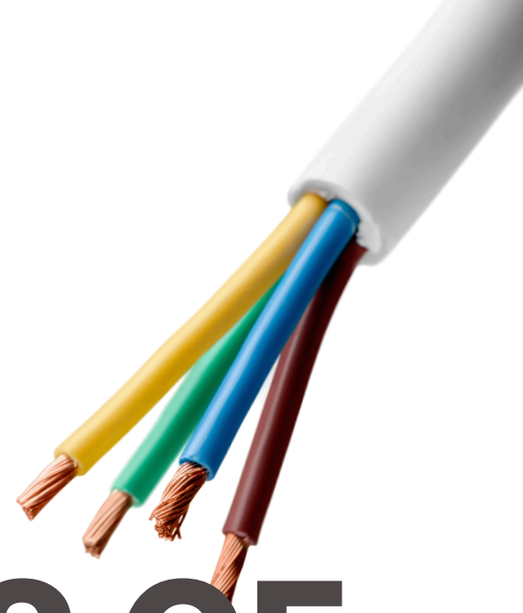
AWS Cloud Engineer | @rajankfl



ADVANCED CONCEPTS OF LAMBDA YOU NEED TO KNOW

SERVERLESS MYTH!

Lambda, Synchronous Invocations

Synchronous invocation occurs when the **caller waits for the function to process an event** and return a response.

This method is typically **used in** applications where an **immediate response is required.**

How Synchronous Invocation Works:

- **Event Source:** The event source (e.g., an API Gateway, AWS SDK) sends an event to the Lambda function.
- **Processing:** Lambda processes the event and executes the function code.
- **Response:** Lambda returns the response to the event source once processing is complete.

Key Points:

- **Timeout:** Up to 15 minutes.
- **Error Handling:** Returns error response to the caller.
- **Scaling:** Automatic based on request volume.

Lambda, Synchronous Invocations - Services

User Invoked:

- Elastic Load Balancing (ALB)
- Amazon API Gateway
- Amazon CloudFront (Lambda@Edge)
- Amazon S3 Batch

Service Invoked:

- Amazon Cognito
- AWS Step Functions

Other Services:

- Amazon Lex
- Amazon Alexa
- Amazon Kinesis Data Firehose

Lambda, Asynchronous Invocations

Asynchronous invocation allows the caller to invoke the Lambda function and **immediately continue executing, without waiting for a response.**

How Asynchronous Invocation Works:

- **Event Source:** The event source (e.g., S3, SNS) sends an event to the Lambda function.
- **Processing:** Lambda queues the event, and then processes it in the background.
- **Response:** No immediate response is returned to the caller; instead, the function returns a **`202 Accepted`** status.

Benefits:

- **Non-Blocking:** The caller can continue execution immediately.
- **Efficient:** Ideal for tasks that do not require immediate response.
- **Reliability:** DLQ and retries ensure reliable processing of events.

Lambda, Asynchronous Invocations – Services

- Amazon Simple Storage Service (S3)
- Amazon Simple Notification Service (SNS)
- Amazon EventBridge
- AWS CodeCommit
  (CodeCommitTrigger: new branch, new tag, new push)
- AWS CodePipeline
  (invoke a Lambda function during the pipeline, Lambda callback)
- Amazon CloudWatch Logs (log processing)
- Amazon Simple Email Service
- AWS CloudFormation
- AWS Config
- AWS IoT
- AWS IoT Events

Lambda Concurrency and Throttling

Lambda Concurrency:

Concurrency is the **number of in-flight requests** that your AWS Lambda function is handling at the same time.

Concurrency Limit:

- Each AWS account has a default **1,000 concurrent execution limit.**
- The limit can be increased by requesting a quota increase from AWS.

Lambda throttling:

When the number of concurrent executions **exceeds the set limits,** Lambda starts throttling new invocation requests.

Throttled requests return a **429 Too Many Requests error**.

Retry Policy:

- Lambda automatically **retries throttled requests twice**, with delays between retries.
- Asynchronous invocations are queued and retried based on the retry policy.

Calculating Concurrency for a Lambda Function

Determine the Expected Traffic:

- Estimate the number of **requests per second** your function will receive.
- This can be based on historical data, application requirements, or traffic patterns.

Measure the Average Execution Duration:

- Measure **how long it takes** for your Lambda function **to execute** on average (in seconds).

Calculate the Concurrency:

Use the formula:

Concurrency = **Requests per Second × Average Execution Duration** (Seconds)

Calculating Concurrency Example Calculation:

Expected Traffic:

- Assume your Lambda function receives 50 requests per second.

Average Execution Duration:

- Assume the average execution duration of your function is 0.2 seconds.

