API Gateway : Build, Deploy and Manage APIs

Example: Building a Serverless API

AWS API Gateway

- AWS Lambda + API Gateway: No infrastructure to manage
- Support for the WebSocket Protocol
- Handle API versioning (v1, v2…)
- Handle different environments (dev, test, prod…)
- Handle security (Authentication and Authorization)
- Create API keys, handle request throttling
- Swagger / Open API import to quickly define APIs
- Transform and validate requests and responses
- Generate SDK and API specifications
- Cache API responses

API Gateway – Integrations High Level

- Lambda Function
  - Invoke Lambda function
  - Easy way to expose REST API backed by AWS Lambda
- HTTP
  - Expose HTTP endpoints in the backend
  - Example: internal HTTP API on premise, Application Load Balancer…
  - Why? Add rate limiting, caching, user authentications, API keys, etc…
- AWS Service
  - Expose any AWS API through the API Gateway?
  - Example: start an AWS Step Function workflow, post a message to SQS
  - Why? Add authentication, deploy publicly, rate control…

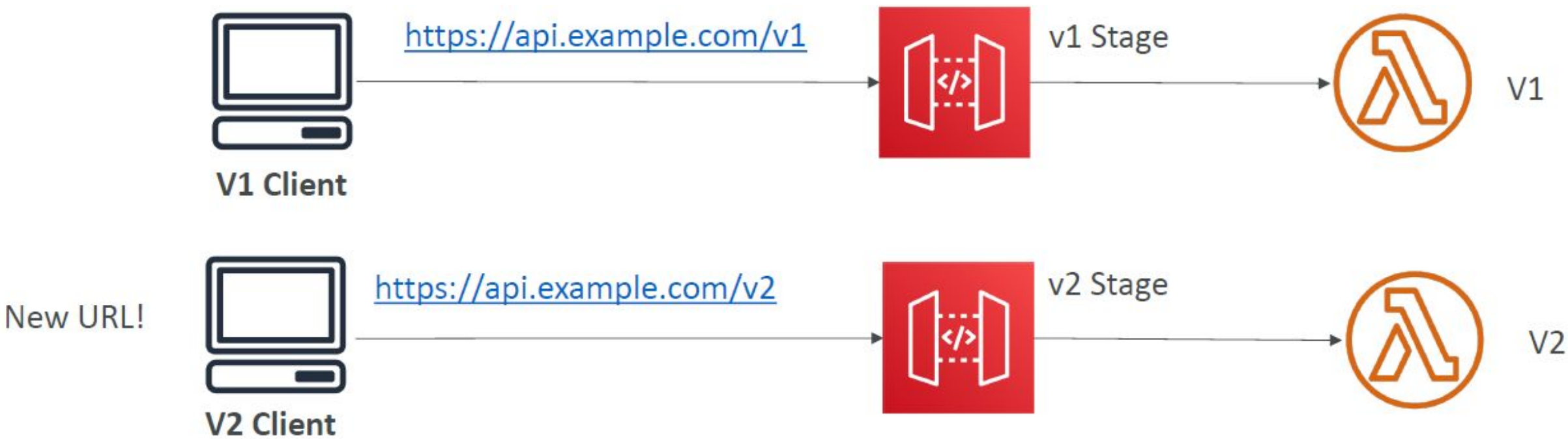
API Gateway - Endpoint Types

- **Edge-Optimized (default):** For global clients
  - Requests are routed through the CloudFront Edge locations (improves latency)
  - The API Gateway still lives in only one region
- **Regional:**
  - For clients within the same region
  - Could manually combine with CloudFront (more control over the caching strategies and the distribution)
- **Private:**
  - Can only be accessed from your VPC using an interface VPC endpoint (ENI)
  - Use a resource policy to define access

API Gateway – Deployment Stages

- Making changes in the API Gateway does not mean they're effective
- You need to make a "deployment" for them to be in effect
- It's a common source of confusion
- Changes are deployed to "Stages" (as many as you want)
- Use the naming you like for stages (dev, test, prod)
- Each stage has its own configuration parameters
- Stages can be rolled back as a history of deployments is kept

API Gateway – Stages v1 and v2
API breaking change

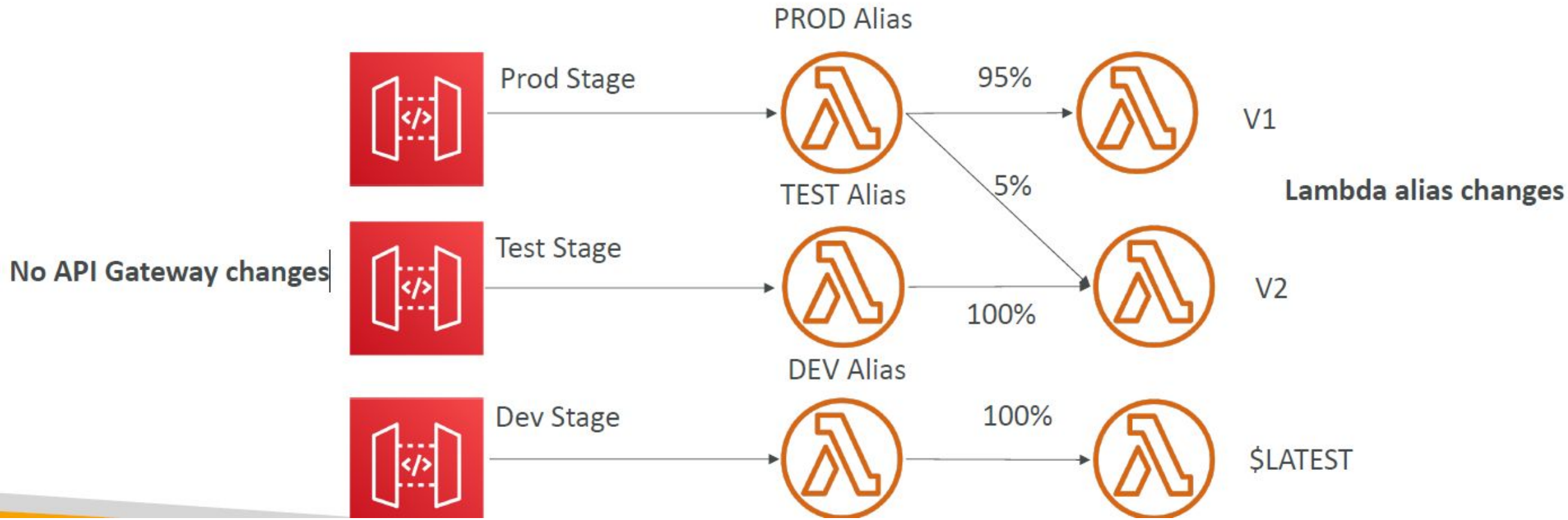


API Gateway – Stage Variables

- Stage variables are like environment variables for API Gateway
- Use them to change often changing configuration values
- They can be used in:
  - Lambda function ARN
  - HTTP Endpoint
  - Parameter mapping templates
- Use cases:
  - Configure HTTP endpoints your stages talk to (dev, test, prod…)
  - Pass configuration parameters to AWS Lambda through mapping templates
- Stage variables are passed to the "context" object in AWS Lambda

