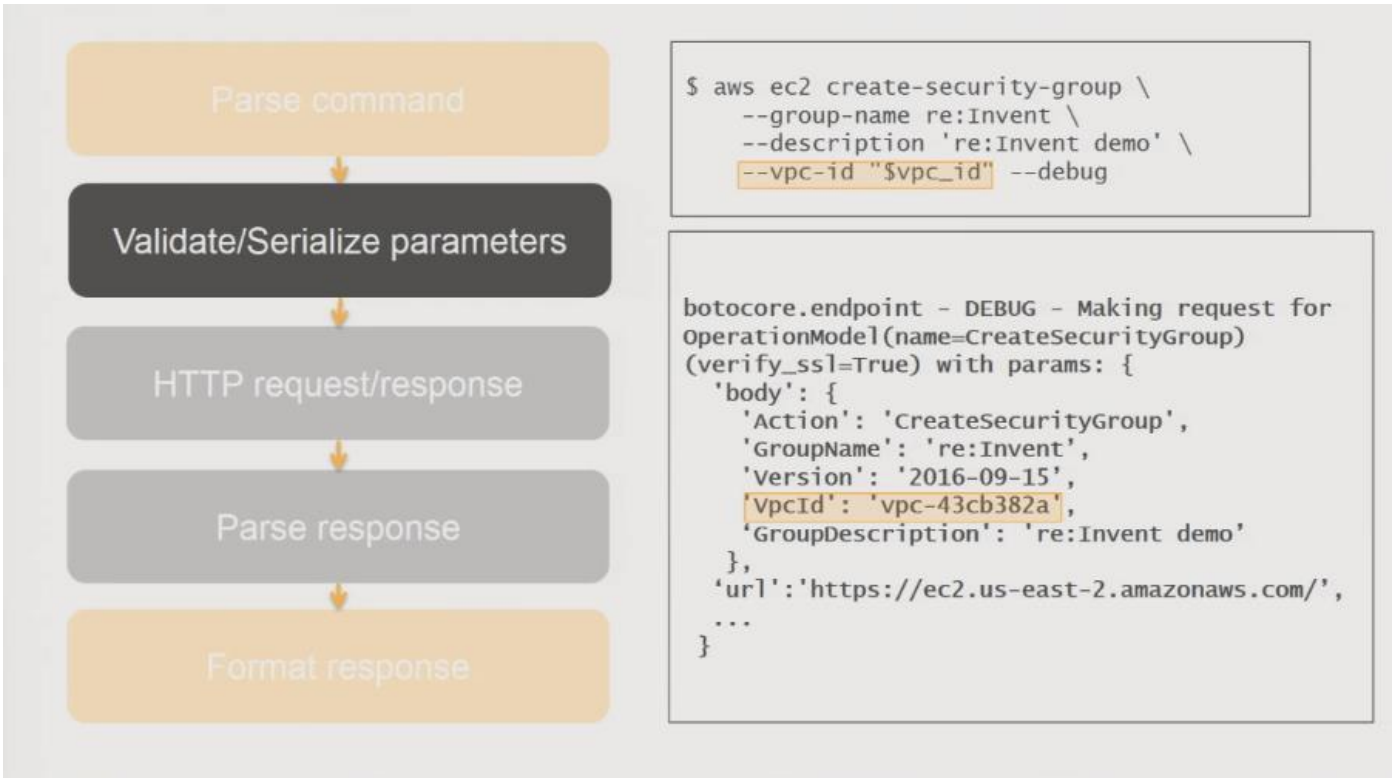


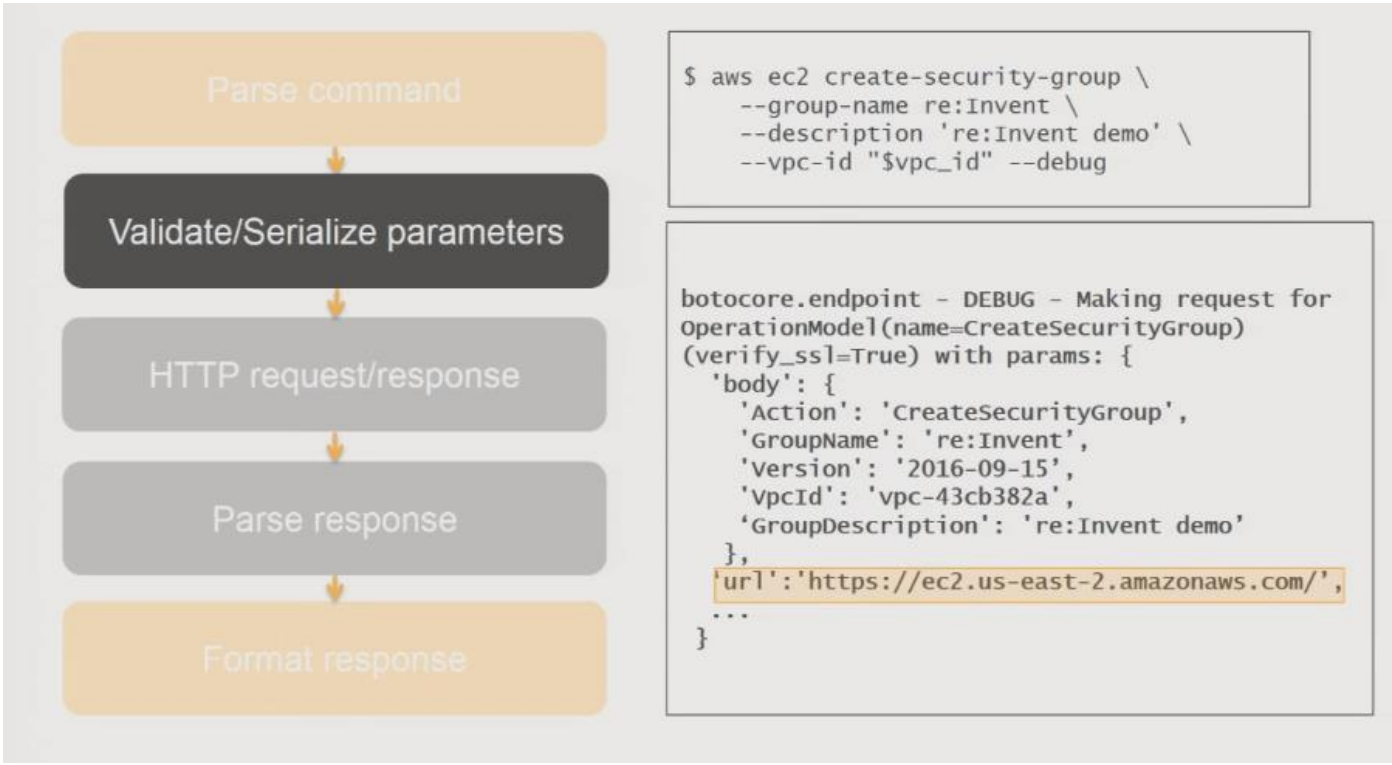
Let us now look at some debug logs at different stages in the CLI call stack

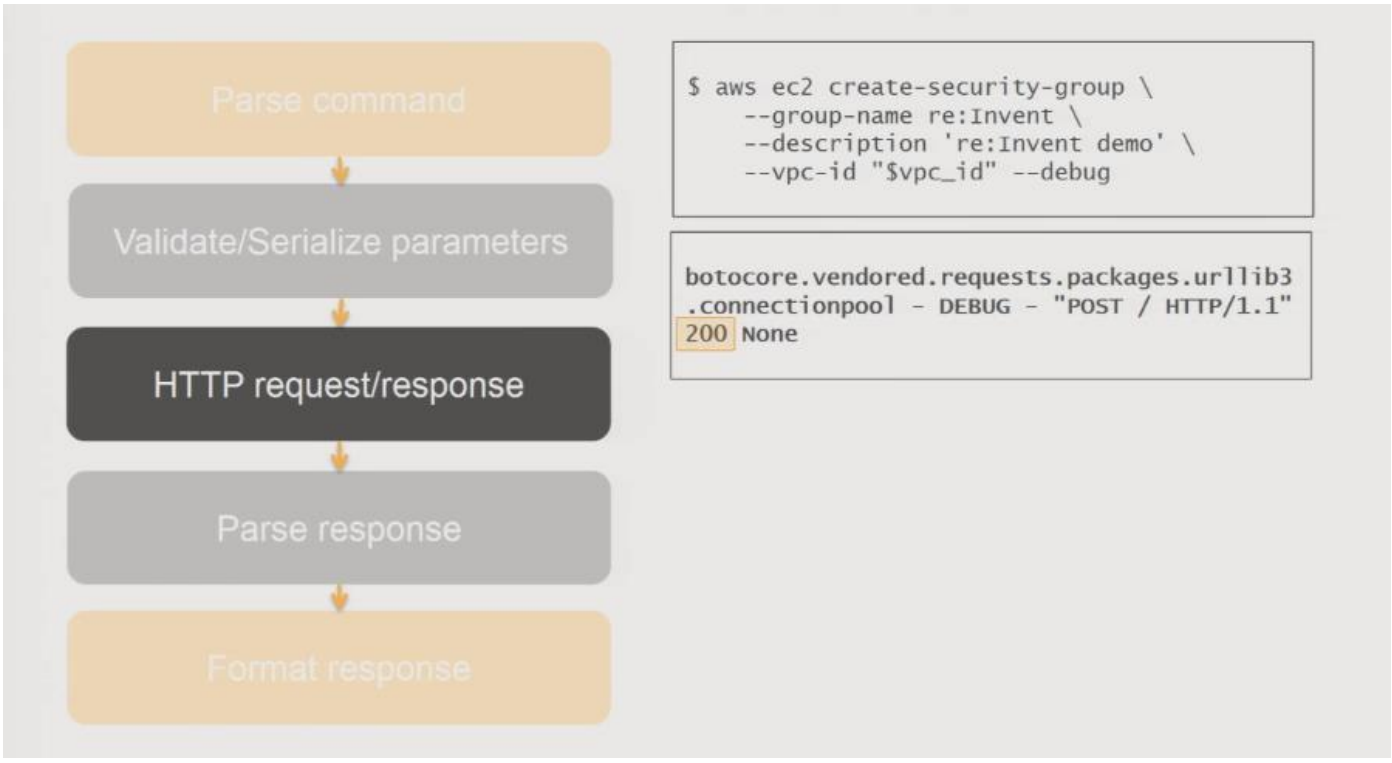

Parse command
Validate/Serialize parameters
HTTP request/response
Parse response
Format response
$ aws ec2 create-security-group \
--group-name re:Invent \
--description 're:Invent demo' \
--vpc-id "$vpc_id" --debug
awscli.clidriver - DEBUG - Arguments entered to CLI: ['ec2', 'create-security-group', '--group-name', 're:Invent', '--description', 're:Invent demo', '--vpc-id', 'vpc-43cb382a', '--debug']

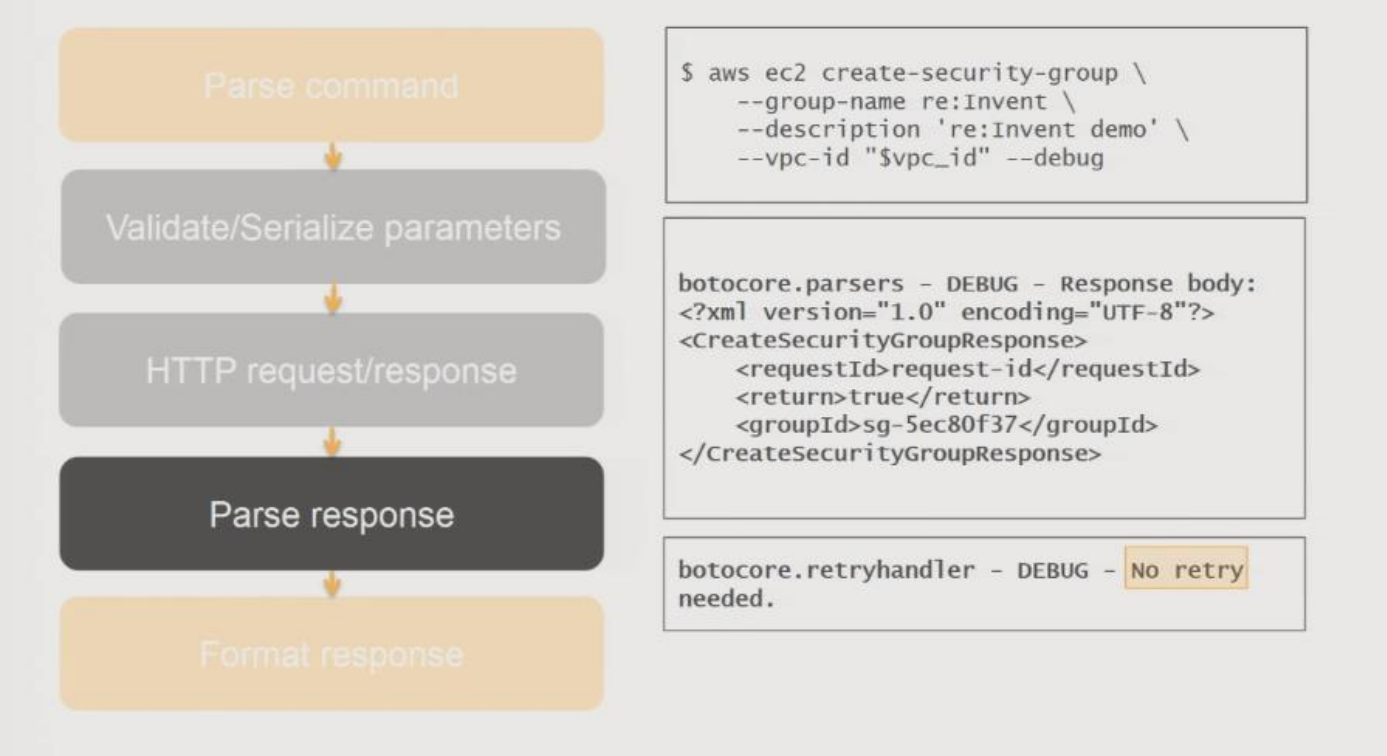

Parse command
Validate/Serialize parameters
HTTP request/response
Parse response
Format response
$ aws ec2 create-security-group \
--group-name re:Invent \
--description file://description.txt \
--vpc-id "$vpc_id" --debug
awscli.clidriver - DEBUG - Arguments entered to CLI: ['ec2', 'create-security-group', '--group-name', 're:Invent', '--description', 'file://description.txt', '--vpc-id', 'vpc-43cb382a', '--debug']
awscli.arguments - DEBUG - Unpacked value of u're:Invent demo' for parameter "description": u're:Invent demo'

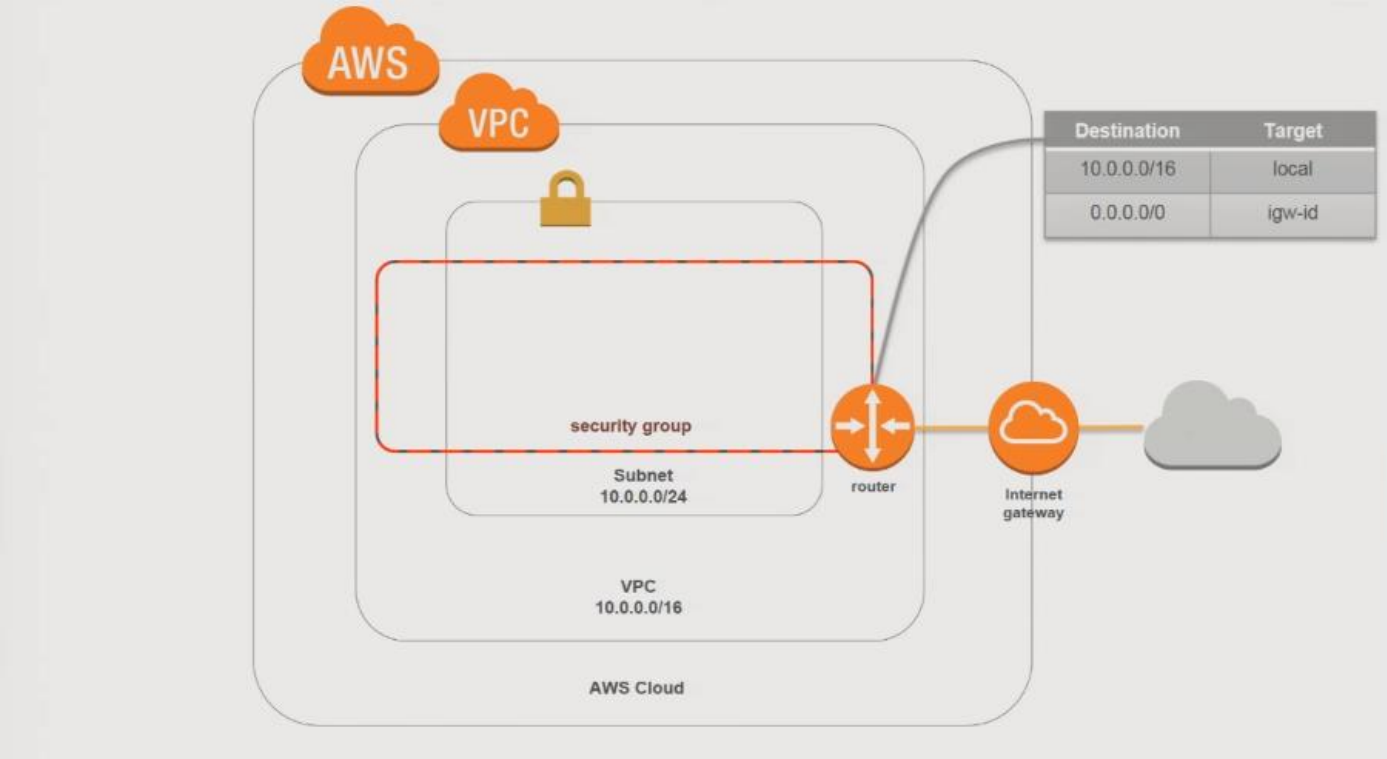

Parse command
Validate/Serialize parameters
HTTP request/response
Parse response
Format response
$ aws ec2 create-security-group \
--group-name re:Invent \
--description 're:Invent demo' \
--vpc-id "$vpc_id" --debug
botocore.endpoint - DEBUG - Making request for
OperationModel(name=CreateSecurityGroup)
(verify_ssl=True) with params: {
'body': {
'Action': 'CreateSecurityGroup',
'GroupName': 're:Invent',
'Version': '2016-09-15',
'VpcId': 'vpc-43cb382a',
'GroupDescription': 're:Invent demo'
},
'url':'https://ec2.us-east-2.amazonaws.com/',
...
}