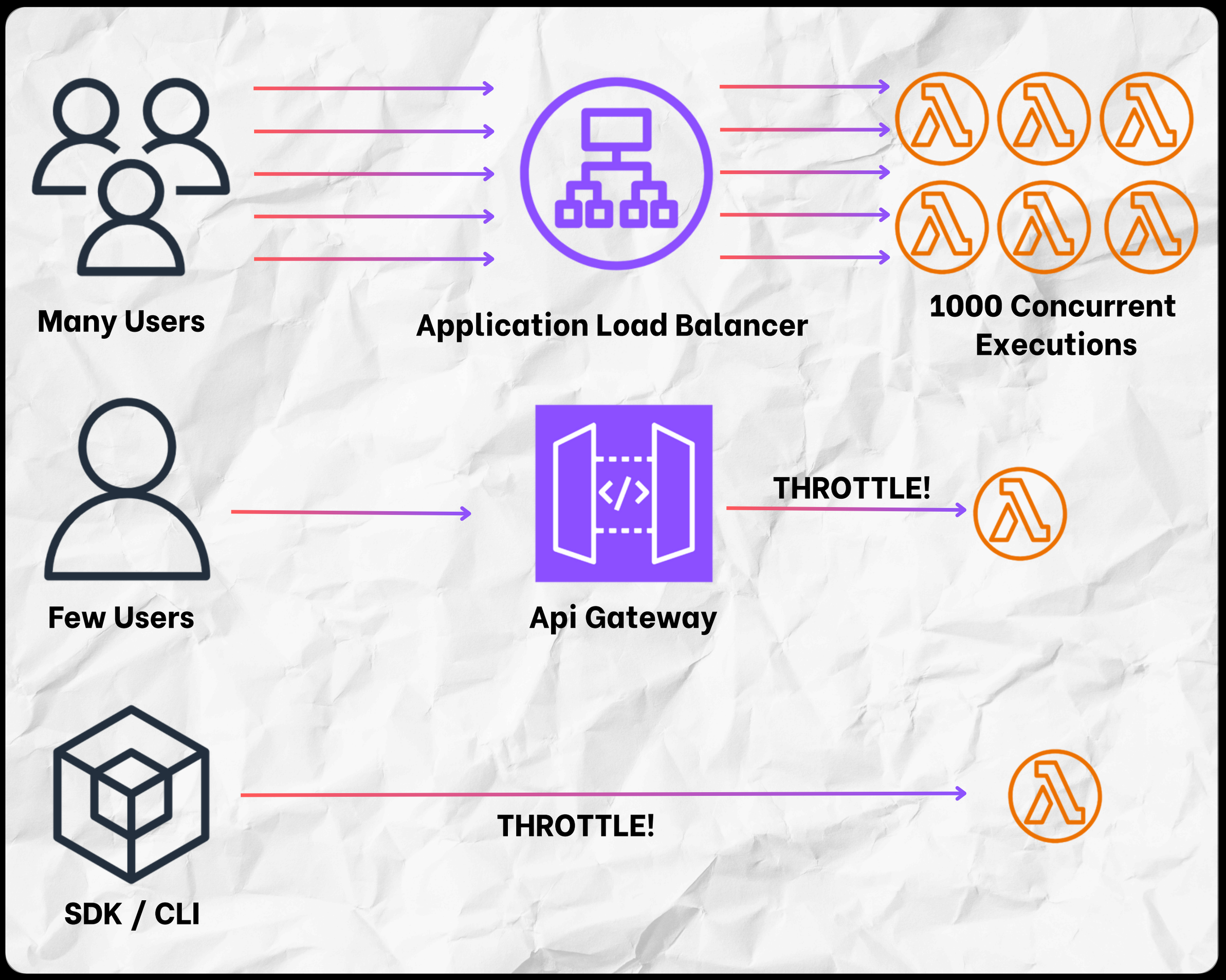
Calculate the Concurrency:

Concurrency = **50 requests / second × 0.2 seconds** = **10**

So, you need a **concurrency of 10** to handle this load without throttling.

Lambda Concurrency Issue

If you don't limit concurrency, the following can happen:



Lambda, Types of Concurrency

Some of your functions might be **more critical** than others.

As a result, you might want to configure concurrency settings to ensure that critical functions **get the concurrency that they need.** There are **two types of concurrency controls available:**

Types of concurrency:

Reserved Concurrency:

Use it to **reserve a portion of your account's concurrency** for a function. This is useful if you don't want other functions taking up all the available unreserved concurrency.

Provisioned Concurrency:

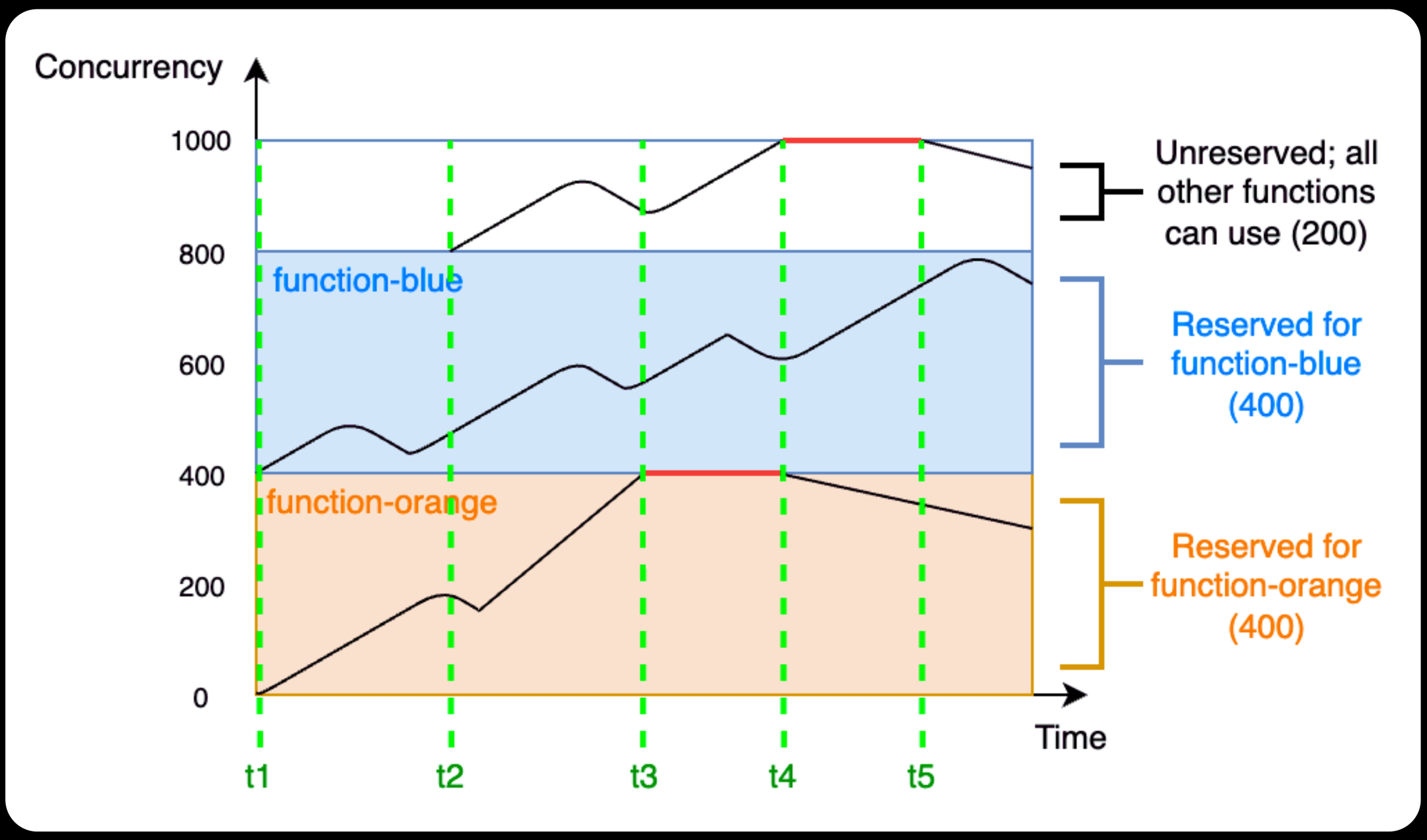
Use it to **pre-initialize several environment instances** for a function. This is useful for reducing cold start latencies.