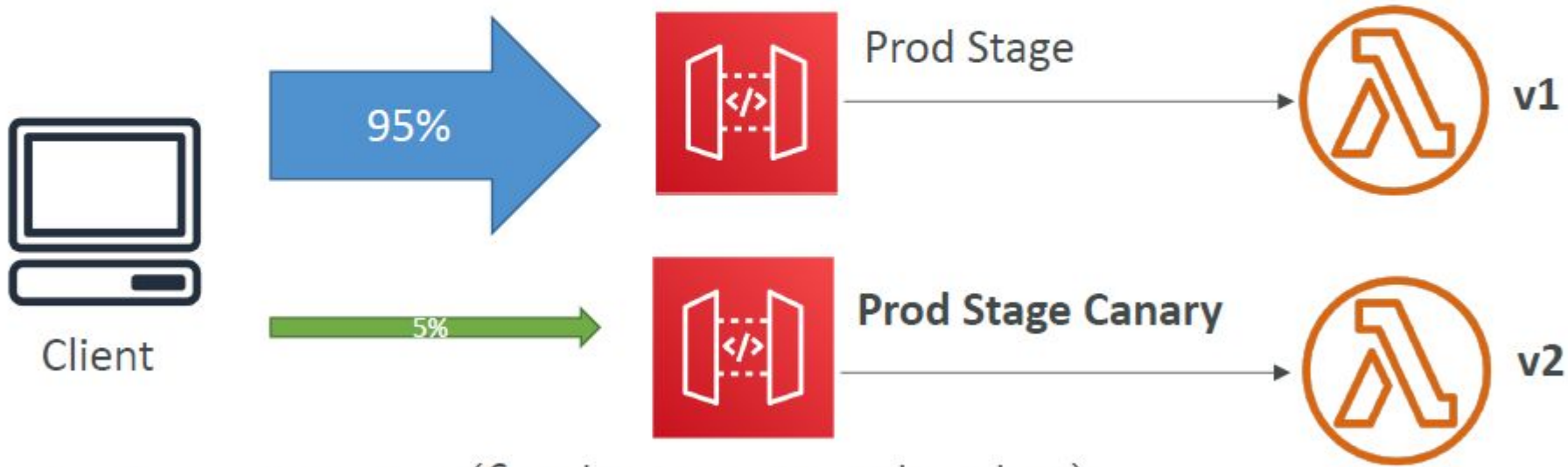
API Gateway Stage Variables & Lambda Aliases

- We create a **stage variable** to indicate the corresponding Lambda alias
- Our API gateway will automatically invoke the right Lambda function!



API Gateway – Canary Deployment

- Possibility to enable canary deployments for any stage (usually prod)
- Choose the % of traffic the canary channel receives

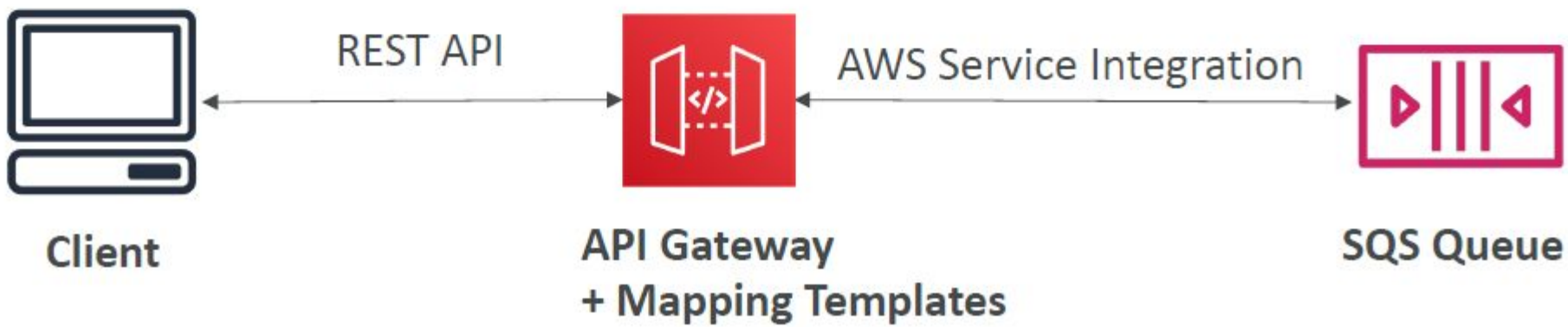


- Metrics & Logs are separate (for better monitoring)
- Possibility to override stage variables for canary
- This is blue / green deployment with AWS Lambda & API Gateway

API Gateway - Integration Types

- Integration Type **MOCK**
  - API Gateway returns a response without sending the request to the backend

- Integration Type **HTTP / AWS (Lambda & AWS Services)**
  - you must configure both the integration request and integration response
  - Setup data mapping using **mapping templates** for the request & response



API Gateway - Integration Types

- Integration Type **AWS_PROXY (Lambda Proxy)**:
  - incoming request from the client is the input to Lambda
  - The function is responsible for the logic of request / response
  - No mapping template, headers, query string parameters… are passed as arguments

```
{
    "resource": "Resource path",
    "path": "Path parameter",
    "httpMethod": "Incoming request's method name",
    "headers": "String containing incoming request headers",
    "multiValueHeaders": "List of strings containing incomin
    "queryStringParameters": "query string parameters ",
    "multiValueQueryStringParameters": "List of query string
    "pathParameters":  "path parameters",
    "stageVariables": "Applicable stage variables",
    "requestContext": "Request context, including authorizer
    "body": "A JSON string of the request payload.",
    "isBase64Encoded": "A boolean flag"
}
```

Lambda function invocation payload

```
{
    "isBase64Encoded": "true|false",
    "statusCode": "httpStatusCode",
    "headers": { "headerName": "headerValue", ... },
    "multiValueHeaders": { "headerName": ["headerValue", "header
    "body": "..."
}
```

Lambda function expected response

API Gateway - Integration Types

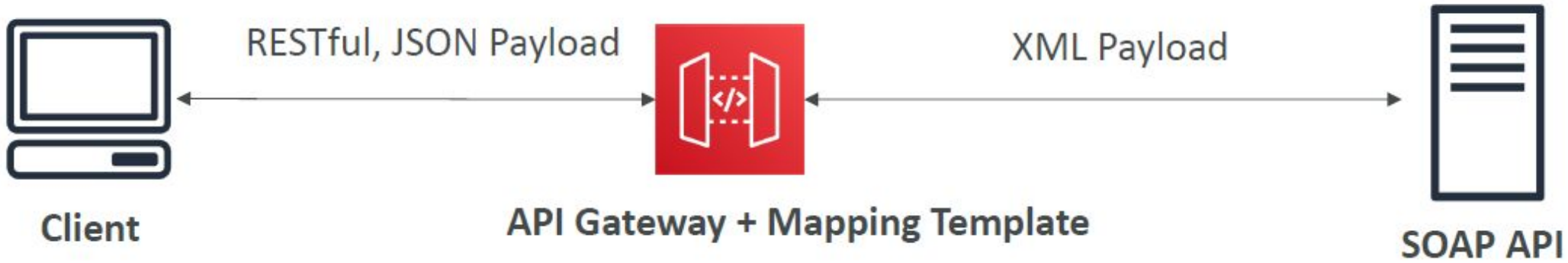
- Integration Type **HTTP_PROXY**
  - No mapping template
  - The HTTP request is passed to the backend
  - The HTTP response from the backend is forwarded by API Gateway