Parse command
Validate/Serialize parameters
HTTP request/response
Parse response
Format response
$ aws ec2 create-security-group \
--group-name re:Invent \
--description 're:Invent demo' \
--vpc-id "$vpc_id" --debug
botocore.endpoint - DEBUG - Making request for
OperationModel(name=CreateSecurityGroup)
(verify_ssl=True) with params: {
'body': {
'Action': 'CreateSecurityGroup',
'GroupName': 're:Invent',
'Version': '2016-09-15',
'VpcId': 'vpc-43cb382a',
'GroupDescription': 're:Invent demo'
},
'url':'https://ec2.us-east-2.amazonaws.com/',
...
}


Parse command
Validate/Serialize parameters
HTTP request/response
Parse response
Format response
$ aws ec2 create-security-group \
--group-name re:Invent \
--description 're:Invent demo' \
--vpc-id "$vpc_id" --debug
botocore.endpoint - DEBUG - Making request for
OperationModel(name=CreateSecurityGroup)
(verify_ssl=True) with params: {
'body': {
'Action': 'CreateSecurityGroup',
'GroupName': 're:Invent',
'Version': '2016-09-15',
'VpcId': 'vpc-43cb382a',
'GroupDescription': 're:Invent demo'
},
'url':'https://ec2.us-east-2.amazonaws.com/',
...
}


Parse command
Validate/Serialize parameters
HTTP request/response
Parse response
Format response
$ aws ec2 create-security-group \
--group-name re:Invent \
--description 're:Invent demo' \
--vpc-id "$vpc_id" --debug
botocore.vendored.requests.packages.urllib3
.connectionpool - DEBUG - "POST / HTTP/1.1"
200 None





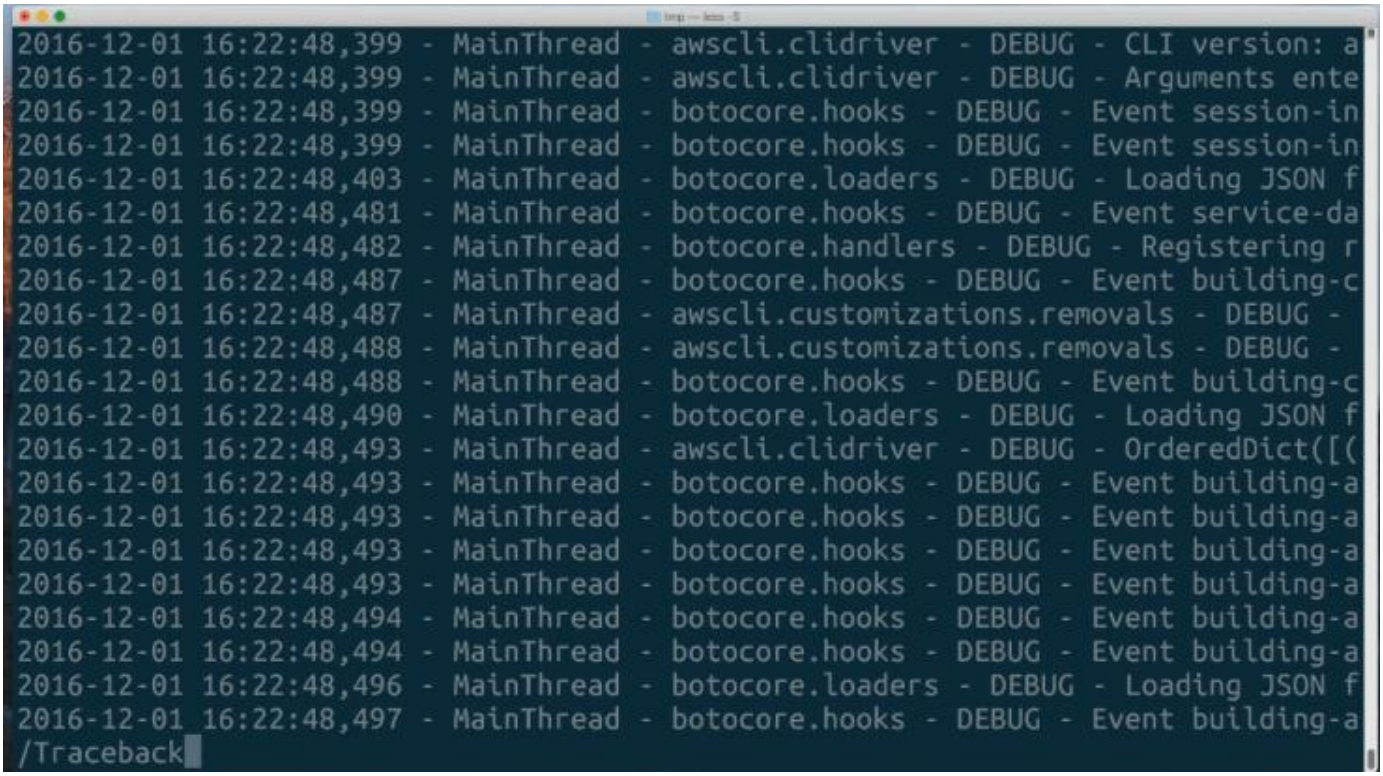
Let us now see a sample debugging scenario related to creating our security group

```
~/tmp$ aws configure set region us-west-1
~/tmp$ aws ec2 create-security-group --group-name reinvent --description reinven
t --vpc-id $vpc_id

An error occurred (InvalidVpcID.NotFound) when calling the CreateSecurityGroup o
peration: The vpc ID 'vpc-4101cb26' does not exist
~/tmp$ aws ec2 create-security-group --group-name reinvent --description reinven
t --vpc-id $vpc_id --debug
```

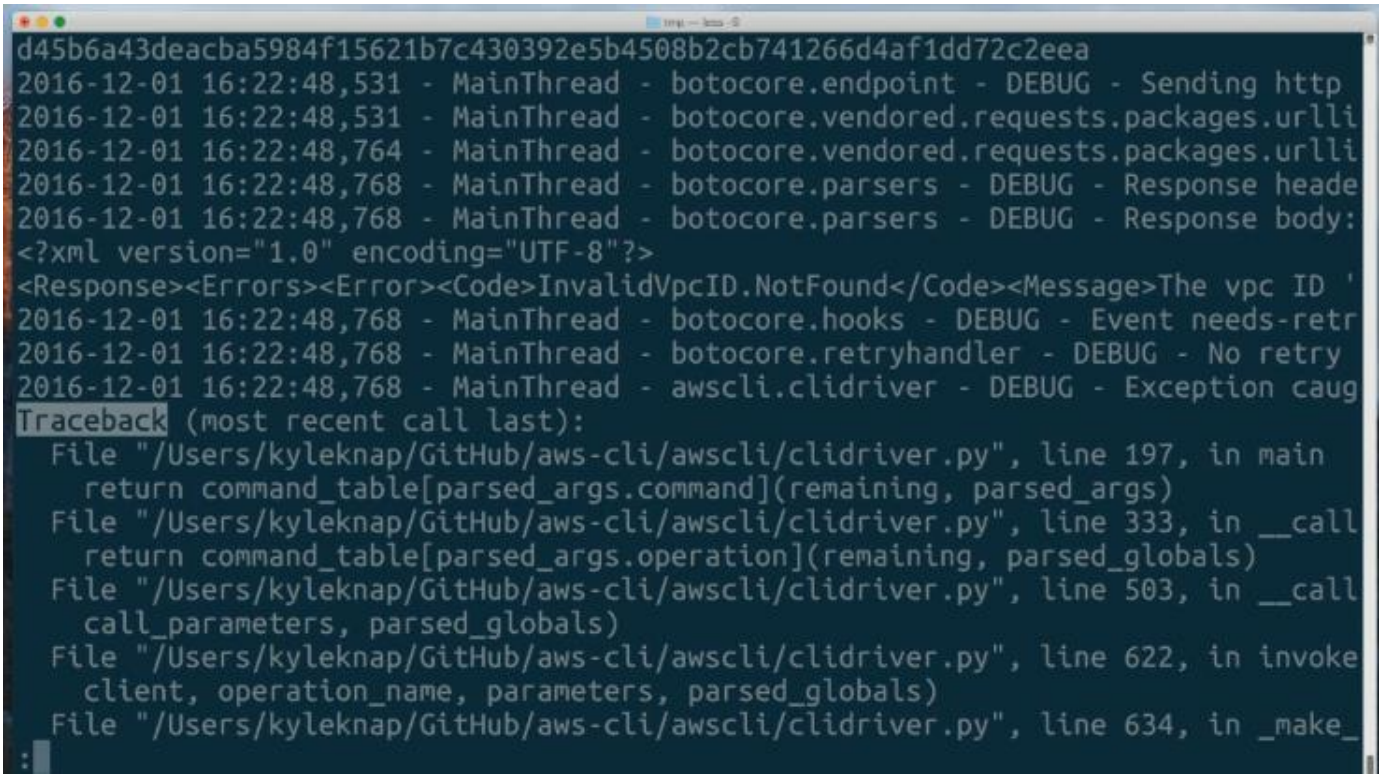
We get this exception when trying to create a SG in the VPC, but we had switched the region to another region from us-west-2 where the VPC is located in. We now need to debug this error log using the command ***$ aws ec2 create-security-group - -group-name reinvent - -description reinvent - -vpc-id $vpc_id - -debug***

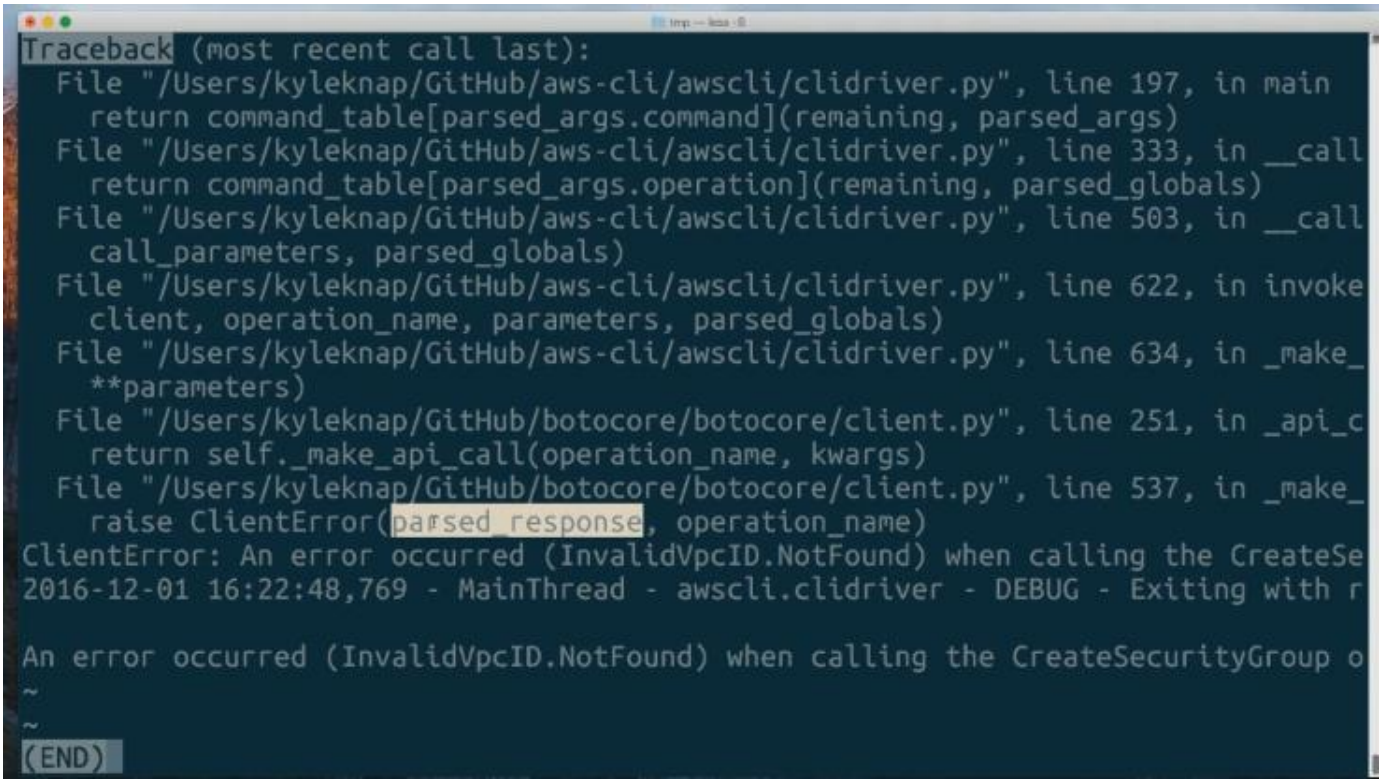
```
--
    call_parameters, parsed_globals)
  File "/Users/kyleknap/GitHub/aws-cli/awscli/clidriver.py", line 622, in invoke
    client, operation_name, parameters, parsed_globals)
  File "/Users/kyleknap/GitHub/aws-cli/awscli/clidriver.py", line 634, in _make_
client_call
    **parameters)
  File "/Users/kyleknap/GitHub/botocore/botocore/client.py", line 251, in _api_c
all
    return self._make_api_call(operation_name, kwargs)
  File "/Users/kyleknap/GitHub/botocore/botocore/client.py", line 537, in _make_
api_call
    raise ClientError(parsed_response, operation_name)
ClientError: An error occurred (InvalidVpcID.NotFound) when calling the CreateSe
curityGroup operation: The vpc ID 'vpc-4101cb26' does not exist
2016-12-01 16:22:31,496 - MainThread - awscli.clidriver - DEBUG - Exiting with r
c 255

An error occurred (InvalidVpcID.NotFound) when calling the CreateSecurityGroup o
peration: The vpc ID 'vpc-4101cb26' does not exist
~/tmp$ aws ec2 create-security-group --group-name reinvent --description reinven
t --vpc-id $vpc_id --debug 2>&1 | less -S
```

There is a lot of error logs here, we can display the logs in lines using the ***$ aws ec2 create-security-group - -group-name reinvent - -description reinvent - -vpc-id $vpc_id - -debug 2>&1 | less -S*** command to see the output in a pager such as ***less*** below

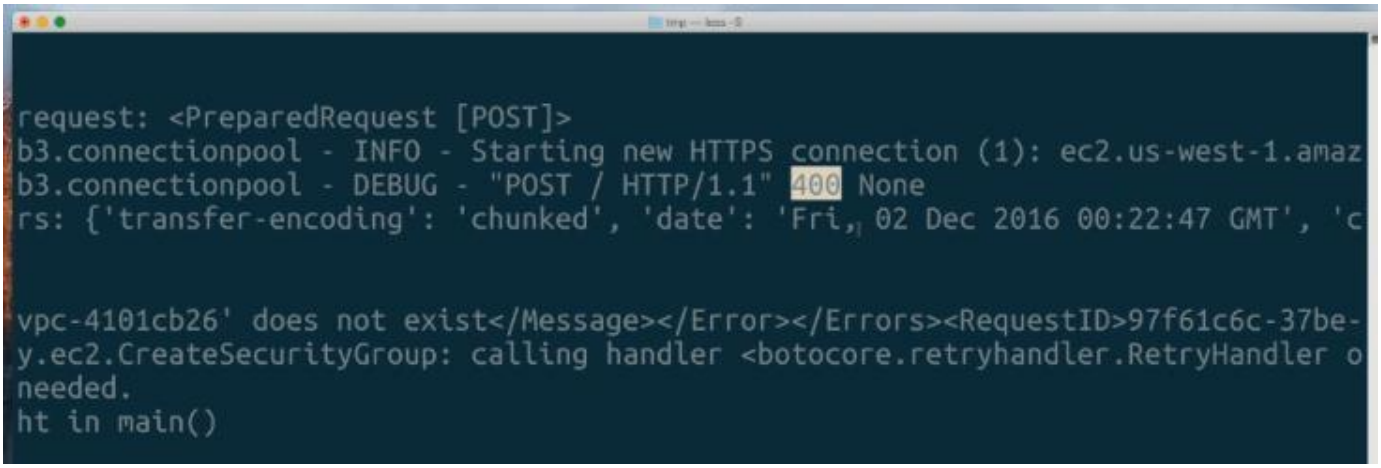


We now have the logs in lines. Whenever there is an error and you look in debug mode, there is always a Traceback that we can check





The Traceback is at the very end of the stack.

