CLOUD COMPUTING

Session 4

Patterns

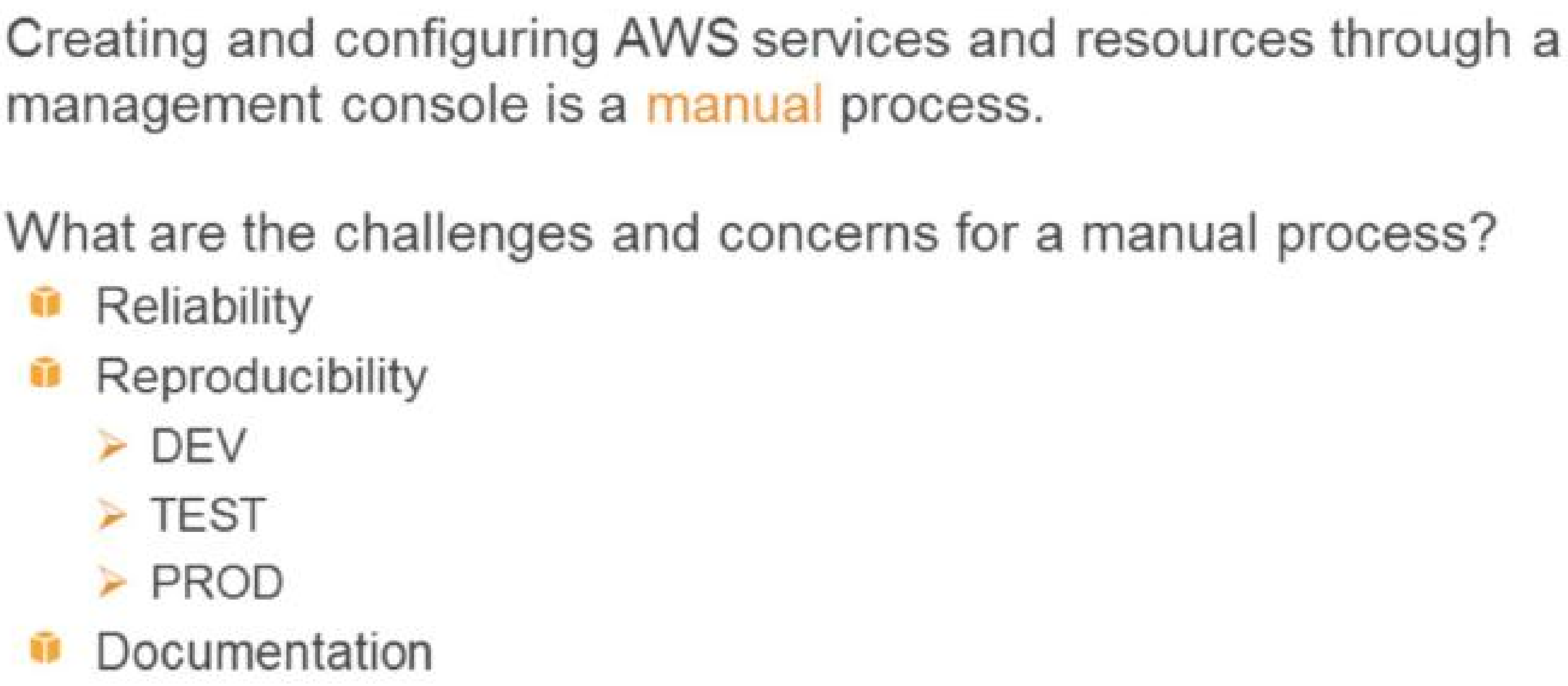
- Infrastructure Automation
- Components Decoupling

APROBAT

Conf. univ. dr. ing. IUSTIN PRIESCU - UTM
Dr. ing. Sebastian NICOLAESCU - Verizone Bussines-US

Infrastructure Automation

Manual Configuration Challenges



Creating and configuring AWS services and resources through a management console is a manual process.

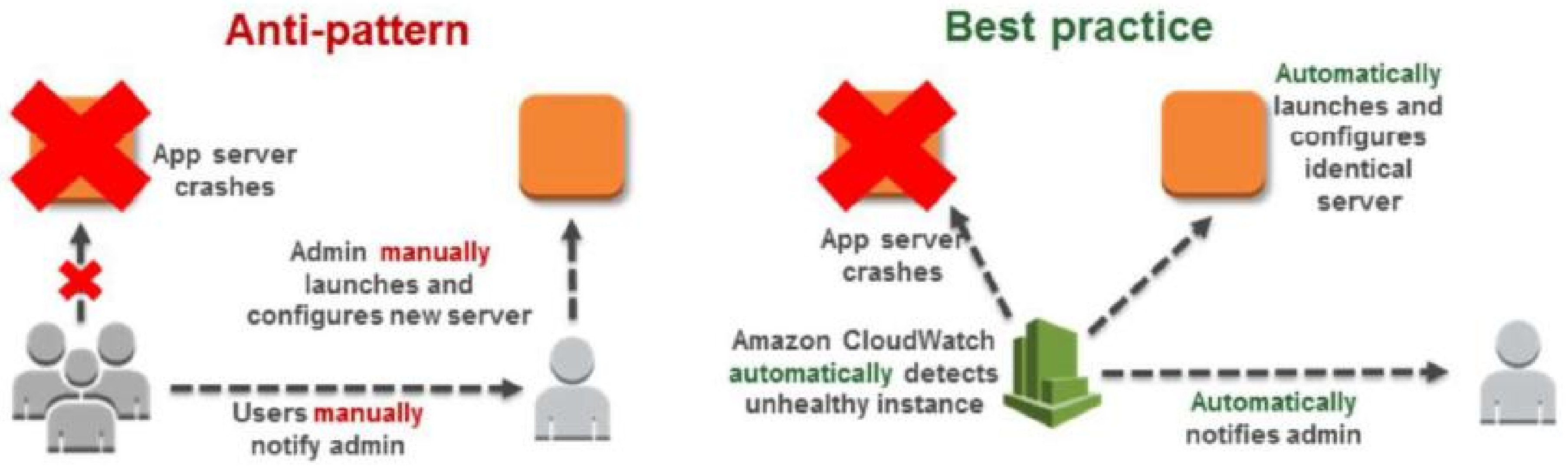
What are the challenges and concerns for a manual process?

- Reliability
- Reproducibility
  - DEV
  - TEST
  - PROD
- Documentation

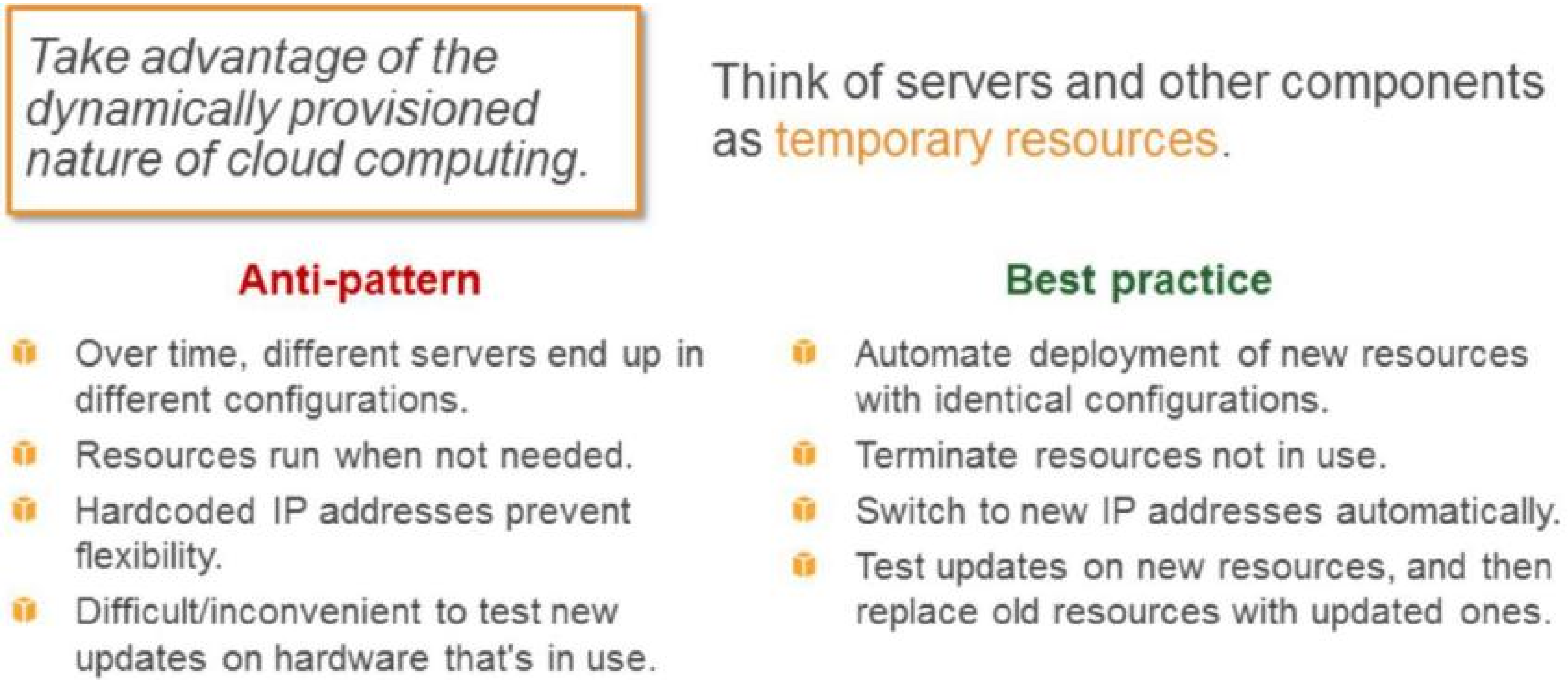
Best Practice: Automate Your Environment

Where possible, automate the provisioning, termination, and configuration of resources.

Improve your system's stability and consistency, as well as the efficiency of your organization, by removing manual processes.


Anti-pattern
App server crashes
Admin manually launches and configures new server
Users manually notify admin
Best practice
Automatically launches and configures identical server
App server crashes
Amazon CloudWatch automatically detects unhealthy instance
Automatically notifies admin

Best Practice: Use Disposable Resources



Take advantage of the dynamically provisioned nature of cloud computing.

Think of servers and other components as temporary resources.

Anti-pattern

- Over time, different servers end up in different configurations.
- Resources run when not needed.
- Hardcoded IP addresses prevent flexibility.
- Difficult/inconvenient to test new updates on hardware that's in use.

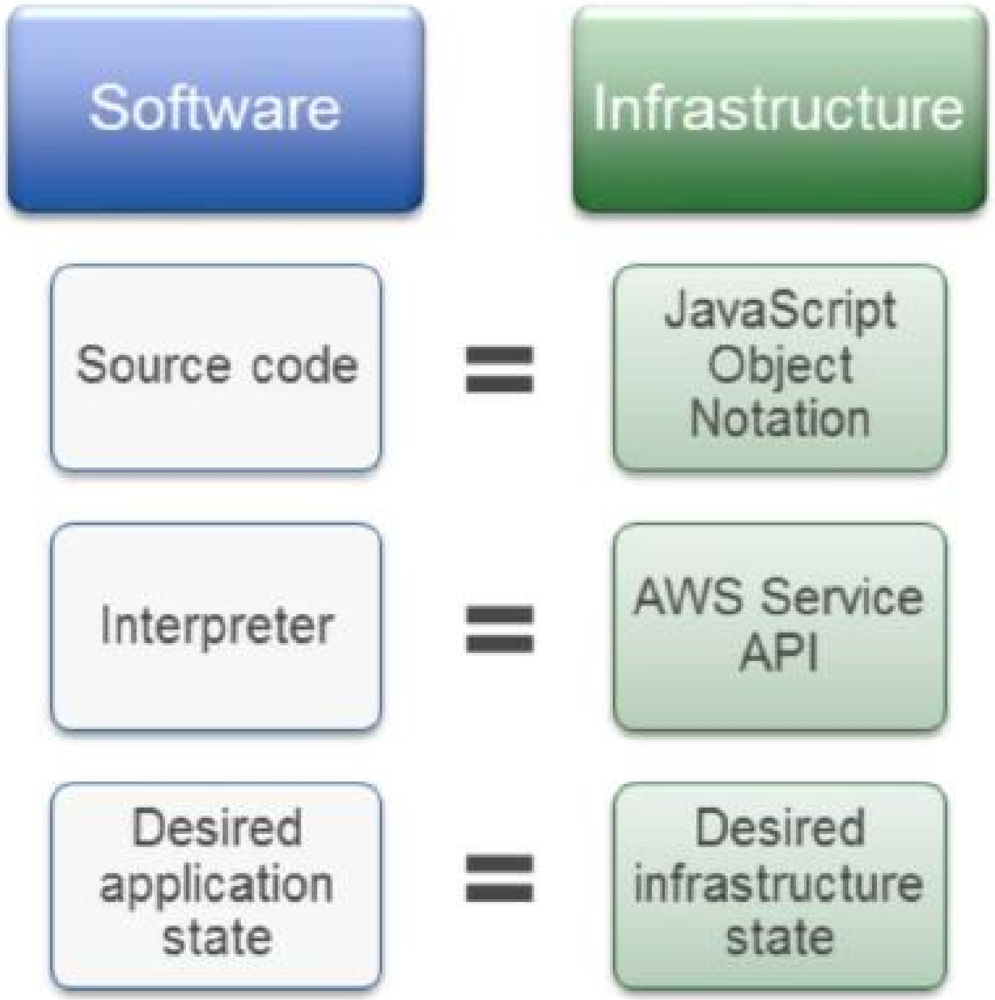
Best practice

- Automate deployment of new resources with identical configurations.
- Terminate resources not in use.
- Switch to new IP addresses automatically.
- Test updates on new resources, and then replace old resources with updated ones.

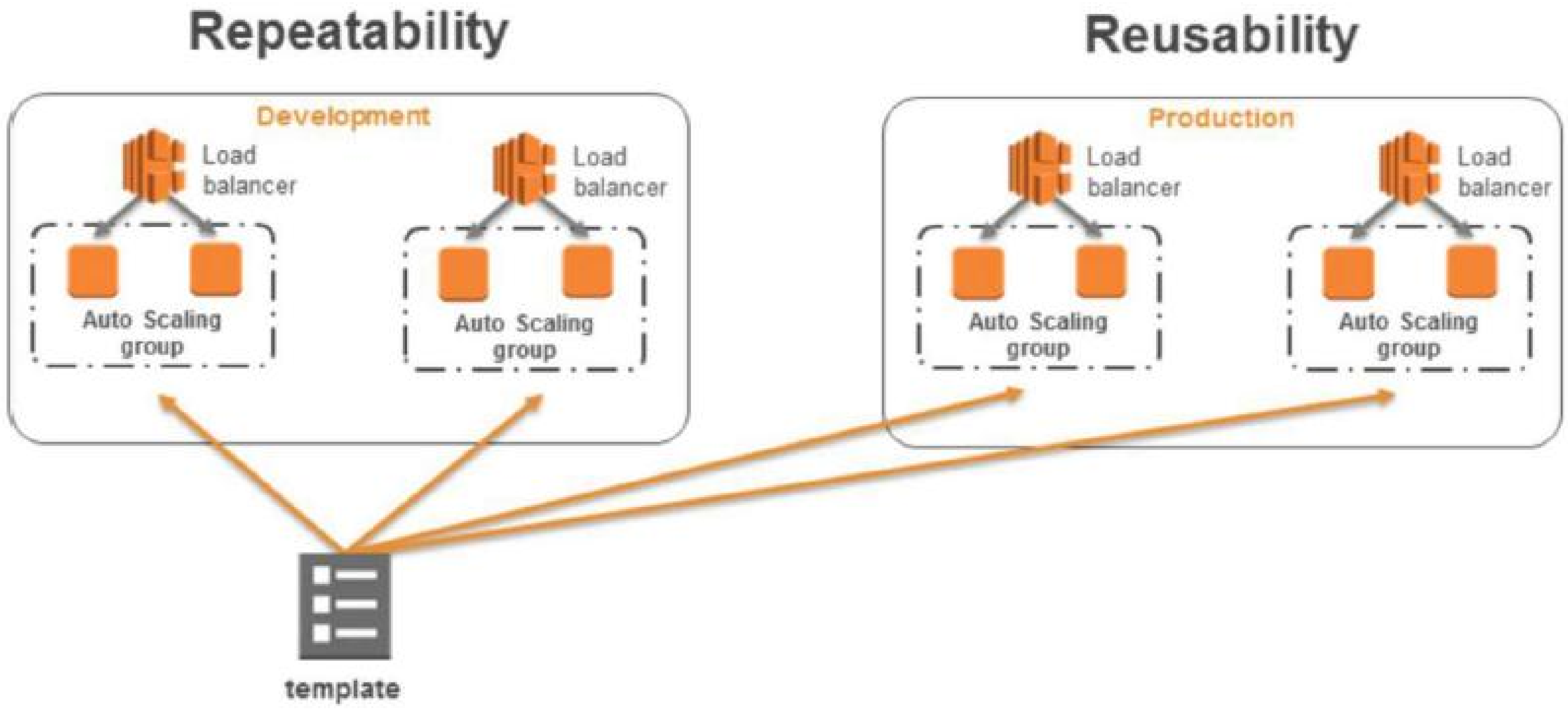
What Does Infrastructure as Code Mean?

Techniques, practices, and tools from software development applied to creating reusable, maintainable, extensible and testable infrastructure.

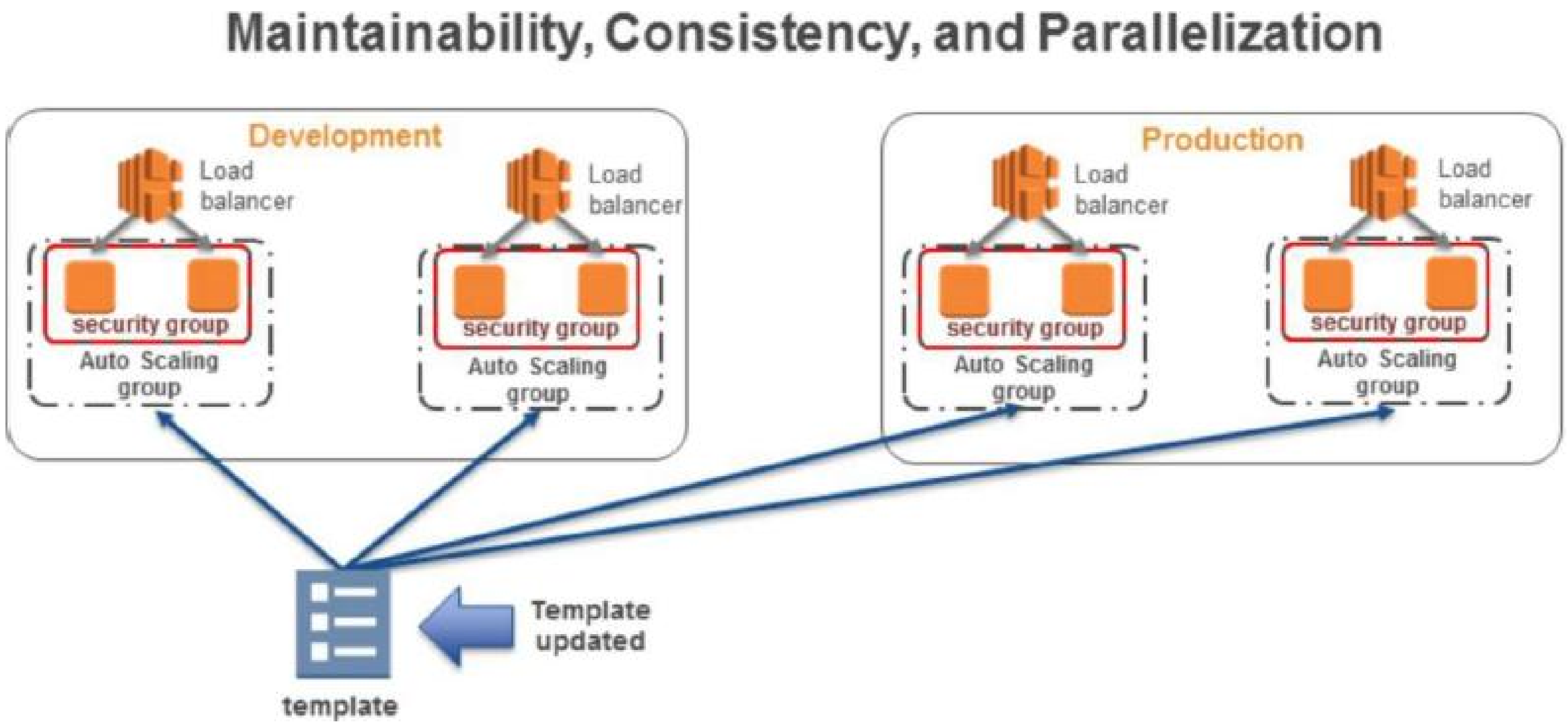
Build and Operate Your Infrastructure Like Software


Software
Infrastructure
Source code
=
JavaScript Object Notation
Interpreter
=
AWS Service API
Desired application state
=
Desired infrastructure state

Benefits of Treating Infrastructure as Code



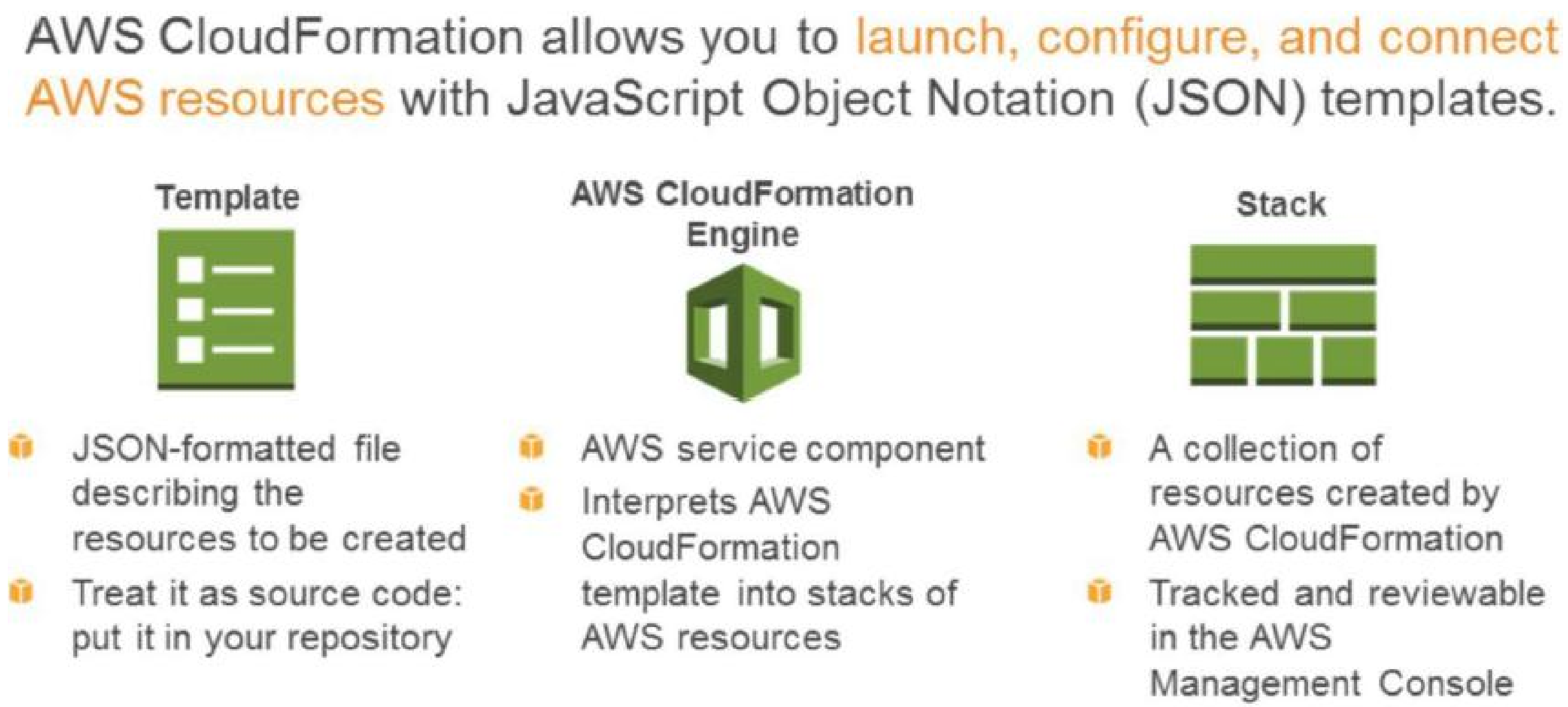
Benefits of Treating Infrastructure as Code



Infrastructure as Code on AWS

AWS CloudFormation

CloudFormation: Infrastructure as Code



AWS CloudFormation allows you to launch, configure, and connect AWS resources with JavaScript Object Notation (JSON) templates.