Mapping Templates (AWS & HTTP Integration)

- Mapping templates can be used to modify request / responses
- Rename / Modify **query string parameters**
- Modify **body content**
- **Add headers**
- Uses Velocity Template Language (VTL): for loop, if etc…
- Filter output results (remove unnecessary data)

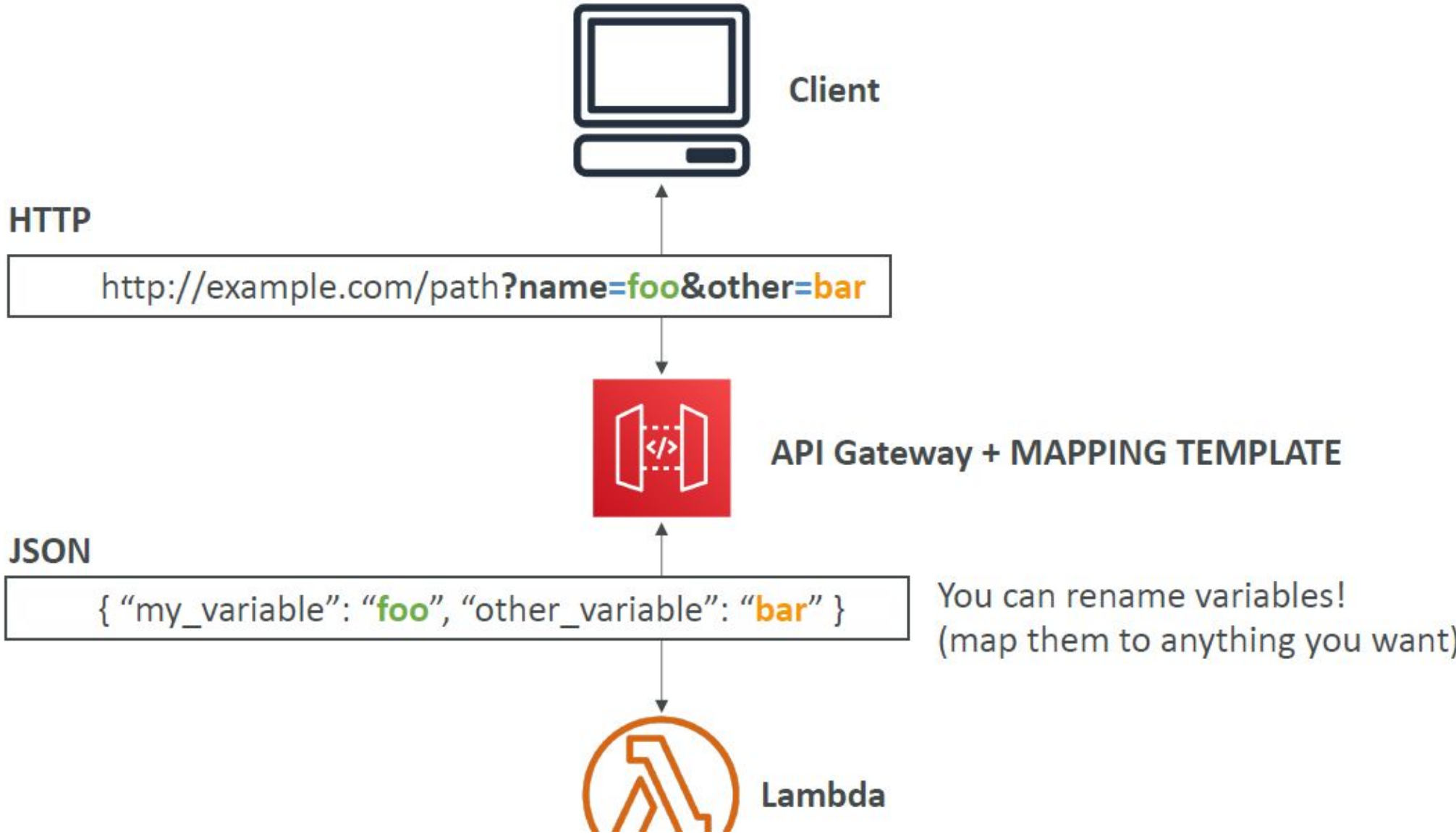
Mapping Example: JSON to XML with SOAP

- SOAP API are XML based, whereas REST API are JSON based



- In this case, API Gateway should:
  - Extract data from the request: either path, payload or header
  - Build SOAP message based on request data (mapping template)
  - Call SOAP service and receive XML response
  - Transform XML response to desired format (like JSON), and respond to the user