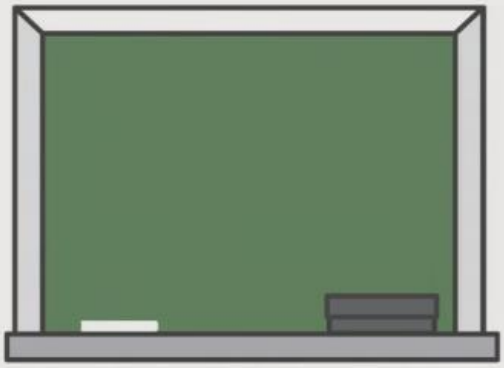


This is pointing at the wrong region that doesn't have our VPC

We then switch back the region to us-west-2 and re-run the ***$ groud_id=$(aws ec2 create-security-group - -group-name reinvent - -description reinvent - -vpc-id $vpc_id - -query GroupId - -output text)*** command again, then save the result into a variable and echo the result just to be sure

Important Takeaways

- `--debug`
- Search for `Traceback`'s
- Leverage knowledge of AWS CLI architecture



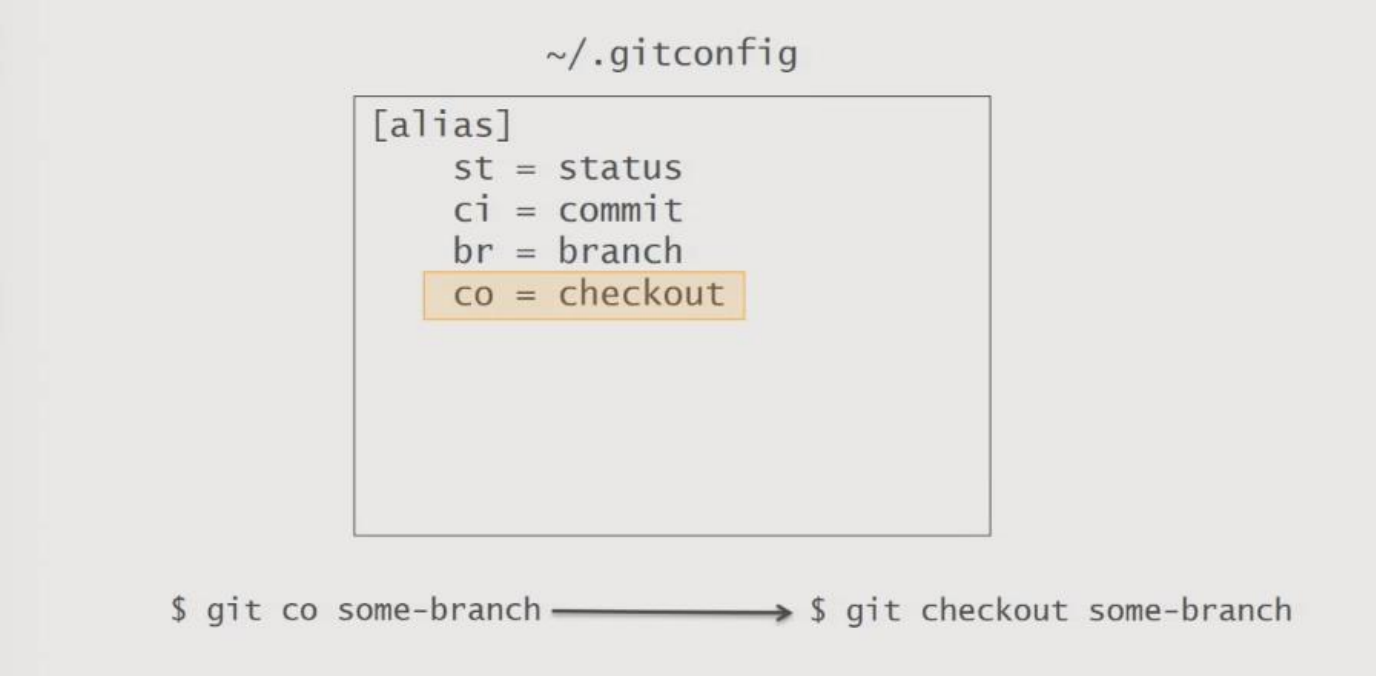
The Effective AWS CLI User Tenets

The effective AWS CLI user:

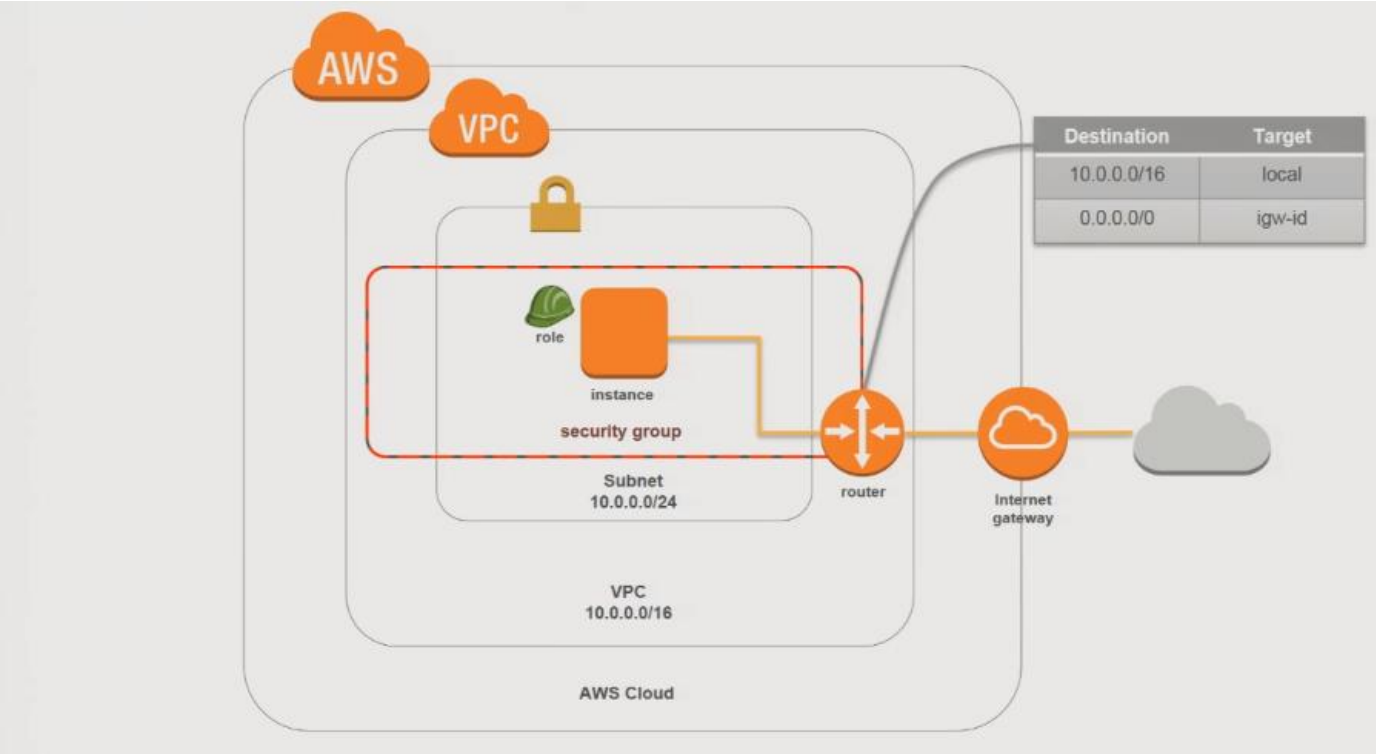
- Uses an iterative workflow
- Troubleshoots well
- Is resourceful with tooling
- Understands performance implications

- BASH: variables, pipes, functions
- UNIX commands: `xargs`, `sort`, `tail`
- External tools: `jp`, `jpterm`, `aws-shell`
- DIY (Do it yourself) tooling

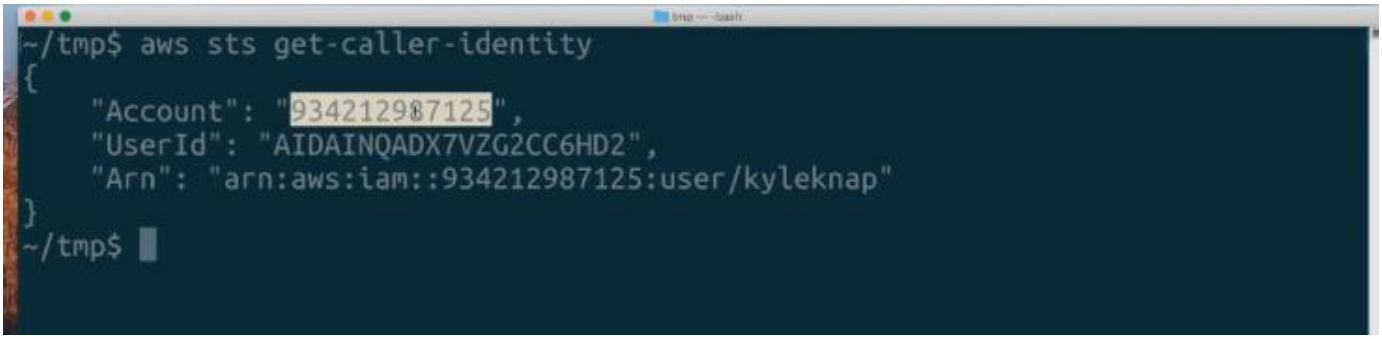
`alias`



The AWS CLI alias command is similar to git alias.

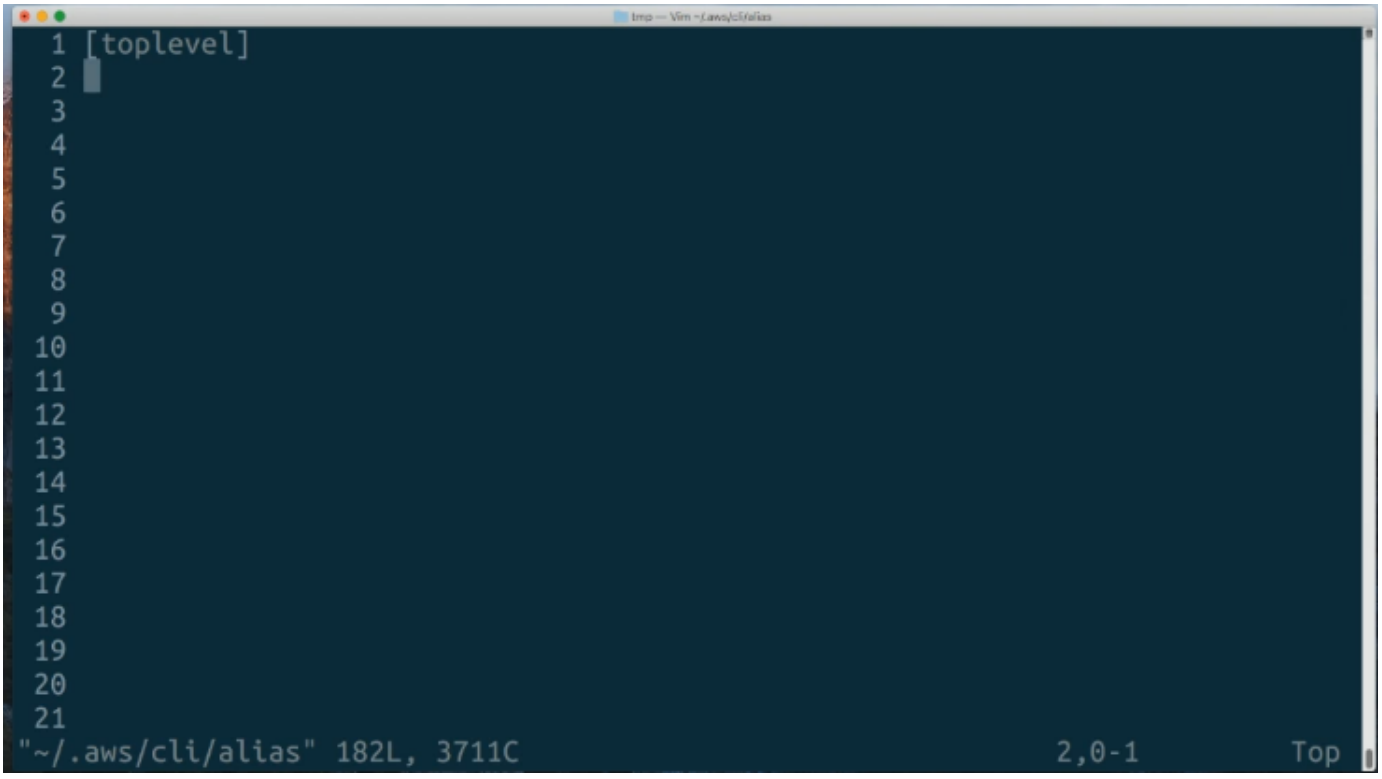


We are going to create an EC2 instance and a Role, then we are going to launch the EC2 instance into our VPC.



The ***$ aws sts get-caller-***identity command tells you what user actually made that request. The ***Account*** value is useful for creating Arns, granting permissions, creating policies. The only other way to get the Account value is to log into the web console or try to parse it out of the ***Arn*** value.

To be able to add the Account as an alias, you need to open up the AWS CLI alias file using the ***$ vim ~/.aws/cli/alias*** command as above



We can now add the Account as an alias by editing this file as below

```
[toplevel]
whoami = sts get-caller-identity
-- INSERT --                                              2,33          Top
```

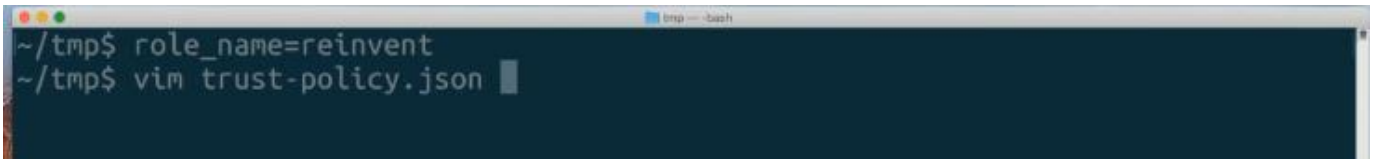
We add in the line ***whoami = sts get-caller-identity*** and save the file and exit

```
~/tmp$ aws sts get-caller-identity
{
    "Account": "934212987125",
    "UserId": "AIDAINQADX7VZG2CC6HD2",
    "Arn": "arn:aws:iam::934212987125:user/kyleknap"
}
~/tmp$ vim ~/.aws/cli/alias
~/tmp$ aws whoami
{
    "Account": "934212987125",
    "UserId": "AIDAINQADX7VZG2CC6HD2",
    "Arn": "arn:aws:iam::934212987125:user/kyleknap"
}
~/tmp$ aws whoami --query Account --output text
934212987125
~/tmp$ vim ~/.aws/cli/alias
```