Template

- JSON-formatted file describing the resources to be created
- Treat it as source code: put it in your repository

AWS CloudFormation Engine

- AWS service component
- Interprets AWS CloudFormation template into stacks of AWS resources

Stack

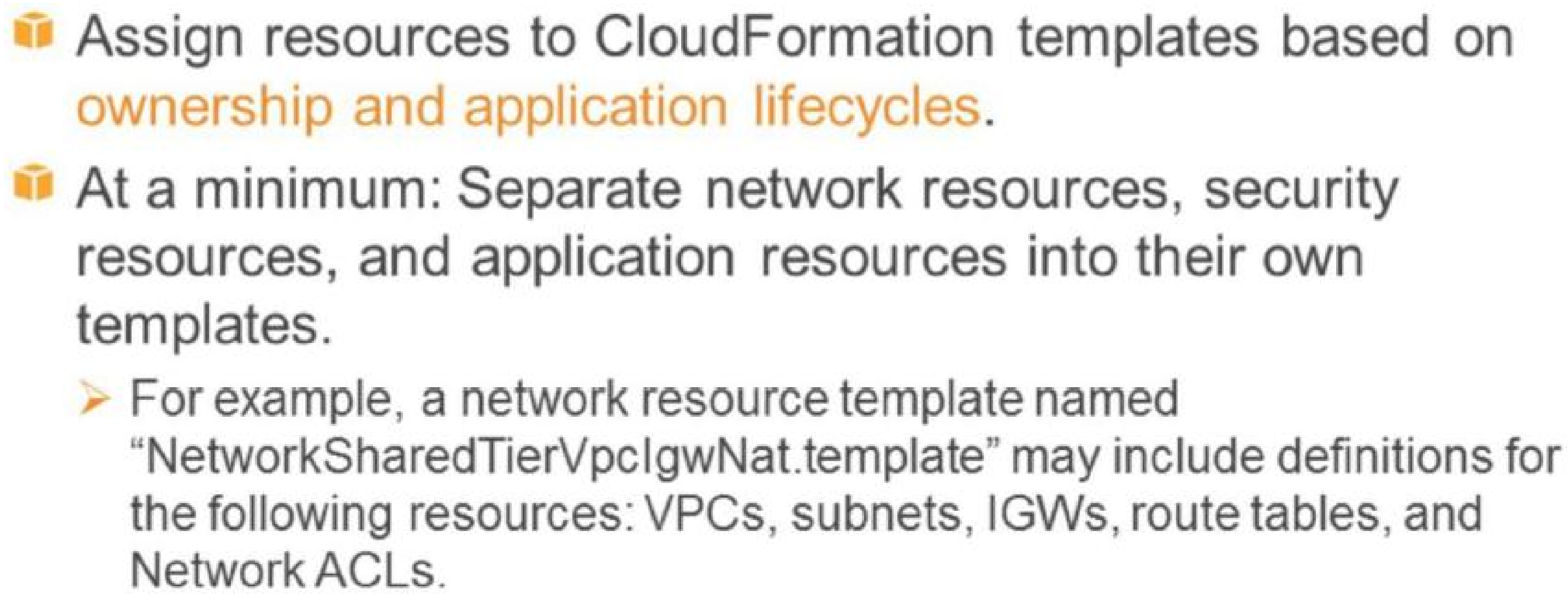
- A collection of resources created by AWS CloudFormation
- Tracked and reviewable in the AWS Management Console

Working With CloudFormation Templates

- Simple JSON text editor
- VisualOps.io
  - APN Partner tool
  - WYSIWYG editor for CloudFormation templates
- CloudFormation Designer
  - Available via the AWS Management Console
  - Lets you drag and drop resources onto a design area to automatically generate a JSON-formatted CloudFormation template
  - Edit the properties of the JSON template on the same page
  - Open and edit existing CloudFormation templates using the CloudFormation Designer tool

How Should Resources Be Grouped Together into Templates?

Organizing Your CloudFormation Templates



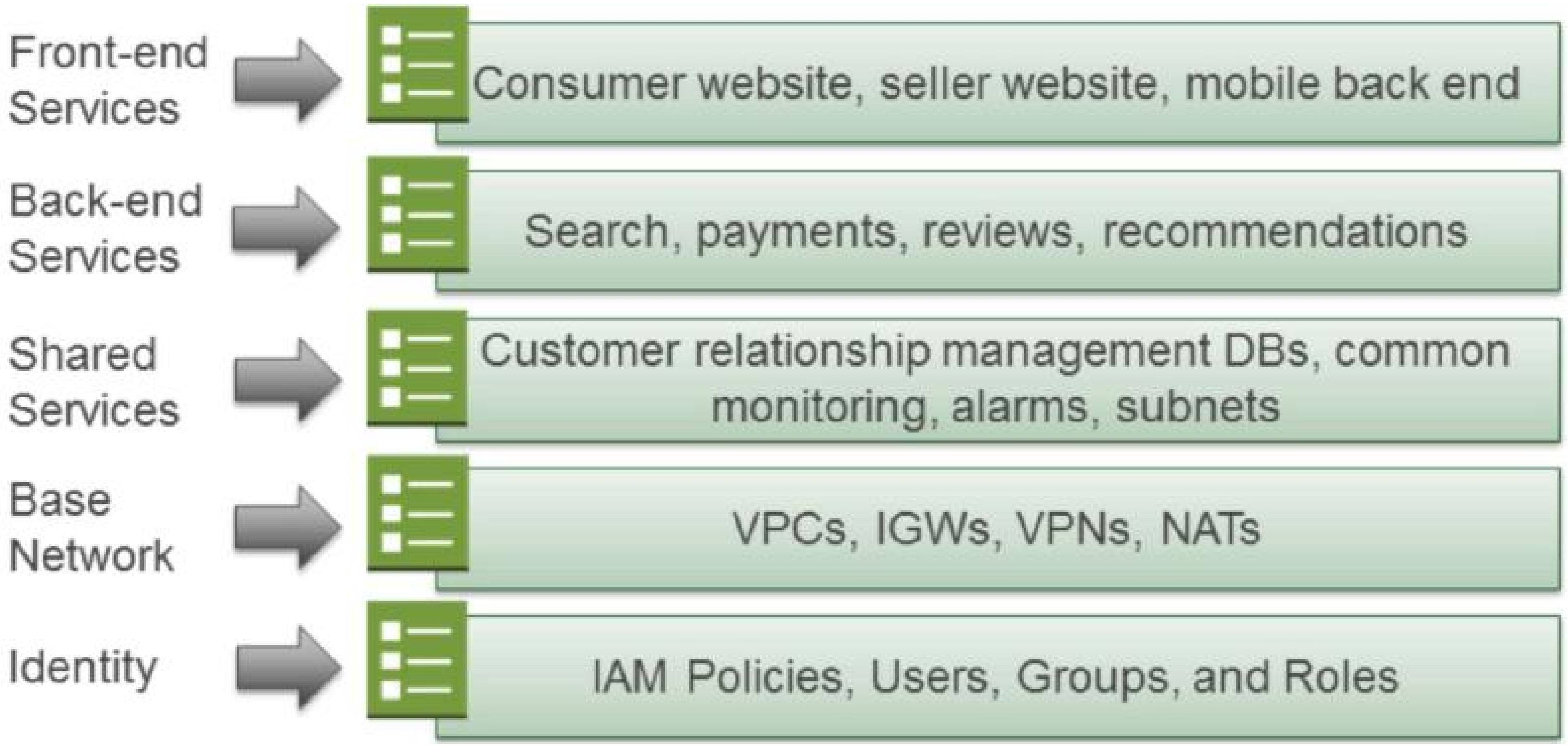
- Assign resources to CloudFormation templates based on ownership and application lifecycles.
- At a minimum: Separate network resources, security resources, and application resources into their own templates.
  - For example, a network resource template named "NetworkSharedTierVpcIgwNat.template" may include definitions for the following resources: VPCs, subnets, IGWs, route tables, and Network ACLs.

Standardized Architecture for NIST-based Assurance Frameworks on the AWS Cloud
http://docs.aws.amazon.com/quickstart/latest/accelerator-nist/welcome.html

Organizing Your CloudFormation Templates

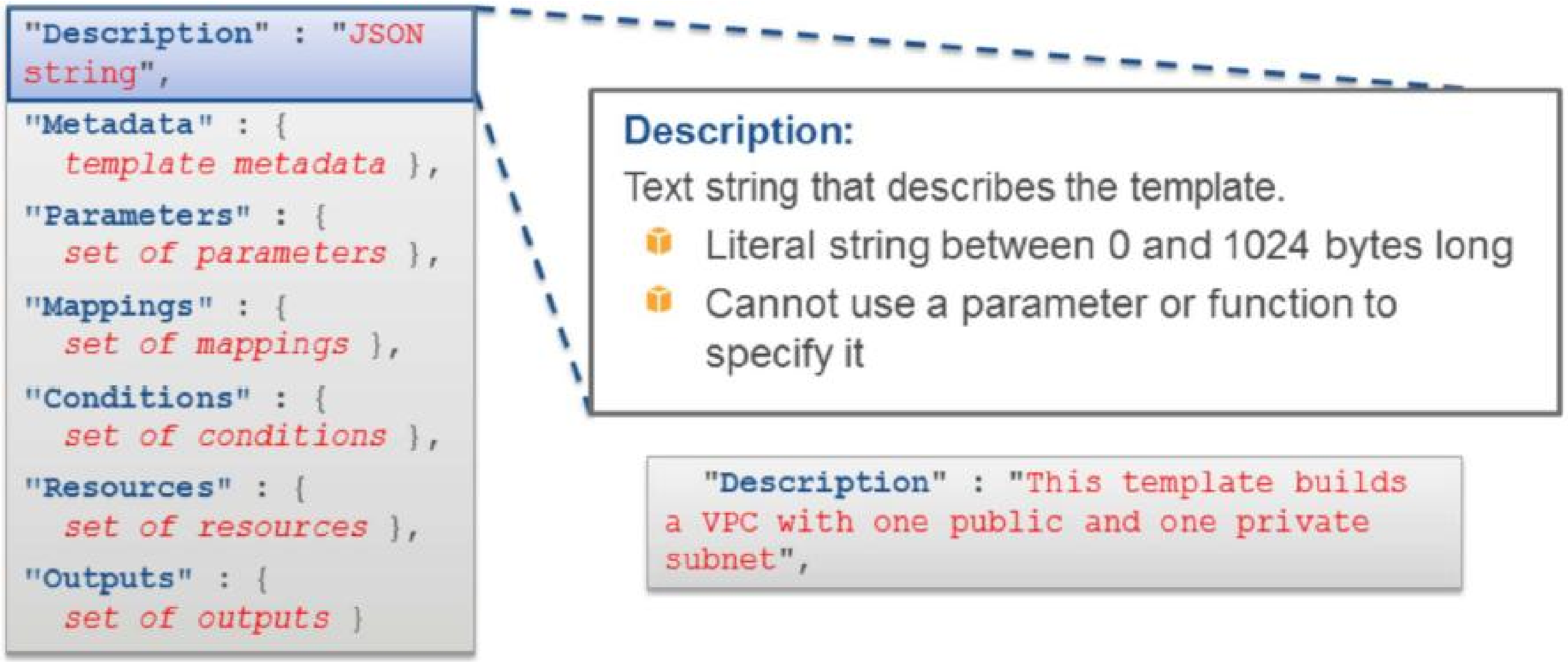
- Avoid sharing a single template across applications for resources of the same type unless you are deliberately centralizing control of that resource type.
    - Ex.: Don't use one template to define the security groups of 10 applications.

Example Cloud Formation Groups



Front-end Services	Consumer website, seller website, mobile back end
Back-end Services	Search, payments, reviews, recommendations
Shared Services	Customer relationship management DBs, common monitoring, alarms, subnets
Base Network	VPCs, IGWs, VPNs, NATs
Identity	IAM Policies, Users, Groups, and Roles

Anatomy Of A CloudFormation Template



```
"Description" : "JSON
string",
"Metadata" : {
    template metadata },
"Parameters" : {
    set of parameters },
"Mappings" : {
    set of mappings },
"Conditions" : {
    set of conditions },
"Resources" : {
    set of resources },
"Outputs" : {
    set of outputs }
```

Description:

Text string that describes the template.

- Literal string between 0 and 1024 bytes long
- Cannot use a parameter or function to specify it

```
  "Description" : "This template builds
a VPC with one public and one private
subnet",
```

http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/template-anatomy.html

Anatomy Of A CloudFormation Template

```
"Description" : "JSON
string",
"Metadata" : {
  template metadata },
"Parameters" : {
  set of parameters },
"Mappings" : {
  set of mappings },
"Conditions" : {
  set of conditions },
"Resources" : {
  set of resources },
"Outputs" : {
  set of outputs }
```

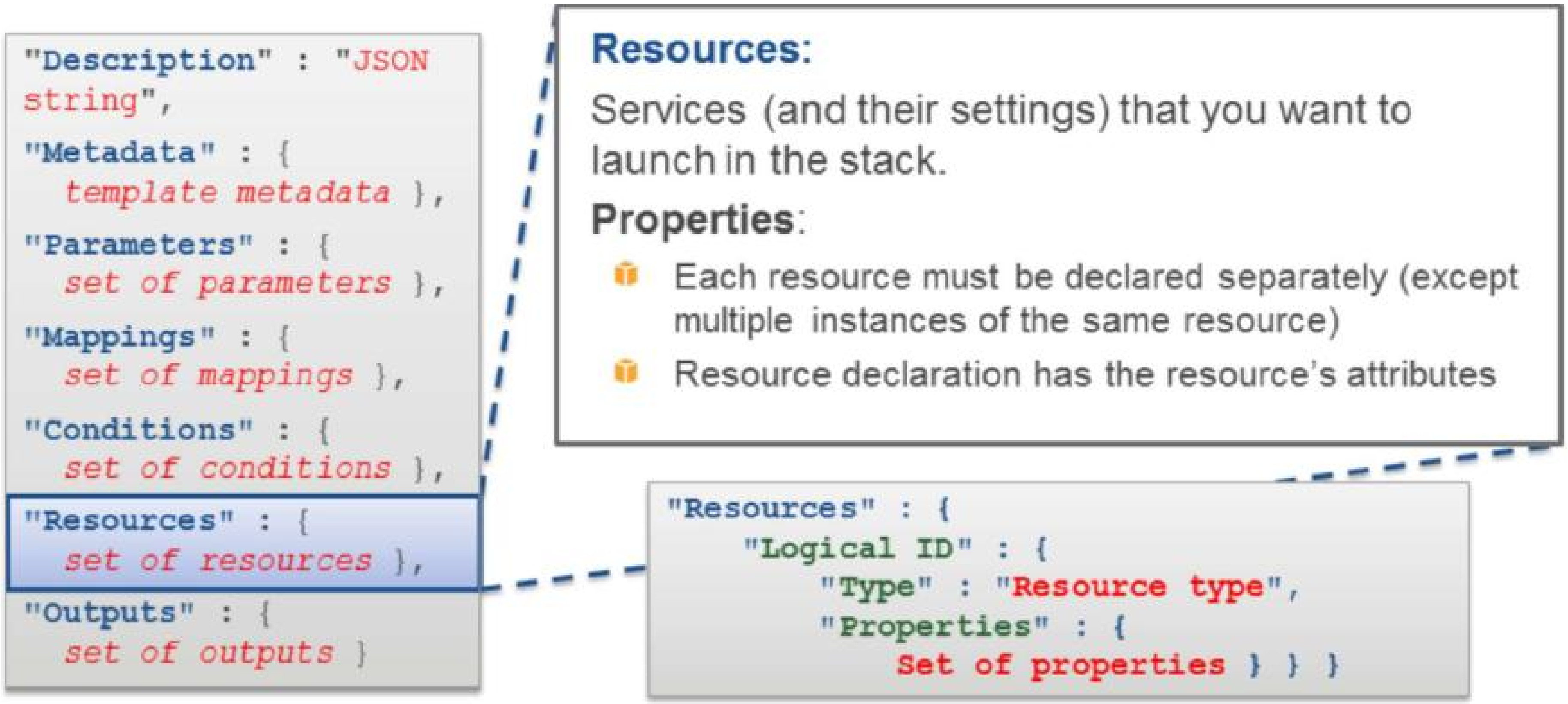
Metadata:

JSON objects that provide additional details about the template.

- Settings or configuration information that some CloudFormation features need to retrieve
- Can be specified at the template or resource level

```
"Metadata" : {
  "Instances" : {"Description" : "<Information
    about the instances>"},
  "Databases" : {"Description" : "<Information
    about the databases>"}}
```

Anatomy Of A CloudFormation Template



```
"Description" : "JSON
string",
"Metadata" : {
   template metadata },
"Parameters" : {
   set of parameters },
"Mappings" : {
   set of mappings },
"Conditions" : {
   set of conditions },
"Resources" : {
   set of resources },
"Outputs" : {
   set of outputs }
```

Resources:

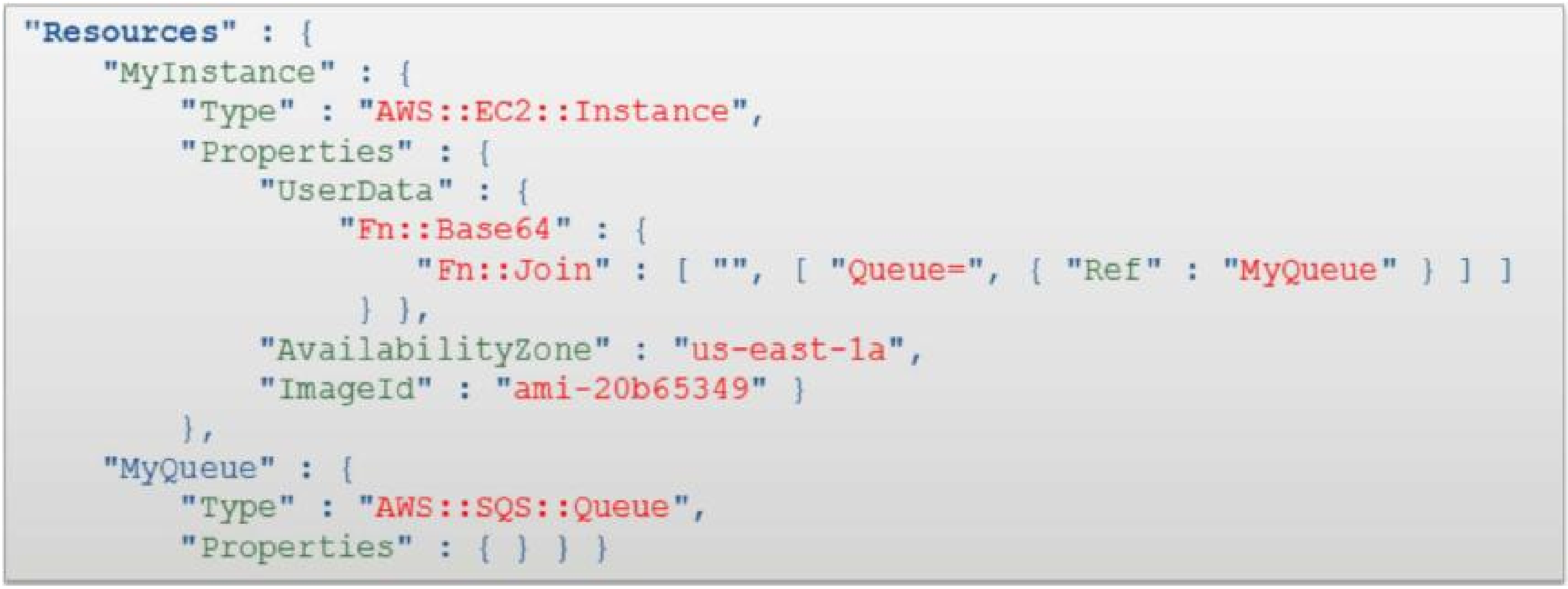
Services (and their settings) that you want to launch in the stack.