Mapping Example: Query String parameters

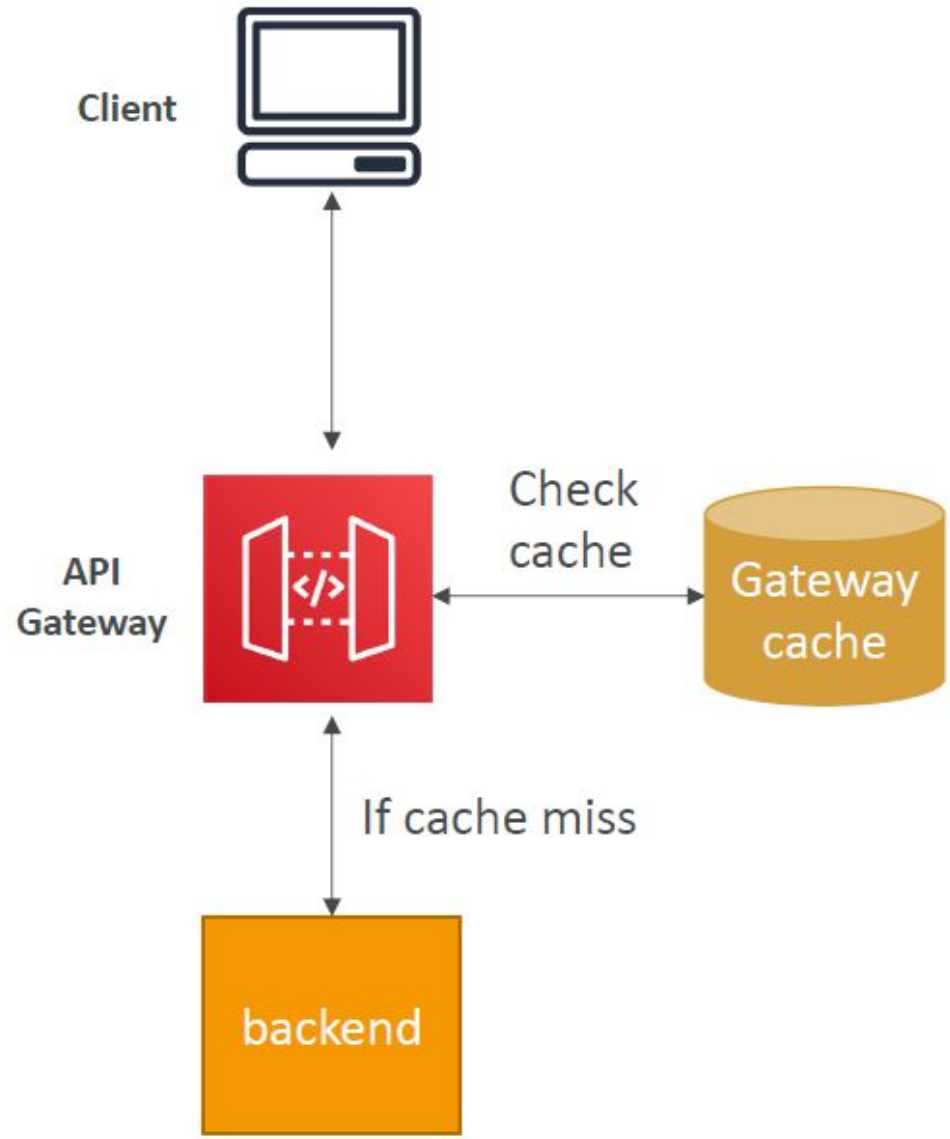


AWS API Gateway Swagger / Open API spec

- Common way of defining REST APIs, using API definition as code
- Import existing Swagger / OpenAPI 3.0 spec to API Gateway
  - Method
  - Method Request
  - Integration Request
  - Method Response
  - + AWS extensions for API gateway and setup every single option
- Can export current API as Swagger / OpenAPI spec
- Swagger can be written in YAML or JSON
- Using Swagger we can generate SDK for our applications

Caching API responses

- Caching reduces the number of calls made to the backend
- Default TTL (time to live) is 300 seconds (min: 0s, max: 3600s)
- **Caches are defined per stage**
- Possible to override cache settings **per method**
- Cache encryption option
- Cache capacity between 0.5GB to 237GB
- Cache is expensive, makes sense in production, may not make sense in dev / test



API Gateway Cache Invalidation

- Able to flush the entire cache (invalidate it) immediately
- Clients can invalidate the cache with **header: Cache-Control: max-age=0** (with proper IAM authorization)
- If you don't impose an InvalidateCache policy (or choose the Require authorization check box in the console), any client can invalidate the API cache