Lambda Reserved Concurrency

If you want to **guarantee that a certain amount of concurrency is available** for your function at any time, use reserved concurrency.

When you dedicate reserved concurrency to a function, **no other function can use that concurrency.**

There is **no charge for configuring reserved concurrency** for a function.

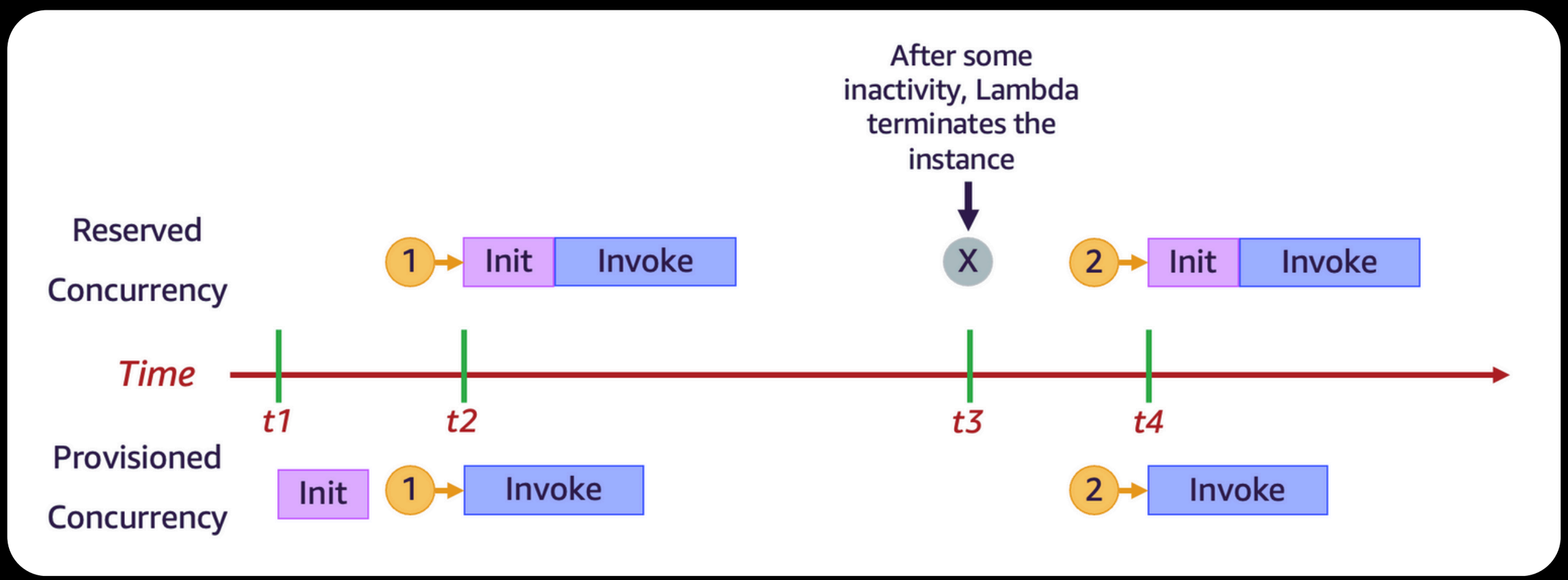


Lambda Provisioned Concurrency

Provisioned concurrency is the number of **pre-initialized execution environments** that you want to allocate to your function.

If you set provisioned concurrency on a function, Lambda initializes that number of execution environments so that they are **prepared to respond immediately to function requests.**

Note: Using provisioned concurrency **incurs additional charges** to your account.