Properties:

- Each resource must be declared separately (except multiple instances of the same resource)
- Resource declaration has the resource's attributes

```
"Resources" : {
    "Logical ID" : {
        "Type" : "Resource type",
        "Properties" : {
            Set of properties } } }
```

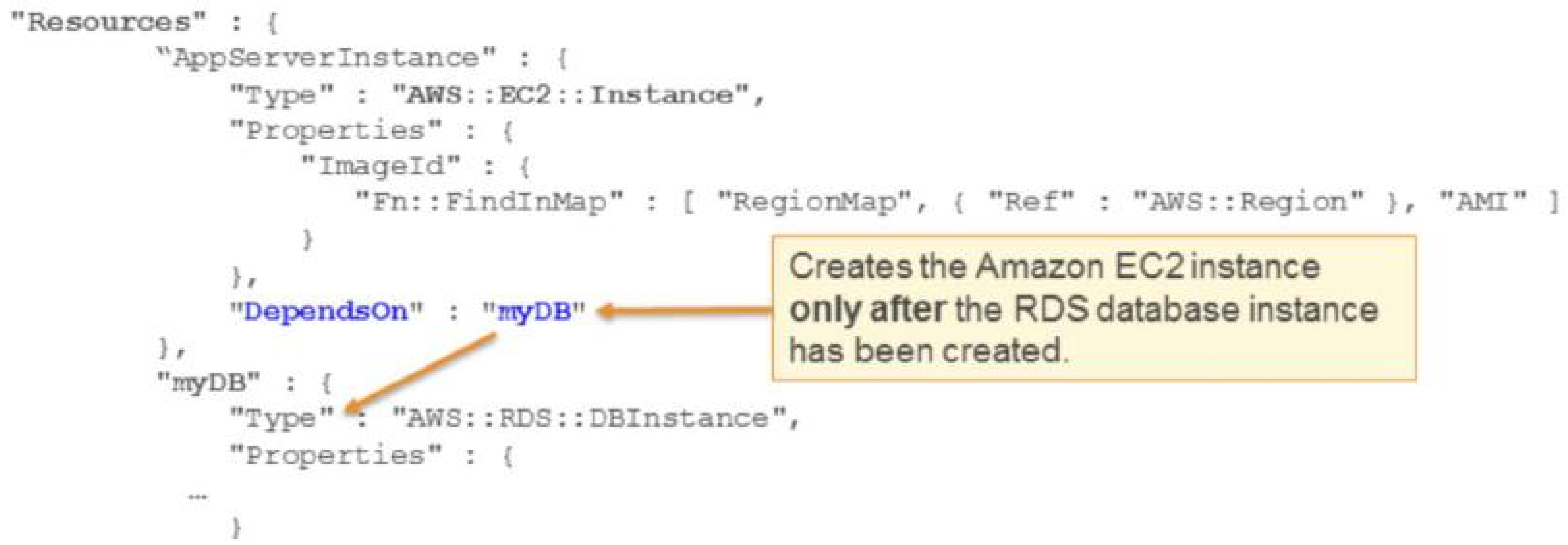
Anatomy Of A CloudFormation Template : Resources



```
"Resources" : {
    "MyInstance" : {
        "Type" : "AWS::EC2::Instance",
        "Properties" : {
            "UserData" : {
                "Fn::Base64" : {
                    "Fn::Join" : [ "", [ "Queue=", { "Ref" : "MyQueue" } ] ]
                } },
            "AvailabilityZone" : "us-east-1a",
            "ImageId" : "ami-20b65349" }
        },
    "MyQueue" : {
        "Type" : "AWS::SQS::Queue",
        "Properties" : { } } }
```

Resource Attribute: Depends On

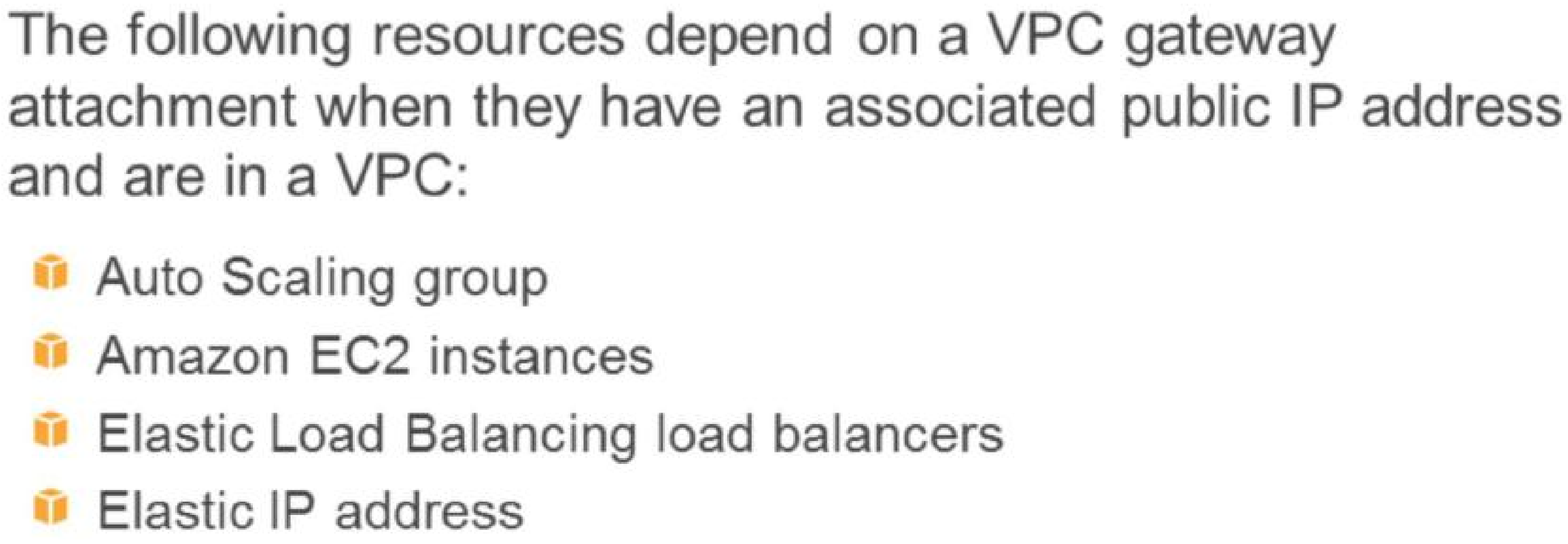
The "DependsOn" attribute specifies that the creation of a specific resource follows another. You can use the DependsOn attribute with any resource.



```
"Resources" : {
        "AppServerInstance" : {
            "Type" : "AWS::EC2::Instance",
            "Properties" : {
                "ImageId" : {
                    "Fn::FindInMap" : [ "RegionMap", { "Ref" : "AWS::Region" }, "AMI" ]
                }
            },
            "DependsOn" : "myDB"
        },
        "myDB" : {
            "Type" : "AWS::RDS::DBInstance",
            "Properties" : {
          …
            }
```

Creates the Amazon EC2 instance **only after** the RDS database instance has been created.

When A DependsOn Attribute Is Required?



The following resources depend on a VPC gateway attachment when they have an associated public IP address and are in a VPC:

- Auto Scaling group
- Amazon EC2 instances
- Elastic Load Balancing load balancers
- Elastic IP address

Special Resource: Wait Condition

Wait conditions are special CloudFormation resources that pause the creation of the stack and wait for a signal before it continues.

Use a wait condition to coordinate the creation of stack resources with other configuration actions external to the stack creation.

```
"myWaitCondition" : {
    "Type" : "AWS::CloudFormation::WaitCondition",
    "DependsOn" : "Ec2Instance",
    "Properties" : {
        "Handle" : { "Ref" : "myWaitHandle" },
         "Timeout" : "4500"
    }
}
```

Wait condition that begins after the successful creation of the "Ec2Instance" resource

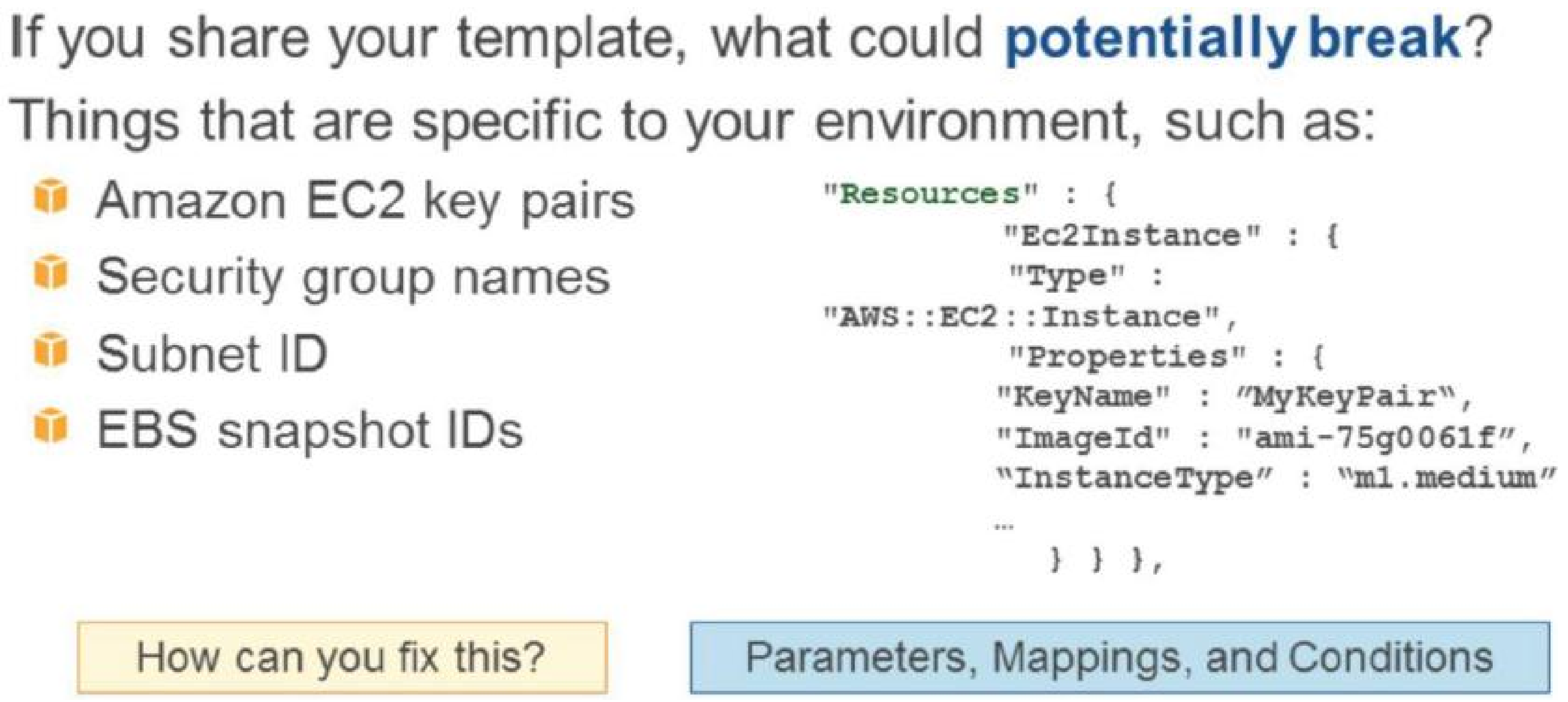
Using Creation Policies In A Template

Pause the creation of the stack and wait for a specified number of success signals before it continues:

```
"AutoScalingGroup": {
  "Type": "AWS::AutoScaling::AutoScalingGroup",
  "Properties": {
    "AvailabilityZones": { "Fn::GetAZs": "" },
    "LaunchConfigurationName": { "Ref": "LaunchConfig" },
    "DesiredCapacity": "3",
    "MinSize": "1",
    "MaxSize": "4"
  },
  "CreationPolicy": {
    "ResourceSignal": {
      "Count": "3",
      "Timeout": "PT15M"
    }
  },
```

Creation policy that waits for three success signals but times out after 15 minutes.

What Is Going To Break?



If you share your template, what could **potentially break**?

Things that are specific to your environment, such as:

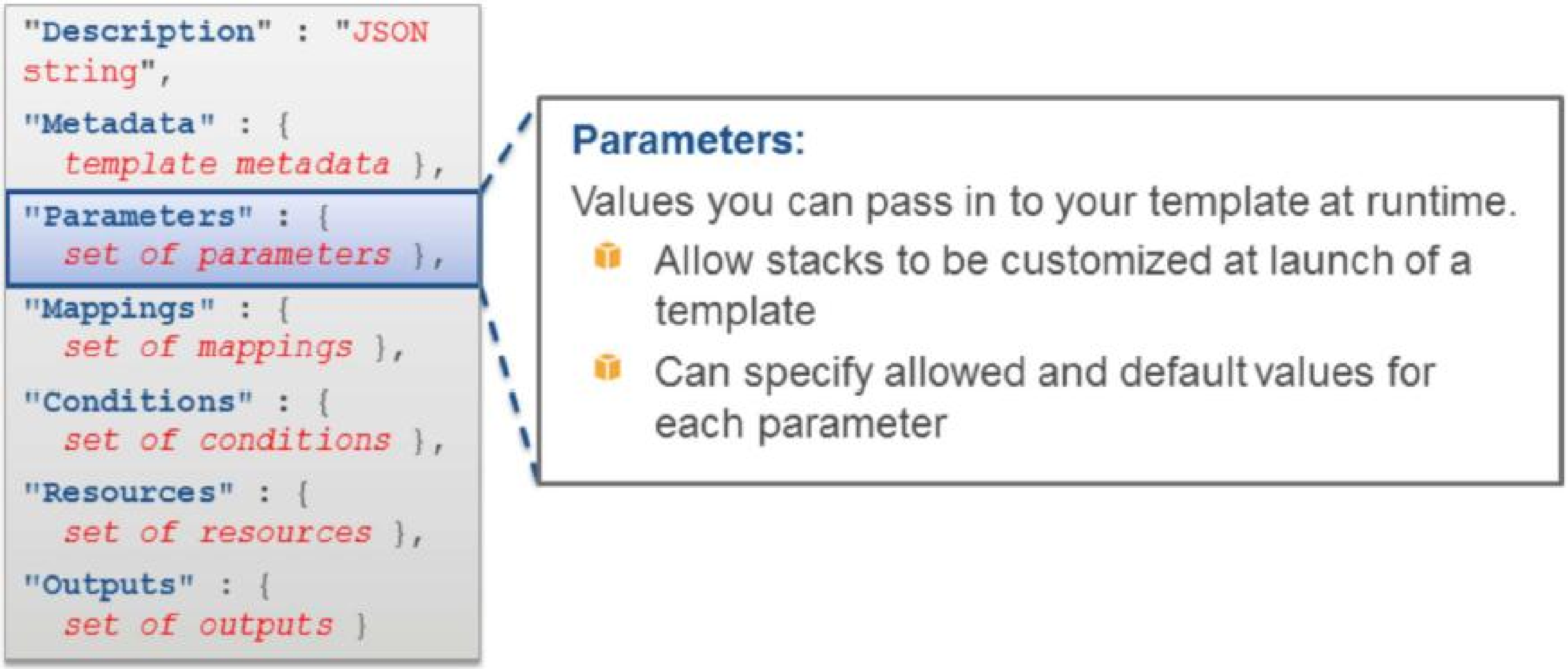
- Amazon EC2 key pairs
- Security group names
- Subnet ID
- EBS snapshot IDs

```
"Resources" : {
           "Ec2Instance" : {
            "Type" :
"AWS::EC2::Instance",
            "Properties" : {
           "KeyName" : "MyKeyPair",
           "ImageId" : "ami-75g0061f",
           "InstanceType" : "m1.medium"
           ...
              } } },
```

How can you fix this?

Parameters, Mappings, and Conditions

Anatomy Of A CloudFormation Template



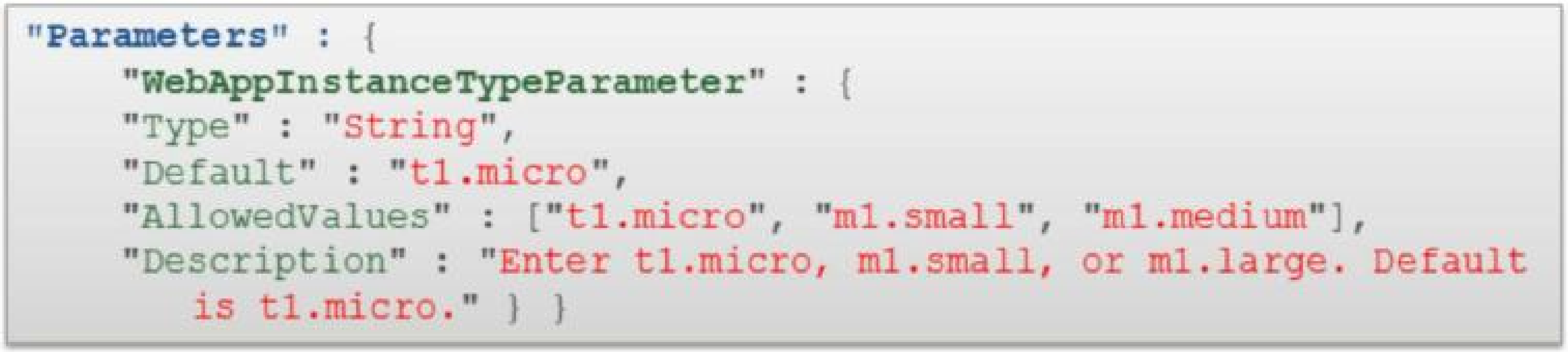
```
"Description" : "JSON
string",
"Metadata" : {
  template metadata },
"Parameters" : {
  set of parameters },
"Mappings" : {
  set of mappings },
"Conditions" : {
  set of conditions },
"Resources" : {
  set of resources },
"Outputs" : {
  set of outputs }
```

Parameters:

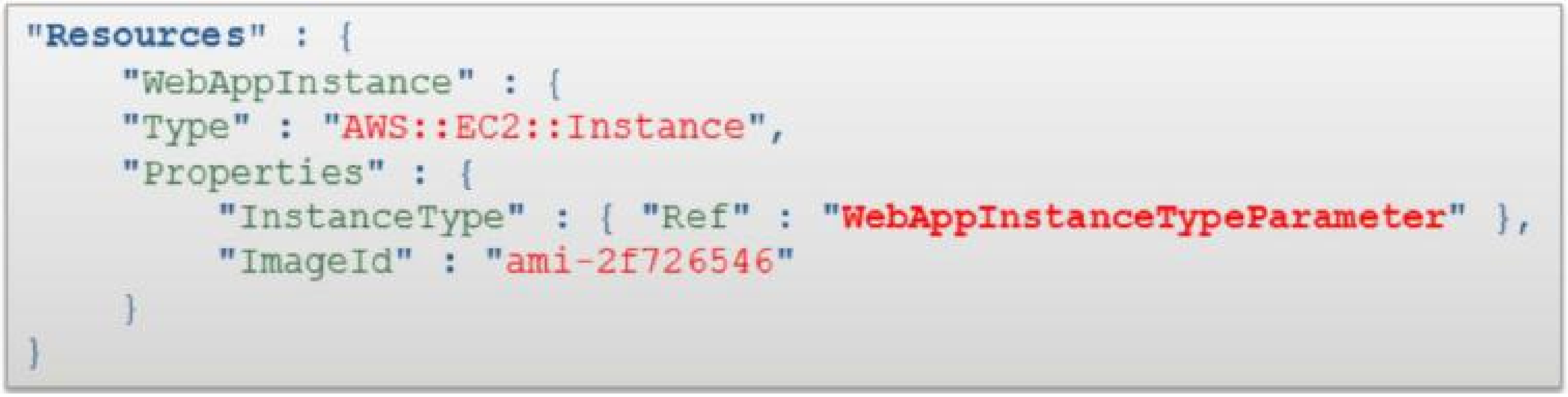
Values you can pass in to your template at runtime.