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "execute-api:InvalidateCache"
            ],
            "Resource": [
                "arn:...:api-id/stage-name/GET/resource-path-specifier"
            ]
        }
    ]
}
```

API Gateway – Usage Plans & API Keys

- If you want to make an API available as an offering ($) to your customers
- Usage Plan:
  - who can access one or more deployed API stages and methods
  - how much and how fast they can access them
  - uses API keys to identify API clients and meter access
  - configure throttling limits and quota limits that are enforced on individual client
- API Keys:
  - alphanumeric string values to distribute to your customers
  - Ex: WBjHxNtoAb4WPKBC7cGm64CBiblb24b4jt8jJHo9
  - Can use with usage plans to control access
  - Throttling limits are applied to the API keys
  - Quotas limits is the overall number of maximum requests

API Gateway – Correct Order for API keys

- To configure a usage plan

1. Create one or more APIs, configure the methods to require an API key, and deploy the APIs to stages.
2. Generate or import API keys to distribute to application developers (your customers) who will be using your API.
3. Create the usage plan with the desired throttle and quota limits.
4. Associate API stages and API keys with the usage plan.

- Callers of the API must supply an assigned API key in the x-api-key header in requests to the API.

API Gateway – Logging & Tracing

- CloudWatch Logs:
  - Enable CloudWatch logging at the Stage level (with Log Level)
  - Can override settings on a per API basis (ex: ERROR, DEBUG, INFO)
  - Log contains information about request / response body

- X-Ray:
  - Enable tracing to get extra information about requests in API Gateway
  - X-Ray API Gateway + AWS Lambda gives you the full picture

API Gateway – CloudWatch Metrics

- Metrics are by stage, Possibility to enable detailed metrics
- **CacheHitCount** & **CacheMissCount**: efficiency of the cache
- **Count**: The total number API requests in a given period.
- **IntegrationLatency**: The time between when API Gateway relays a request to the backend and when it receives a response from the backend.
- **Latency**: The time between when API Gateway receives a request from a client and when it returns a response to the client. The latency includes the integration latency and other API Gateway overhead.
- **4XXError** (client-side) & **5XXError** (server-side)

API Gateway - Errors

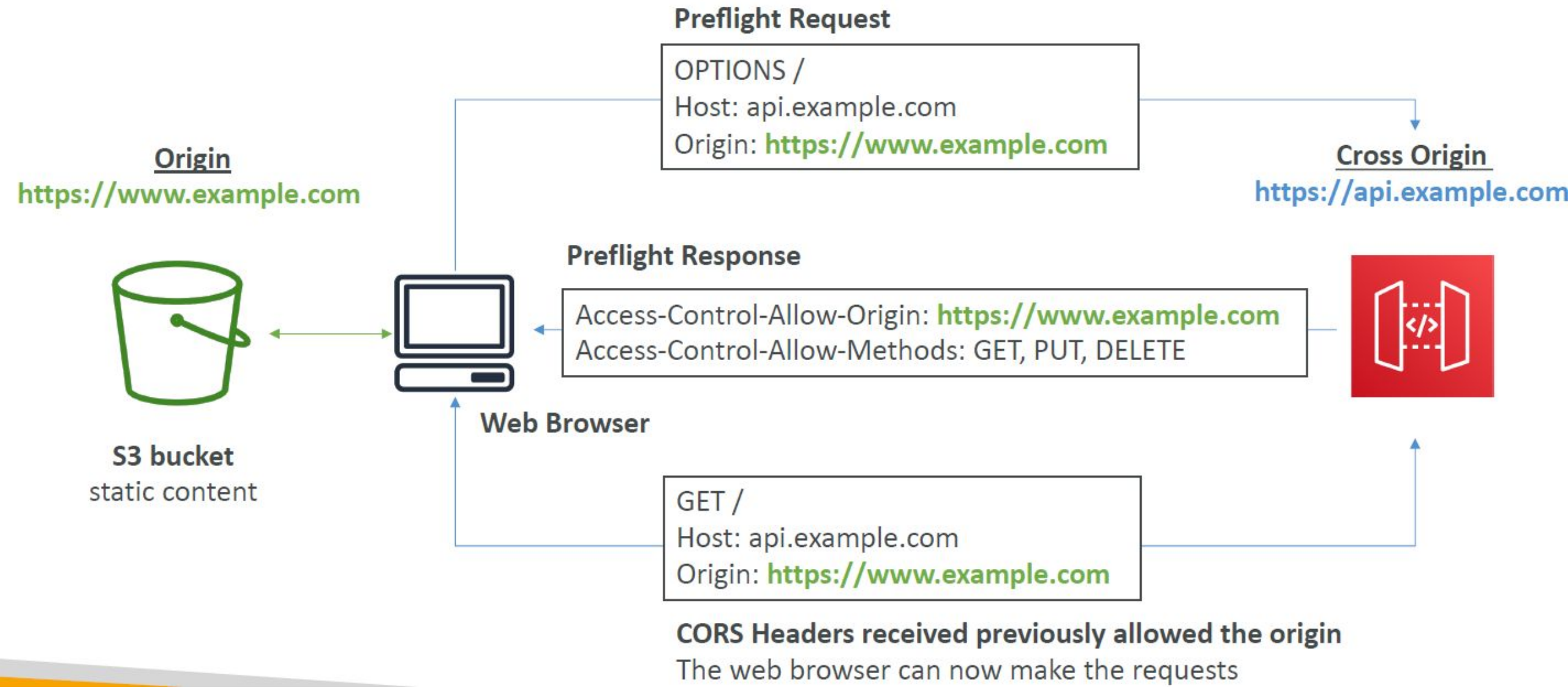
- 4xx means Client errors
  - 400: Bad Request
  - 403: Access Denied, WAF filtered
  - 429: Quota exceeded, Throttle

- 5xx means Server errors
  - 502: Bad Gateway Exception, usually for an incompatible output returned from a Lambda proxy integration backend and occasionally for out-of-order invocations due to heavy loads.
  - 503: Service Unavailable Exception
  - 504: Integration Failure – ex Endpoint Request Timed-out Exception
    API Gateway requests time out after 29 second maximum

AWS API Gateway - CORS

- CORS must be enabled when you receive API calls from another domain.
- The OPTIONS pre-flight request must contain the following headers:
  - Access-Control-Allow-Methods
  - Access-Control-Allow-Headers
  - Access-Control-Allow-Origin
- CORS can be enabled through the console

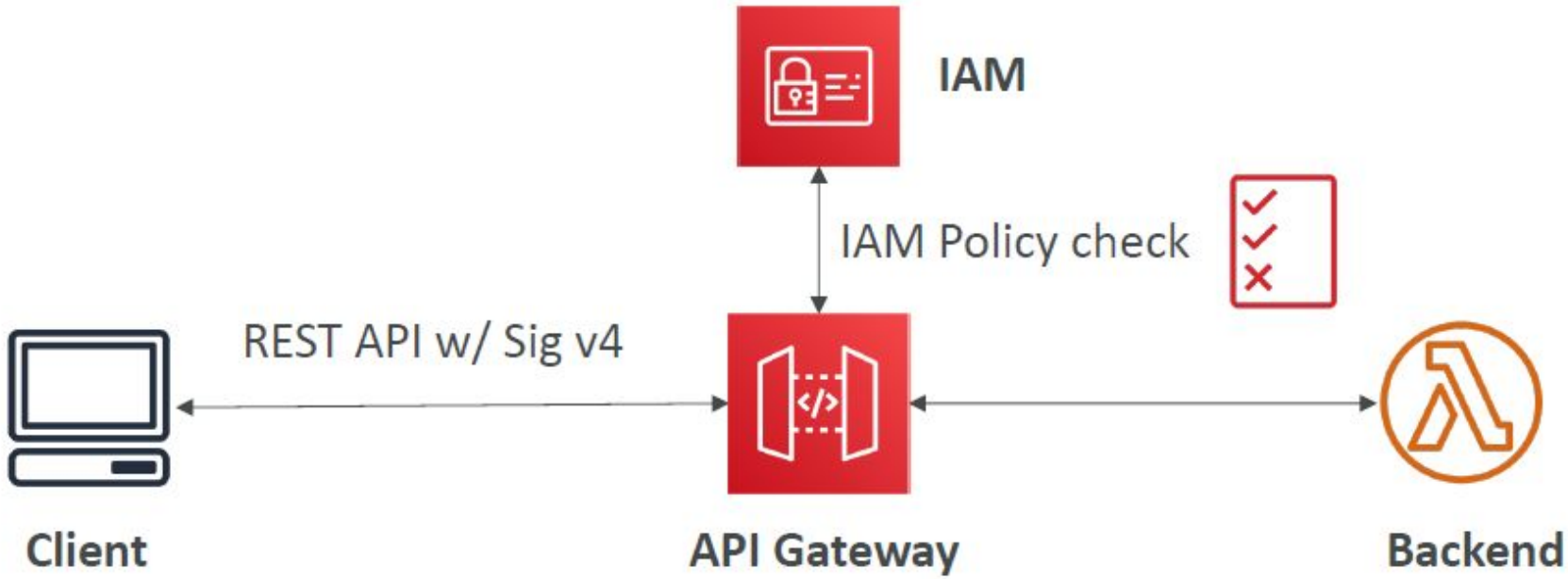
CORS – Enabled on the API Gateway



API Gateway – Security

IAM Permissions

- Create an IAM policy authorization and attach to User / Role
- Authentication = IAM | Authorization = IAM Policy
- Good to provide access within AWS (EC2, Lambda, IAM users…)
- Leverages "Sig v4" capability where IAM credential are in headers



API Gateway – Resource Policies

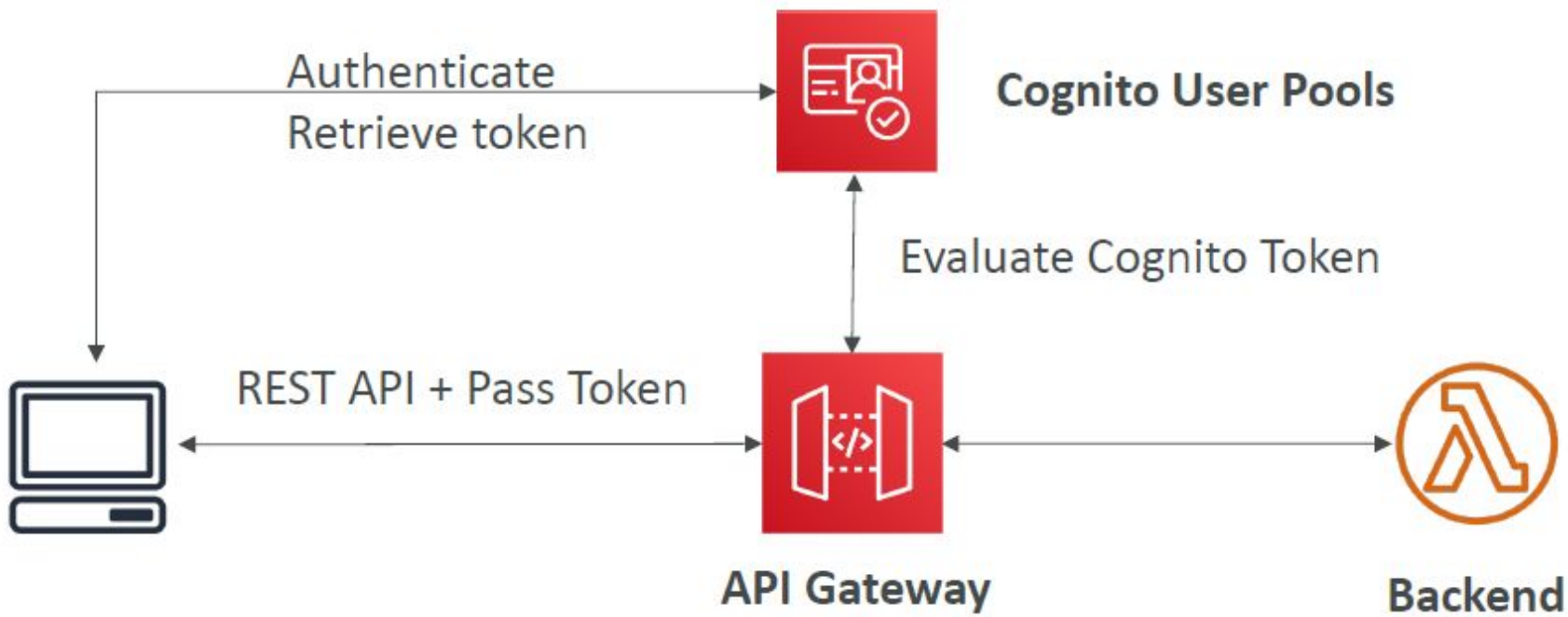
- **Resource policies** (similar to Lambda Resource Policy)
- **Allow for Cross Account Access (combined with IAM Security)**
- Allow for a specific source IP address
- Allow for a VPC Endpoint

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "AWS": [
                    "arn:aws:iam::account-id-2:user/Alice",
                    "account-id-2"
                ]
            },
            "Action": "execute-api:Invoke",
            "Resource": [
                "arn:aws:execute-api:region:account-id-1:api-id/stage/GET/pets"
            ]
        }
    ]
}
```

