AWS Lambda - Serverless Architecture

Area of Focus	Summary
Introduction	Lambda is the heart of serverless - run code without provisioning and managing servers, triggers on your behalf based on events, scales automatically, provides built in code monitoring and logging) Big benefit of cloud computing you can abstract the infrastructure layer (not working about deployment and operations related to configuration, patching, updating OS, install app platform, scale, automating scaling). With serverless platform - dev can focus on the code that makes your business unique to experiment and innovate faster. Is available in all 18 AWS regions - to create highly available and secure services. In traditional infrastructure architecture you need LB, Scaling up and down, handling system and datacenter failures (health checks based on key service metrics), repeat for each system and service component. Developers spend alot of time in system administration. Lambda handles load balancing, auto scaling, handling failures, security isolations, utilization.

Architecture	Control Plane - dev interacts with developer tools (Lambda console, SAM CLI) and control plane APIs (Configuration, Resource Management) Data Plane - Asynchronous Invoke & Events (Pollers, State Manager, Leasing Service) working together to process events. Once events are processed - they are handed over to the Synchronous Invoke (Front End Invoke, Counting Service, Worker Manager, Worker, Placement service) • Front End Invoke - orchestration both sync and async invokes (authenticate callers to allow valid invokers) and load the function metadata and confirm the concurrency. May customer function to worker manager - as function concurrency scales up the workers are scaled up. • Counting Service - provide a region wide view of customer concurrency to help enforce set limits. If it hits concurrency limit - it may or may not throttle. Critical component - which is resilient to failure • Worker Manager - tracks container idle and busy state and schedule incoming invoke requests to available containers. Function invocation, environment variable setup, customer execution role, spin down workers when it's idle. • Worker - provisions a secure environment for customer code execution. Creates and manages a collection of sandboxes and sets limit on CPU and memory, manage multiple language runtime, manage AWS agents for monitoring (CloudWatch), notify worker manager with sandbox invoke completes. • Placement Service - places sandbox on workers to maximize packing density without impacting customer experience or cold-path latency. Where to place sandbox when its ready for execution. Mark a worker when it's unhealthy. **Load Balancing** - routing function traffic across across hosts distributed across Availability Zones. Front end will validate the caller (authentication), retrieve the function meta data and check with counting service to see current concurrency vs. limit. Front end talk to Worker Manager to reserve a sandbox - Worker manager identifies a worker will create sandbox, download code, and initialize the runtime and call the customer code in it. This is a warm sandbox. Then Front end can call the invoke. Worker lets the worker manager knows when it's idle (it has a warm sandbox). Front end tells Worker Manager when it's done. Load Balancing is required for high TPS and high availability. **Auto Scaling** - provision function capacity when needed and releasing when not needed. Call invoke - hits the app load balancer. Front End authenticates and retrieve function meta data, concurrency check, reserve sandbox with worker manager. Worker manager doesn't have a worker or sandbox. It claims a worker from the placement service - placement service does evaluation and identify a good place to provision the sandbox. The worker manager is going to create the sandbox, download code, initialize the runtime and call init on the code. This a warm sandbox. The front end calls invoke. Then tells worker manager when it's done. The placement service is responsible for managing the worker to the worker manager (lease of 6 to 10 hours), lease duration is impacted by function duration. When worker gets closer to the lease expire it needs to return the worker. If worker manager finds worker close to expiration it will stop reserving sandboxes on the worker (all sandboxes become idle) then the worker can be returned. **Handling Failures** - handling host and availability zone failures. Built across multiple availability zones - when worker become unhealthy the worker manager detects and stops provisioning sandboxes on the worker. When entire AZ fails, the system components will not route to the failed AZ. **Isolation** - keeping workloads safe and separate. Lambda functions run on same hardware and servers. Isolation - security and operational isolation (ensure functions run at a consistent performance when other functions are on the same hardware, noisy neighbors). Worker - Your Code, Lambda runtime, Sandbox, Guest OS (amazon linux) with 1000's isolated using hypervisor, Hypervisor, Host OS (amazon Linux), Hardware • One Function (your Code, Lambda runtime, sandbox) leveraging cgroups, namespaces, seccomp, iptables, chroot • One Account (Guest OS, Hypervisor) - virtualization and device emulation • Many Accounts (Hypervisor, Host OS, Hardware) EC2 instances (above the Hypervisor) - each lambda worker is a separate EC2 instance (security and fast way to build it) Firecracker (VMM) - instead of running 1 instance per account we run 1 bare metal and EC2 instances and use firecracker to launch many micro-VMs and it's more flexible and agile boundary. Simplifies the security model - one function in micro VM and multiple microVM across 1 hardware. Provides strong isolation between functions. Each microVM seize a network device - works through virtualization. Gives better startup time, memory overhead, and flexibility. Firecracker unlocks higher utilization and scale. **Utilization** - % of resources doing useful work (vs. idle of waste) to get better container reuse. You pay for only useful work. Optimize to keep servers busy (workers) - *bad* - balance the load (how do you know how busy the server is) - *good* - cache locality (smaller number of sandboxes busy) and ability to autoscale. There is only 1 invoke in a sandbox (busy in a binary way), count # of sandbox and you get a clear picture of load. *bad* - pack server with one workload and loads are correlated. Loads will be spikey. *good* - pack with many loads (diverse set of workloads), take advantadge of statistical multiplexing. *Gr eat* - Placement Optimization - pick workloads that pack together well. Minimize contention.
Features	Bring Your Own Code, Integration with and extends other AWS services, flexible resource and concurrency model, permissions model, availability and fault tolerance, don’t pay for idle SAM - Serverless Application Model, developer tools to simplify deployments