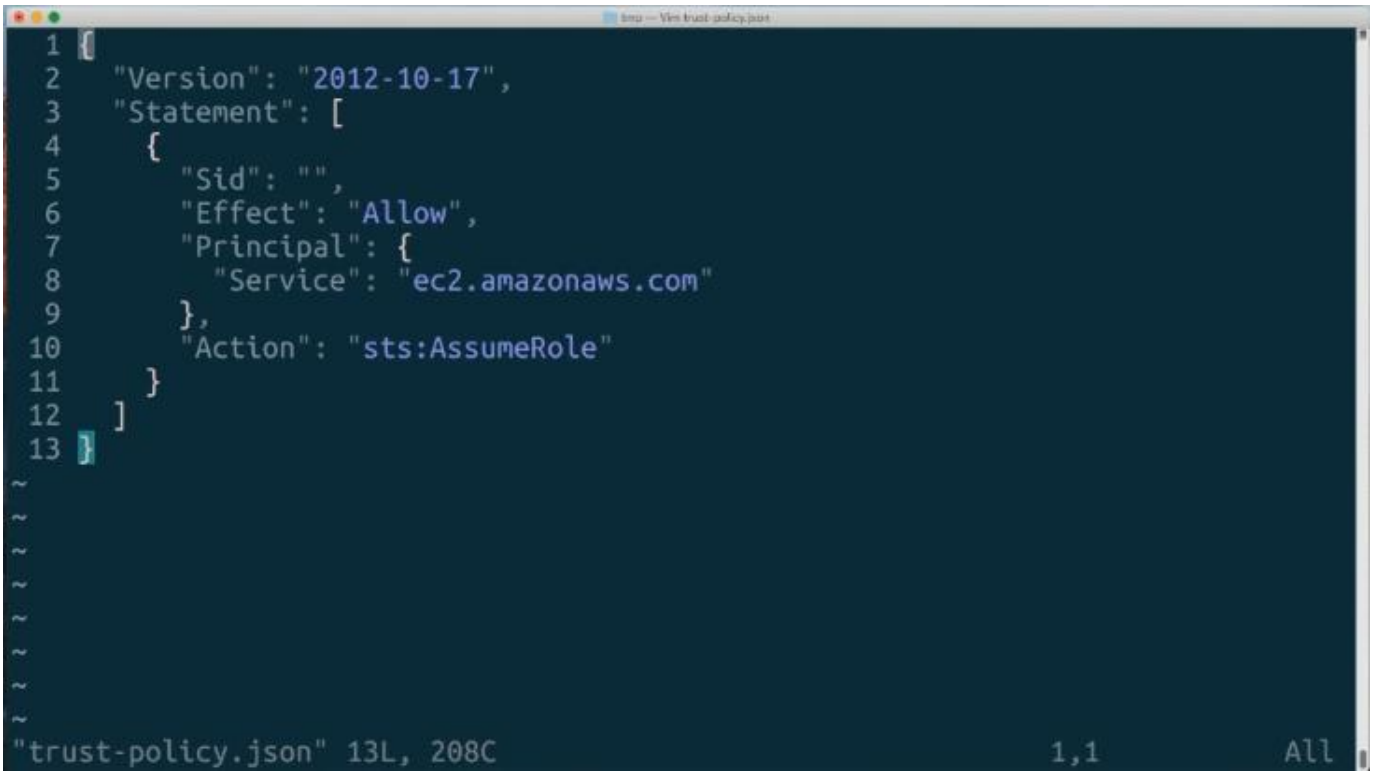
We can now use the ***$ aws whoami*** command to get the same value back from the git caller identity as above. We can also query the response too and then add the query to our alias file also

```
[toplevel]
whoami = sts get-caller-identity --query Account --output tex
```

We now get the unquoted Account value back when we use the ***$ aws whoami*** command. Next, let us see some deeper functionality that we can get when we use aliases.



Let us create an IAM Role, for this we need a role name and a trust policy which we already have on disk



The trust policy allows the EC2 service to assume the role

We can now create the IAM role using the ***$ aws iam create-role - -role-name $role_name - -assume-role-policy-document file://trust-policy.json*** command as above. But we can use an alias here for 2 reasons, the first reason is because you always have to reference the file on disk when running this command, the second reason is that if we want to have a different service to assume the role we will have to go in and modify the trust policy file. It will be better if we can instead specify what service we want to assume the role when giving the command.

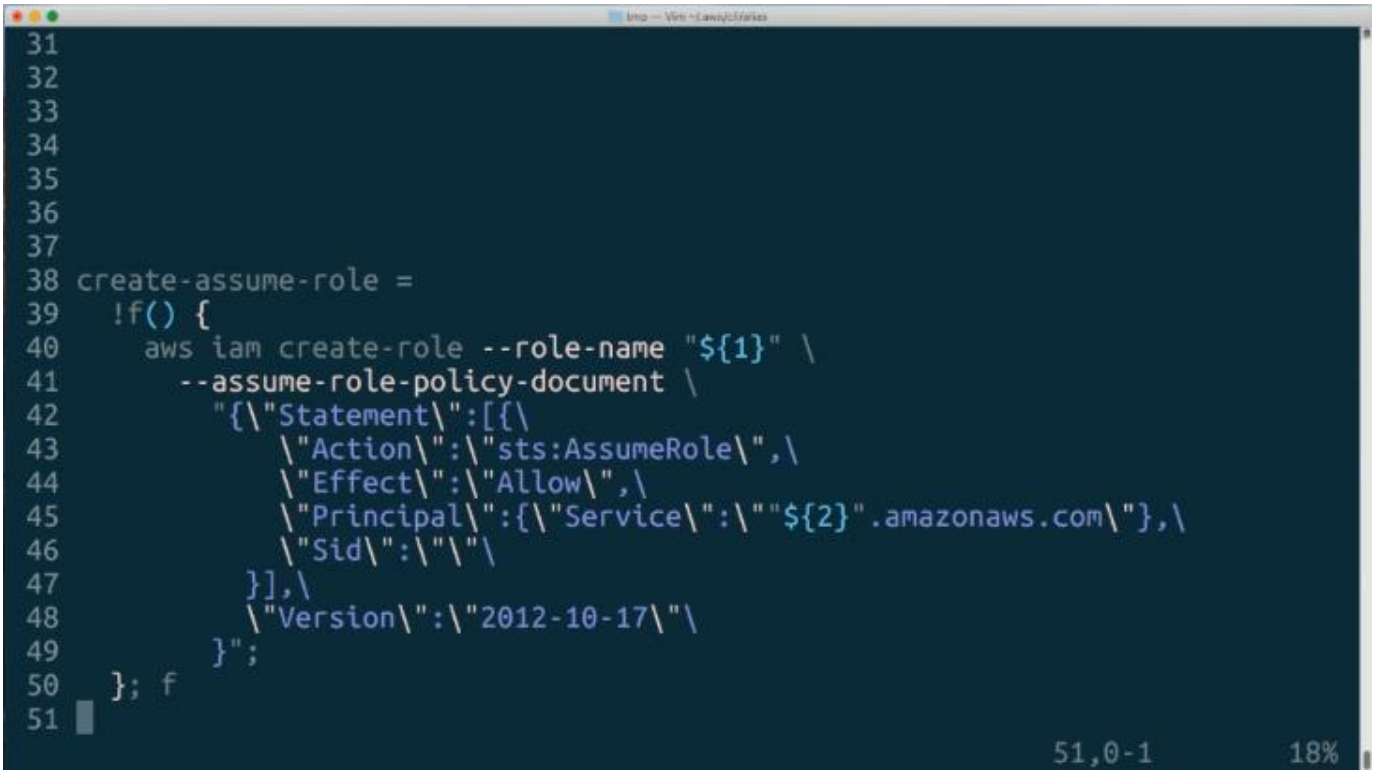
```
~/tmp$ role_name=reinvent
~/tmp$ vim trust-policy.json
~/tmp$ aws create-assume-role $role_name ec2
```

Using the ***$ aws create-assume-role $role_name ec2*** command, we can create another alias called ***create-assume-role*** that will require us to pass in the ***role_name*** and the service we want to assume the role such as ***ec2***.

```
~/tmp$ role_name=reinvent
~/tmp$ vim trust-policy.json
~/tmp$ aws create-assume-role $role_name ec2
{
    "Role": {
        "AssumeRolePolicyDocument": {
            "Version": "2012-10-17",
            "Statement": [
                {
                    "Action": "sts:AssumeRole",
                    "Sid": "",
                    "Effect": "Allow",
                    "Principal": {
                        "Service": "ec2.amazonaws.com"
                    }
                }
            ]
        },
        "RoleId": "AROAIA6CAQM5IQSRPTI7W",
```

```
        "AssumeRolePolicyDocument": {
            "Version": "2012-10-17",
            "Statement": [
                {
                    "Action": "sts:AssumeRole",
                    "Sid": "",
                    "Effect": "Allow",
                    "Principal": {
                        "Service": "ec2.amazonaws.com"
                    }
                }
            ]
        },
        "RoleId": "AROAIA6CAQM5IQSRPTI7W",
        "CreateDate": "2016-12-02T00:31:47.196Z",
        "RoleName": "reinvent",
        "Path": "/",
        "Arn": "arn:aws:iam::934212987125:role/reinvent"
    }
}
~/tmp$ vim ~/.aws/cli/alias
```

We now have the alias created as above with the role name and service inserted into the alias document. We can then see the alias file using the ***$ vim ~/.aws/cli/alias*** command



Note that we are still calling the ***aws iam create-role*** but the CLI command is now wrapped by a bash function that is prefixed by the exclamation mark on line 39, this tells the CLI to run this command as an external command and to sub-process it to the bash shell. This allows it to map the 2 needed arguments as ***${1}*** for the role name and ***${2}*** for the service on lines 40 and 45 to be passed in when running the command.

```
~/tmp$ jp -f trust-policy.json "to_string(@)"
"{\"Statement\":[{\"Action\":\"sts:AssumeRole\",\"Effect\":\"Allow\",\"Principal
\":{\"Service\":\"ec2.amazonaws.com\"},\"Sid\":\"\"}],\"Version\":\"2012-10-17\"
}"
~/tmp$
```

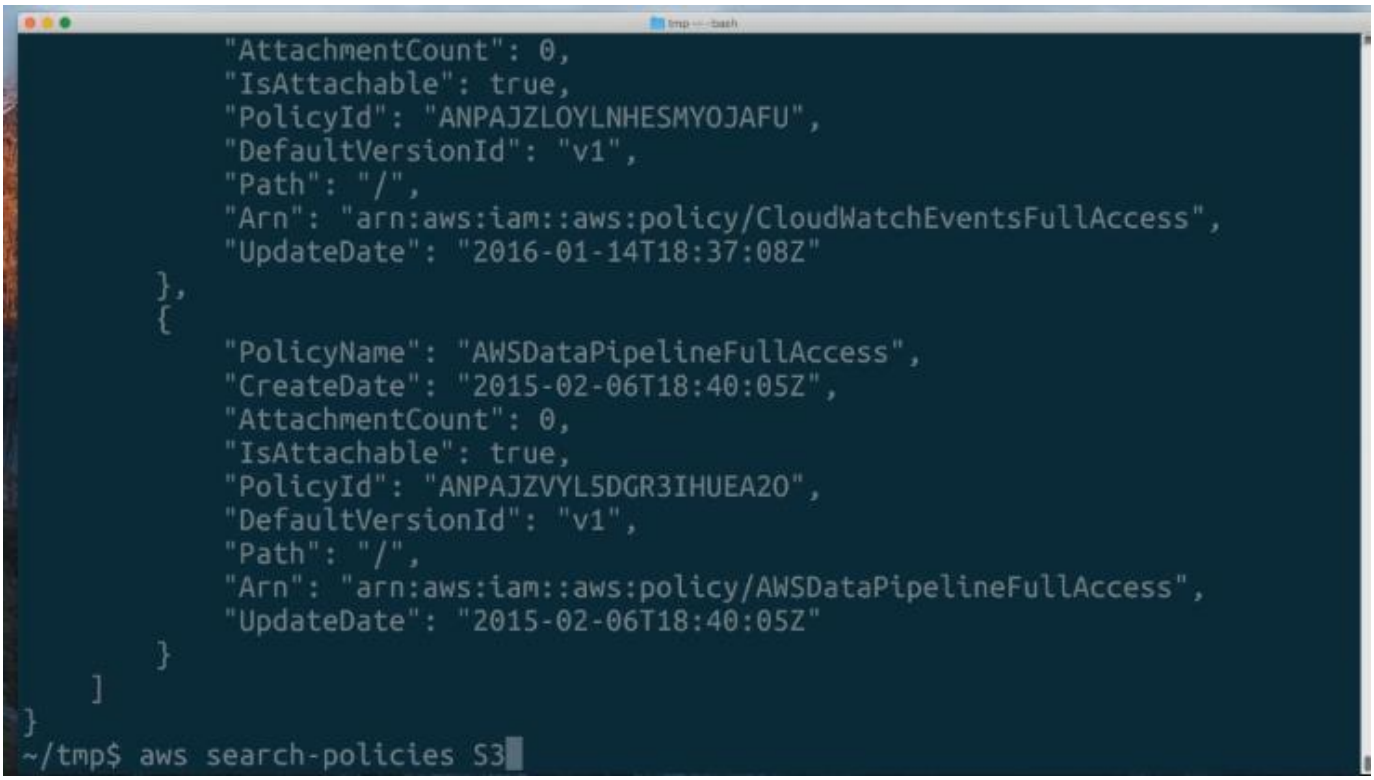
We can use JP tool and the ***$ jp -f trust-policy.json "to_string(@)"*** command to escape the quotes in the trust-policy.json file to convert the file into a string

```
                "Sid": "",
                "Effect": "Allow",
                "Principal": {
                    "Service": "ec2.amazonaws.com"
                }
            }
        ]
    },
    "RoleId": "AROAIA6CAQM5IQSRPTI7W",
    "CreateDate": "2016-12-02T00:31:47.196Z",
    "RoleName": "reinvent",
    "Path": "/",
    "Arn": "arn:aws:iam::934212987125:role/reinvent"
  }
}
~/tmp$ vim ~/.aws/cli/alias
~/tmp$ jp -f trust-policy.json "to_string(@)"
"{\"Statement\":[{\"Action\":\"sts:AssumeRole\",\"Effect\":\"Allow\",\"Principal
\":{\"Service\":\"ec2.amazonaws.com\"},\"Sid\":\"\"}],\"Version\":\"2012-10-17\"
}"
~/tmp$ aws iam list-policies
```

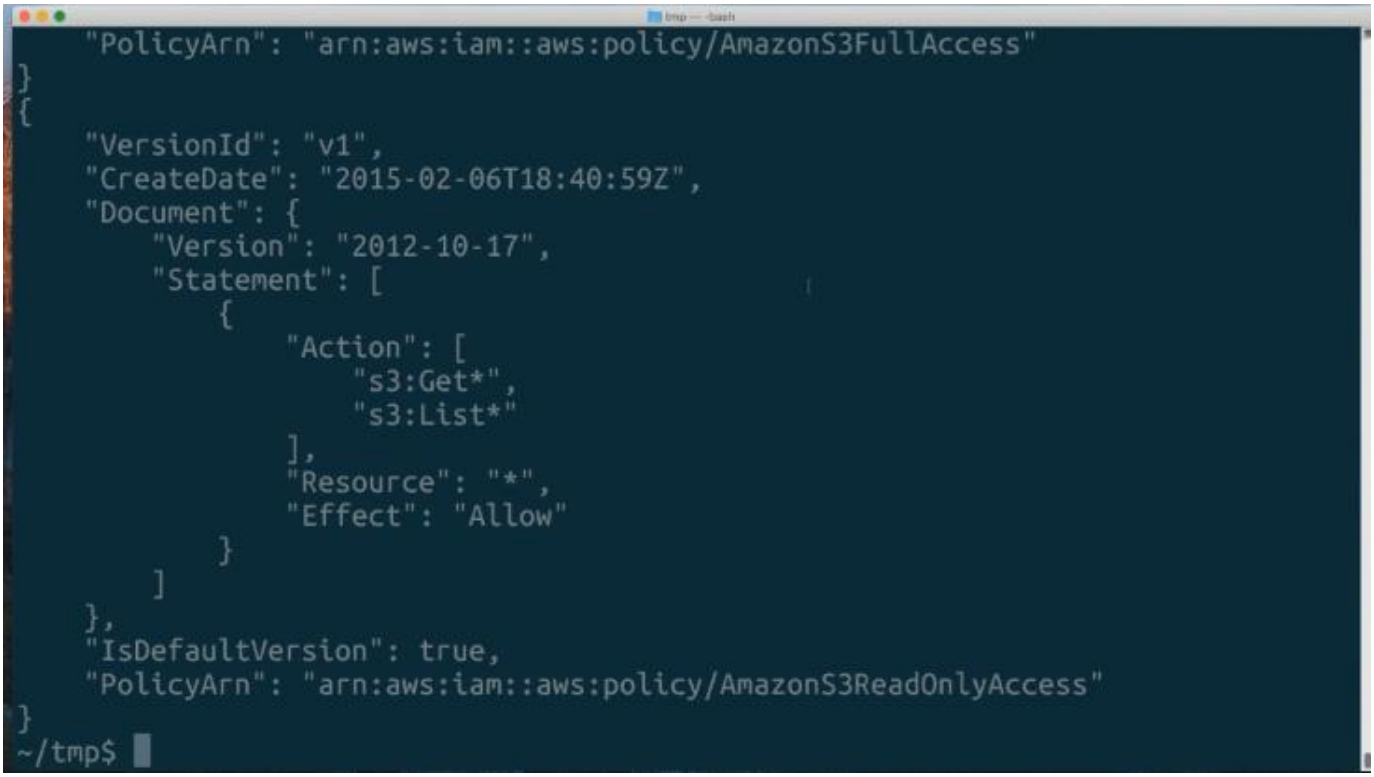
We now need to figure out what policy we want to assign the role we just create, like a policy to allow access to S#. We run the ***$ aws iam list-policies*** command to get the long list of available IAM policies

```
            "IsAttachable": true,
            "PolicyId": "ANPAJYCHABBP6VQIVBCBQ",
            "DefaultVersionId": "v1",
            "Path": "/",
            "Arn": "arn:aws:iam::aws:policy/AWSCertificateManagerFullAccess",
            "UpdateDate": "2016-01-21T17:02:36Z"
        },
        {
            "PolicyName": "PowerUserAccess",
            "CreateDate": "2015-02-06T18:39:47Z",
            "AttachmentCount": 0,
            "IsAttachable": true,
            "PolicyId": "ANPAJYRXTHIB4FOVS3ZXS",
            "DefaultVersionId": "v1",
            "Path": "/",
            "Arn": "arn:aws:iam::aws:policy/PowerUserAccess",
            "UpdateDate": "2015-02-06T18:39:47Z"
        },
        {
```