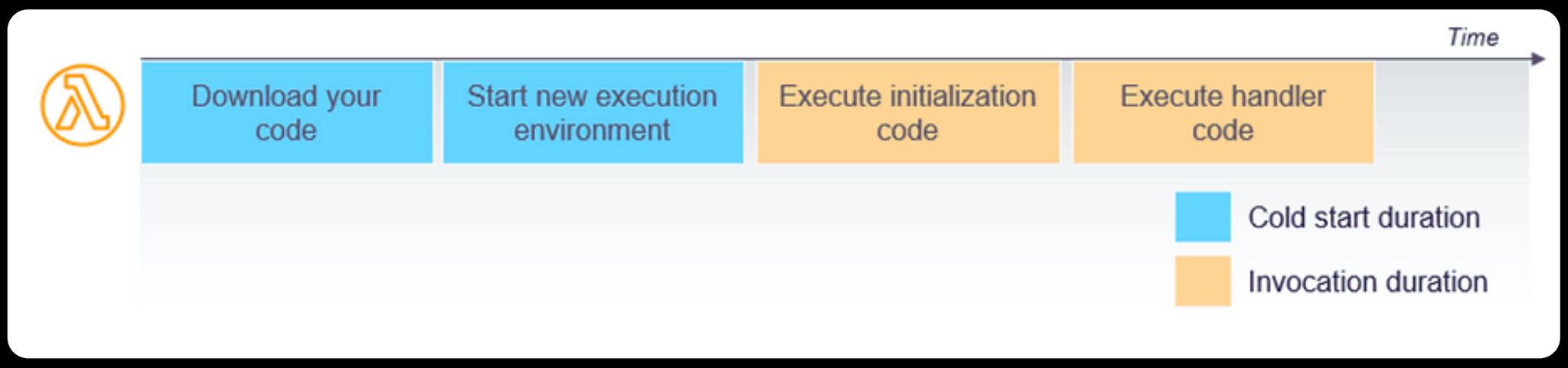


Time	Reserved concurrency	Provisioned concurrency
t1	Nothing happens.	Lambda pre-initializes an execution environment instance.
t2	Request 1 comes in. Lambda must initialize a new execution environment instance.	Request 1 comes in. Lambda uses the pre-initialized environment instance.
t3	After some inactivity, Lambda terminates the active environment instance.	Nothing happens.
t4	Request 2 comes in. Lambda must initialize a new execution environment instance.	Request 2 comes in. Lambda uses the pre-initialized environment instance.

Lambda Cold Start

A cold start **occurs when AWS Lambda has to set up a new execution environment to run a function.**

This setup includes **initializing** the **runtime environment, downloading the function code, and running the initialization code.**



Factors Affecting Cold Starts:

Runtime: Different runtimes have different cold start durations.

Language Choice:

- Statically typed languages (Java, C#) generally have longer cold start times due to more complex runtime initialization.
- Dynamically typed languages (Python, Node.js) tend to have shorter cold start times.

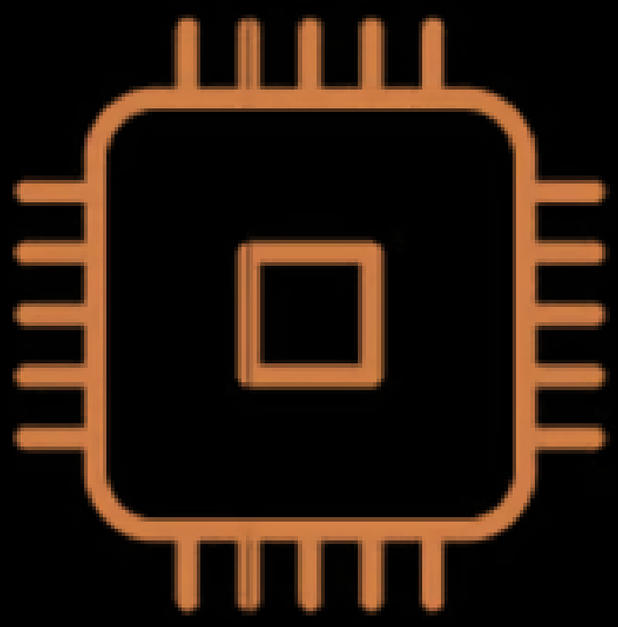
Package Size: Larger function packages take longer to download and initialize.

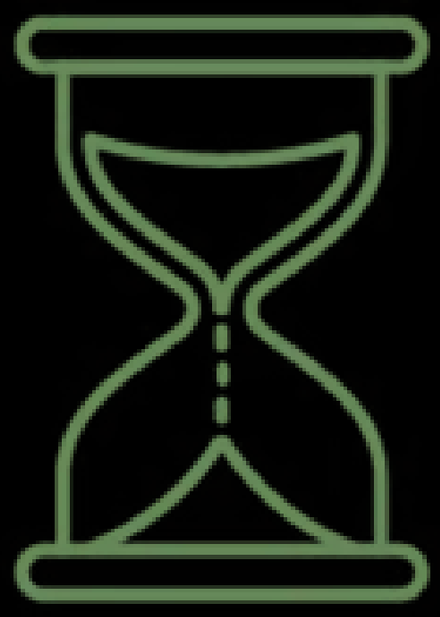
VPC Configuration: Functions inside a Virtual Private Cloud (VPC) experience longer cold starts due to additional networking setup.

Provisioned Concurrency: Functions with provisioned concurrency have reduced cold starts as environments are pre-warmed.

Lambda Memory And Timeout

When developing and evaluating a function, it's essential to define three key configuration parameters: **memory allocation, timeout duration, and concurrency level.**





These settings play a crucial role in **measuring the performance** of the function.

Determining the optimal configuration for memory, timeout, and concurrency involves **testing in real-world scenarios and under peak loads.**

Continuously monitoring your functions allows for adjustments to be made **to optimize costs and maintain the desired customer experience** within your application.

Lambda Memory

Lambda functions allow for allocating up to **10 GB of memory**. Lambda **allocates CPU and other resources in direct proportion to the amount of memory configured.**

Scaling up the memory size results in a corresponding **increase in** available **CPU** resources for your function. To determine the optimal memory configuration for your functions, consider utilizing the **AWS Lambda Power Tuning tool.**

AWS Lambda Power Tuning:

It is a state machine powered by AWS Step Functions, **optimizes Lambda functions for cost and performance.**

It's language agnostic and suggests the best power configuration for your function, **based on multiple invocations across various memory settings (128MB to 10GB)**, aiming to minimize costs or maximize performance.

Memory Configuration:

- **Setting Memory:** You set the memory allocation when you create or update a Lambda function.
- **AWS Management Console:** Set memory in the configuration tab.
- **AWS CLI/SDK:** Use the **update-function-configuration** command.

bash Copy code

```
aws lambda update-function-configuration --function-name myFunction --memory-size 1024
```

Lambda Timeout

The AWS Lambda timeout value **specifies the duration a function can run before Lambda terminates it.** Currently capped at **900 seconds**, this limit means a Lambda function invocation cannot exceed 15 minutes.