- Allow stacks to be customized at launch of a template
- Can specify allowed and default values for each parameter

CloudFormation Template: Parameters Example

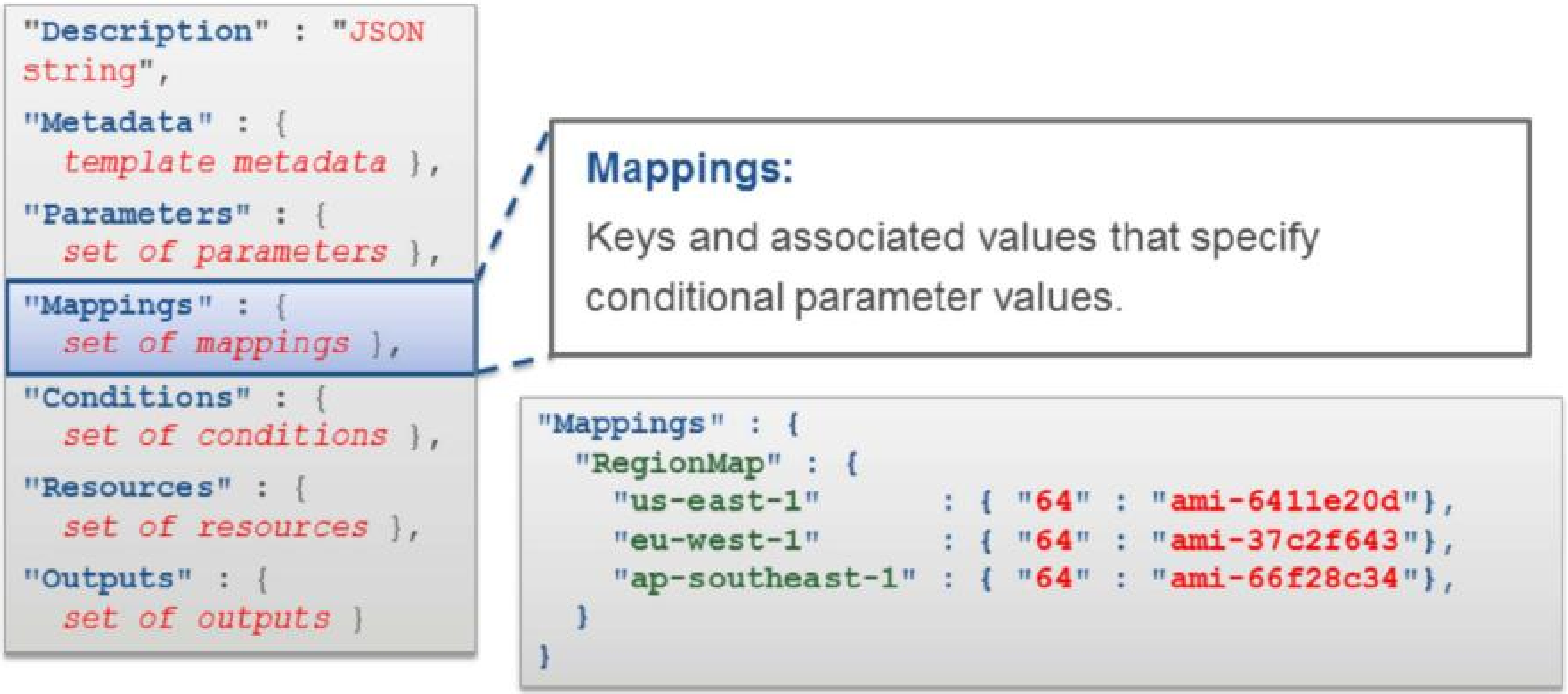


```
"Parameters" : {
     "WebAppInstanceTypeParameter" : {
     "Type" : "String",
     "Default" : "t1.micro",
     "AllowedValues" : ["t1.micro", "m1.small", "m1.medium"],
     "Description" : "Enter t1.micro, m1.small, or m1.large. Default
        is t1.micro." } }
```



```
"Resources" : {
     "WebAppInstance" : {
     "Type" : "AWS::EC2::Instance",
     "Properties" : {
          "InstanceType" : { "Ref" : "WebAppInstanceTypeParameter" },
          "ImageId" : "ami-2f726546"
     }
}
```

Anatomy Of A CloudFormation Template



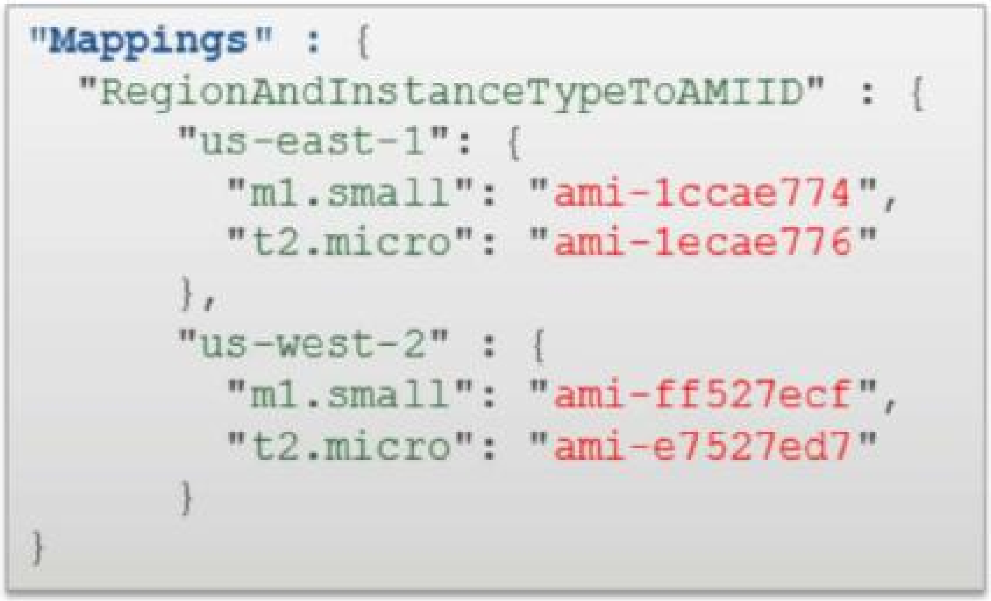
```
"Description" : "JSON
string",
"Metadata" : {
  template metadata },
"Parameters" : {
  set of parameters },
"Mappings" : {
  set of mappings },
"Conditions" : {
  set of conditions },
"Resources" : {
  set of resources },
"Outputs" : {
  set of outputs }
```

Mappings:

Keys and associated values that specify conditional parameter values.

```
"Mappings" : {
  "RegionMap" : {
    "us-east-1"      : { "64" : "ami-6411e20d"},
    "eu-west-1"      : { "64" : "ami-37c2f643"},
    "ap-southeast-1" : { "64" : "ami-66f28c34"},
  }
}
```

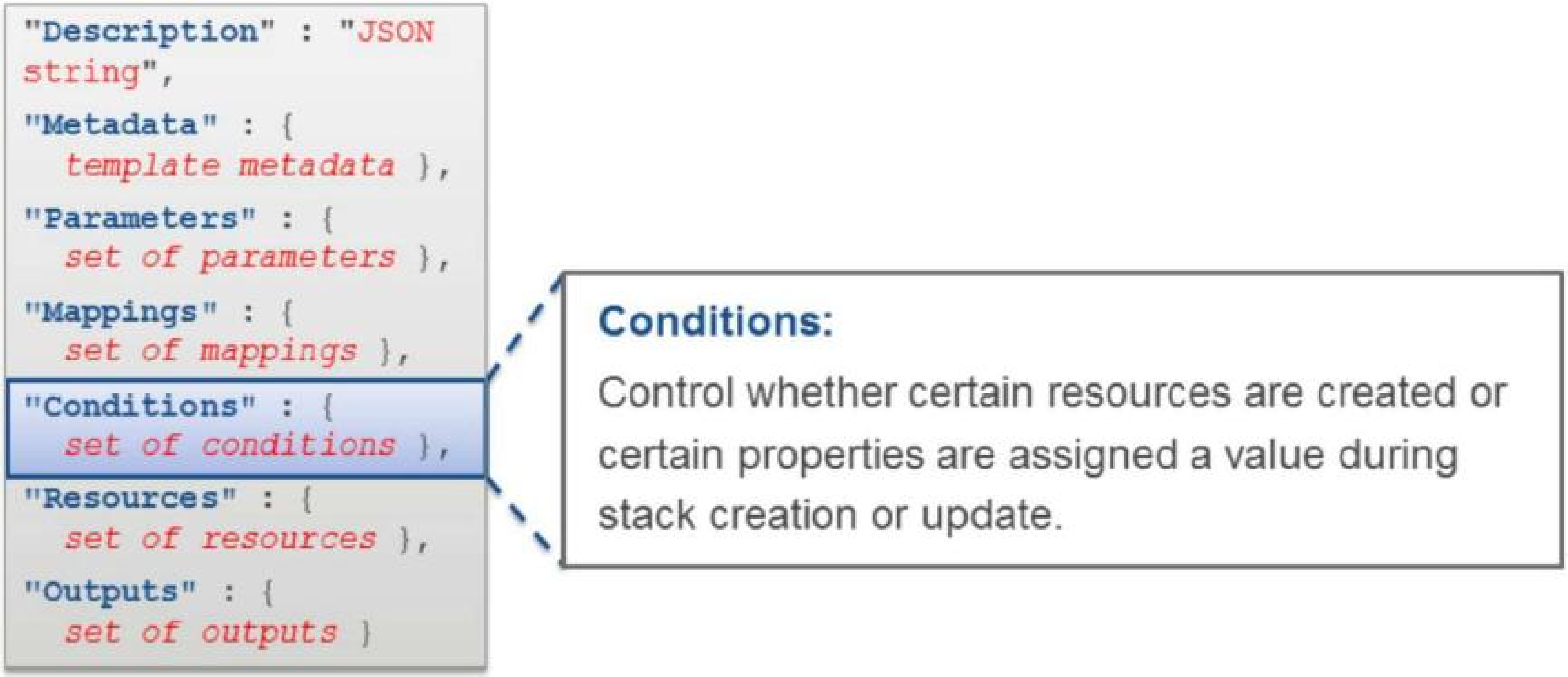
CloudFormation Template: Mappings Example



```
"Mappings" : {
  "RegionAndInstanceTypeToAMIID" : {
      "us-east-1": {
        "m1.small": "ami-1ccae774",
        "t2.micro": "ami-1ecae776"
      },
      "us-west-2" : {
        "m1.small": "ami-ff527ecf",
        "t2.micro": "ami-e7527ed7"
      }
}
```

Specify multiple mapping levels

Anatomy Of A CloudFormation Template

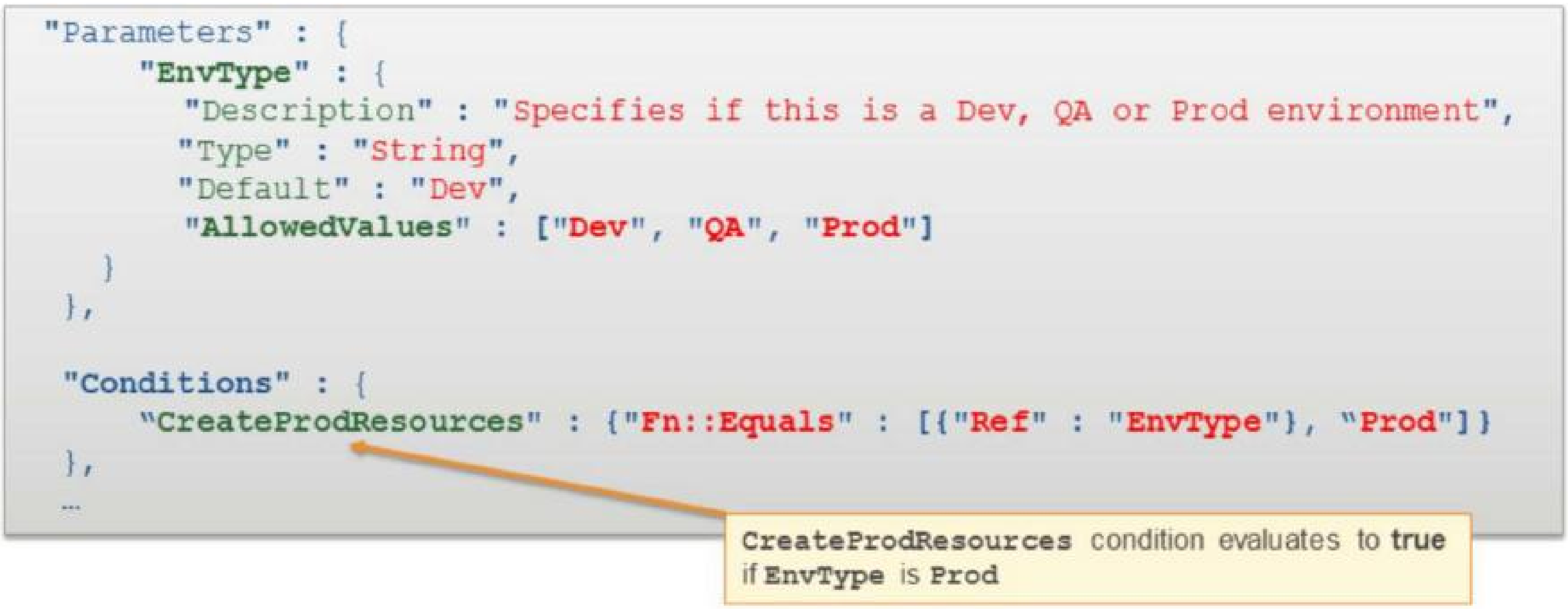


```
"Description" : "JSON
string",
"Metadata" : {
  template metadata },
"Parameters" : {
  set of parameters },
"Mappings" : {
  set of mappings },
"Conditions" : {
  set of conditions },
"Resources" : {
  set of resources },
"Outputs" : {
  set of outputs }
```

Conditions:

Control whether certain resources are created or certain properties are assigned a value during stack creation or update.

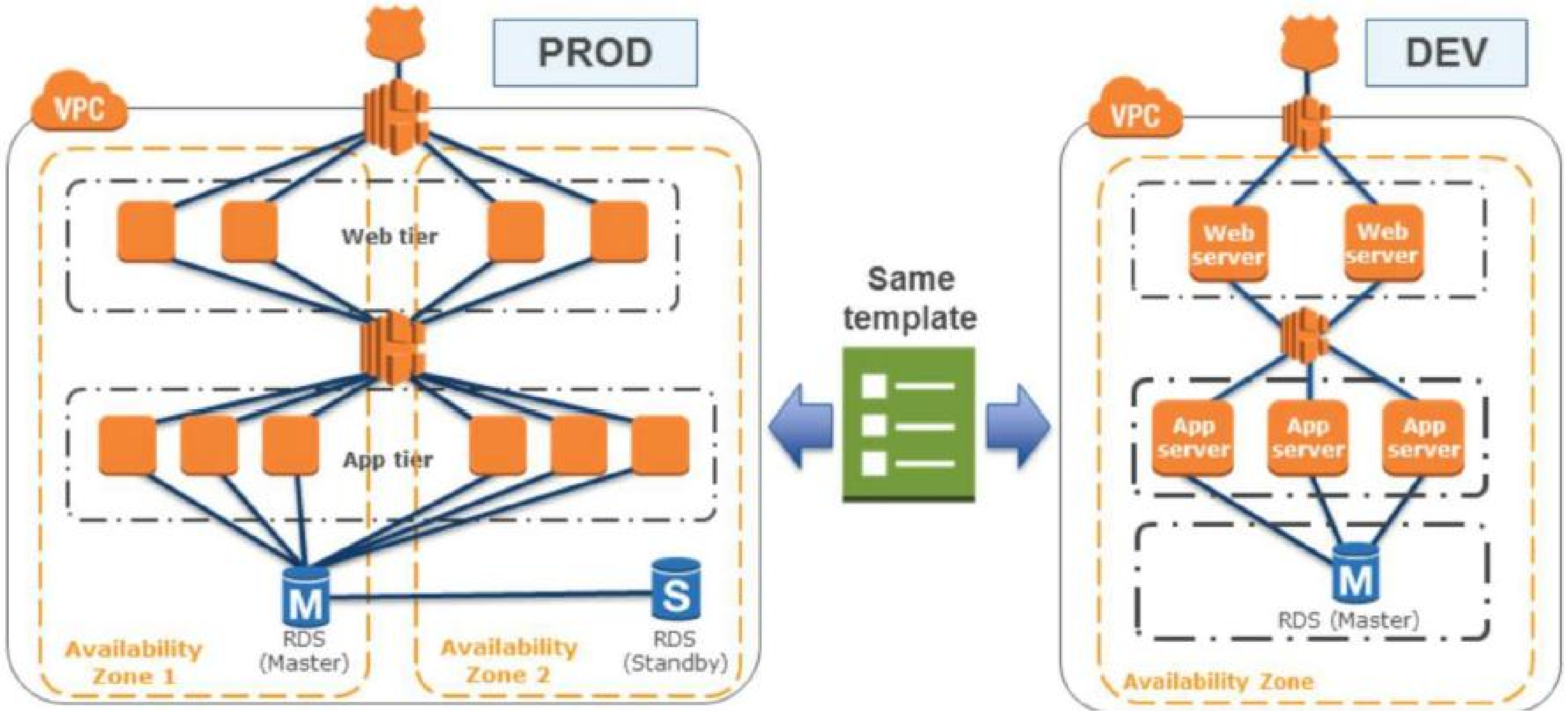
Anatomy Of A CloudFormation Template : Conditions



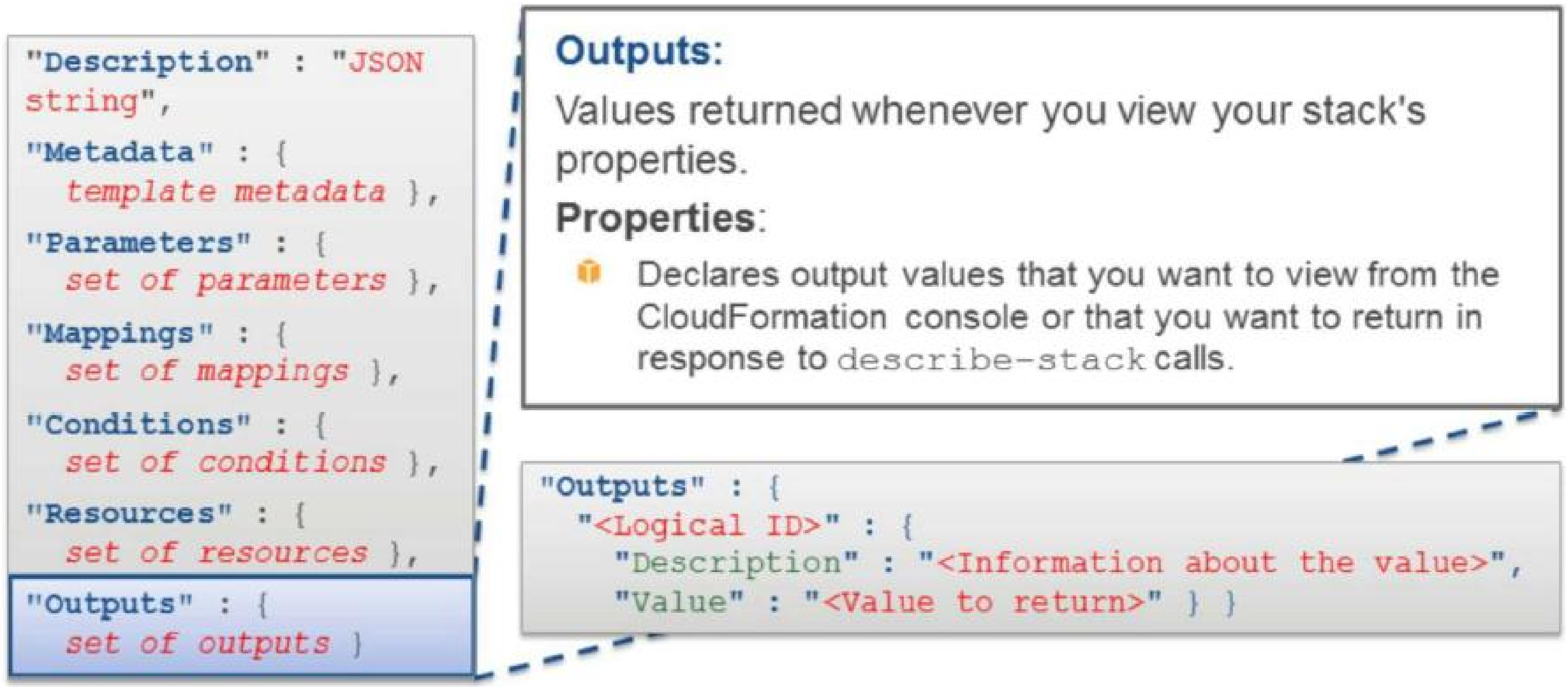
```
"Parameters" : {
     "EnvType" : {
       "Description" : "Specifies if this is a Dev, QA or Prod environment",
       "Type" : "String",
       "Default" : "Dev",
       "AllowedValues" : ["Dev", "QA", "Prod"]
   }
 },

 "Conditions" : {
     "CreateProdResources" : {"Fn::Equals" : [{"Ref" : "EnvType"}, "Prod"]}
 },
 ...
```

CreateProdResources condition evaluates to true if EnvType is Prod

Building Environments With Conditions



Anatomy Of A CloudFormation Template



```
"Description" : "JSON
string",
"Metadata" : {
  template metadata },
"Parameters" : {
  set of parameters },
"Mappings" : {
  set of mappings },
"Conditions" : {
  set of conditions },
"Resources" : {
  set of resources },
"Outputs" : {
  set of outputs }
```

Outputs:

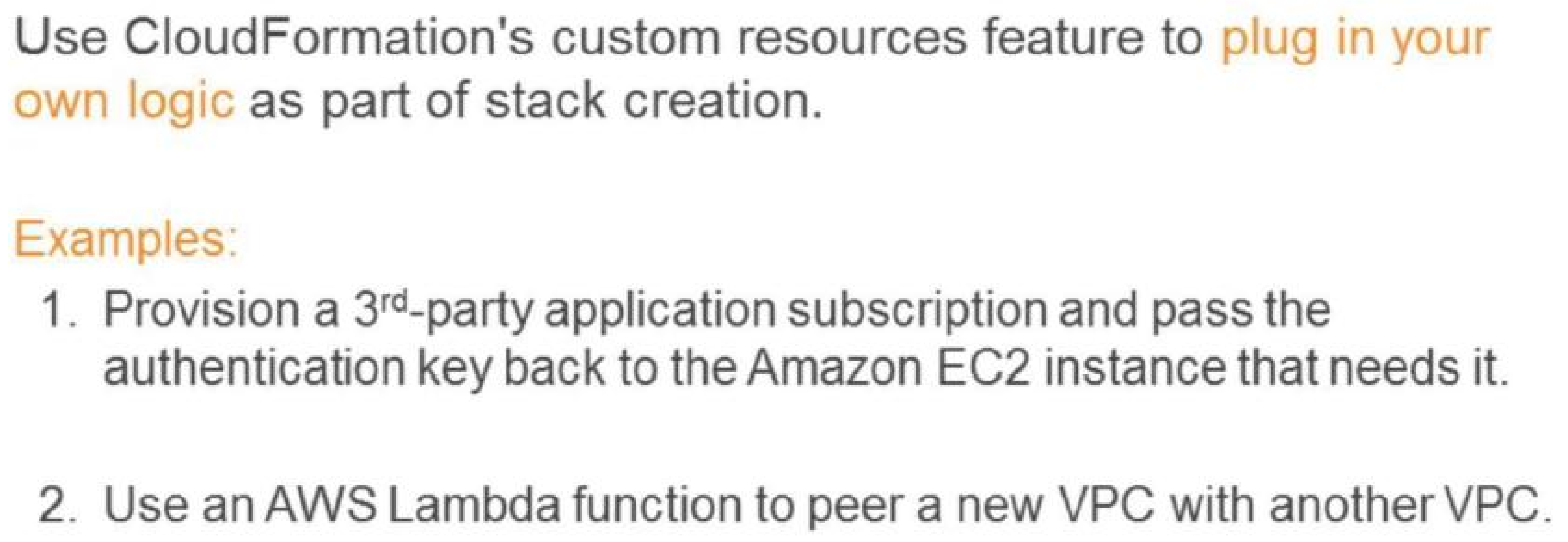
Values returned whenever you view your stack's properties.

Properties:

- Declares output values that you want to view from the CloudFormation console or that you want to return in response to `describe-stack` calls.

```
"Outputs" : {
  "<Logical ID>" : {
    "Description" : "<Information about the value>",
    "Value" : "<Value to return>" } }
```

Extending CloudFormation With *Custom Resources*



Use CloudFormation's custom resources feature to plug in your own logic as part of stack creation.

Examples:

1. Provision a 3rd-party application subscription and pass the authentication key back to the Amazon EC2 instance that needs it.

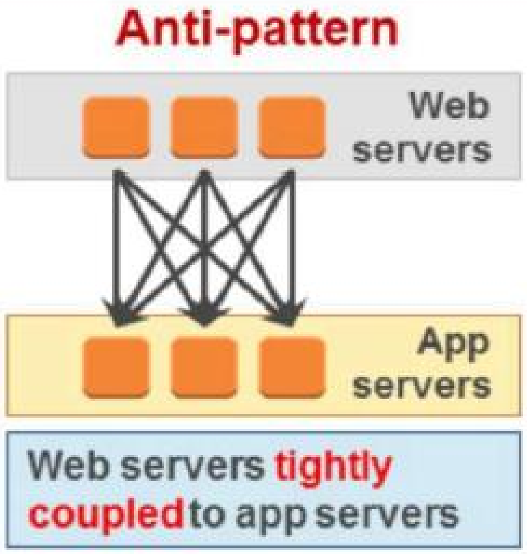
2. Use an AWS Lambda function to peer a new VPC with another VPC.