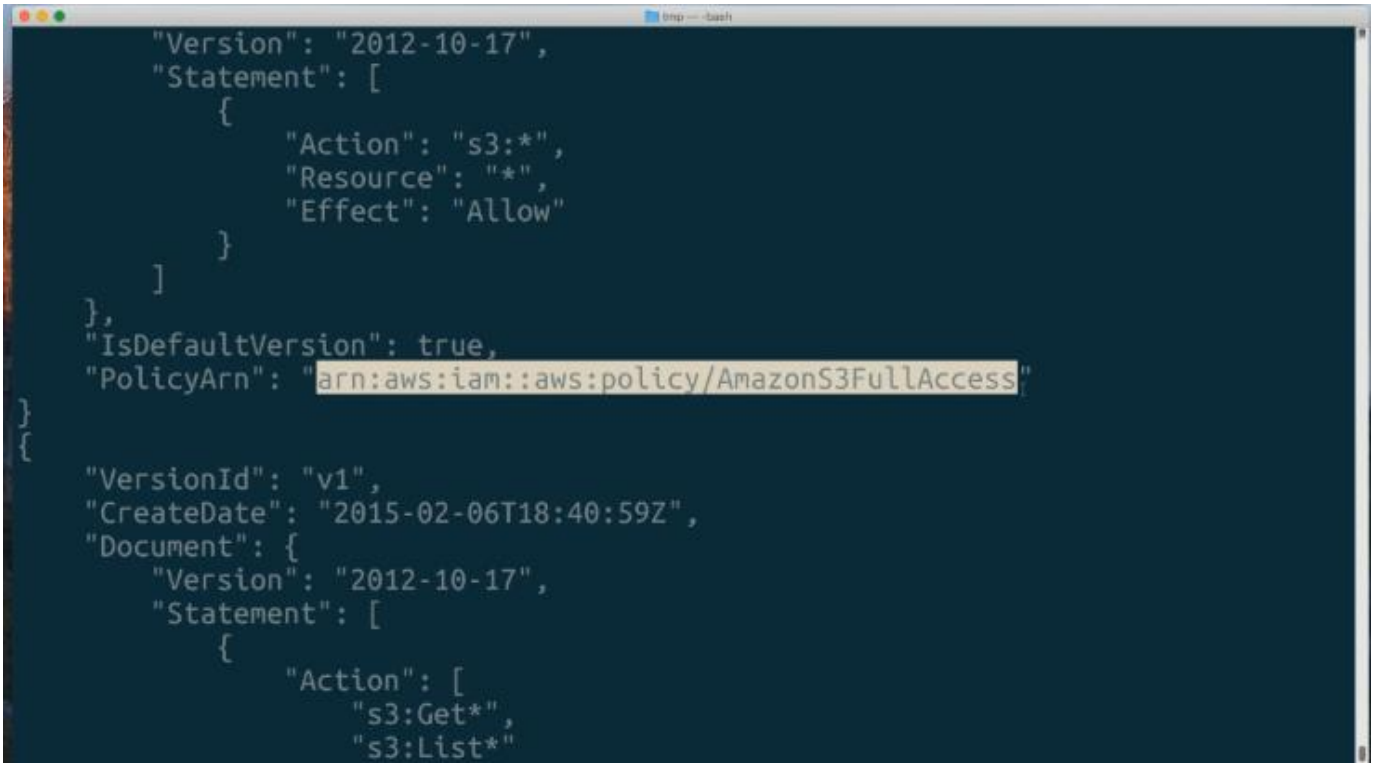
Most of these policies are not related to S3, so we need to filter the list for policy names that contain S3



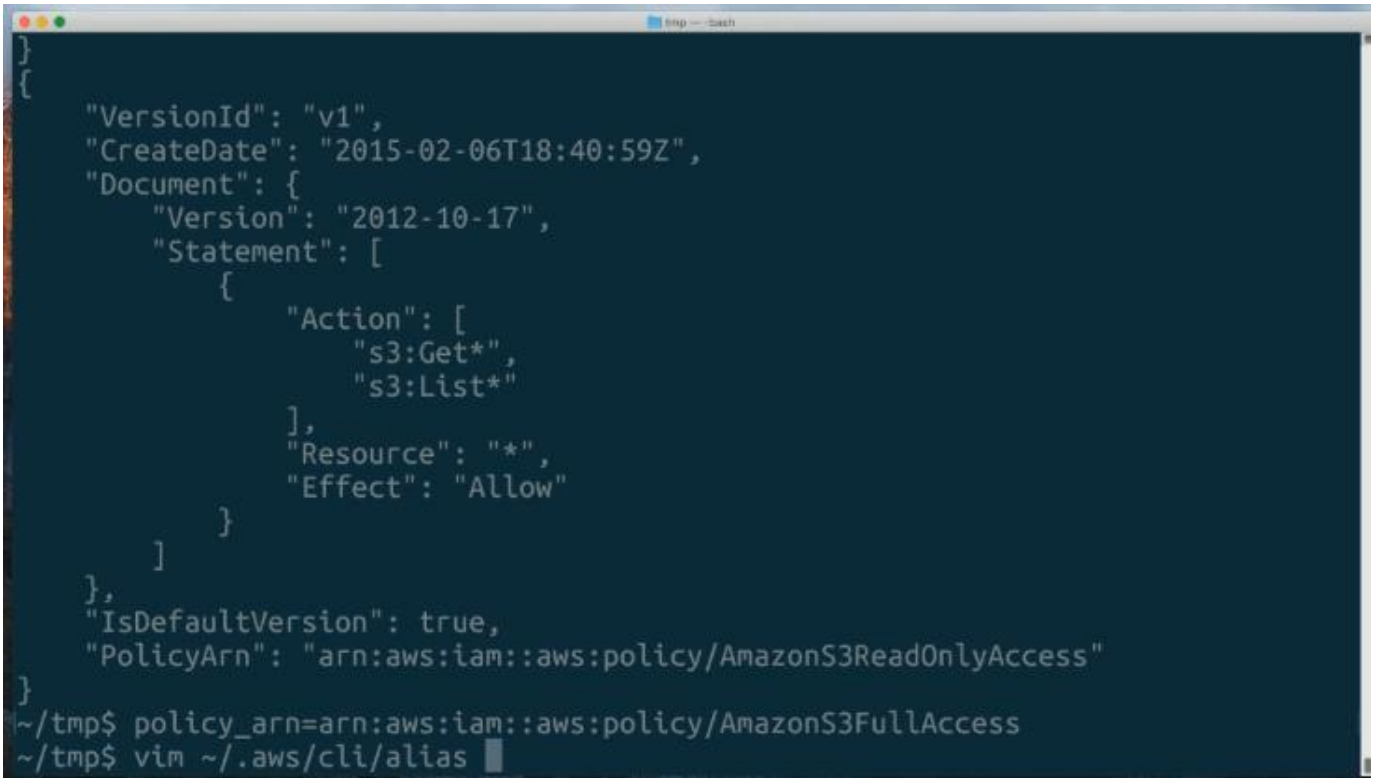
We have created an alias called ***search-policies*** that will search for all the policy names that contain the ***argument*** value of 'S3'. We then use the command ***$ aws search-policies S3***



We can now see all the S3 related policies



We want to use the ***AmazonS3FullAccess*** policy above,



We then save the ***arn*** value into a variable policy_arn using ***policy_arn=arn:aws:iam::aws:policy/AmazonS3FullAccess*** command

```
toplevel
whoami = sts get-caller-identity --query Account --output text
"~/.aws/cli/alias" 182L, 3773C                                1,1           Top
```

We can now show what the search-policies alias looks like

```
          \"Version\":\"2012-10-17\"\
        }";
  }; f


search-policies =
  !f() {
    aws iam list-policies \
      --query "Policies[?contains(PolicyName, \`"${1}"\`)] \
               | [*].[to_string({PolicyArn: Arn, \
                                 VersionId: DefaultVersionId})]" \
      --output text \
    | gxargs -d '\n' -I"params" \
        aws iam get-policy-version \
          --cli-input-json "params" \
          --query "merge(PolicyVersion,\`"params"\`)";
  }; f
```

This shows that we can have more than one CLI command in an alias file as seen above, whatever your shell supports.

```
~/tmp$ aws iam attach-role-policy --role-name $role_name --policy-arn $policy_ar
n
~/tmp$
```

We then attach the policy to our role using the command ***$ aws iam attach-role-policy - -role-name $role_name - - policy-arn $policy_arn***

```
~/tmp$ aws setup-instance-profile $role_name
{
    "InstanceProfile": {
        "InstanceProfileId": "AIPAITMIUDJ6YELWTH4HC",
        "Roles": [],
        "CreateDate": "2016-12-02T00:37:00.970Z",
        "InstanceProfileName": "reinvent",
        "Path": "/",
        "Arn": "arn:aws:iam::934212987125:instance-profile/reinvent"
    }
}
~/tmp$
```

We now need to create an EC2 instance profile from EC2 for the role, we are going to use another alias called ***setup-instance-profile*** that will take an EC2 instance profile using the argument role name to add the role to the instance profile. This is how we create the ***reinvent*** InstanceProfileName above.

```
~/tmp$ aws setup-instance-profile $role_name
{
    "InstanceProfile": {
        "InstanceProfileId": "AIPAITMIUDJ6YELWTH4HC",
        "Roles": [],
        "CreateDate": "2016-12-02T00:37:00.970Z",
        "InstanceProfileName": "reinvent",
        "Path": "/",
        "Arn": "arn:aws:iam::934212987125:instance-profile/reinvent"
    }
}
~/tmp$ vim ~/.aws/cli/alias
```

We now have all we need to start launching our EC2 instances in the VPC, we also created another alias for this purpose called ***launch-reinvent***

```
 1 [toplevel]
 2 whoami = sts get-caller-identity --query Account --output text
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
16
17
18
19
20
21
"~/.aws/cli/alias" 182L, 3773C                                1,1           Top
```

```
    role_name="$1"
    aws iam create-instance-profile --instance-profile $role_name
    aws iam add-role-to-instance-profile \
        --instance-profile $role_name --role-name $role_name
}; f



launch-reinvent =
   ec2 run-instances --image-id ami-8a50feea --key-name us_west_2_key \
     --associate-public-ip-address --query Instances[0].[InstanceId] \
     --output text
```

In the ***launch-reinvent*** alias, we are not using any external commands but instead using the EC2 sub-command ***run-instances***.

```
~/tmp$ id_1=$(aws launch-reinvent --instance-type m4.large --security-group-ids
$group_id --iam-instance-profile Name=$role_name --subnet-id $subnet_id)
~/tmp$ echo $id_1
i-0ef142f10385a85dc
~/tmp$
```

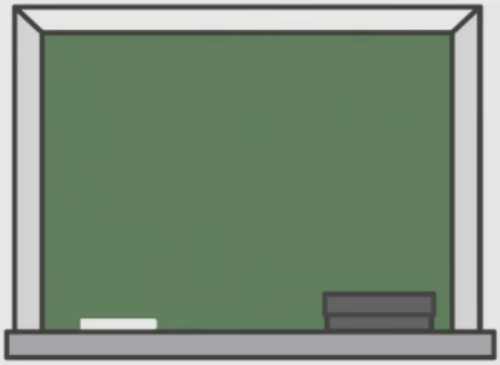
We then launch the EC2 instance using the ***$ id_1=$(aws launch-reinvent - -instance-type m4.large - -security-group-ids $group_id - -iam-instance-profile Name=$role_name - -subnet_id $subnet_id)*** command and also saved the EC2 instanceId in a variable called ***id_1***.

```
~/tmp$ id_1=$(aws launch-reinvent --instance-type m4.large --security-group-ids
$group_id --iam-instance-profile Name=$role_name --subnet-id $subnet_id)
~/tmp$ echo $id_1
i-0ef142f10385a85dc
~/tmp$ id_2=$(aws launch-reinvent --instance-type m4.4xlarge --security-group-id
s $group_id --iam-instance-profile Name=$role_name --subnet-id $subnet_id)
~/tmp$ echo $id_2
i-04cba0c935b0dcda3
~/tmp$
```

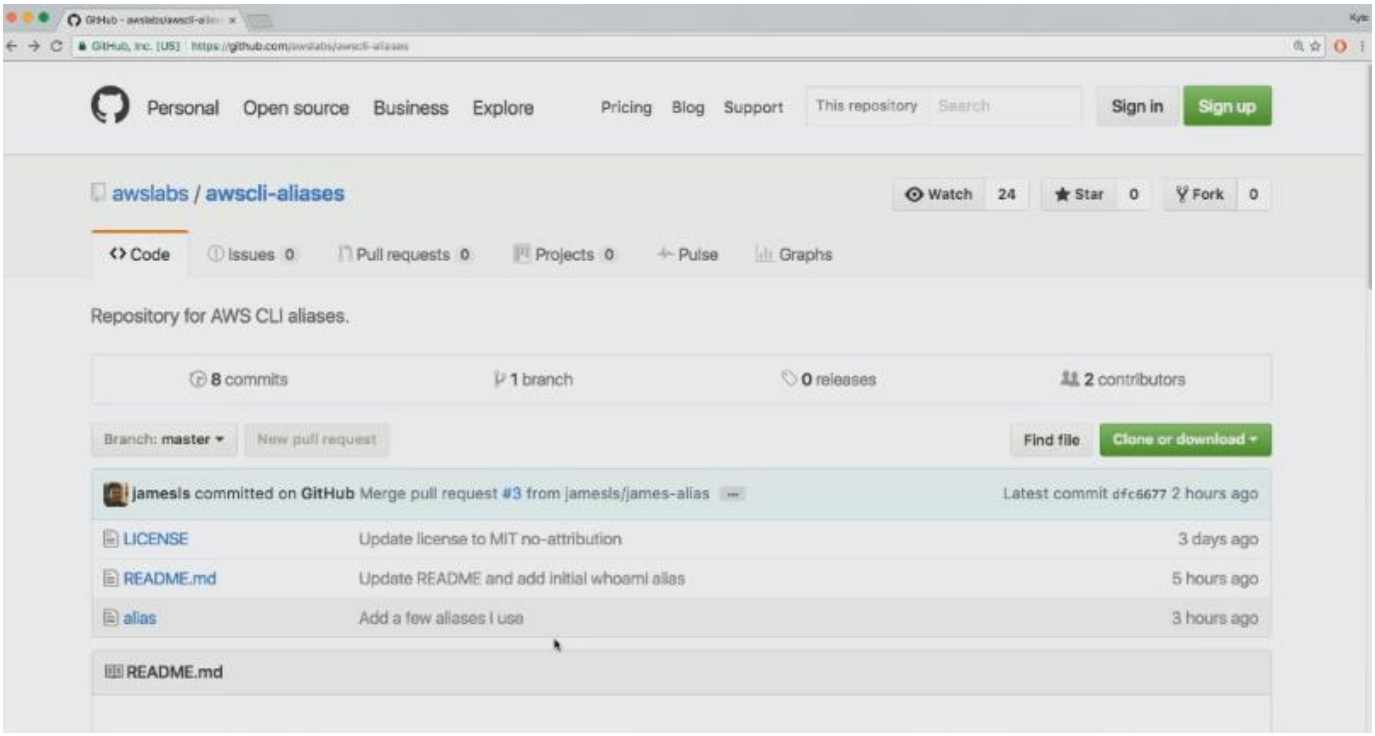
We can also spin up a second larger EC2 instance using the same command with minor changes, then we echo out the instance id as above

Important Takeaways

- `alias`
- Leverage BASH and external tooling



There are also several ***awslabs/awscli-aliases GitHub repository for sample aliases*** available to use.



The Effective AWS CLI User Tenets

The effective AWS CLI user:

- Uses an iterative workflow
- Troubleshoots well
- Is resourceful with tooling
- Understands performance implications

Performance Implications

- Client-side vs. server-side filtering
- Pagination
- aws s3 cp

```
$ aws ec2 describe-images \
    --query 'Images[?starts_with(to_string(Name),`amzn-ami-hvm-`)]'
```

This command can be used to describe all the amazon Linux hvm AMIs by specifying the query parameter. This command will list all the images available on to the client side, then perform the filtering which is not very efficient and causes the command to be slow.

```
$ aws ec2 describe-images \
    --query 'Images[?starts_with(to_string(Name),`amzn-ami-hvm-`)]'
```

```
$ aws ec2 describe-images \
    --filters Name=name,Values='amzn-ami-hvm-*' --query 'Images'
```

The best approach is to use server-side filtering as above by using server-side parameters like the - -filter parameter.