A Lambda Serverless application is made up of **three major components.** Each of these components can time out, affecting your serverless application:

- **Event source** – commonly the AWS API Gateway
- **Lambda function** – affected by service limits of AWS Lambda
- **Services** – other resources the Lambda function integrates with, commonly DynamoDB, S3, and third party apps

Serverless Component	Max Timeout	Comments
API Gateway	50 milliseconds – 29 seconds	Configurable
Lambda Function	900 seconds (15 minutes)	Also limited to 1,000 concurrent executions. If not handled, can lead to throttling issues.
DynamoDB Streams	40,000 write capacity units per table	No timeout by default
S3	No timeout by default, can be configured to 5-10 seconds	Unlimited objects per bucket
Downstream Applications	Check your applications	

Lambda Timeout Best Practices

Use Short Timeouts:

- 3-6 seconds for API calls.
- Adjust for Kinesis, DynamoDB, SQS based on batch size.

Monitor Timeouts:

- Use CloudWatch and X-Ray.
- Fine-tune based on data.

Fallback Methods:

- Return error codes.
- Use cached data or alternatives (Hystrix/Spring Retry for Java, oibackoff for Node.js).

Manage DynamoDB Writes:

- Avoid exceeding 40,000 writes.
- Use node-rate-limiter for Node.js.

Optimize Functions:

- Break long-running tasks into smaller steps with Step Functions.

Balance Performance and Cost:

- Increase memory for CPU tasks to reduce execution time.
- For DB-heavy tasks, more memory won't help.
- Adjust memory to keep execution time below 100ms increments to save costs.

Lambda Functions /tmp space

You can use the /tmp directory if:

- Your Lambda function needs to **download a big file to work.**
- Your Lambda function **needs disk space** to perform operations.

By default, /tmp comes with **512MB** of storage, but you can configure it to be as large as **10GB.**

How is this different from using Lambda with EFS?

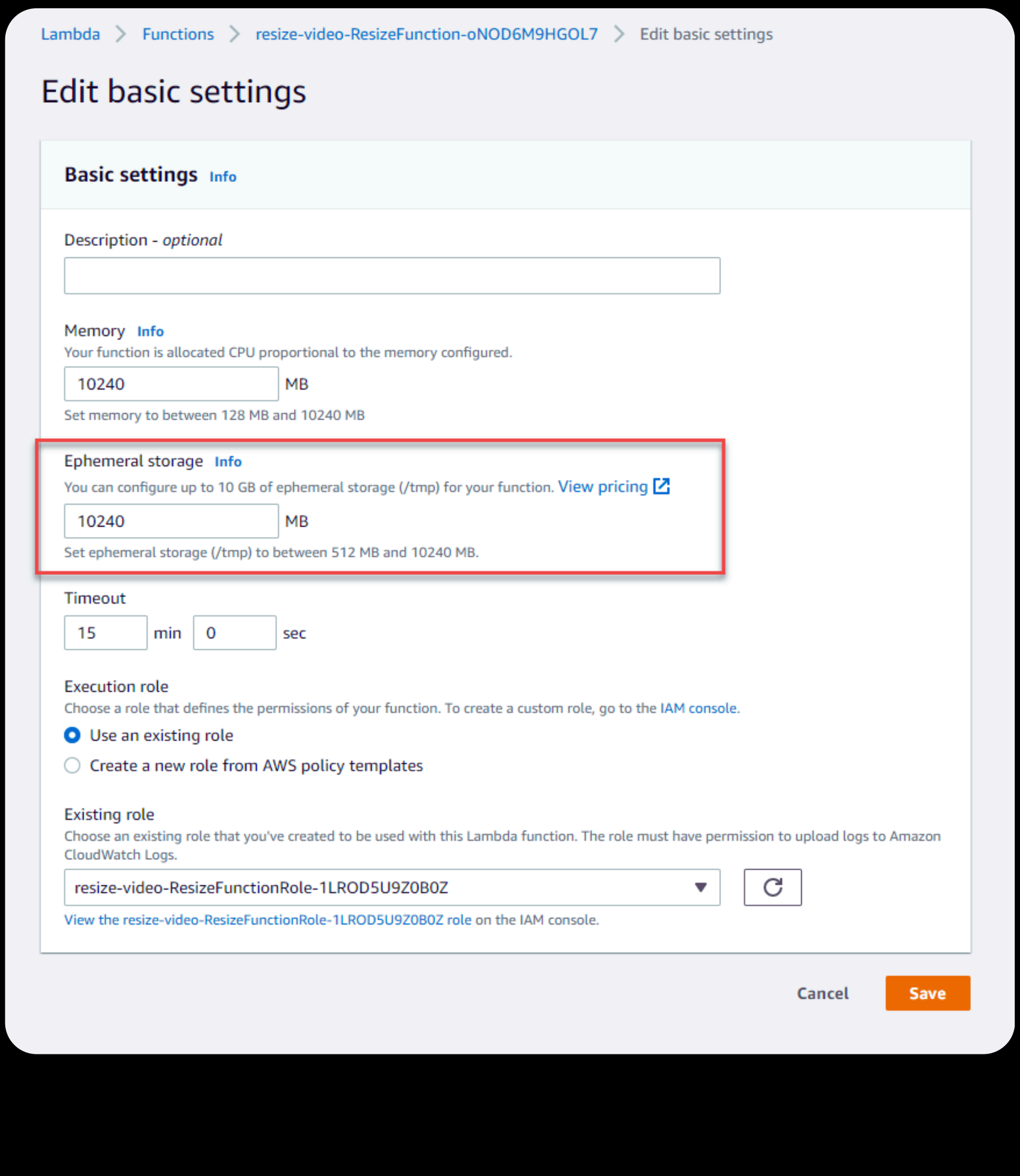
- **Performance:** EFS is a network file system and its read and write **latency is** therefore much **higher** (~5-10x) than the ephemeral storage.