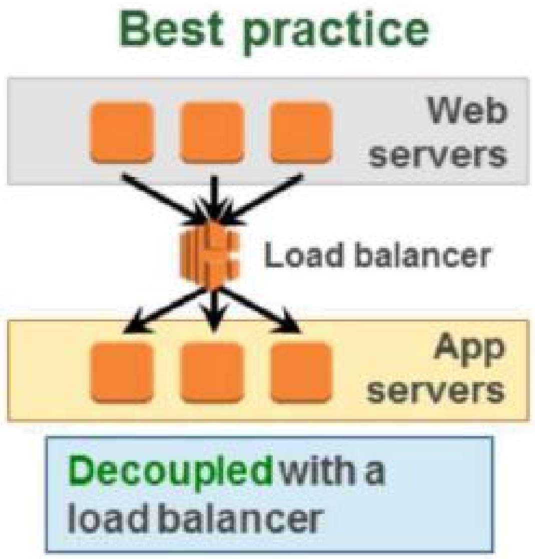
Decoupling Infrastructure

Best Practice: Loosely Couple Your App Components

Design architectures with independent components.

Reduce interdependencies so that the change or failure of one component does not affect other components.

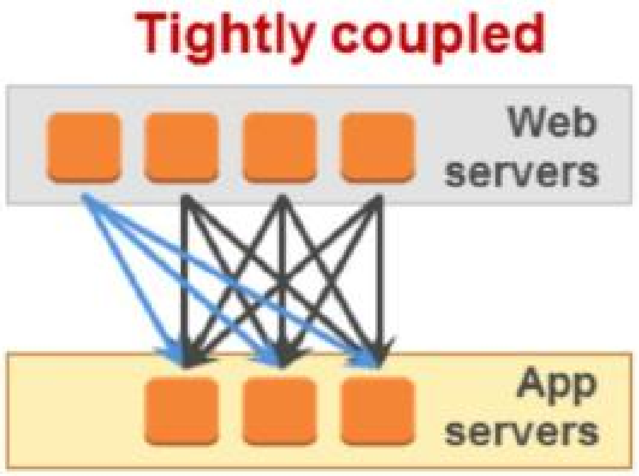


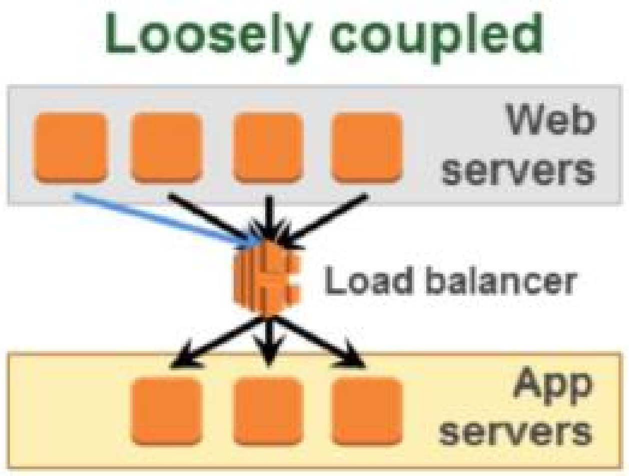


Decoupling

The more loosely your system is coupled:

- The more easily it scales.

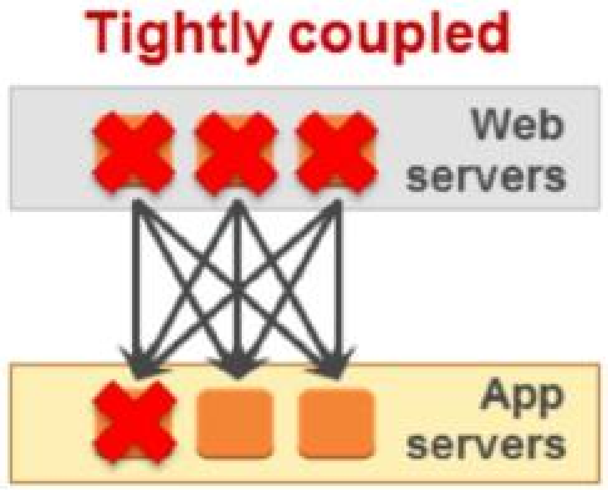


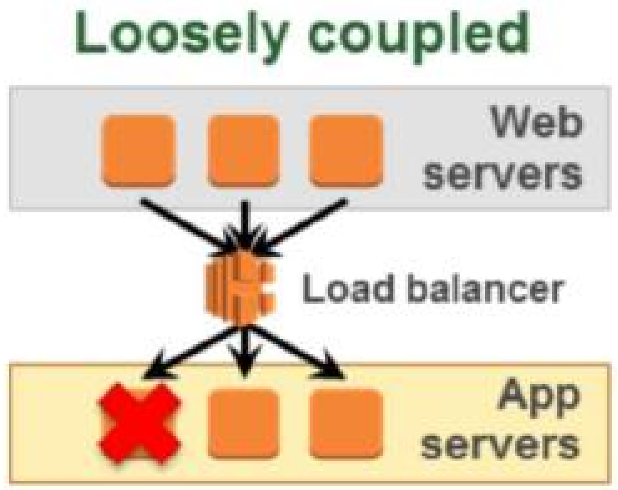


Decoupling

The more loosely your system is coupled:

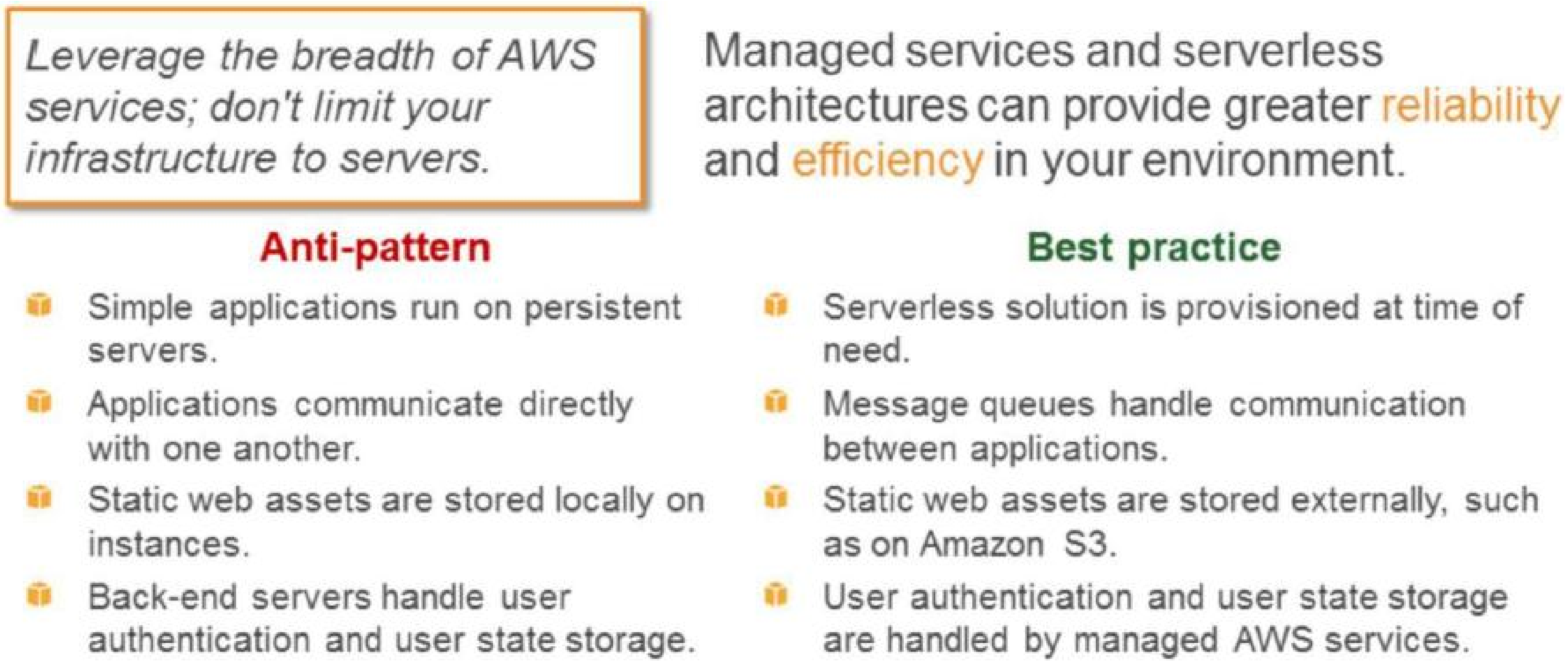
- The more easily it scales.
- The more fault-tolerant it can be.





Loose Coupling Strategies

Best Practices: Design Services, Not Servers

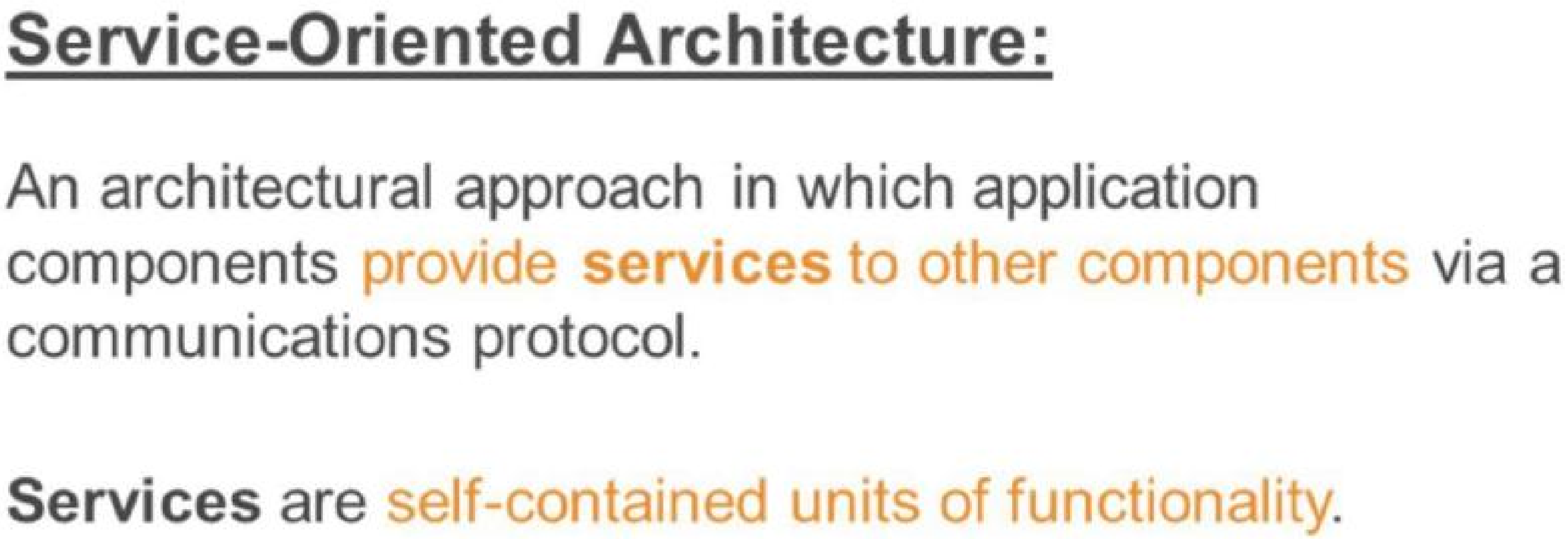


Leverage the breadth of AWS services; don't limit your infrastructure to servers.

Managed services and serverless architectures can provide greater reliability and efficiency in your environment.

Anti-pattern	Best practice
Simple applications run on persistent servers.	Serverless solution is provisioned at time of need.
Applications communicate directly with one another.	Message queues handle communication between applications.
Static web assets are stored locally on instances.	Static web assets are stored externally, such as on Amazon S3.
Back-end servers handle user authentication and user state storage.	User authentication and user state storage are handled by managed AWS services.

Implement a SOA
(Service Oriented Architecture)

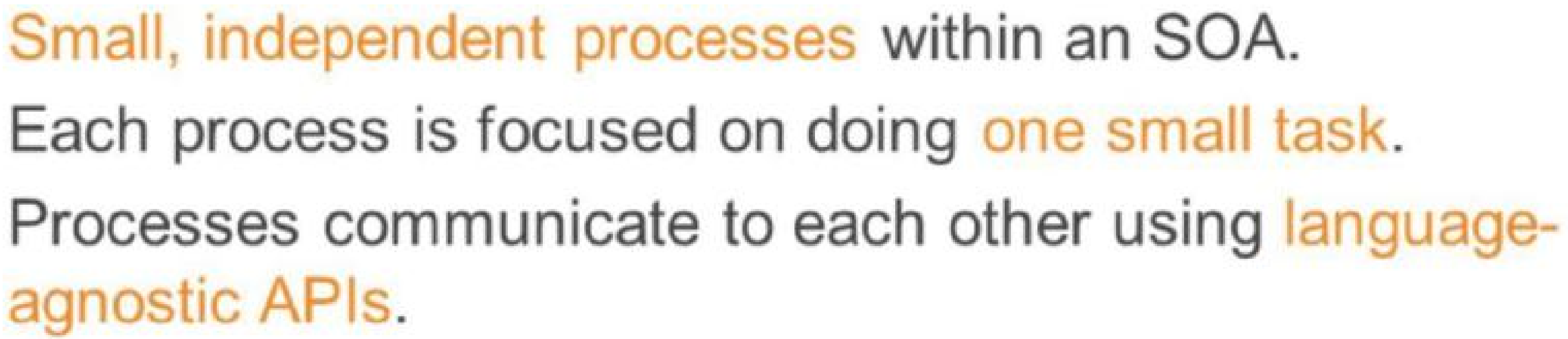


Service-Oriented Architecture:

An architectural approach in which application components provide **services** to other components via a communications protocol.

Services are self-contained units of functionality.

Microservices and Decoupling



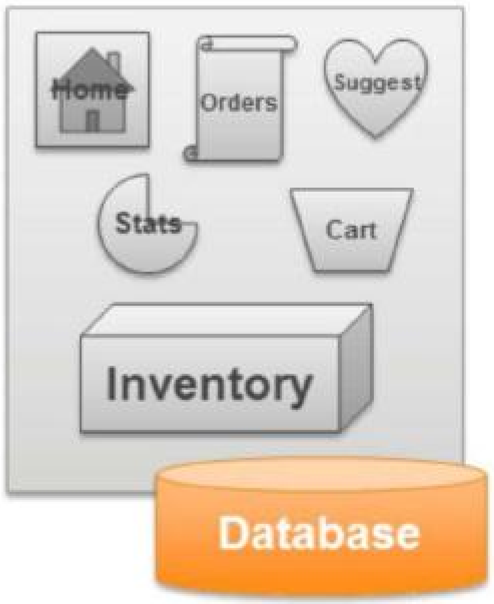
Small, independent processes within an SOA.

Each process is focused on doing one small task.

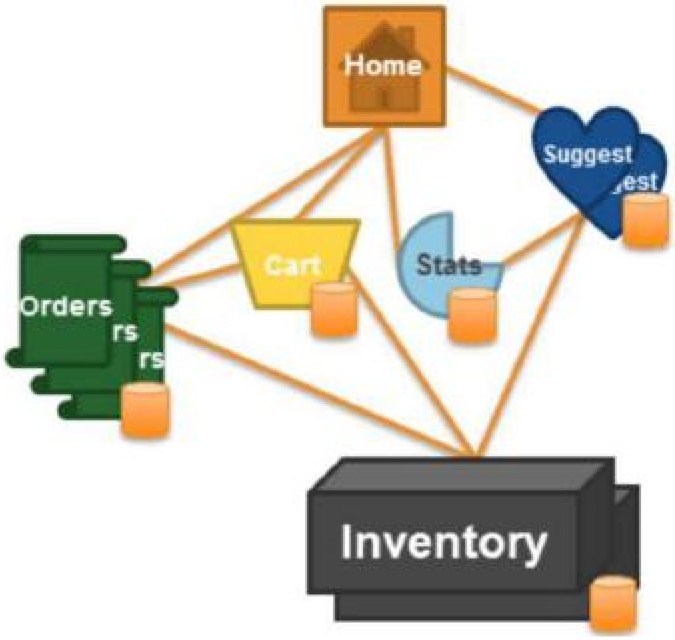
Processes communicate to each other using language-agnostic APIs.

Comparing Architectural Styles

Traditional app architectures are monolithic:

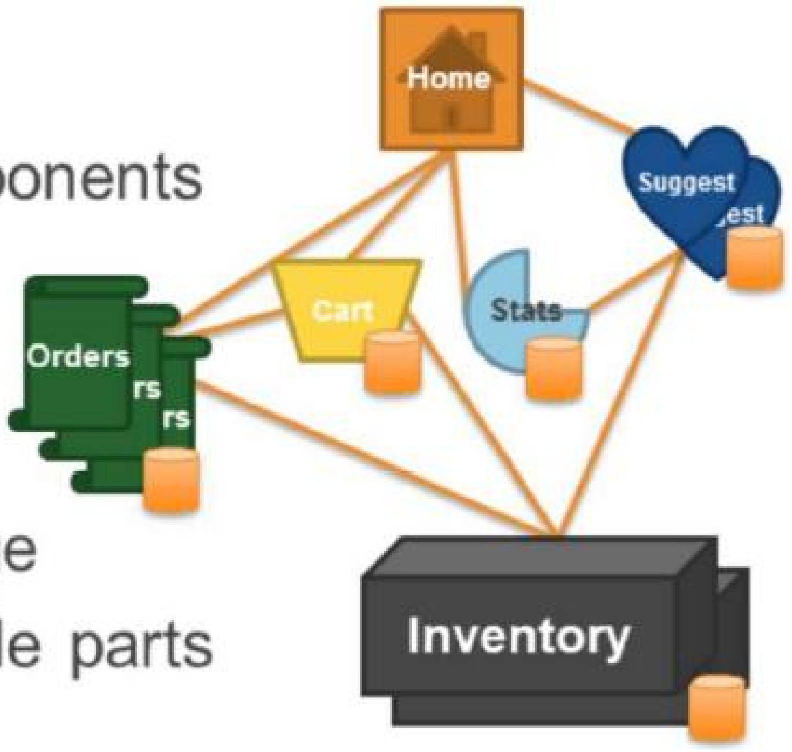


Microservice-based architectures are loosely coupled:



Microservices

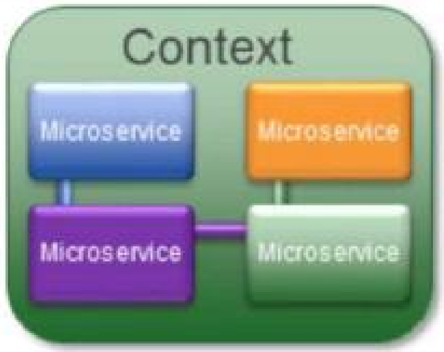
- Split features into individual components
- Have smaller parts to iterate on
- Have a reduced test surface area
- Benefit from a lower risk of change
- Use individual horizontally scalable parts



Microservices Concepts : Bounding

Each business domain can be divided into contexts.

- Contexts are made up of microservices.
- Each context has its own functions, objects, and language.
- Contexts can comprise single operations like:
  - Transaction handling
  - User registration, authentication, tracking
  - Content publishing or syndication
  - Catalog, warehouse management



Best Practices 1/2

Change components without breaking them:

- The interface is a contract.
- Modifying capabilities should not affect consumers.

Use a simple API:

- Lowers the cost of using your service.
- More complexity means more resistance to change.
- The less you share, the less will break.
- Allows you to hide the details.

Best Practices 2/2

Keep it technology-agnostic:

- Enable change: be ready to embrace the next evolution
- Be tech-omnivorous

Design with failure in mind:

- “Everything fails, all the time.”

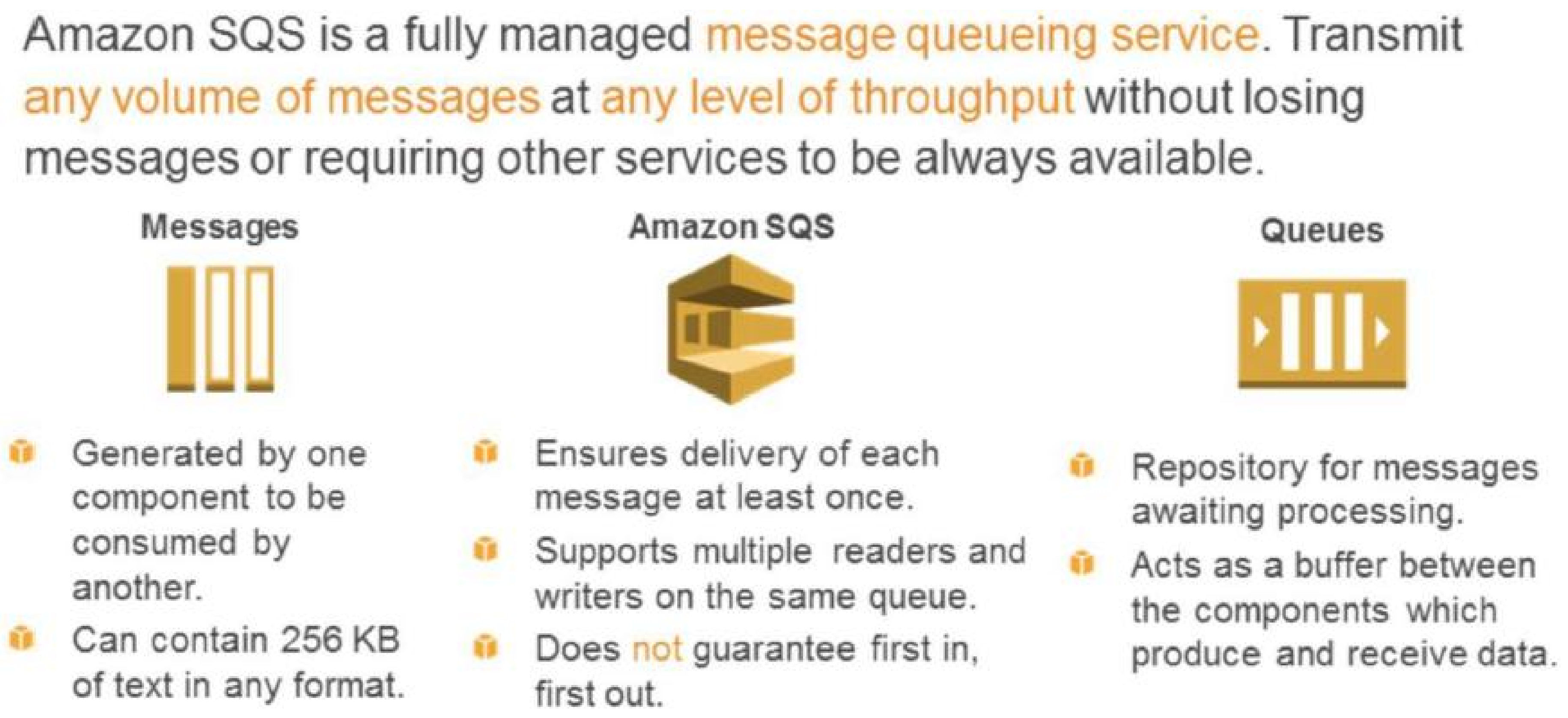
Monitor your environment:

- Not just the infrastructure
- Extracting health status means collecting it from services

What can be used to easily and reliably communicate between components?