How It Works	Event Source - entity that publish events (triggers your Lambda functions) Custom Event Sources - Data stores (Kinesis, DynamoDB,S3), End Points (AWS services like Alexa, API Gateway), Repositories (CloudWatch, CloudFormation, CloudTrail, CodeCommit), Message Services (SES, SNS, SQS, Cron Events) Push Events - Synchronous, Asynchronous (S3, SNS, SES, CloudWatch logs, CloudWatch Events, Config, CodeCommit) Polling Events - Stream based (DynamoDB, Kinesis), Not Stream based (Amazon SQS). Lambda polls for changes for processing
Lifecycle	Launch Container, Bootstrap (invocation request) to get environment setup then Lambda fees the container. If another invocation happens it goes through a warm start (the frozen container is thawed and executed) and will stay in this state as long as their are requests. If it's idle timeout - it will shutdown. Store and reference dependencies locally, limit reinitialization of variables, check for and resume existing connections, use tmp space as transit cache (500 MB in tmp directory), check that background process have completed before code exists Function is invoked - it will start container and download code and initiative runtime. You can optimize the initialization packages and dependencies and execute code (this is where billing begins)
Functions	Languages that support Lambda (Node.js, Python, Java, Go, C#) - Cloud9 web based IDE Lambda function - Handler Method (java Main) takes 2 objects (event object, context object) - Event Objects provides format, Context Object is generated by AWS and provides metadata (requestId, LogStreamName, getRemainingTime)
Best Practices	Separate business logic (make code more portable, target unit test), Functions modular (functions for specific task), Treat your functions as stateless (tmp directory might not be available in cold start) Persist Data using DynamoDB, ElastiCache less expensive than DynamoDB, S3 can be used also - if throughput isn't critical Minimize dependencies and overall deployment package size, Choose modules you need (not entire AWS SDK), Minimize complexity (Dagger, Juice) over Spring framework Lambda function should incorporate log function Node.js has context methods to terminate functions and return value to caller Use Environment variables to pass operational parameters (updated configuration settings) and store sensitive information (database password, API keys, connection strings) . Avoid Recursive Code - function calls it's self and spawn new invocations. Set the concurrent execution limit to 0
Authoring	• Lambda Management Console - doesn't require custom libraries, can create deployment package using IDE and upload package using upload function, also utilize SAM CLI to simulate locally (must be less than 10 mb in size), not suitable for team approach. Java / C# you need to compile then upload the package • Upload deployment package to Lambda console • Upload deployment package to S3 - specify url if your deployment package is greater than 10 mb. Also good for deployment workflows with automated CI/CD
Integrations	• API Gateway • Kinesis • DynamoDB (store and retrieve items) • S3 (simple storage services) to manage large files and images • IoT • CloudWatch Events • CloudWatch Logs • SNS • Cognito (user integrations)