API Gateway – Security

Cognito User Pools

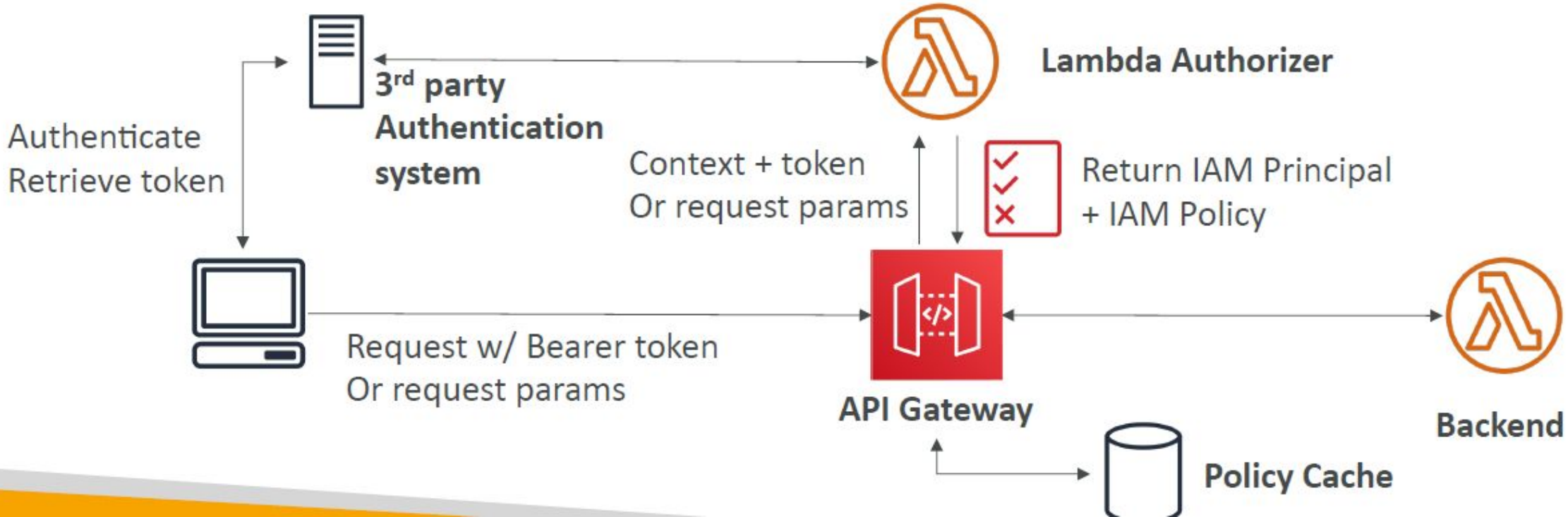
- Cognito fully manages user lifecycle, token expires automatically
- API gateway verifies identity automatically from AWS Cognito
- No custom implementation required
- Authentication = Cognito User Pools | Authorization = API Gateway Methods



API Gateway – Security

Lambda Authorizer (formerly Custom Authorizers)

- **Token-based authorizer** (bearer token) – ex JWT (JSON Web Token) or Oauth
- A **request parameter-based** Lambda authorizer (**headers, query string**, stage var)
- Lambda must return an IAM policy for the user, result policy is cached
- Authentication = External | Authorization = Lambda function



API Gateway – HTTP API vs REST API

- HTTP APIs
  - low-latency, cost-effective AWS Lambda proxy, HTTP proxy APIs and private integration (no data mapping)
  - support OIDC and OAuth 2.0 authorization, and built-in support for CORS
  - No usage plans and API keys
- REST APIs
  - All features (except Native OpenID Connect / OAuth 2.0)

Authorizers	HTTP API	REST API
AWS Lambda		✓
IAM		✓
Amazon Cognito	✓ *	✓
Native OpenID Connect / OAuth 2.0	✓	

API Gateway – WebSocket API – Overview

- What's WebSocket?
  - Two-way interactive communication between a user's browser and a server
  - Server can push information to the client
  - This enables **stateful** application use cases
- WebSocket APIs are often used in **real-time applications** such as chat applications, collaboration platforms, multiplayer games, and financial trading platforms.

- Works with AWS Services (Lambda, DynamoDB) or HTTP endpoints

AWS Serverless Application Model (SAM)

AWS SAM

- SAM = Serverless Application Model
- Framework for developing and deploying serverless applications
- All the configuration is YAML code
- Generate complex CloudFormation from simple SAM YAML file
- Supports anything from CloudFormation: Outputs, Mappings, Parameters, Resources…
- Only two commands to deploy to AWS
- SAM can use CodeDeploy to deploy Lambda functions
- SAM can help you to run Lambda, API Gateway, DynamoDB locally

AWS SAM – Recipe

- Transform Header indicates it's SAM template:
  - `Transform: 'AWS::Serverless-2016-10-31'`
- Write Code
  - `AWS::Serverless::Function`
  - `AWS::Serverless::Api`
  - `AWS::Serverless::SimpleTable`
- Package & Deploy:
  - `aws cloudformation package / sam package`
  - `aws cloudformation deploy / sam deploy`

Deep dive into SAM deployment

aws cloudformation package

aws cloudformation deploy

SAM Template
YAML file

Transform

reference

Generated Template
CloudFormation YAML

Create and execute change set

change set

AWS
CloudFormation

Zip and upload

Code S3
bucket

Application Code
+ Swagger File (optional)