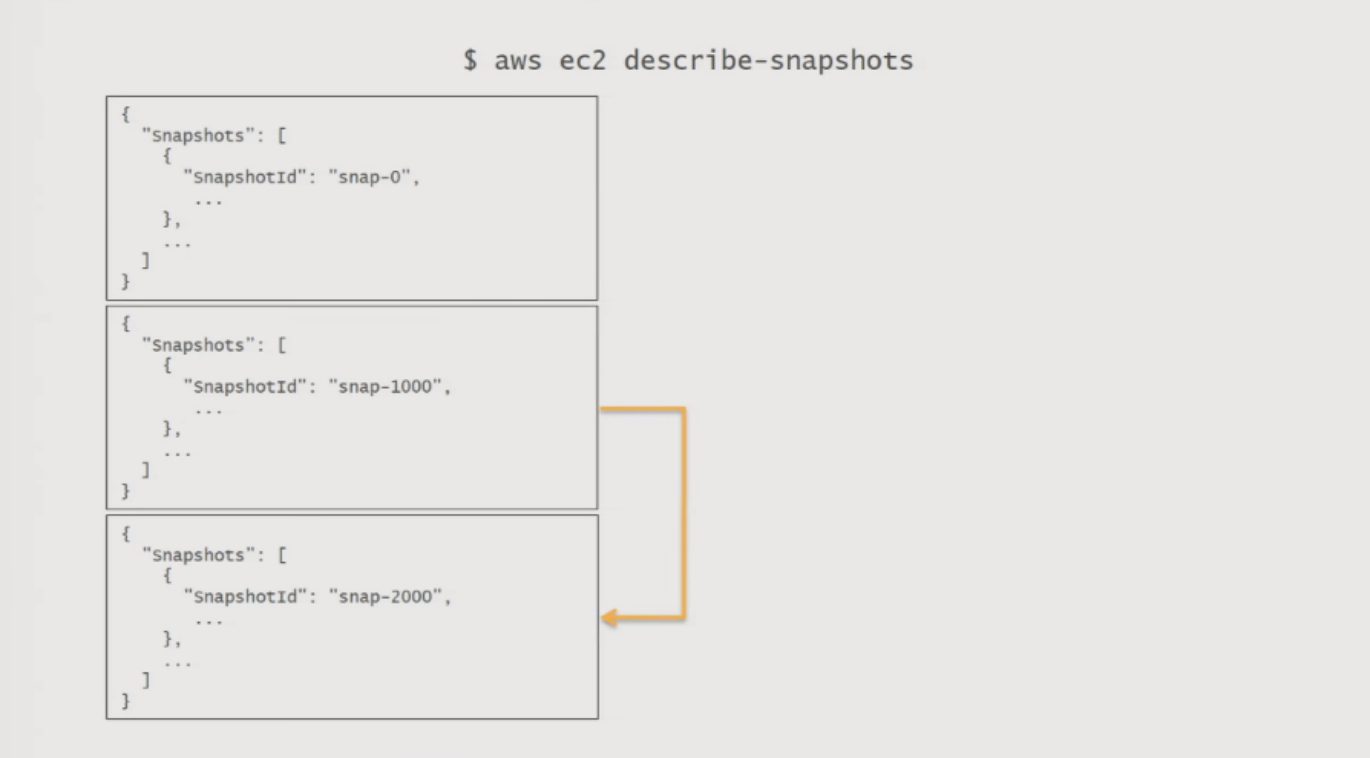
Performance Implications

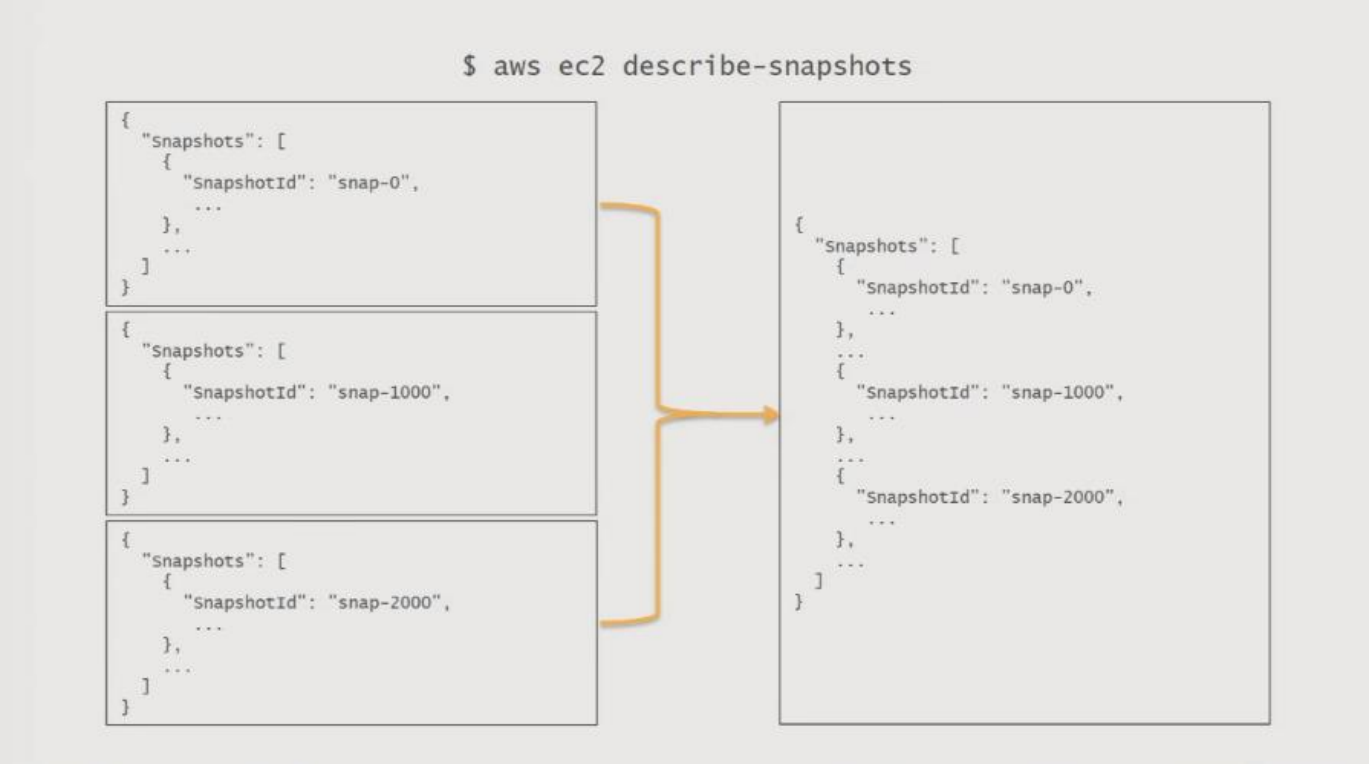
- Client-side vs. server-side filtering
- Pagination
- aws s3 cp

```
$ aws ec2 describe-snapshots
```

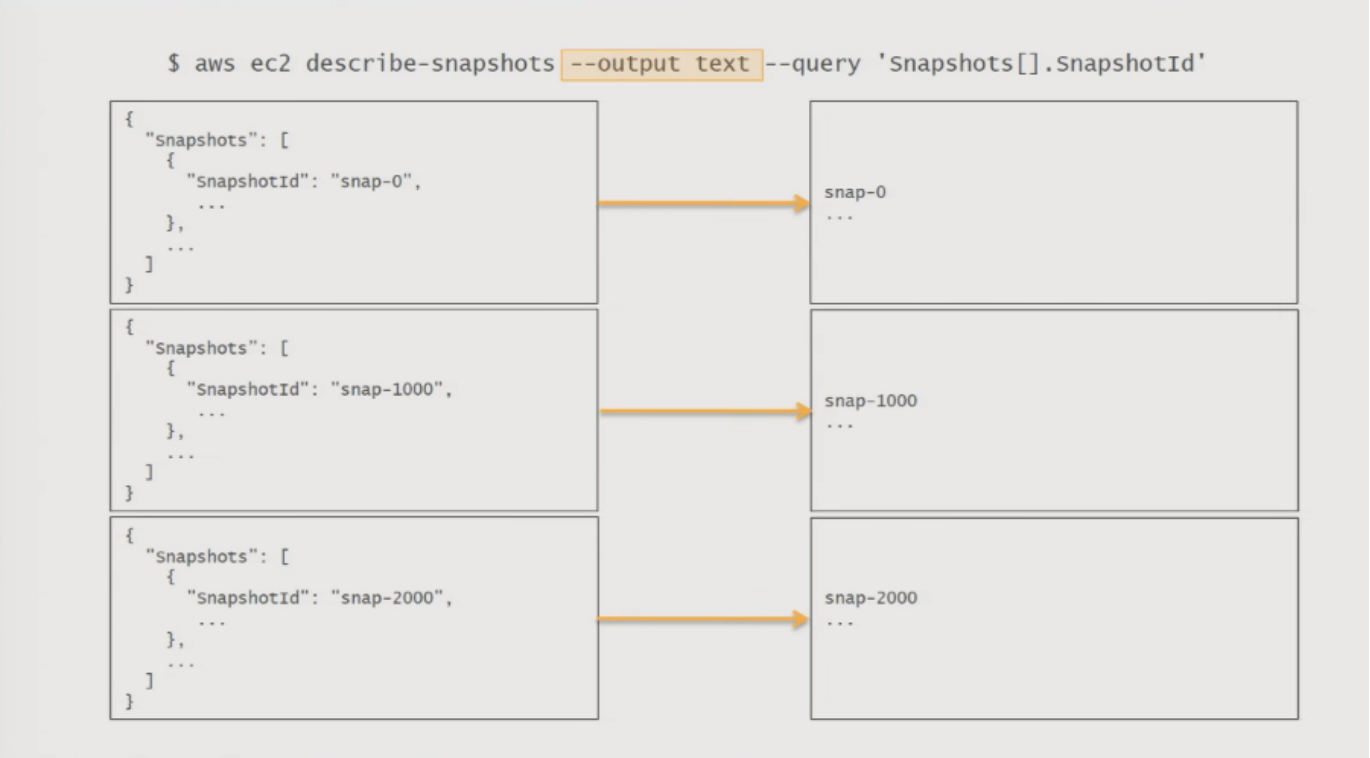
Above is an example command that is paginated by default,

The CLI will keep making the requests as long as there is an output token in the response

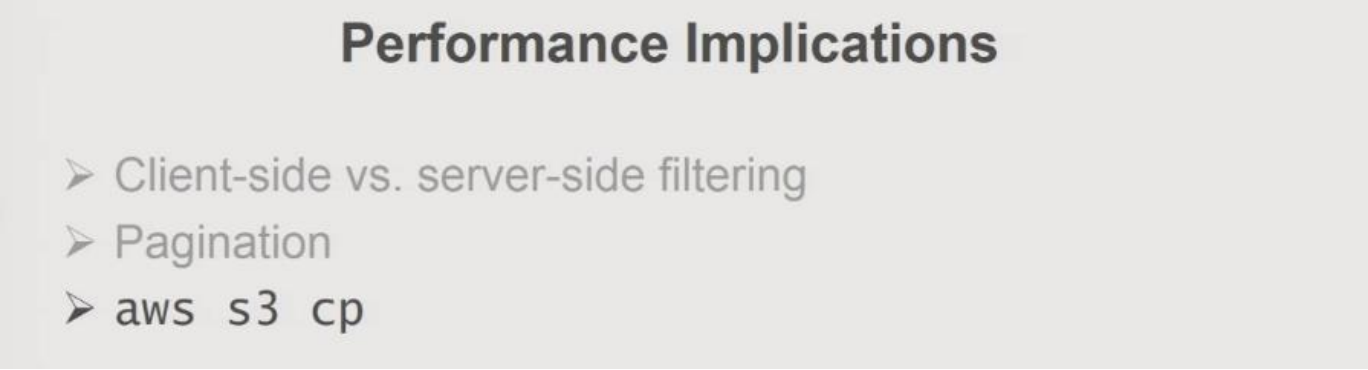


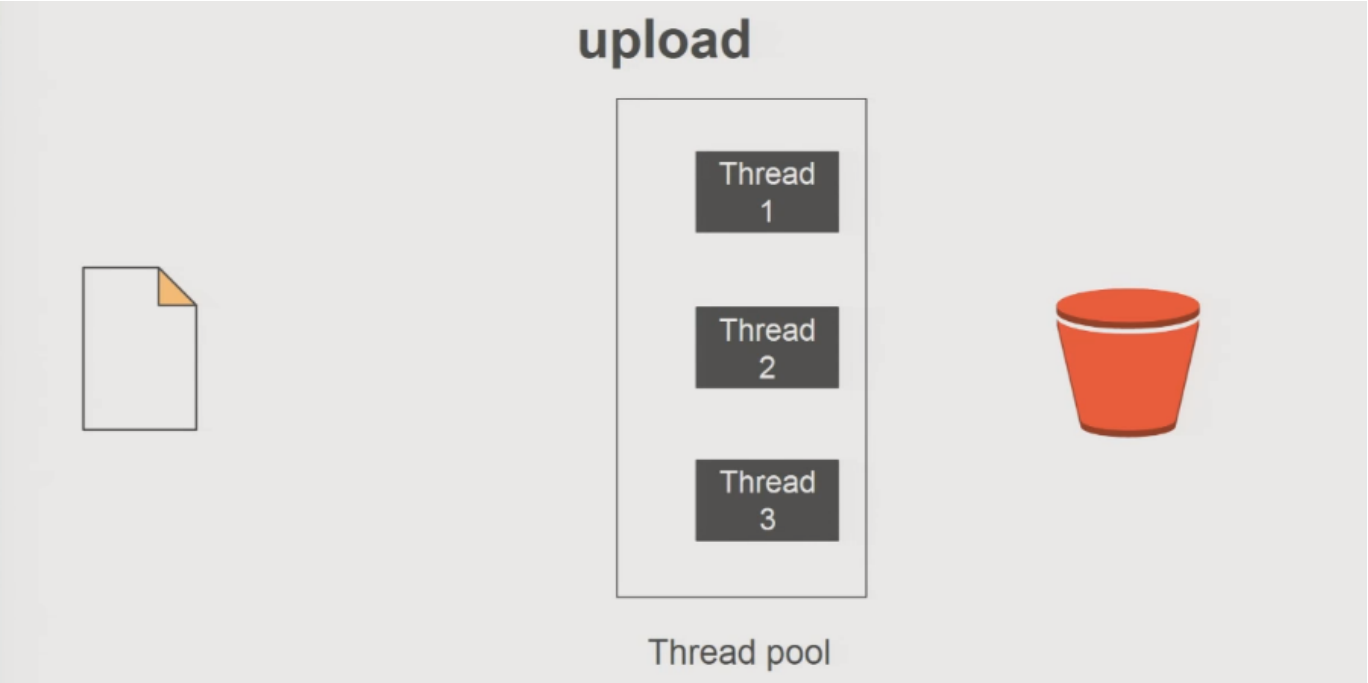


This is how we get a single list of snapshots

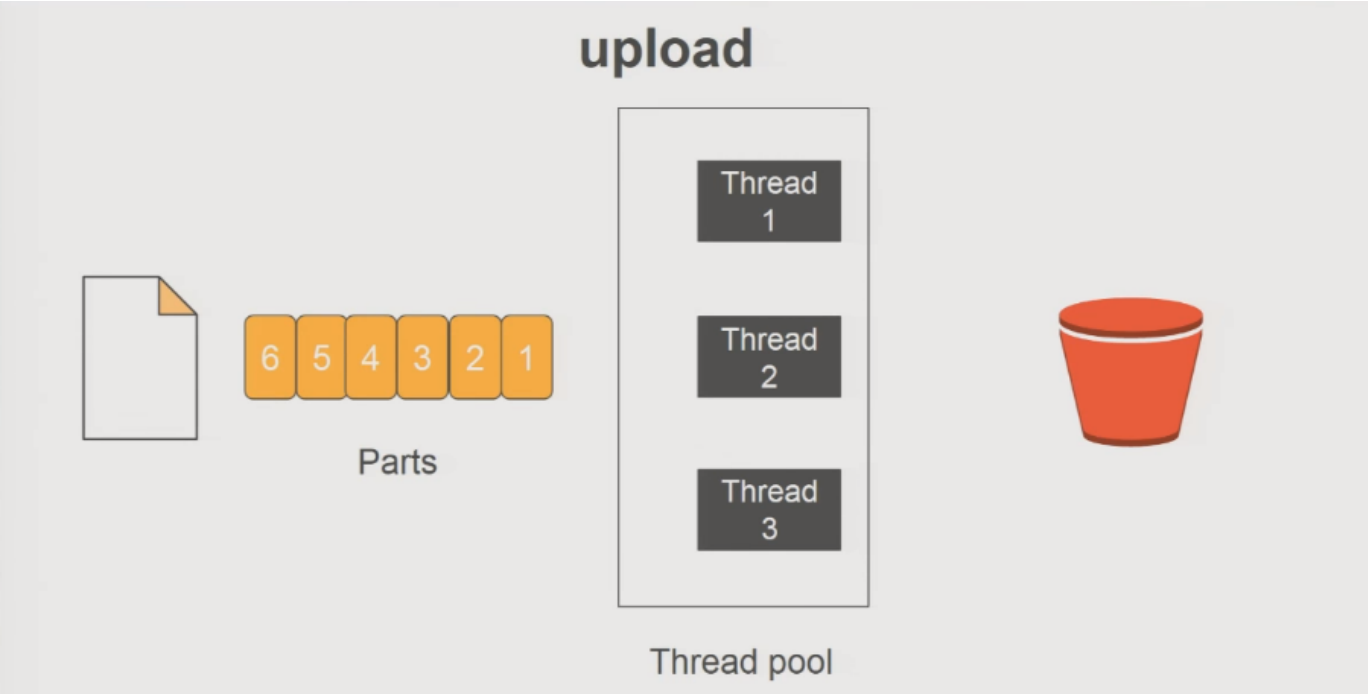


The behavior of the pagination changes based on the parameters passed along with the command, using ***- -output text*** as above means that the CLI will output the results as it becomes available. This becomes more like streaming data and the caller will not have to wait.