- **Data sharing:** Each Lambda function worker has its own instance of /tmp directory and they don't share any data. Whereas with EFS, **data can be shared.**

Lambda Functions /tmp space

Increase /tmp space:

- Via **CloudFormation** or any tools that use CloudFormation under the hood.
- Via the AWS **CLI** or **SDK** or AWS **Console.**

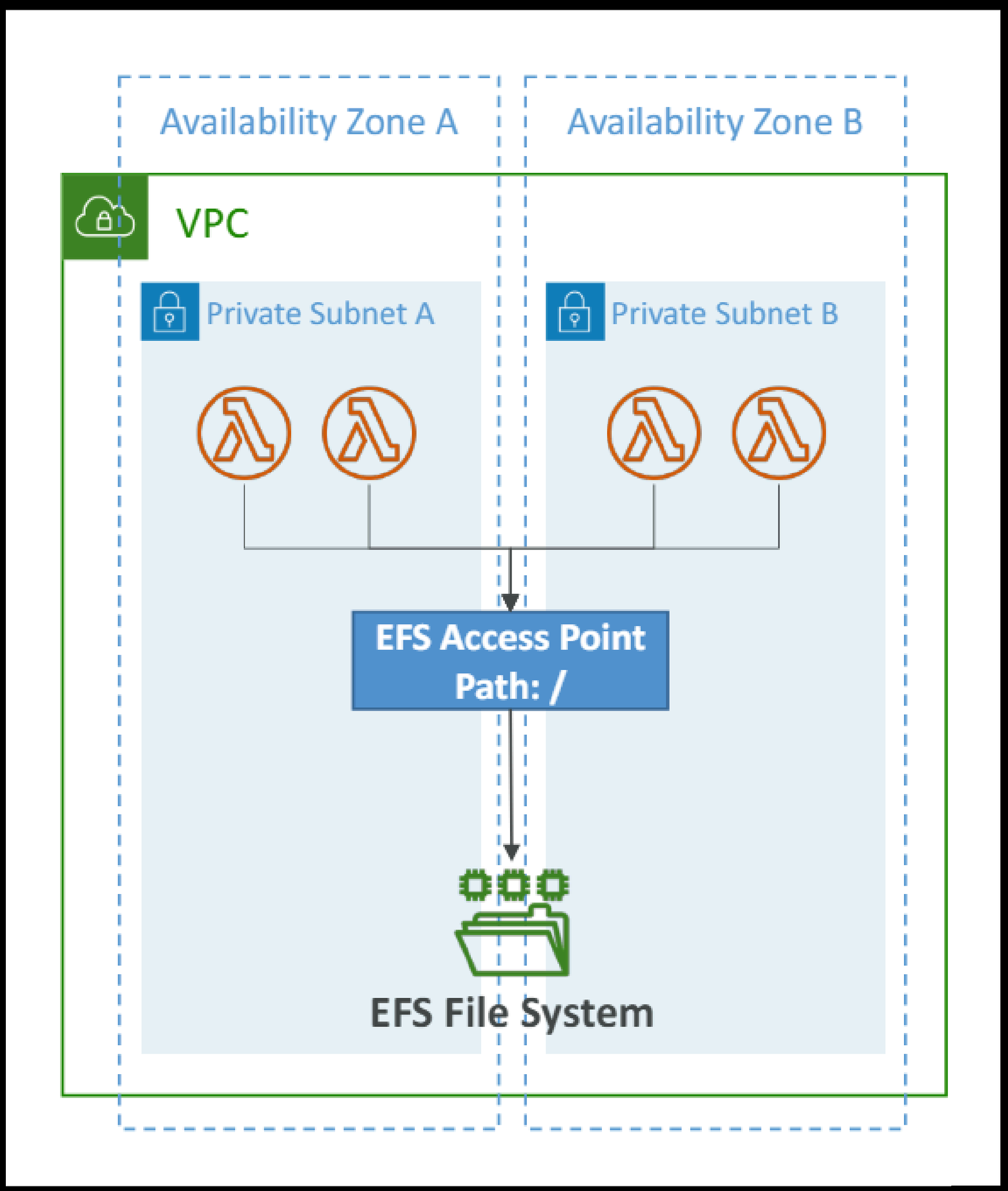


Lambda File System Mounting

Lambda functions can **access EFS file systems if they are running in a VPC.**

Configure Lambda to mount EFS file systems to the local directory during initialization, Must **leverage EFS Access Points**.

Limitations: watch out for the EFS connection limits **(one function instance = one connection)** and connection burst limits.



Lambda Storage Options

Ephemeral Storage /tmp	Lambda Layers	Amazon S3	Amazon EFS
Max. Size	10,240 MB	5 layers per function up to 250MB total	Elastic
Persistence	Ephemeral	Durable	Durable
Content	Dynamic	Static	Dynamic
Storage Type	File System	Archive	Object
Operations Supported	any File System operation	Immutable	Atomic with Versioning
Pricing	Included in Lambda	Included in Lambda	Storage + Requests + Data Transfer
Sharing/Permissions	Function Only	IAM	IAM
Relative Data Access Speed from Lambda	Fastest	Fastest	Fast
Shared Across All Invocations	No	Yes	Yes

Lambda Monitoring

Why Is AWS Lambda Monitoring Important?

Optimize Performance and Resource Usage:

- Detect bottlenecks, high latency, and resource constraints. Gain insights into memory, CPU, and execution duration for informed resource allocation and cost control.

Error Detection and Troubleshooting:

- Capture and analyze error messages, exceptions, and anomalies for quick fixes.

Maintain Reliability and Availability:

- Proactively detect and address issues to minimize user impact.

Gain Insights into Application Behavior:

- Understand application behavior under various conditions for better design and architecture decisions.

Ensure Compliance and Security:

- Identify security vulnerabilities, meet compliance requirements, and set up proactive alerts to quickly resolve issues.

Lambda Monitoring

Key Concepts of AWS Lambda Monitoring

Metrics:

Lambda publishes metrics like request rate, error rate, and invocation duration. Create custom metrics for specific use cases.

Logs:

Default logging with Amazon CloudWatch logs discrete events. Option to use third-party logging systems.

Alerts:

CloudWatch alerts notify operators when metrics exceed expected bounds. Integrate with monitoring systems for comprehensive anomaly reporting.

Visualization:

Display metrics visually and organize them in dashboards.

Distributed Tracing:

Identify the full request path in microservice architectures.