SAM Policy Templates

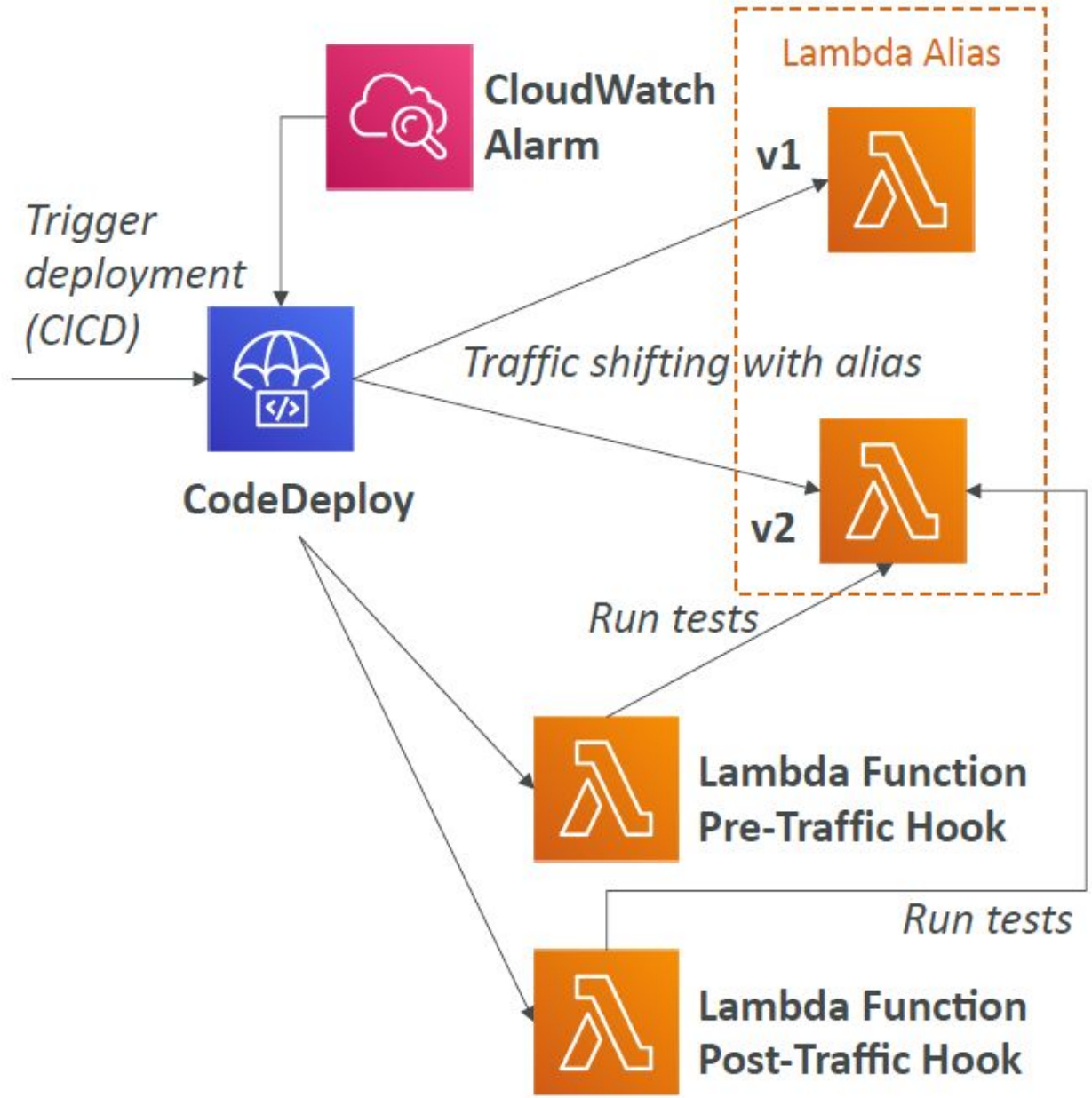
- List of templates to apply permissions to your Lambda Functions
- Full list available here: https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/serverless-policy-templates.html#serverless-policy-template-table
- Important examples:
  - **S3ReadPolicy**: Gives read only permissions to objects in S3
  - **SQSPollerPolicy**: Allows to poll an SQS queue
  - **DynamoDBCrudPolicy**: CRUD = create read update delete

```
MyFunction:
  Type: 'AWS::Serverless::Function'
  Properties:
    CodeUri: ${codeuri}
    Handler: hello.handler
    Runtime: python2.7
    Policies:
      - SQSPollerPolicy:
          QueueName:
            !GetAtt MyQueue.QueueName
```

SAM and CodeDeploy

- SAM framework natively uses CodeDeploy to update Lambda functions

- Traffic Shifting feature
- Pre and Post traffic hooks features to validate deployment (before the traffic shift starts and after it ends)
- Easy & automated rollback using CloudWatch Alarms



SAM – Exam Summary

- SAM is built on CloudFormation
- SAM requires the **Transform** and **Resources** sections
- Commands to know:
  - sam build: fetch dependencies and create local deployment artifacts
  - sam package: package and upload to Amazon S3, generate CF template
  - sam deploy: deploy to CloudFormation
- SAM Policy templates for easy IAM policy definition
- SAM is integrated with CodeDeploy to do deploy to Lambda aliases

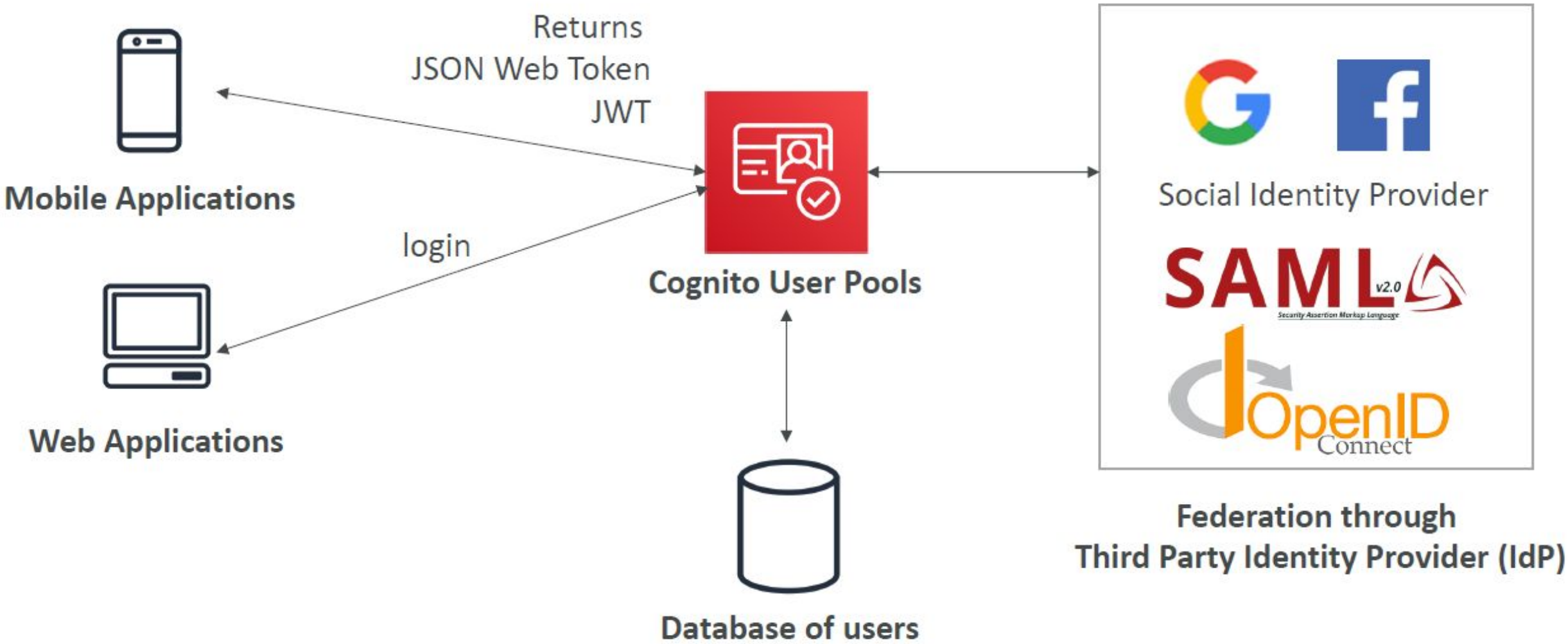
Amazon Cognito

- We want to give our users an identity so that they can interact with our application.
- Cognito User Pools:
  - Sign in functionality for app users
  - Integrate with API Gateway & Application Load Balancer
- Cognito Identity Pools (Federated Identity):
  - Provide AWS credentials to users so they can access AWS resources directly
  - Integrate with Cognito User Pools as an identity provider
- Cognito Sync:
  - Synchronize data from device to Cognito.
  - Is deprecated and replaced by AppSync
- Cognito vs IAM: "hundreds of users", "mobile users", "authenticate with SAML"