Amazon Simple Queue Service (SQS)

Amazon SQS



Amazon SQS is a fully managed message queueing service. Transmit any volume of messages at any level of throughput without losing messages or requiring other services to be always available.

Messages

- Generated by one component to be consumed by another.
- Can contain 256 KB of text in any format.

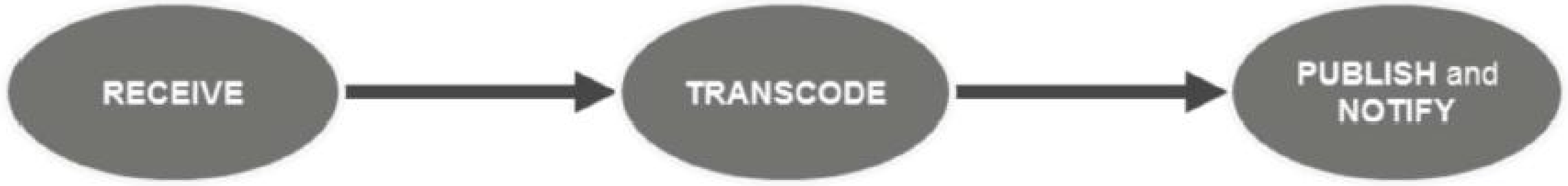
Amazon SQS

- Ensures delivery of each message at least once.
- Supports multiple readers and writers on the same queue.
- Does not guarantee first in, first out.

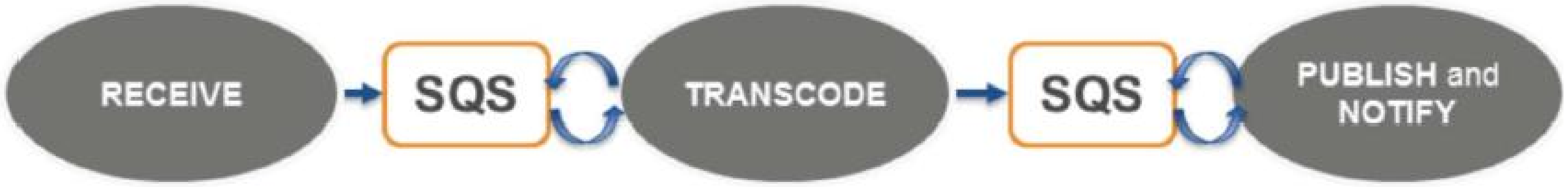
Queues

- Repository for messages awaiting processing.
- Acts as a buffer between the components which produce and receive data.

Tightly Coupled System

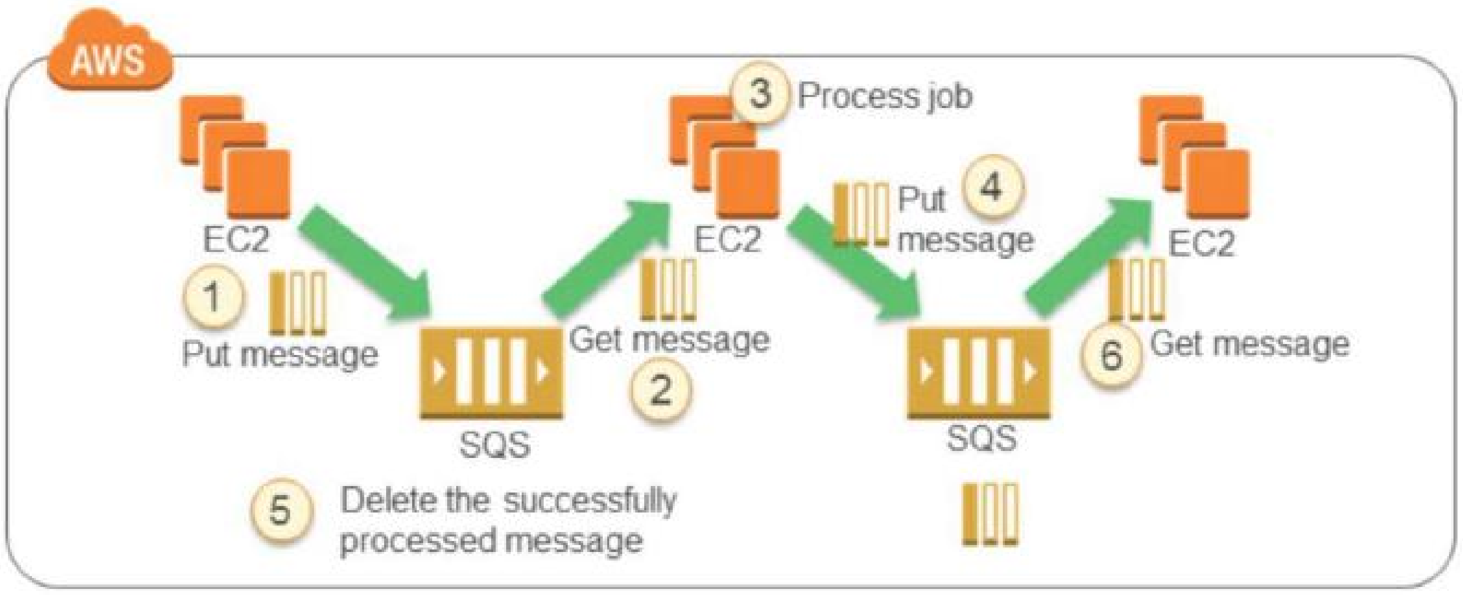


Loosely Coupled System



Loose Coupling With Amazon SQS

The queuing chain pattern enables asynchronous processing:



Note: Queuing chain pattern is one of the cloud design patterns. For more information, see http://en.clouddesignpattern.org/index.php/CDP:Queuing_Chain_Pattern

Amazon SQS Characteristics

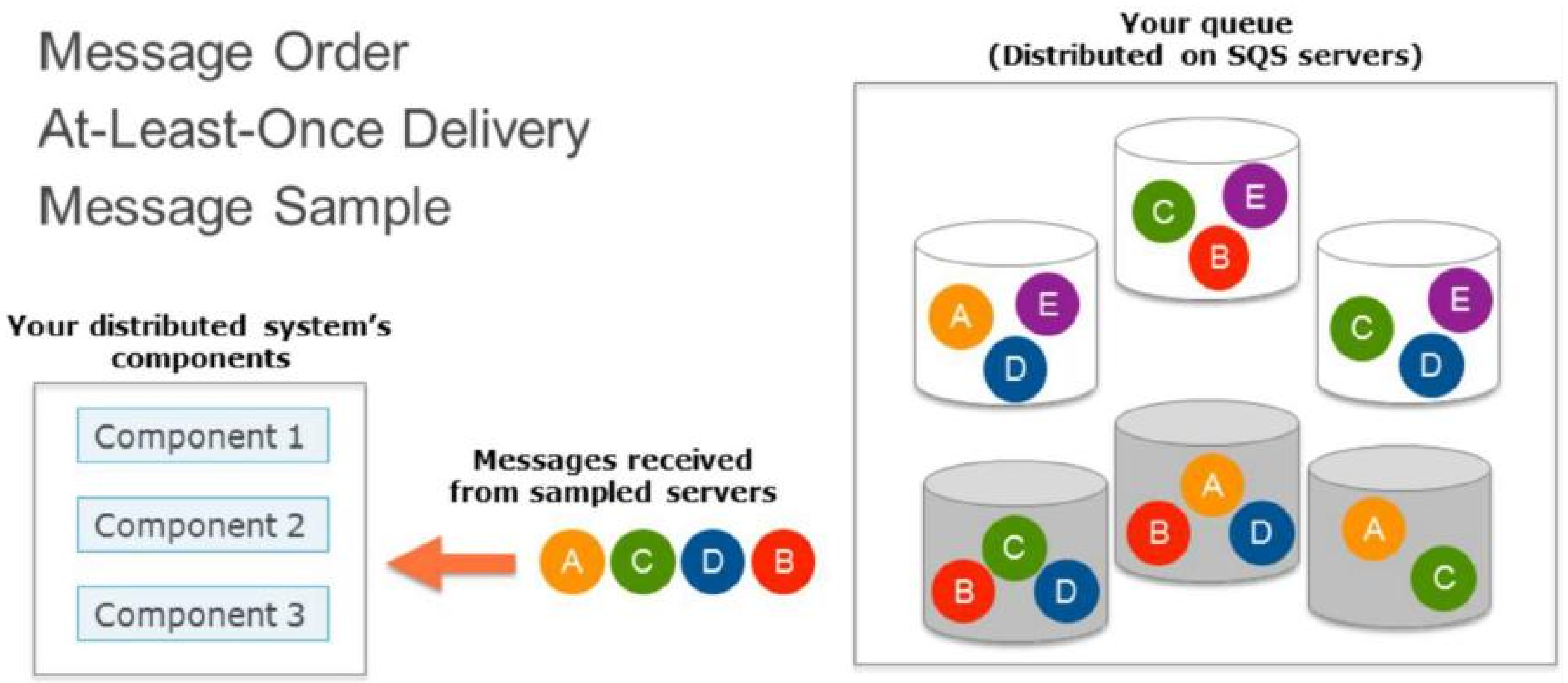
- Extremely scalable
  - Potentially millions of messages.
- Extremely reliable
  - All messages are stored redundantly on multiple servers and in multiple data centers.
- Simple to use
  - Messages get sent in, messages get pulled out.
- Simultaneous read/write
- Secure
  - API credentials are needed.

Understanding Distributed Queues

Message Order

At-Least-Once Delivery

Message Sample

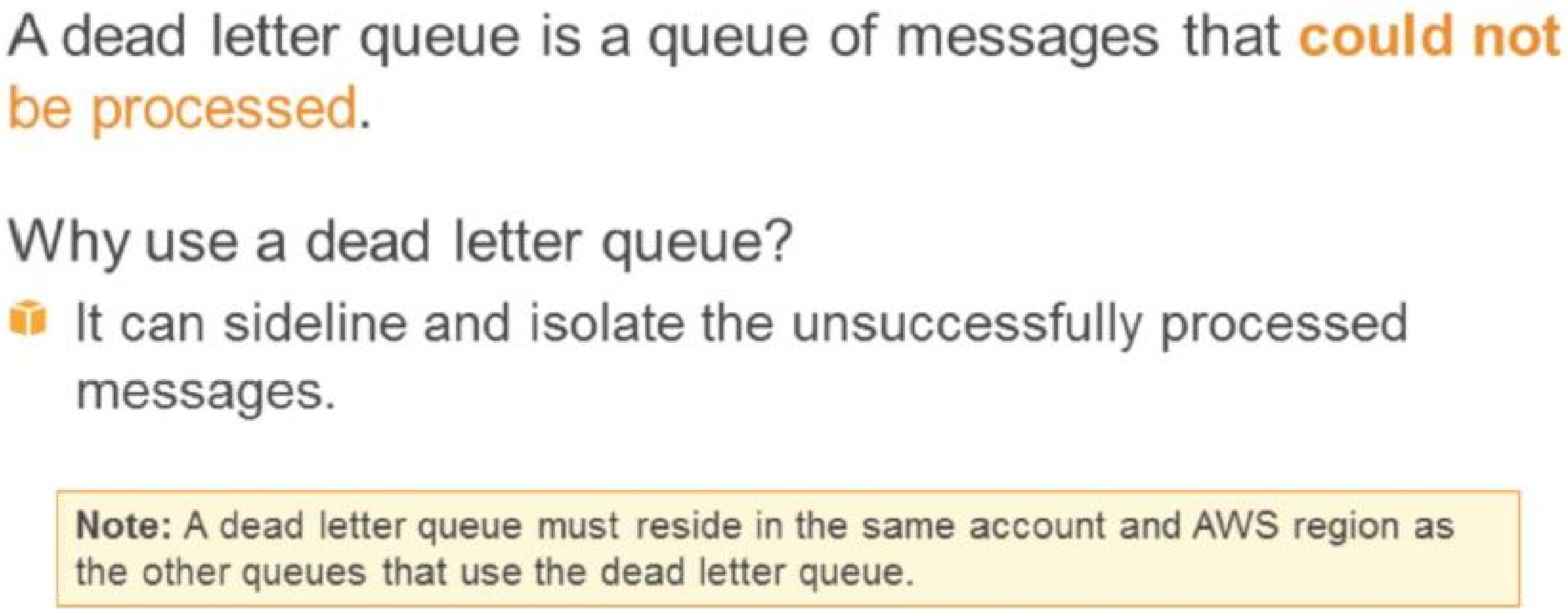

Your queue (Distributed on SQS servers)
Your distributed system's components
Component 1
Component 2
Component 3
Messages received from sampled servers
A
C
D
B
E

Amazon SQS Visibility TimeOut

Visibility timeout prevents multiple components from processing the same message.

- When a message is received, it becomes “locked” while being processed. This prevents it from being processed by other computers.
- The component that receives the message processes it and then deletes it from the queue.
- If the message processing fails, the lock will expire and the message will be available again (fault tolerance).

Amazon SQS : Dead Letter Queue



A dead letter queue is a queue of messages that **could not be processed**.

Why use a dead letter queue?

- It can sideline and isolate the unsuccessfully processed messages.

Note: A dead letter queue must reside in the same account and AWS region as the other queues that use the dead letter queue.

Queue Sharing

Shared queues

- Queues can be shared with other AWS accounts and anonymously.
- A **permission** gives access to another person to use your queue in some particular way.
- A **policy** is the actual document that contains the permissions you granted.

Who pays for shared queue access?

- The queue **owner** pays for shared queue access.

AWS SQS Use Cases

- **Work Queues**

  Decouple components of a distributed application that may not all process the same amount of work simultaneously.

- **Buffering Batch Operations**

  Add scalability and reliability to your architecture and smooth out temporary volume spikes without losing messages or increasing latency.

- **Request Offloading**

  Move slow operations off of interactive request paths by enqueueing the request.

- **Fan-out**

  Combine Amazon SQS with Amazon SNS to send identical copies of a message to multiple queues in parallel for simultaneous processing.

- **Auto Scaling**

  Use Amazon SQS queues to help determine the load on an application, and when combined with Auto Scaling, you can scale the number of Amazon EC2 instances out or in, depending on the volume of traffic.

Other Decoupling Services in AWS

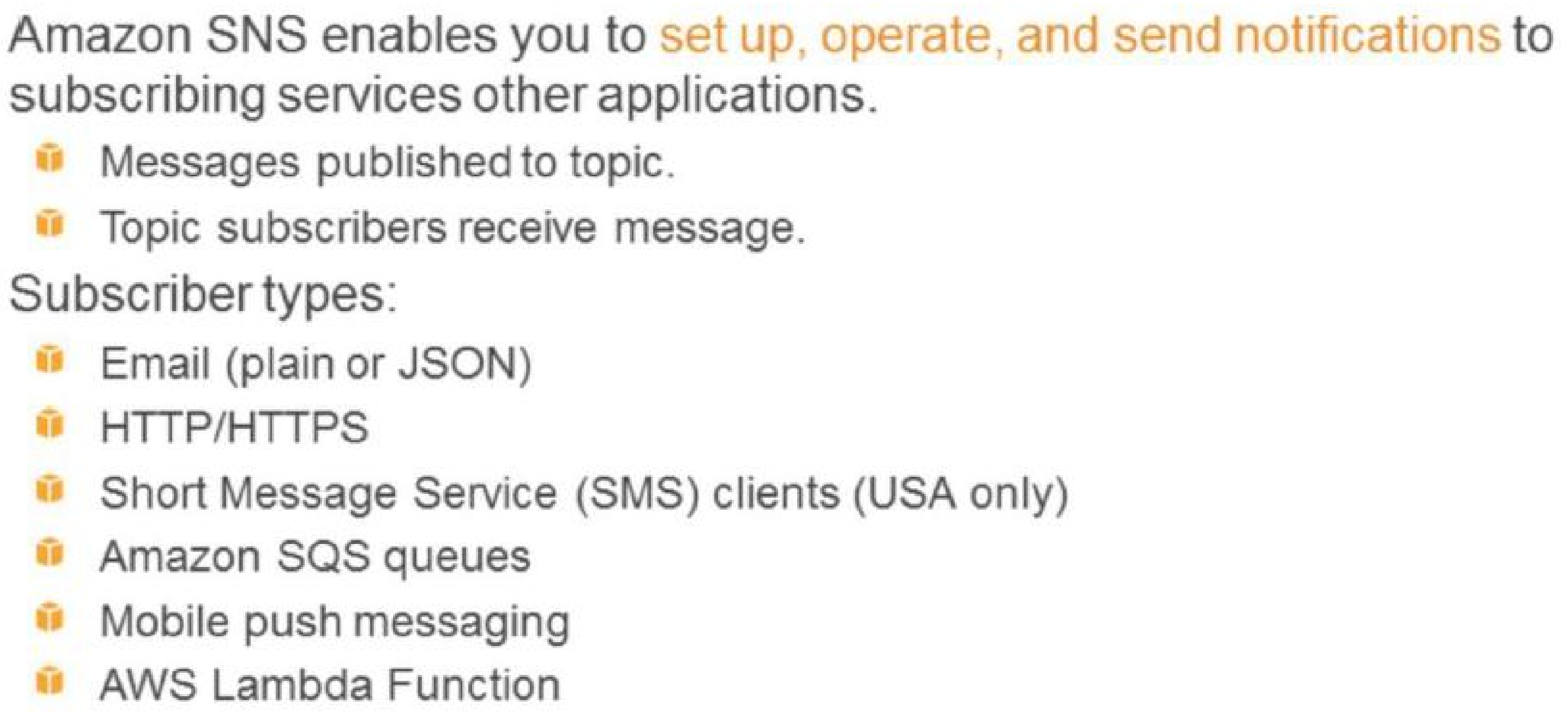
Amazon Simple Notification Service (SNS)

Amazon DynamoDB

Amazon API Gateway

AWS Lambda

AWS Simple Notification Service (SNS)



Amazon SNS enables you to set up, operate, and send notifications to subscribing services other applications.

- Messages published to topic.
- Topic subscribers receive message.

Subscriber types:

- Email (plain or JSON)
- HTTP/HTTPS
- Short Message Service (SMS) clients (USA only)
- Amazon SQS queues
- Mobile push messaging
- AWS Lambda Function

AWS SNS Characteristics

- **Single published message**

  All notification messages will contain a single published message.

- **Order is not guaranteed**

  Amazon SNS will attempt to deliver messages from the publisher in the order they were published into the topic. However, network issues could potentially result in out-of-order messages at the subscriber end.

- **No recall**

  When a message is delivered successfully, there is no recall feature.

- **HTTP/HTTPS retry**

  An Amazon SNS Delivery Policy can be used to control the retry pattern (linear, geometric, exponential backoff), maximum and minimum retry delays, and other parameters.

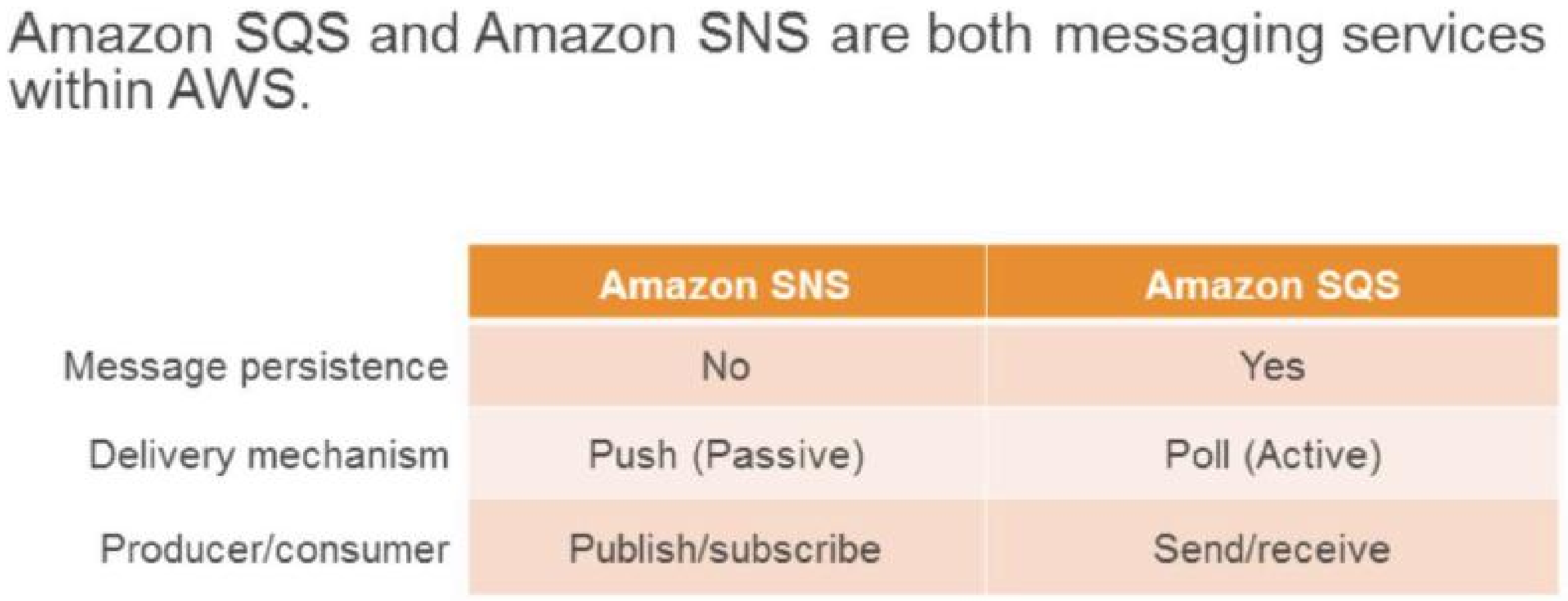
- **64 KB per message (non-SMS)**

  XML, JSON, and unformatted text.

- **256 KB max (four requests of 64 KB each)**

  Each 64-KB chunk of published data is billed as one request. For example, a single API call with a 256-KB payload will be billed as four requests.