Lambda Metrics

Lambda invocation metrics include:

Invocations
The total number of times Lambda executes the function code (including successful and failed executions).

Errors
The number of Lambda invocations resulting in function errors.

DeadLetterErrors
The number of failed attempts Lambda makes to send an event to the dead letter queue (for asynchronous invocations).

Throttles
The number of throttled invocation requests (Lambda rejects these requests intentionally).

ProvisionedConcurrencyInvocations
The number of times Lambda executes the function code on provisioned concurrency.

ProvisionedConcurrencySpilloverInvocations
The number of times Lambda executes the function code on standard concurrency while provisioned concurrency is all in use.

Lambda Metrics

Lambda performance metrics include:

Duration

The time the function code takes to process an event.

IteratorAge

The age of the oldest event record (for event source mappings reading from streams).

Lambda Monitoring with CloudWatch

CloudWatch Logs:

- AWS Lambda execution logs are stored in AWS CloudWatch Logs
- Make sure your AWS Lambda function **has** an **execution role with an IAM policy that authorizes writes to CloudWatch Logs**

CloudWatch Metrics:

- AWS Lambda metrics are displayed in AWS CloudWatch Metrics
- Invocations, Durations, Concurrent Executions
- Error count, Success Rates, Throttles
- Async Delivery Failures
- Iterator Age (Kinesis & DynamoDB Streams)

Lambda Tracing with X-Ray

- Enable in Lambda configuration **(Active Tracing)** under **Configuration > Monitoring tools > X-Ray > Active tracing.**
- **Runs** the **X-Ray daemon** for you
- Use AWS X-Ray SDK in Code
- Ensure Lambda Function **has** a correct **IAM Execution Role**
- The **managed policy** is called **AWSXRayDaemonWriteAccess**
- Environment variables to communicate with X-Ray
  - **_X_AMZN_TRACE_ID**: contains the tracing header
  - **AWS_XRAY_CONTEXT_MISSING**: by default, LOG_ERROR
  - **AWS_XRAY_DAEMON_ADDRESS**: the X-Ray Daemon IP_ADDRESS:PORT

Best Practices for Lambda Monitoring

- **Custom Metrics and Logs:** Create custom metrics and logs for targeted insights.
- **Structured Logging:** Use formats like JSON for easier analysis in CloudWatch Logs.
- **Monitor Cold Starts:** Track and optimize to reduce latency.
- **Optimize Timeouts and Memory:** Balance performance and cost by monitoring and adjusting these settings.
- **Concurrency and Throttling:** Track and adjust limits to maintain performance.
- **Monitor Security Events:** Track unauthorized access and configuration changes.
- **Third-Party Tools:** Use Datadog, New Relic, or Dynatrace for deeper insights.
- **Establish a Baseline:** Define normal performance to detect deviations and optimize effectively.

Lambda Versions

Lambda function versions are resources you create to encapsulate function code and configuration **as a snapshots.**

Key Features

- **Immutable:** Each version is a snapshot of your function code and configuration.
- **Identifier:** Each version has a unique ARN (Amazon Resource Name).
- **$LATEST Version:** Represents the most recent unpublished changes.
- **Version Numbers:** Start from 1 and increment with each new version.

If you apply concepts from software version control systems, like Git.

You will notice that the Lambda Function version is similar to the git commit.

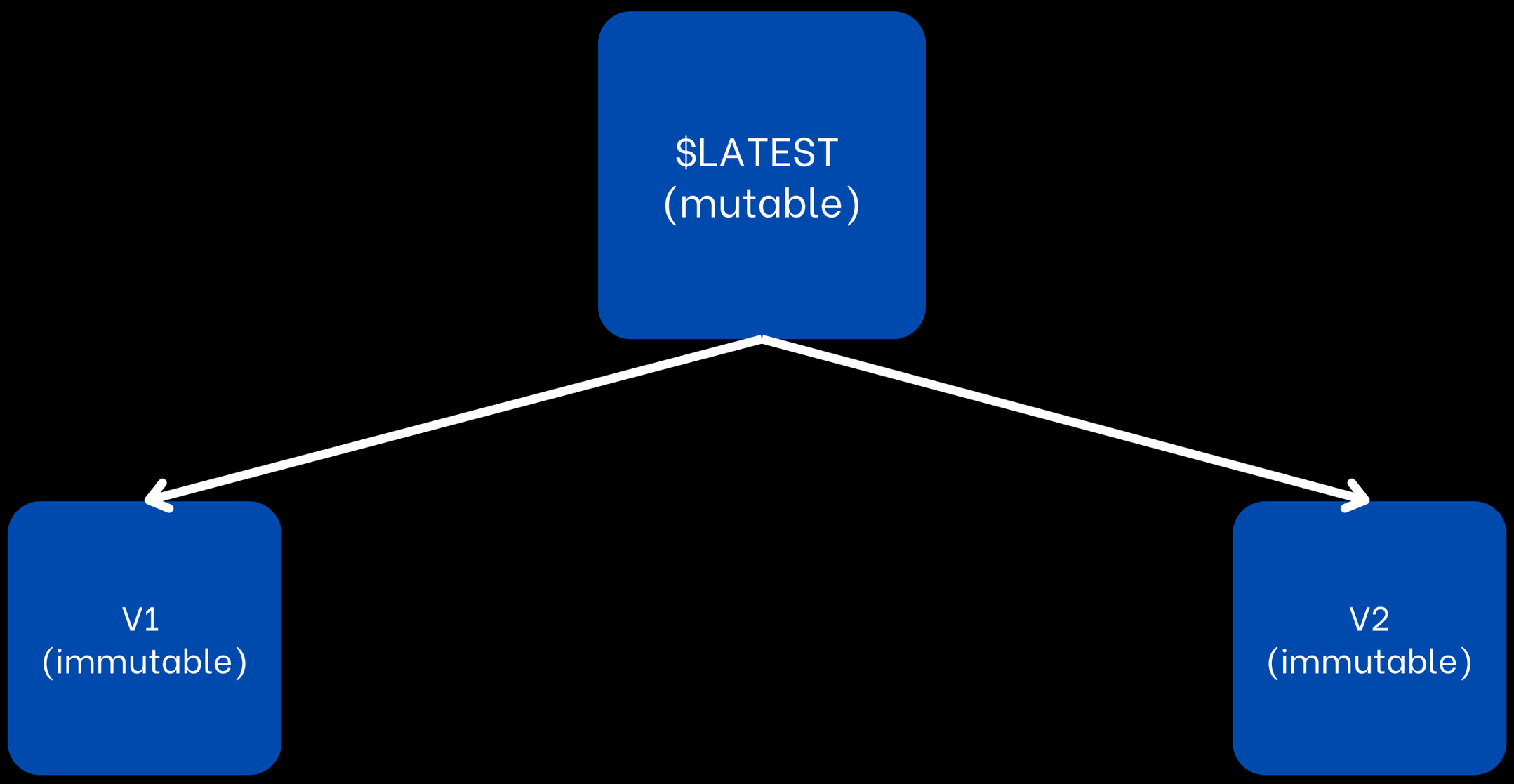
A git commit captures a snapshot of the project's currently staged changes, and each commit is assigned an ID that can be used as a reference value.