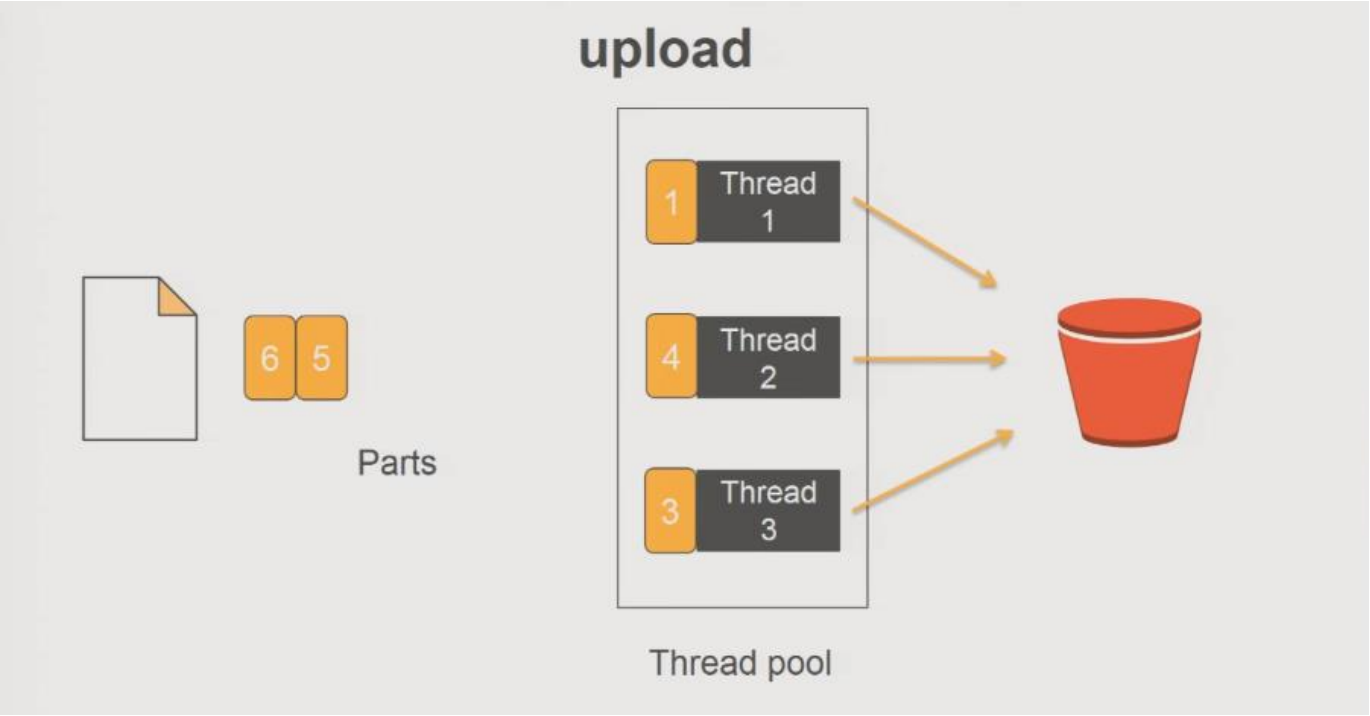


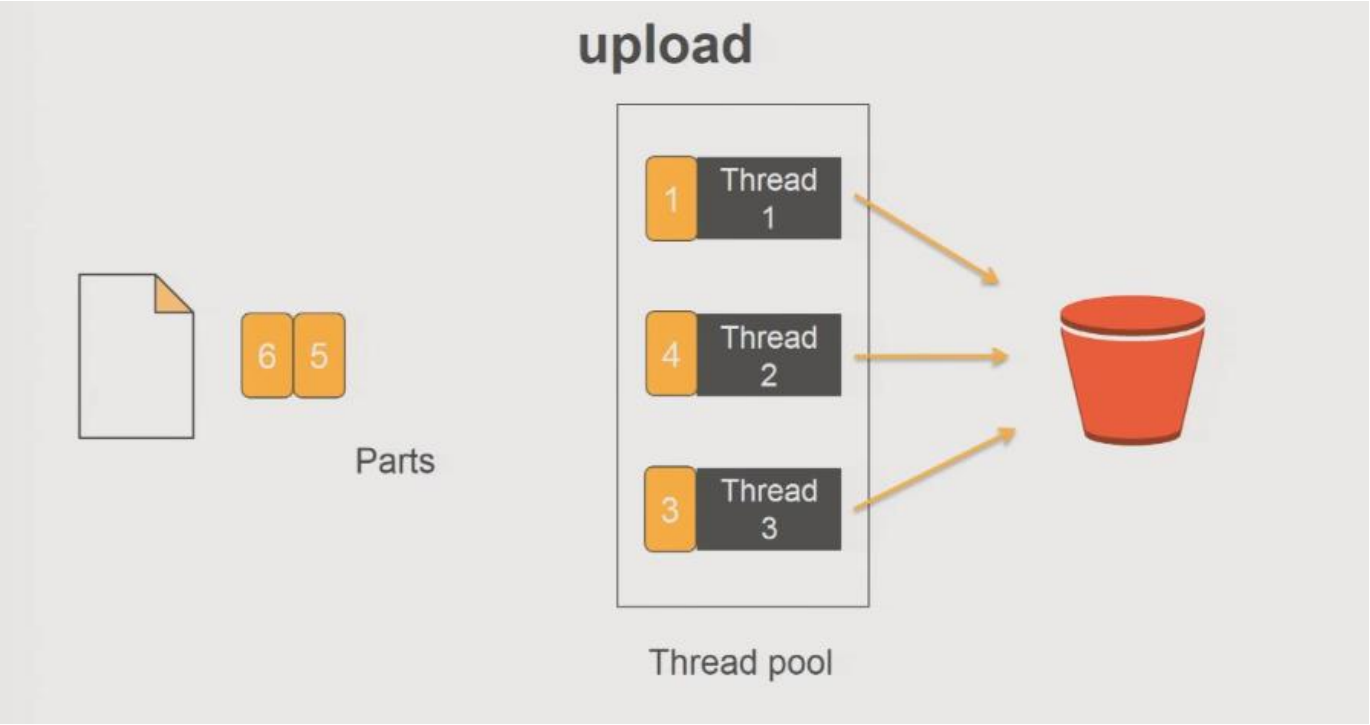


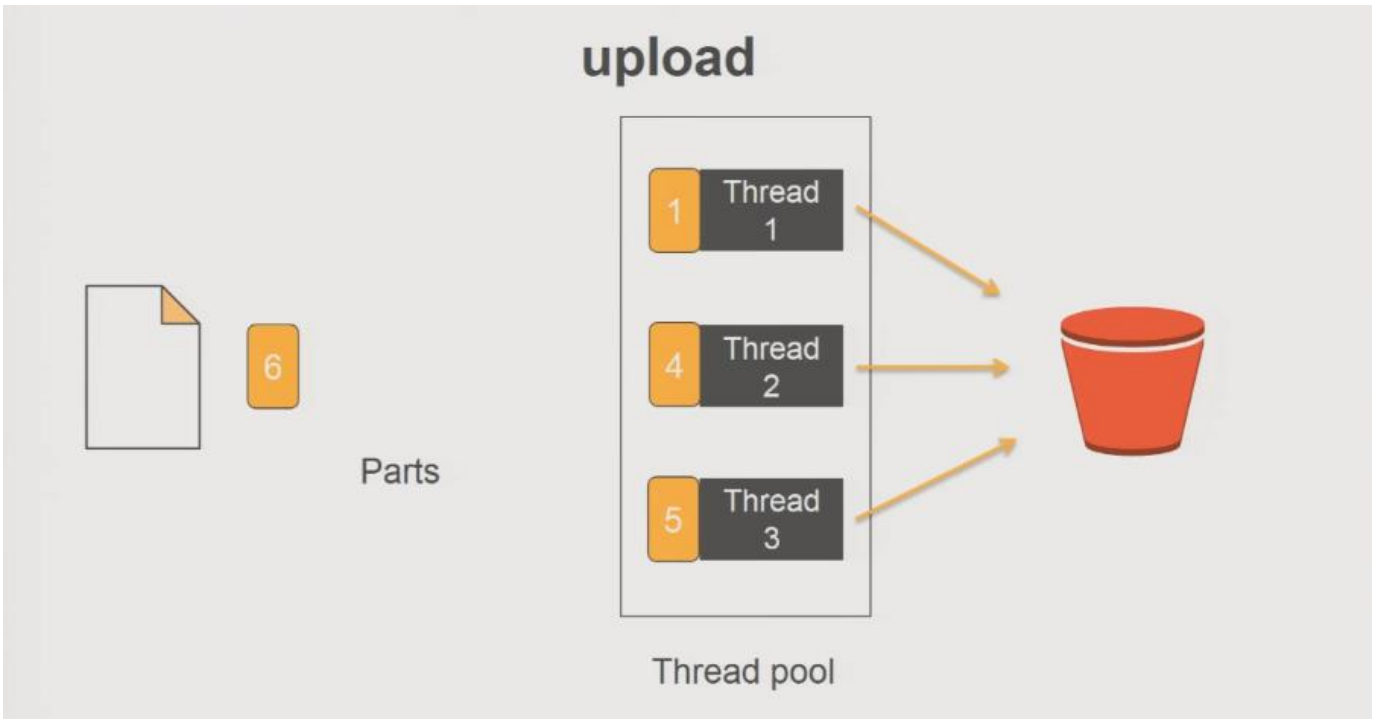
Above is a sample architecture of how a S3 command looks, all S3 commands have a threadpool backing it.

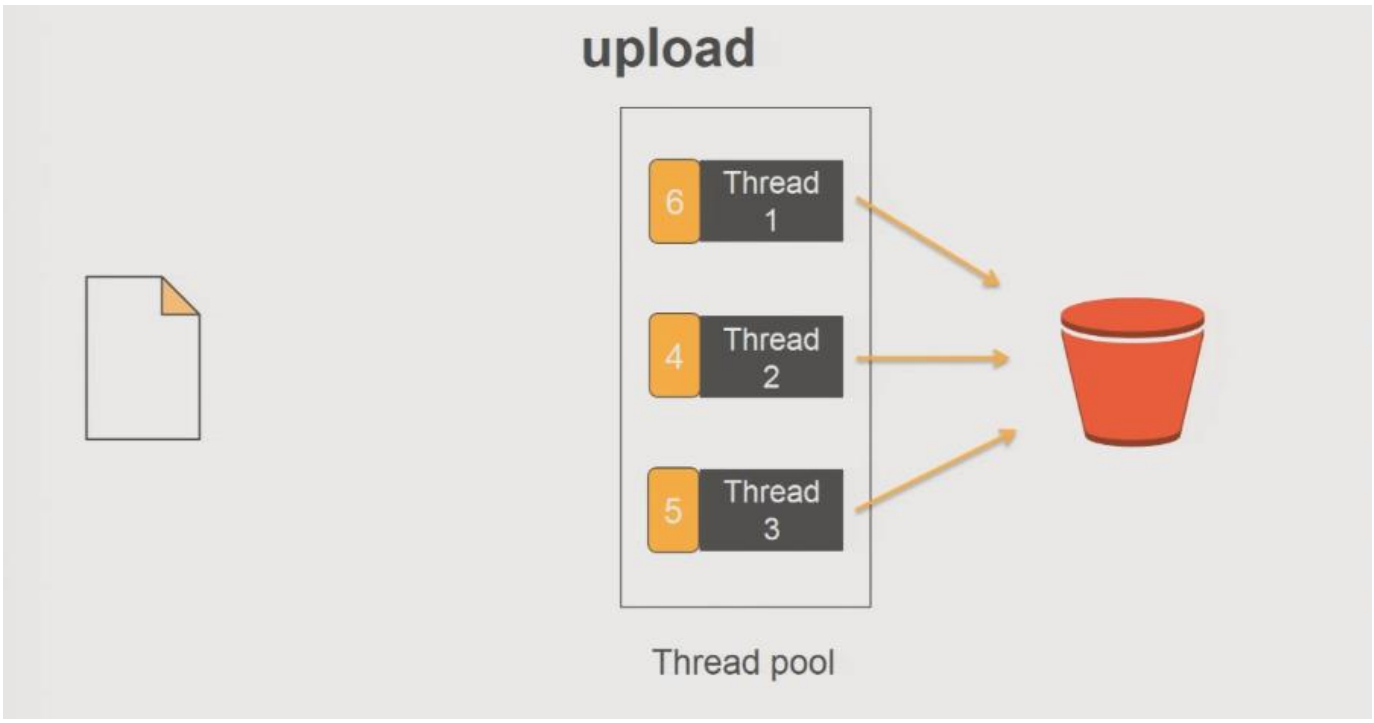


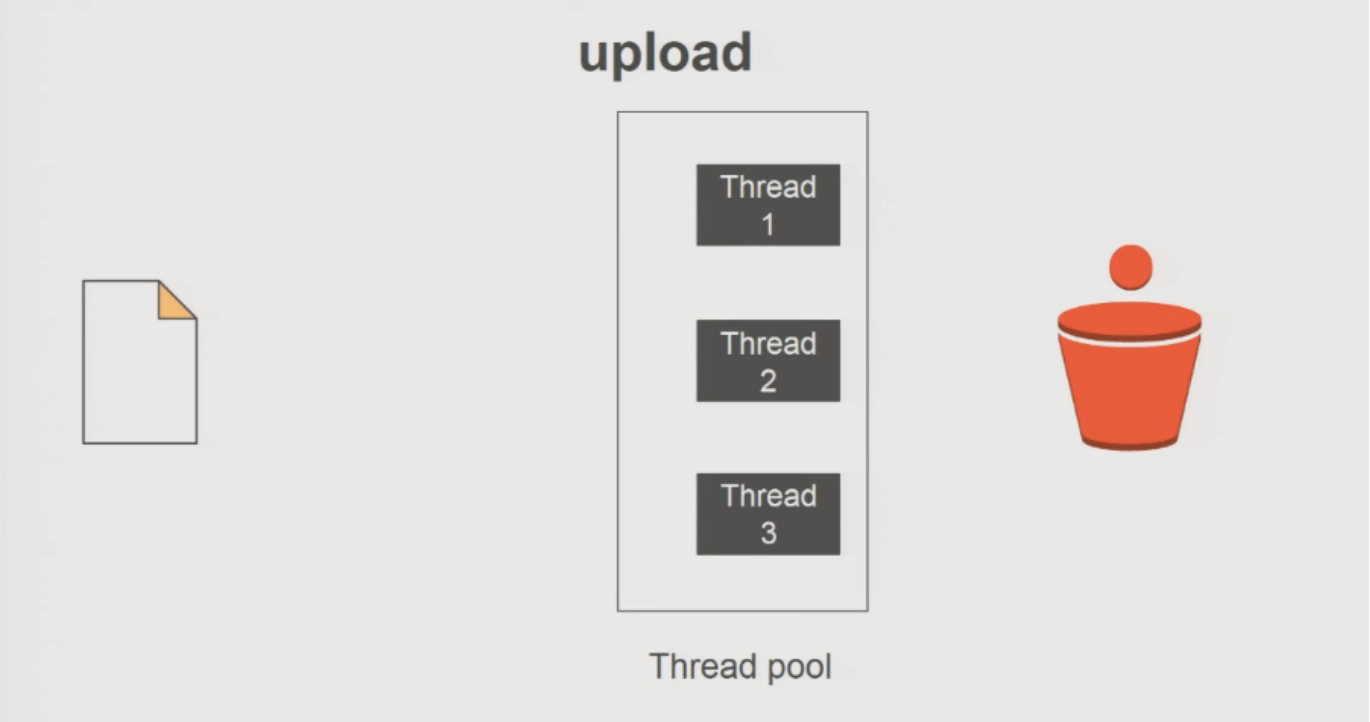
This is if the task needs to be multipart