Configuring	Components to Configuration - Memory, Timeout, Concurrency Memory / Timeout are interdependent to optimize cost and speed. Specify memory and Lambda allocate CPU power. Lambda price goes up as your higher memory. Timeout prevents long running requests - so they don't run too long (3 sec is default) and minimize cost. Lambda is priced on memory and duration of execution. Speed is directly proportional to the memory configured Concurrency - unit of scaling is measured (capacity to server at any specific time). Count limit is set to 1000 concurrent executions, you can set reserves at the function level (saving seats for concurrent functions). This numbers represent a limit of concurrent for specific function. Account Limit - Reserved Concurrency = Unreserved Concurrency Lambda will keep the unreserved pool at a minimum of 100 to ensure that functions without reserver can be invoked. Reserve function concurrency to handle anticipated peaks. Concurrency bursts are managed based on limits and predetermined concurrency increases. Lambda checks every minute to see if more concurrent increase is needed (added). Lambda will not increase concurrent limit past the function concurrency limit Estimating concurrency = request rate x duration of event If # of records > max batch size, Lambda must split into batches. Batch 1 must first process before processing Batch 2 to keep sequence. Request rate = concurrent executions / average function duration
Deploy & Test	Server Based vs. Serverless Deployments - need to know infrastructure is setup to deploy applications. Devops is responsible for build and deploy to desired environment. Serverless you would construct on-demand and convey in standard way. Deploying lambda function is like using AWS cloud formation (infrastructure as code) as a template to deploy as a defined stack. Key Difference for serverless - code, dependencies, and blueprint for infrastructure (cloud formation template - IAM roles, DB connections, etc). SAM as your serverless application framework - minimize the work you need to do. SAM template - allows you to create CloudFormation blueprint (template) with simplified information transformed into CloudFormation template. SAM makes it easy to setup resources that are commonly needed for Serverless applications. Transform Directive = Sam Template • Resources: • Type: create a lambda function • Properties - referenced zip file, handler, runtime to be used • Policies - IAM policies referenced • Events - API Gateway end points • ListTable - Create tables with 5 read and write units SAM to deploy stack to multiple accounts - ensures environment parity, simplifies feature branch testing and experimentation. You don't have overhead to setup environments - SAM allows the creation of on demand environments. SAM CLI - invokes a docker container locally before you deploy to the cloud (run API locally, debug code, etc) Versioning and Aliases Deployments - mitigate risk to production roll out. Utilize CI/CD to promote code. When you create Lambda function it creates as $LATEST. The snapshot becomes the new version and is immutable. You can publish multiple versions. Aliases to easily mange references to versions in your code. Promote and roll back versions by updating the aliases that point to them. When an alias is updated to point to a new version, if there is a problem - Lambda allows you to specific a second version for an alias and limit how much traffic goes to each version. Lambda is integrated with CodeDeploy for automated rollout - Canary (shifted in 2 increments), Linear (equal increments of X with Y minutes between each increment). • AutoPublishAlias = live • Type: Canary10percent10Minutes (implement traffic shifting) • Alarm - roll back deployment based on specified cloud watch alarms • Hooks - run sanity checks before traffic shifting starts and after traffic shifting completes • Rollback - everything in the SAM template are rolled back • Use 1 template per service
Monitoring	• Amazon CloudWatch (Invocation, Errors, Duration, Throttles, IteratorAge, ConcurrentExecutions, Unreserved Concurrent Executions) • Dead Letter Queues - capture application errors that you can't discard. Dead Letter Queue (asynchronous push) or discarded after 3 attempts, Failed invokes with be put into your queue for you to deal with later • AWS CloudTrail - audit API calls made by or on behalf of a function, Audit actions made against your application, Integrate with CloudWatch event rule and react if production function is changed, • AWS X-Ray - trace path and timing of an invocation to locate bottlenecks and failures, for performance tuning, identify call flow of functions (high level view of your function) and trace individual API calls

Lambda@Edge	AWS Edge Network - 138 Pops across 63 cities (Edge high capacity Pops, Regional Edge Cache with higher storage capacity) Edge locations - sit on AWS backbone which is private. Edge Network - AWS Shield (protection of DDOS), AWS WAF (Web application firewall), Route 53 (DNS), CloudFront (Content Delivery Network for video streaming, static content, accelerating access to API), Lambda@Edge (Distributed serverless capability that works with CloudFront) Lambda - • Just your runtime • Upload your code to a region • Define how you want your code to be triggered What if you can run Lambda function closer to the user - Lambda@Edge is triggered by CloudFront events that is triggered closer to the user. User gets faster response to content. Publish to CloudFront - and will associate to global regions. The nearest lambda function will be invoked. Use Cases - Simple HTTP Manipulations (user agent header normalization, adding HSTS security headers, enforcing cache control headers, A/B testing), Dynamic content generations (image manipulation, render pages, redirections, Search engine optimization), Origin independence (pretty URLs to mask directory structure, API wrapper to mask legacy API, Authorization, Bot mitigation)