$LATEST version acts as a git staging area, containing all changes that are not yet committed or in Lambda function version terms published.

Lambda Versions

A function version includes the following information:

- The function code and all associated dependencies.
- The Lambda runtime that invokes the function.
- All the function settings, including the environment variables.
- A unique Amazon Resource Name (ARN) to identify the specific version of the function.



Lambda Aliases

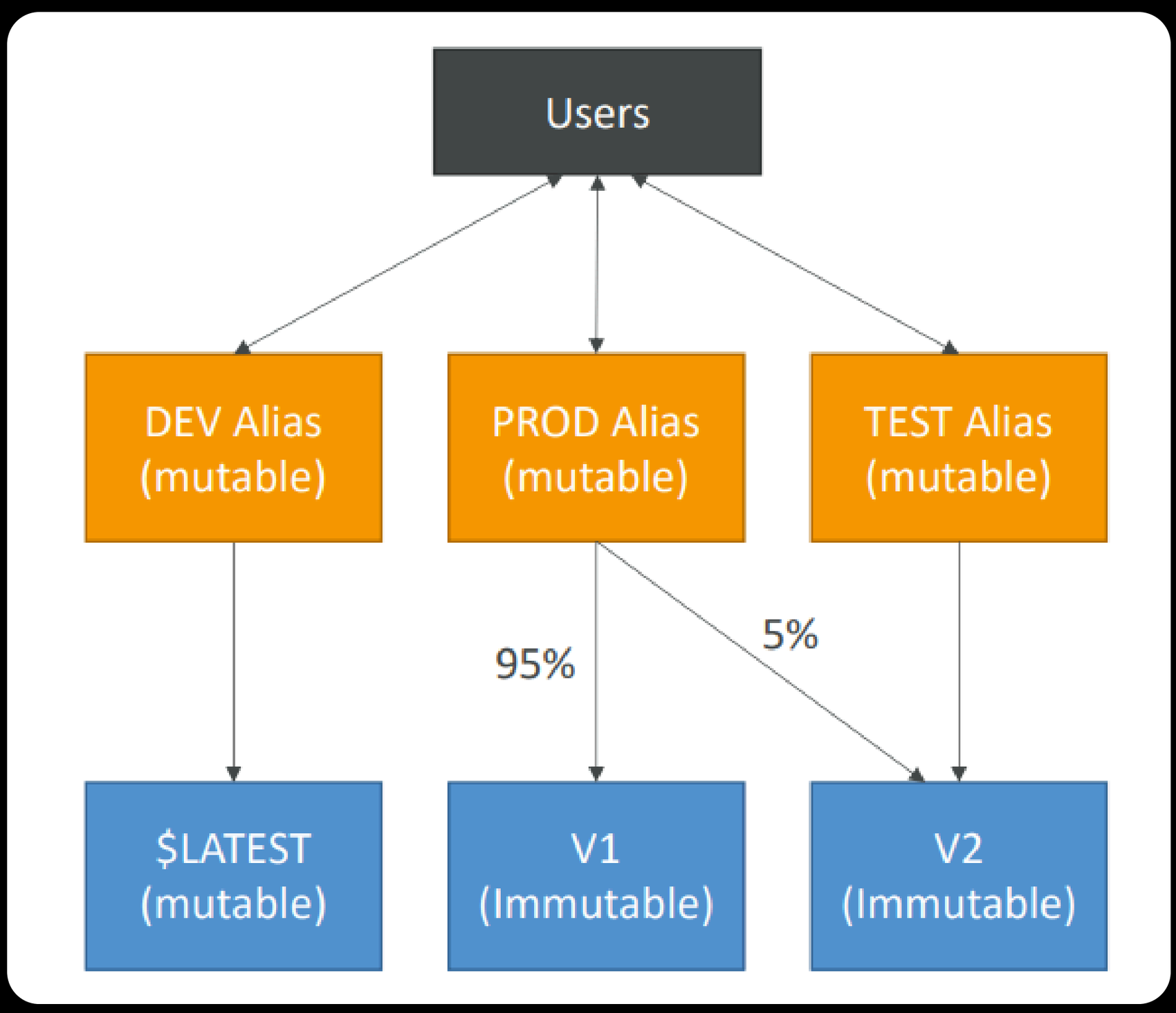
The alias is simply a **pointer to a specific Lambda function version.** Each alias also **has a unique ARN**.

- We can define a **“dev”, ”test”, and “prod”** aliases and **have them point at different lambda versions.**

- Aliases enable **Canary deployment** by assigning weights to lambda functions.

- Aliases enable stable configuration of our event triggers / destinations.

- Aliases **cannot reference aliases.**



Next Week I'm covering the Part 2 of Advanced Lambda Topics on my Newsletter.

Subscribe to the Newsletter

rajankfl.substack.com

REPOST

FOLLOW FOR MORE GUIDES!