SQS vs SNS



Amazon SQS and Amazon SNS are both messaging services within AWS.

	Amazon SNS	Amazon SQS
Message persistence	No	Yes
Delivery mechanism	Push (Passive)	Poll (Active)
Producer/consumer	Publish/subscribe	Send/receive

Use Case: Fan-Out

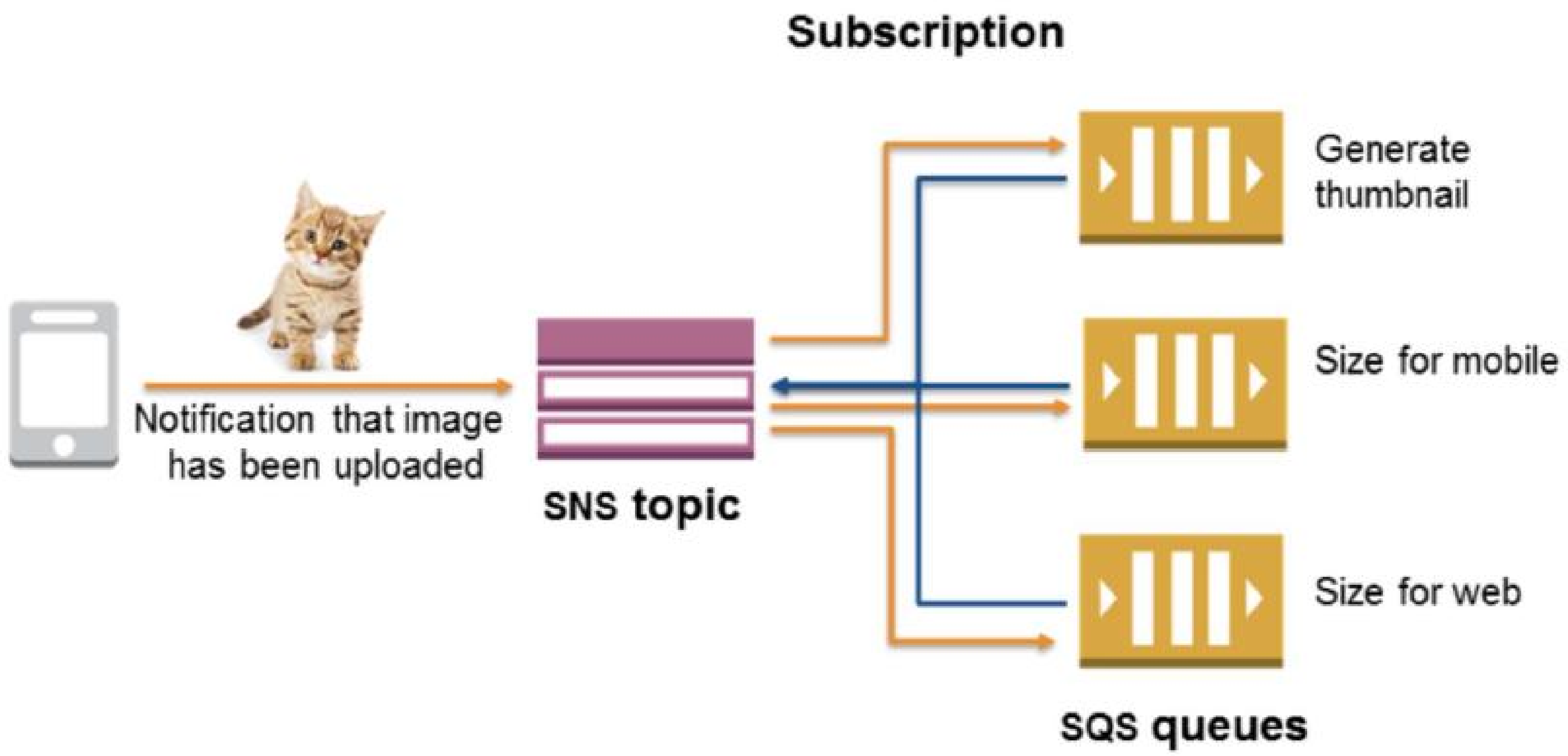
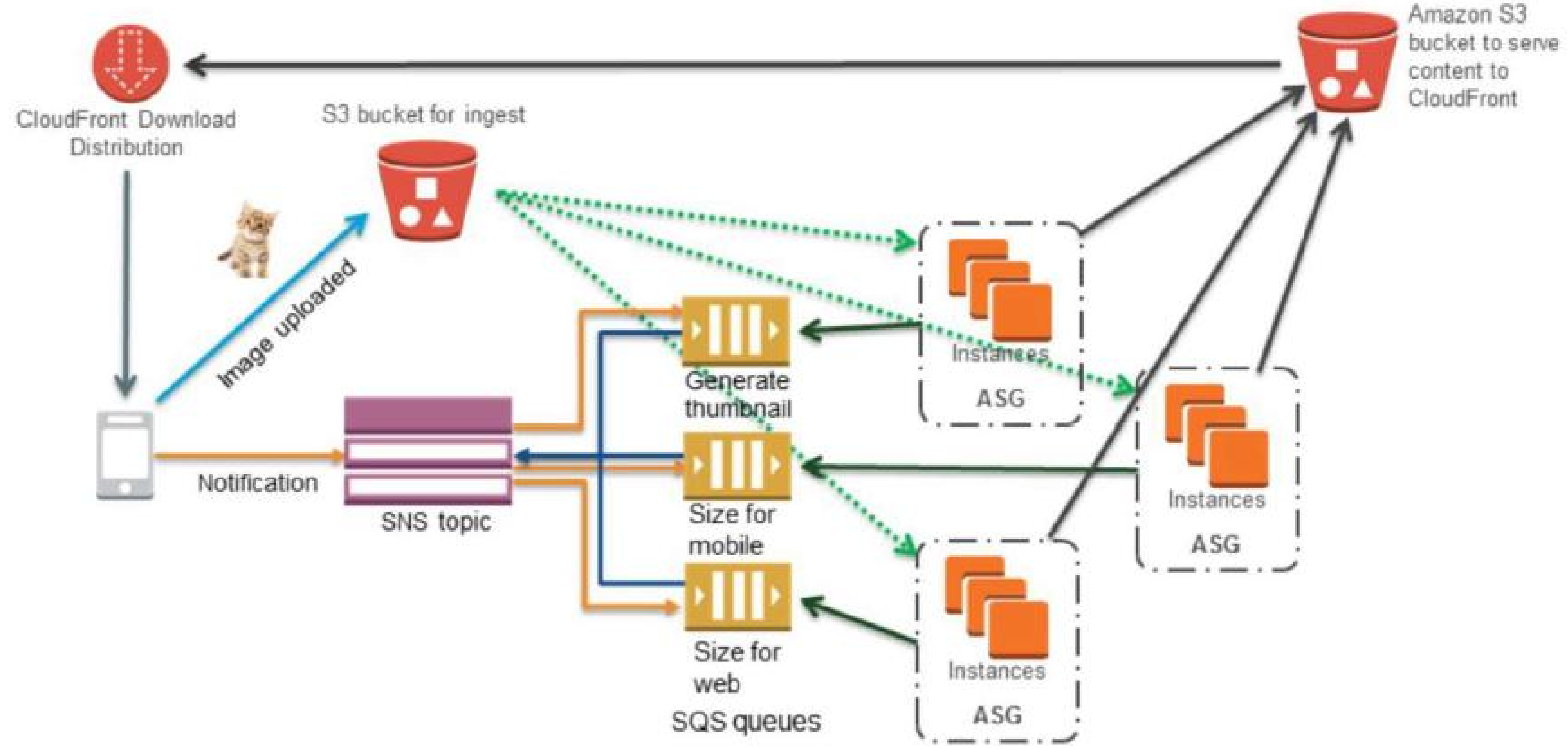
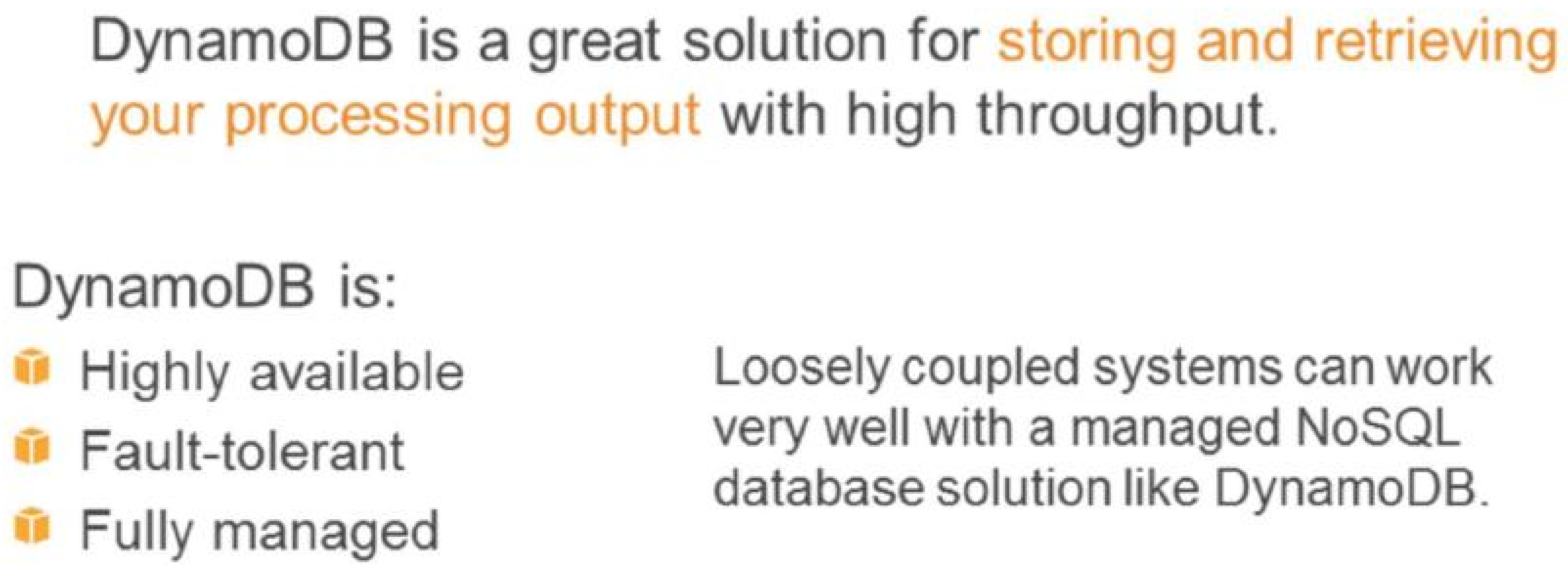


Image Processing Scenario



AWS DynamoDB



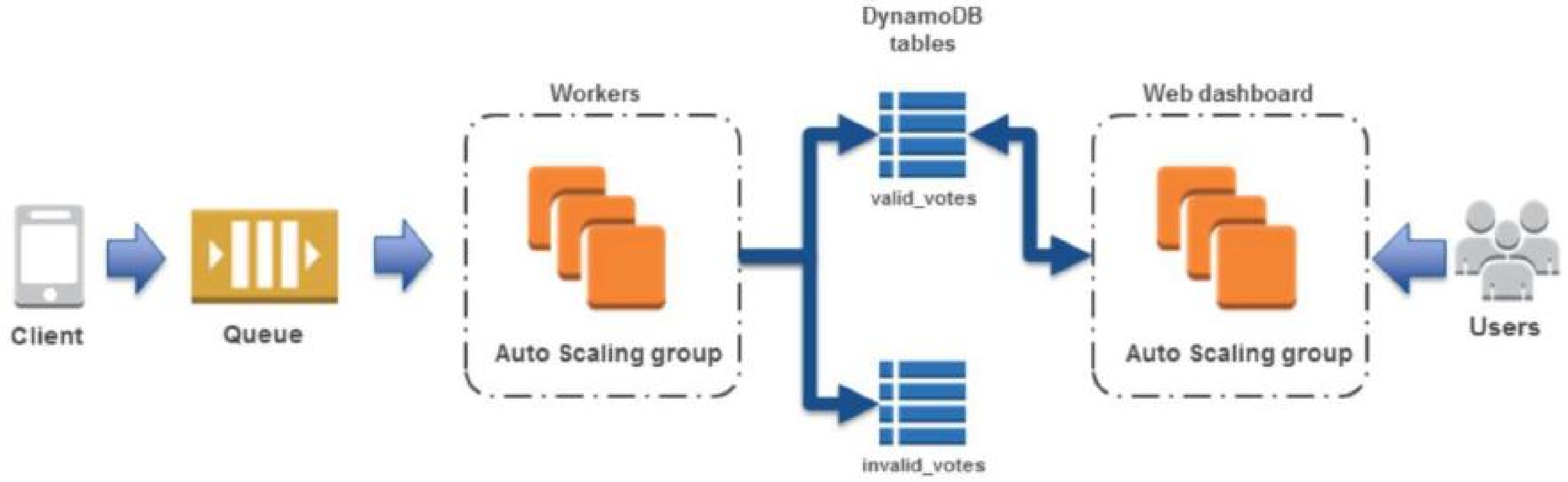
DynamoDB is a great solution for storing and retrieving your processing output with high throughput.

DynamoDB is:

- Highly available
- Fault-tolerant
- Fully managed

Loosely coupled systems can work very well with a managed NoSQL database solution like DynamoDB.

Example Mobile Voting Platform with SQS and DynamoDB



API Gateway

Allows you to create APIs that act as "front doors" for your applications to access data, business logic, or functionality from your back-end services.

Fully managed and handles all tasks involved in accepting and processing up to hundreds of thousands of concurrent API calls.

Can handle workloads running on:

- Amazon EC2
- AWS Lambda
- Any web application

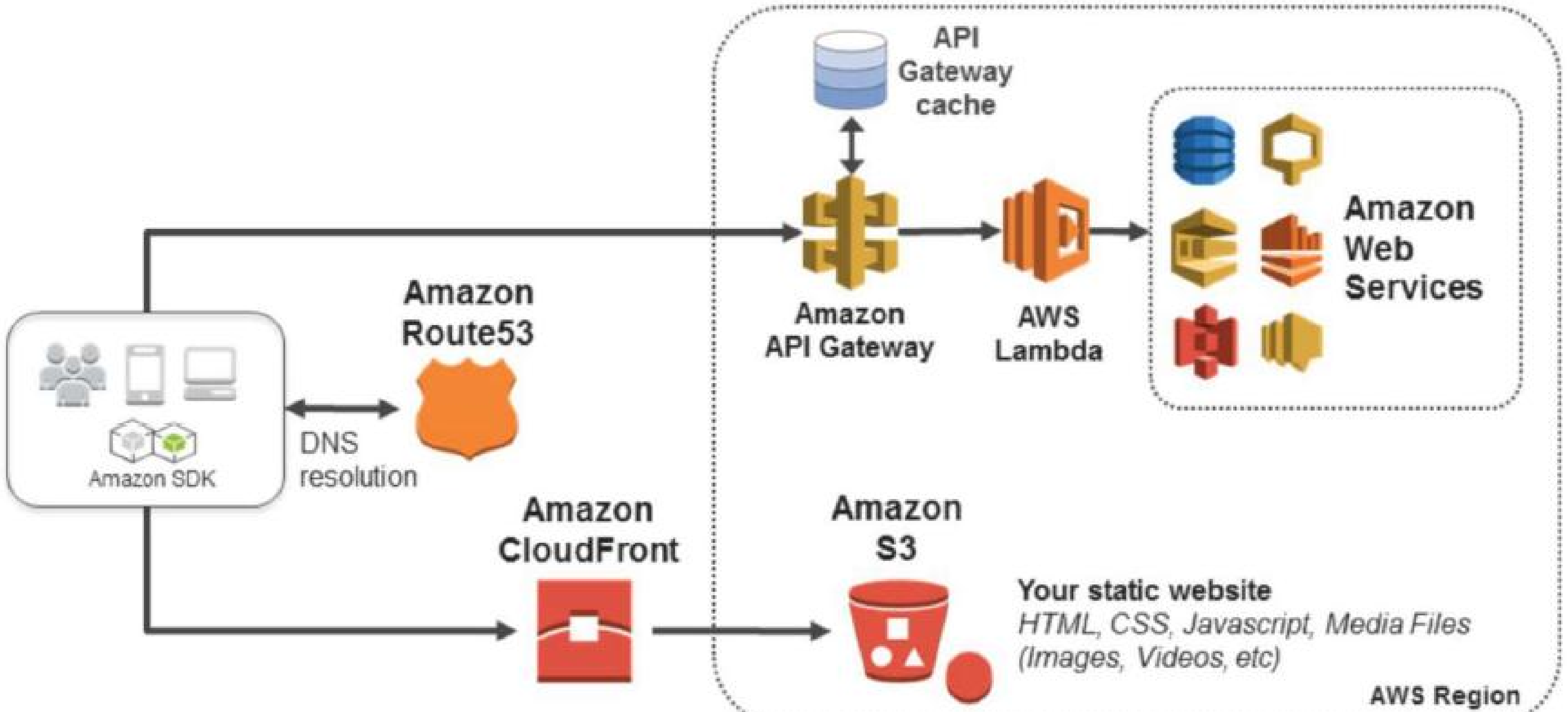
API Gateway Features

- Host and use multiple versions and stages of your APIs.
- Create and distribute API keys to developers.
- Leverage signature version 4 to authorize access to APIs.
- Throttle and monitor requests to protect your back end.
- Deeply integrated with AWS Lambda.

API Gateway Benefits

- Managed cache to store API responses
- Reduced latency and Distributed Denial of Service (DDoS) protection through Amazon CloudFront
- SDK generation for iOS, Android, and JavaScript
- OpenAPI Specification (Swagger) support
- Request/response data transformation

Serverless Architecture Using API Gateway

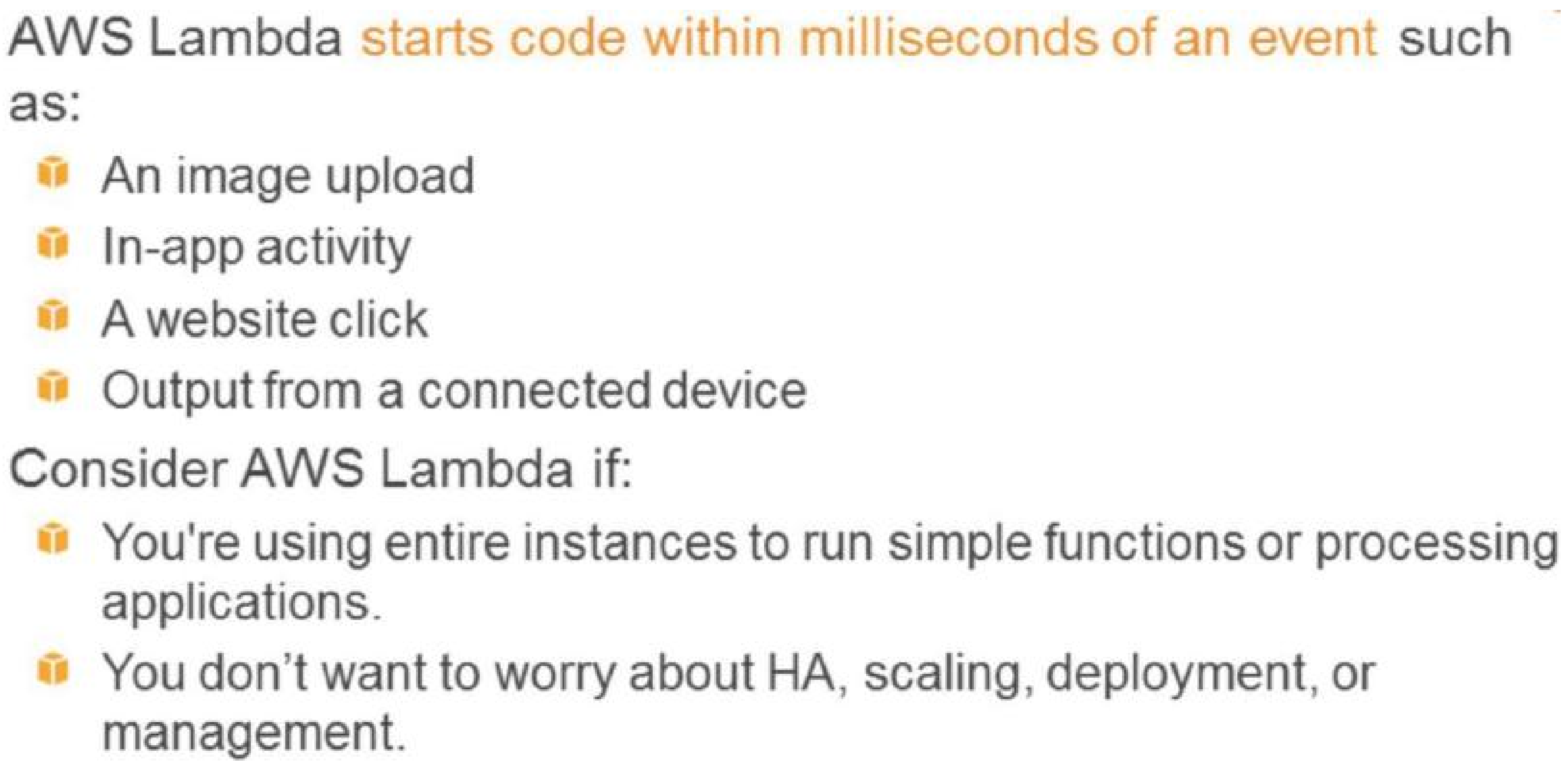


Use AWS Lambda To Decouple Infrastructure

AWS Lambda is a great solution for processing data with high availability and a limited cost footprint.

AWS Lambda allows you to further decouple your infrastructure by replacing traditional servers with simple microprocesses.

Serverless Computing With AWS Lambda



AWS Lambda starts code within milliseconds of an event such as:

- An image upload
- In-app activity
- A website click
- Output from a connected device

Consider AWS Lambda if:

- You're using entire instances to run simple functions or processing applications.
- You don't want to worry about HA, scaling, deployment, or management.

Triggers for AWS Lambda

Amazon DynamoDB
Amazon S3
Amazon SNS
Amazon Kinesis
Amazon SES
Scheduled Events
Amazon Cognito
AWS SDKs via Amazon API Gateway
Amazon CloudWatch
Amazon Echo: Alexa Skills
https://
HTTPS via API Gateway

How To Use AWS Lambda

1. Upload your code to AWS Lambda (in .zip form)
   - You can also write your code directly into an editor in the console or import it from an Amazon S3 bucket.
2. Scheduled function? Specify how often it will run.
   Event-driven function? Identify the event source.
3. Specify its necessary compute resources (128MB-1.5GB of memory).
4. Specify its timeout period.
5. Specify the VPC whose resources it needs to access (if applicable).
6. Launch the function.