Cognito User Pools (CUP) – User Features

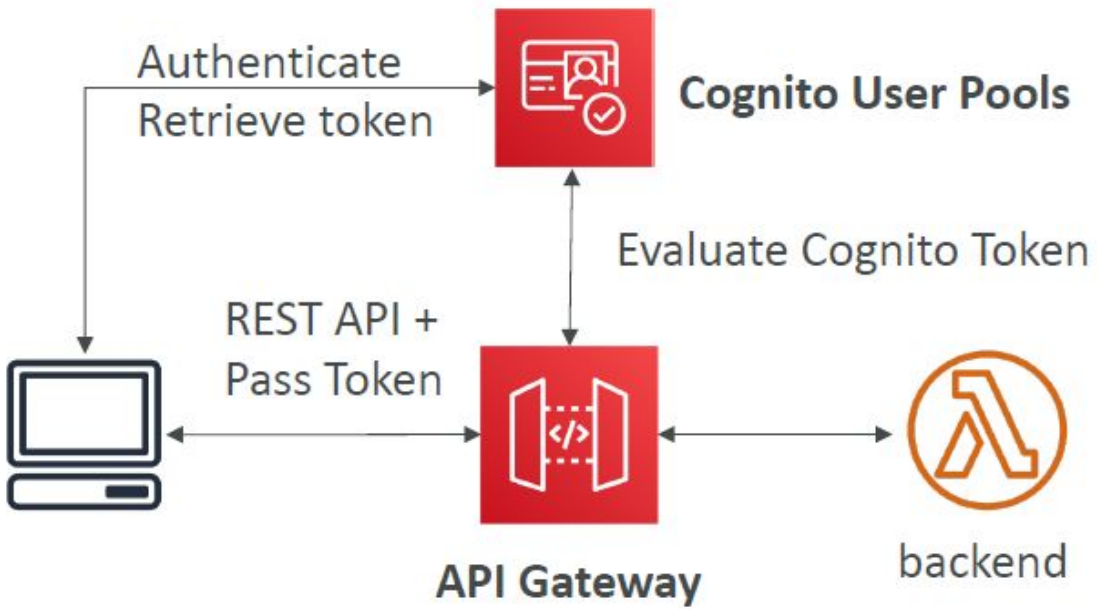
- **Create a serverless database of user for your web & mobile apps**
- Simple login: Username (or email) / password combination
- Password reset
- Email & Phone Number Verification
- Multi-factor authentication (MFA)
- Federated Identities: users from Facebook, Google, SAML…
- Feature: block users if their credentials are compromised elsewhere
- Login sends back a JSON Web Token (JWT)

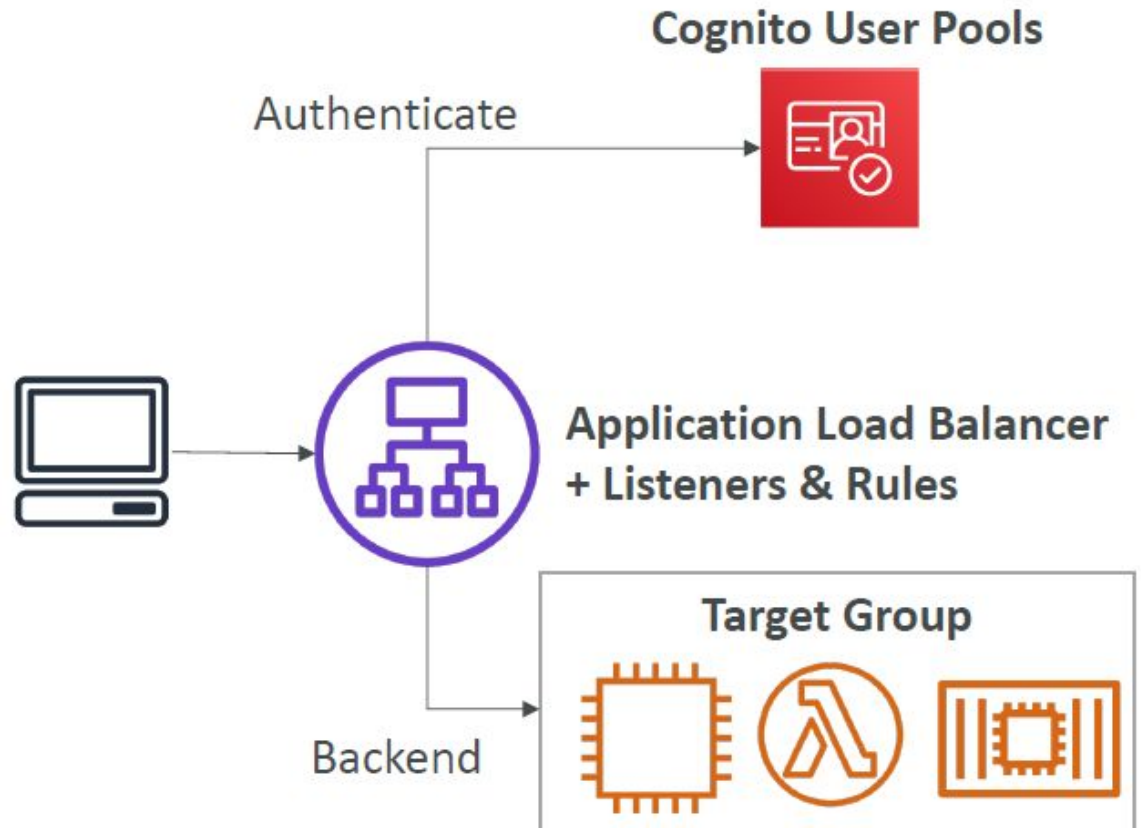
Cognito User Pools (CUP) – Diagram



Cognito User Pools (CUP) - Integrations

- CUP integrates with **API Gateway** and **Application Load Balancer**





Cognito User Pools – Lambda Triggers

- CUP can invoke a Lambda function synchronously on these triggers:

User Pool Flow	Operation	Description
Authentication Events	Pre Authentication Lambda Trigger	Custom validation to accept or deny the sign-in request
	Post Authentication Lambda Trigger	Event logging for custom analytics
	Pre Token Generation Lambda Trigger	Augment or suppress token claims
Sign-Up	Pre Sign-up Lambda Trigger	Custom validation to accept or deny the sign-up request
	Post Confirmation Lambda Trigger	Custom welcome messages or event logging for custom analytics
	Migrate User Lambda Trigger	Migrate a user from an existing user directory to user pools
Messages	Custom Message Lambda Trigger	Advanced customization and localization of messages
Token Creation	Pre Token Generation Lambda Trigger	Add or remove attributes in Id tokens

Cognito User Pools – Hosted Authentication UI

- Cognito has a **hosted authentication UI** that you can add to your app to handle sign-up and sign-in workflows
- Using the hosted UI, you have a foundation for integration with social logins, OIDC or SAML
- Can customize with a **custom logo** and **custom CSS**