upload
1
Thread 1
4
Thread 2
3
Thread 3
6
5
Parts
Thread pool


upload
1
Thread 1
4
Thread 2
3
Thread 3
6
5
Parts
Thread pool


upload
1
Thread 1
4
Thread 2
5
Thread 3
6
Parts
Thread pool


upload
6
Thread 1
4
Thread 2
5
Thread 3
Thread pool

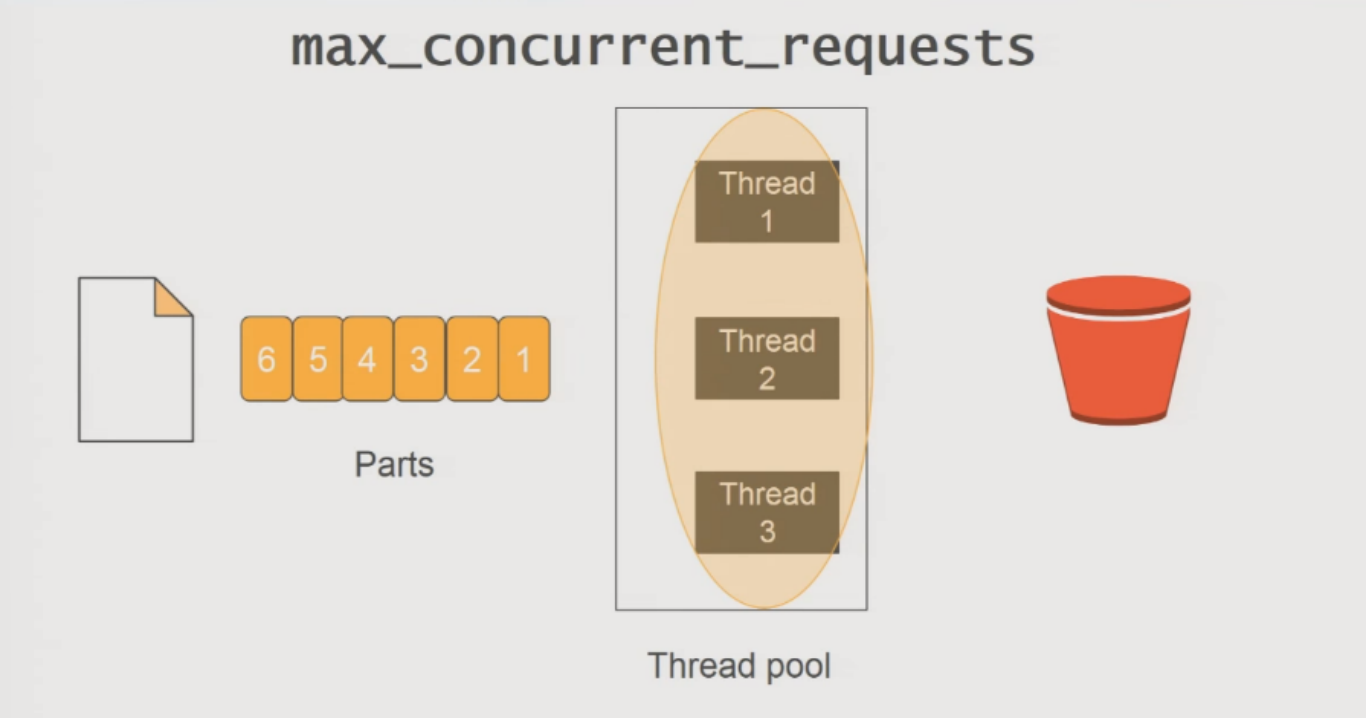


It will then call a ***CompleteMultiPartUpload*** action to finalize the S3 object

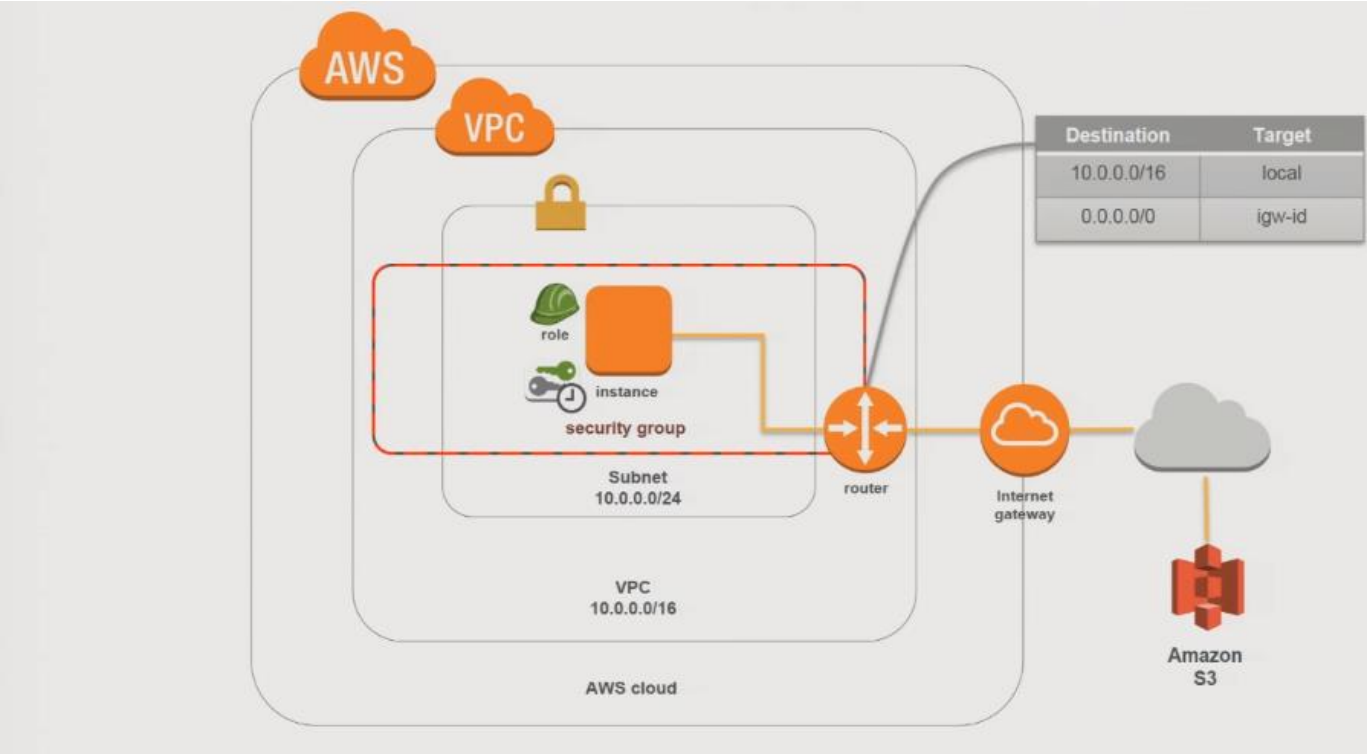
~/.aws/config

```
[default]
region = us-east-1
s3 =
    max_concurrent_requests = 20
    multipart_chunksize = 16MB
    multipart_threshold = 64MB
    max_queue_size = 10000
```

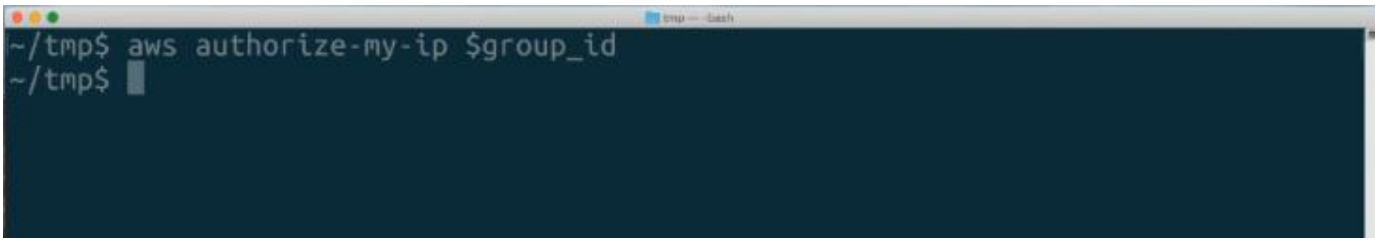
There are 4 S3 parameters that you can set when using S3 commands



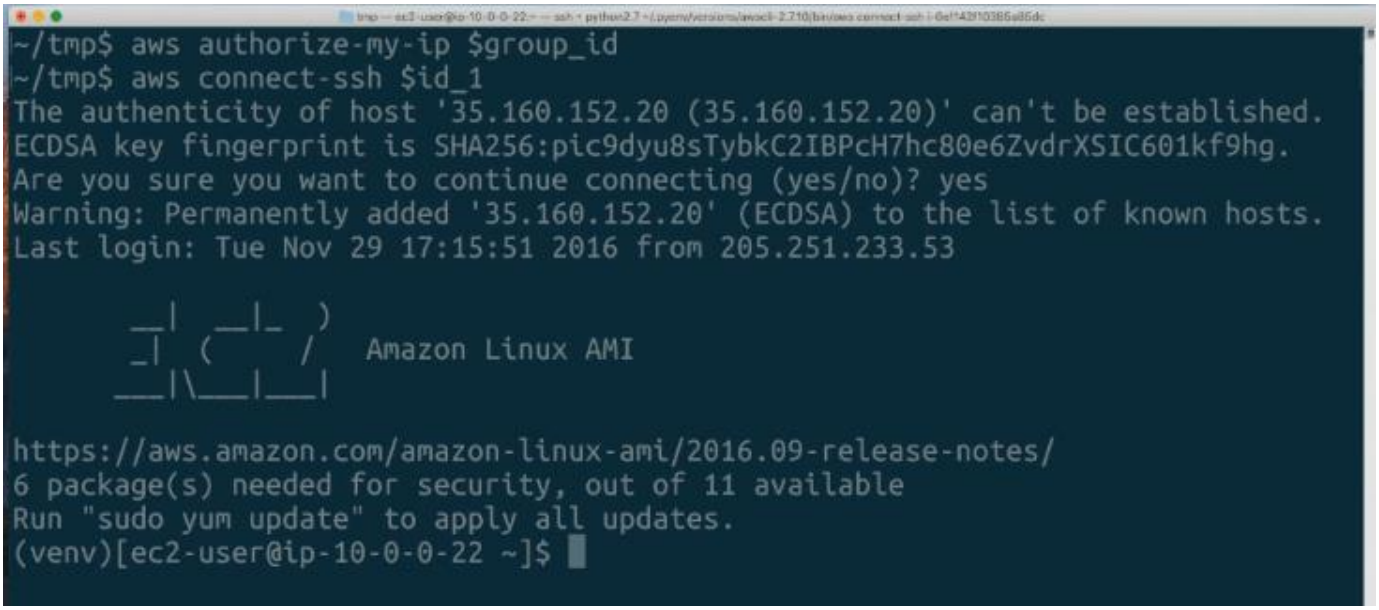
This is directly related to the number of threads and bandwidth to use for your commands



Let us now complete our set up by uploading files from our EC2 instance to S3.

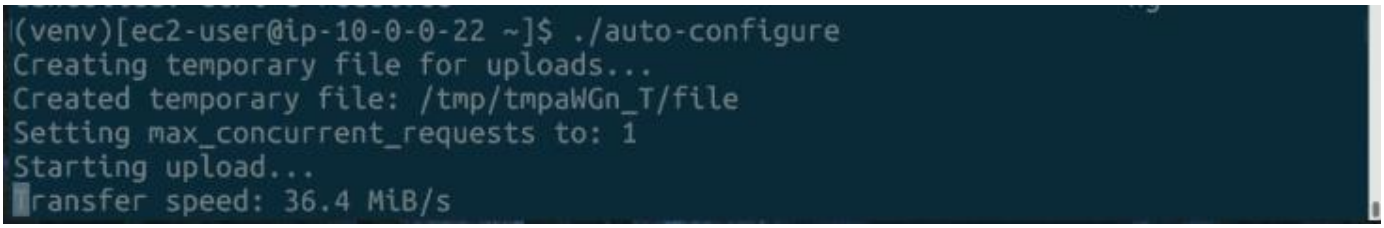


Before we can SSH into the running EC2 instance in our VPC, we need to authorize our IP for the SG using the command ***$ aws authorize-my-ip $group_id***. We are using the alias called ***authorize-my-ip*** for this. This prevents us from opening up the EC2 instance to the entire world and restrict access to only our current IP.



We then use the ***connect-ssh*** alias to connect to the first EC2 instance using the command ***$ aws connect-ssh $id_1***. On the instance we already have the AWS CLI installed and a few files inside a folder.

We then upload a file using the command ***$ aws s3 cp 10GB s3://kyleknapp-reinvent***. We can see the trafer speed being used to upload this file, this can help us determine if we need to use the max_concurrent_requests parameter with a high value for faster uploading.



We can use the ***auto-configure*** script we wrote to determine the value to use for the max_concurrent_requests parameter by tracking the transfer speed for 10secs and deciding whether to increase or keep the current value.

```
(venv)[ec2-user@ip-10-0-0-22 ~]$ ./auto-configure
Creating temporary file for uploads...
Created temporary file: /tmp/tmpaWGn_T/file
Setting max_concurrent_requests to: 1
Starting upload...
Transfer speed: 25.3 MiB/s
Setting max_concurrent_requests to: 4
Starting upload...
Transfer speed: 61.1 MiB/s
Setting max_concurrent_requests to: 7
Starting upload...
Transfer speed: 63.2 MiB/s
Done tuning max_concurrent_requests...
Setting max_concurrent_requests to: 7
(venv)[ec2-user@ip-10-0-0-22 ~]$
```

```
~/tmp$ aws connect-ssh $id_2
The authenticity of host '35.164.190.186 (35.164.190.186)' can't be established.
ECDSA key fingerprint is SHA256:fCREOsLCbLwP6iFqqPNnlIeKVwtxD4MXN+0V1wcKQ74.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '35.164.190.186' (ECDSA) to the list of known hosts.
Last login: Tue Nov 29 17:15:51 2016 from 205.251.233.53

       __|  __|_  )
       _|  (     /   Amazon Linux AMI
      ___|\___|___|

https://aws.amazon.com/amazon-linux-ami/2016.09-release-notes/
6 package(s) needed for security, out of 11 available
Run "sudo yum update" to apply all updates.
(venv)[ec2-user@ip-10-0-0-64 ~]$
```

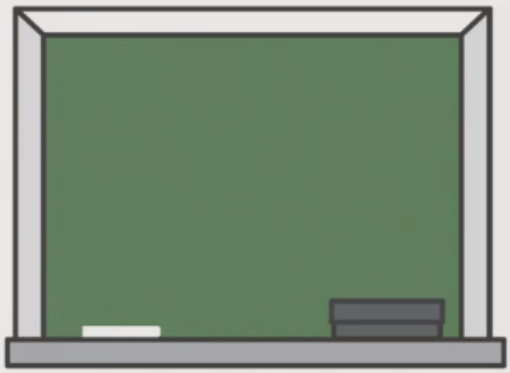
We now SSH into the larger EC2 instance we have running using the command ***$ aws connect-ssh $id_2***

```
Run "sudo yum update" to apply all updates.
(venv)[ec2-user@ip-10-0-0-64 ~]$ aws auto-configure
Creating temporary file for uploads...
Created temporary file: /tmp/tmp3Qcr57/file
Setting max_concurrent_requests to: 1
Starting upload...
Transfer speed: 21.9 MiB/s
Setting max_concurrent_requests to: 4
Starting upload...
Transfer speed: 70.7 MiB/s
Setting max_concurrent_requests to: 7
Starting upload...
Transfer speed: 139.4 MiB/s
Setting max_concurrent_requests to: 10
Starting upload...
Transfer speed: 201.0 MiB/s
Setting max_concurrent_requests to: 11
Starting upload...
Transfer speed: 206.2 MiB/s
Done tuning max_concurrent_requests...
Setting max_concurrent_requests to: 11
(venv)[ec2-user@ip-10-0-0-64 ~]$
```

We now use an alias called ***auto-configure*** to do the same upload again by tracking and increasing the transfer speed with the max_concurrent_requests value.

Important Takeaways

- Use server-side filtering when possible
- Pagination with `--output text`
- `s3` configuration options



The Effective AWS CLI User Tenets

The effective AWS CLI user:

- Uses an iterative workflow
  - Example: `--generate-cli-skeleton output`
- Troubleshoots well
  - Example: `--debug`
- Is resourceful with tooling
  - Example: `alias`
- Understands performance implications
  - Example: `s3` configurations

Thank you!

amazon web services