That's it. From there, AWS deploys and manages it for you.

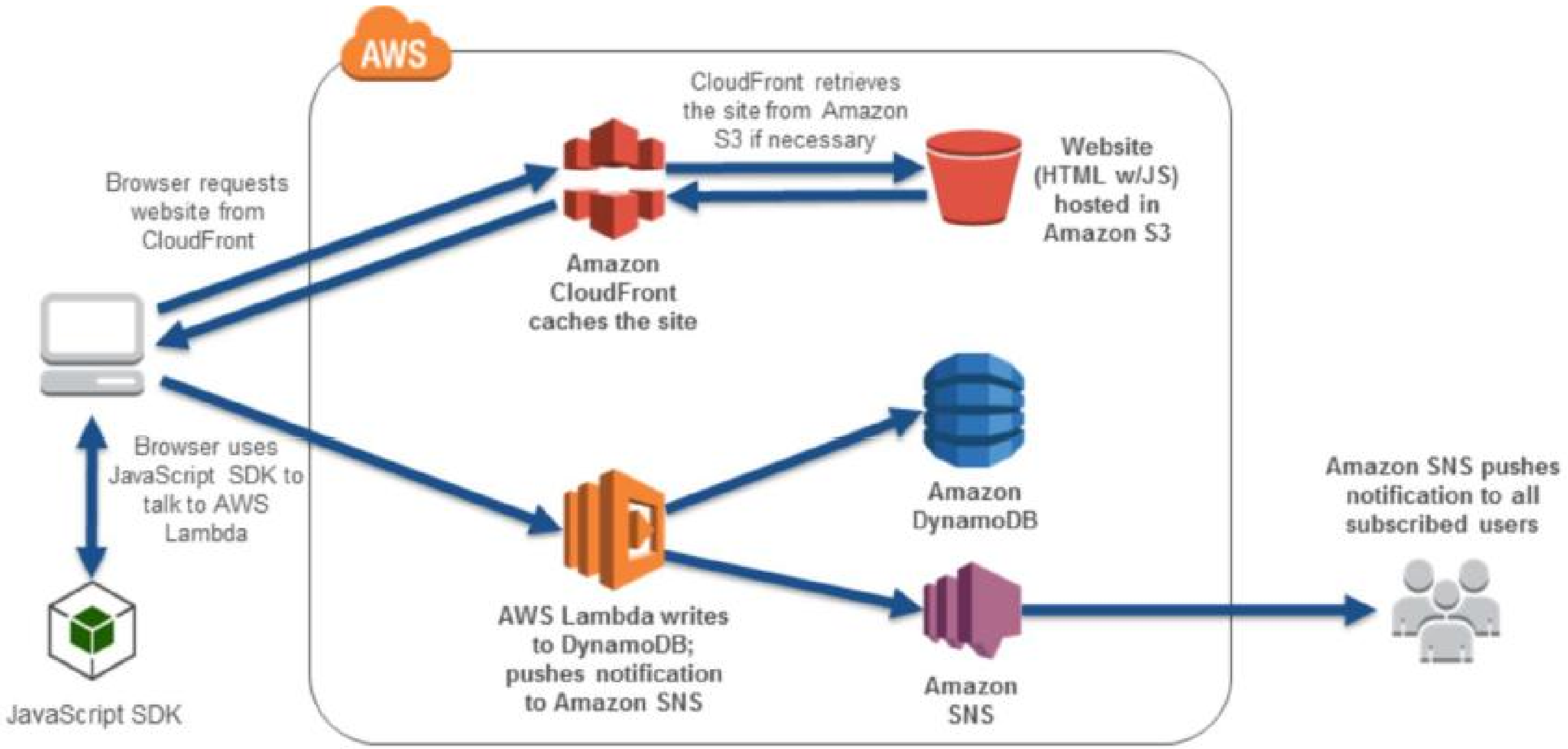
Resource Sizing With Lambda

- AWS Lambda currently offers 23 different levels of resource allocation, which range from:
  - 128 MB of memory and the lowest CPU power, to
  - 1.5 GB of memory and the highest CPU power
- More resources = lower latency for CPU-intensive workloads.
- Compute price scales with resource level.
- Functions can run between 100 ms and five minutes in length.
- Free tier: 1M free requests and 400,000 GB-s/mo of compute time.

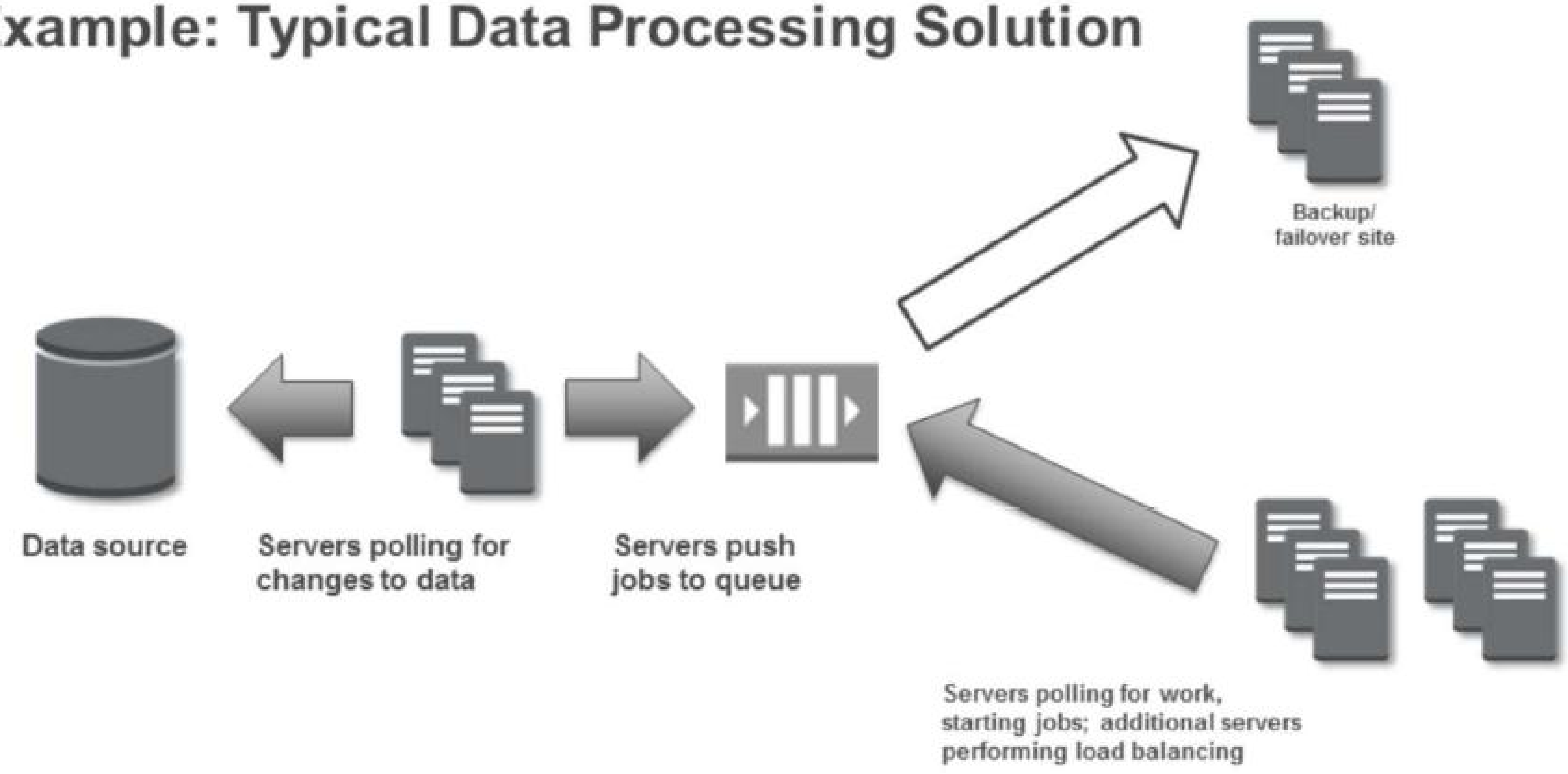
AWS Lambda Caveats

- Immature technology results in:
  - Component fragmentation
  - Unclear best-practices
- Discipline required against function sprawl
- Multitenancy means it’s technically possible that neighbor functions could hog the system resources behind the scenes
- Testing locally becomes tricky
- Further limitations:
  - Significant restrictions around local state
  - Execution duration is capped

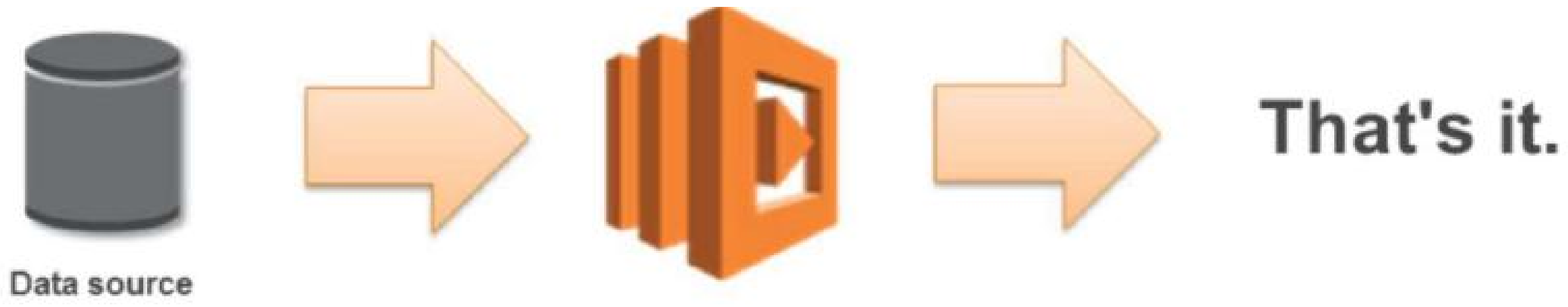
Example: Using Lambda as a Web Server


AWS
CloudFront retrieves the site from Amazon S3 if necessary
Browser requests website from CloudFront
Website (HTML w/JS) hosted in Amazon S3
Amazon CloudFront caches the site
Browser uses JavaScript SDK to talk to AWS Lambda
Amazon DynamoDB
Amazon SNS pushes notification to all subscribed users
AWS Lambda writes to DynamoDB; pushes notification to Amazon SNS
Amazon SNS
JavaScript SDK

Example: Typical Data Processing Solution



Example: Data Processing With Lambda

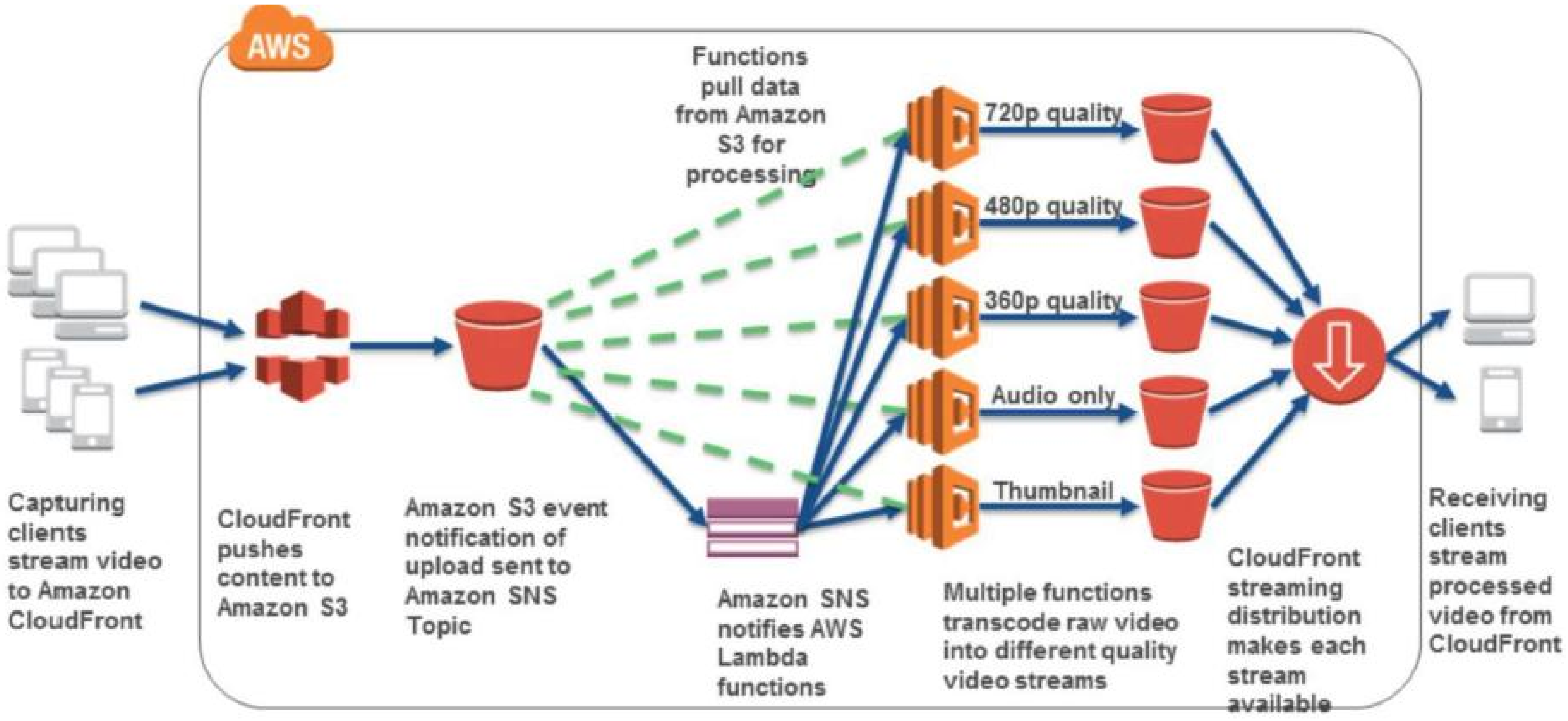


AWS Lambda handles:

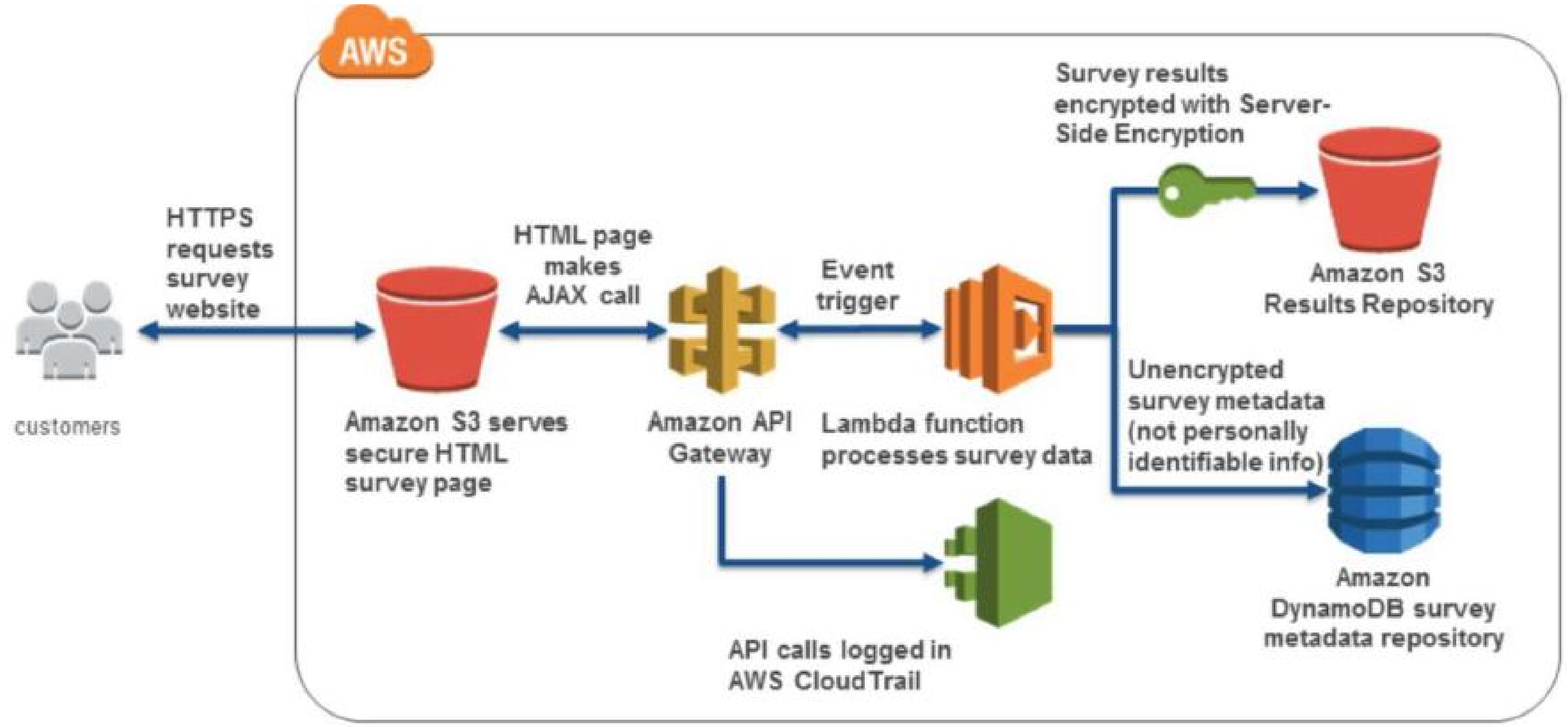
- Listening/polling
- Queueing
- Processing
- Autoscaling
- Redundancy
- Load balancing

Decoupling Examples

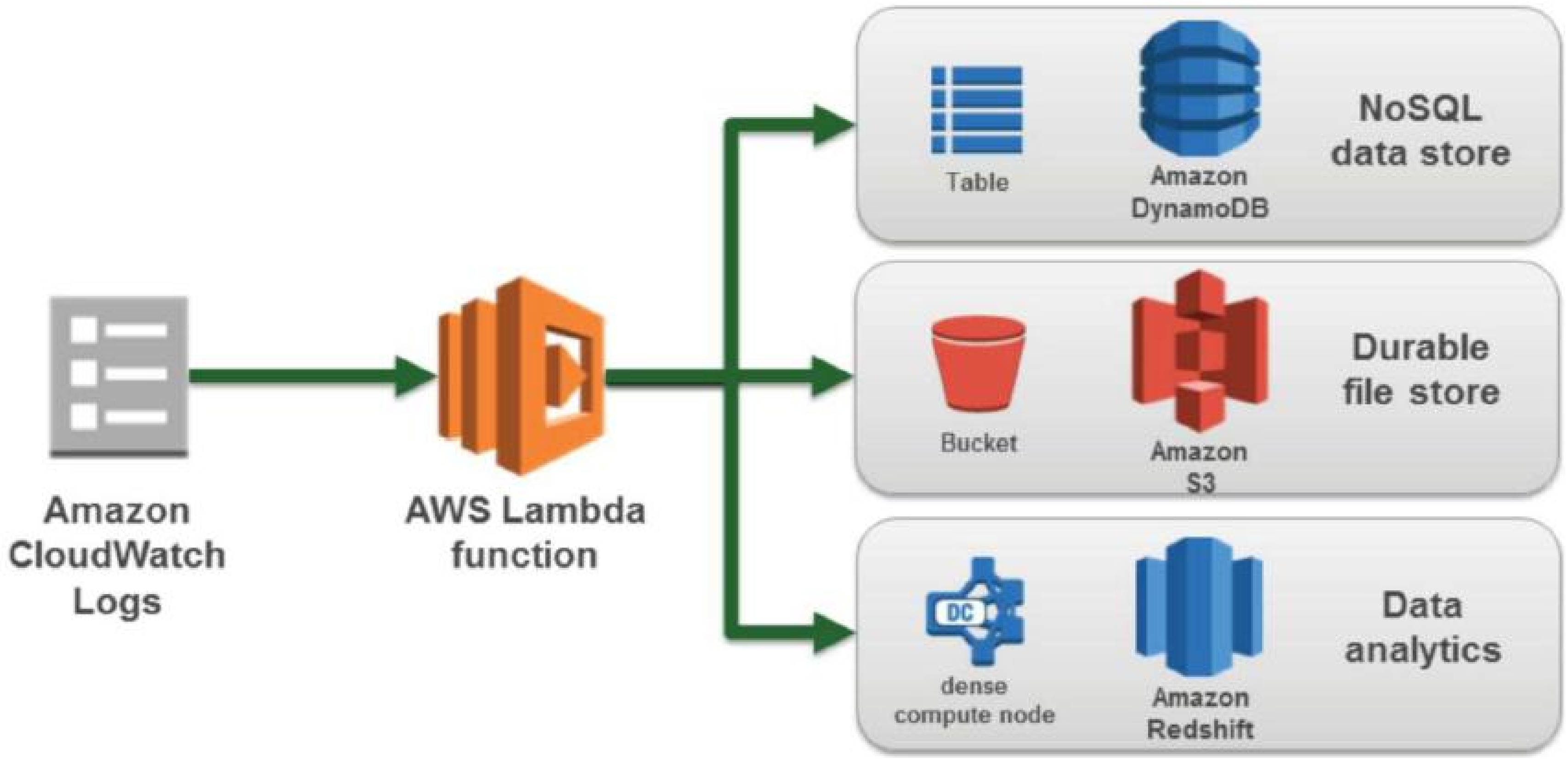
Livestreaming


AWS
Functions pull data from Amazon S3 for processing
720p quality
480p quality
360p quality
Audio only
Thumbnail
Capturing clients stream video to Amazon CloudFront
CloudFront pushes content to Amazon S3
Amazon S3 event notification of upload sent to Amazon SNS Topic
Amazon SNS notifies AWS Lambda functions
Multiple functions transcode raw video into different quality video streams
CloudFront streaming distribution makes each stream available
Receiving clients stream processed video from CloudFront

Serving Secure Surveys


AWS
HTTPS requests survey website
customers
Amazon S3 serves secure HTML survey page
HTML page makes AJAX call
Amazon API Gateway
Event trigger
Lambda function processes survey data
Survey results encrypted with Server-Side Encryption
Amazon S3 Results Repository
Unencrypted survey metadata (not personally identifiable info)
Amazon DynamoDB survey metadata repository
API calls logged in AWS CloudTrail

Processing Cloud Watch Logs



Amazon CloudWatch Logs
AWS Lambda function
Table
Amazon DynamoDB
NoSQL data store
Bucket
Amazon S3
Durable file store
DC
dense compute node
Amazon Redshift
Data analytics