AWS IoT Lens

AWS Well-Architected Framework

December 2019

Notices

Customers are responsible for making their own independent assessment of the information in this document. This document: (a) is for informational purposes only, (b) represents current AWS product offerings and practices, which are subject to change without notice, and (c) does not create any commitments or assurances from AWS and its affiliates, suppliers or licensors. AWS products or services are provided “as is” without warranties, representations, or conditions of any kind, whether express or implied. The responsibilities and liabilities of AWS to its customers are controlled by AWS agreements, and this document is not part of, nor does it modify, any agreement between AWS and its customers.

Contents

Abstract

This whitepaper describes the **AWS IoT Lens** for the AWS Well-Architected Framework, which enables customers to review and improve their cloud-based architectures and better understand the business impact of their design decisions. The document describes general design principles, as well as specific best practices and guidance for the five pillars of the Well-Architected Framework.

Introduction

The AWS Well-Architected Framework helps you understand the pros and cons of the decisions you make when building systems on AWS. Using the Framework allows you to learn architectural best practices for designing and operating reliable, secure, efficient, and cost-effective systems in the cloud. The Framework provides a way for you to consistently measure your architectures against best practices and identify areas for improvement. We believe that having well-architected systems greatly increases the likelihood of business success.

In this “Lens” we focus on how to design, deploy, and architect your IoT workloads (Internet of Things) in the AWS Cloud. To implement a well-architected IoT application, you must follow well-architected principles, starting from the procurement of connected physical assets (things) to the eventual decommissioning of those same assets in a secure, reliable, and automated fashion. In addition to AWS Cloud best practices, this document also articulates the impact, considerations, and recommendations for connecting physical assets to the internet.

This document only covers IoT specific workload details from the Well-Architected Framework. We recommend that you read the AWS Well-Architected Framework whitepaper and consider the best practices and questions for other lenses.

This document is intended for those in technology roles, such as chief technology officers (CTOs), architects, developers, embedded engineers, and operations team members. After reading this document, you will understand AWS best practices and strategies for IoT applications.

Definitions

The AWS Well-Architected Framework is based on five pillars — operational excellence, security, reliability, performance efficiency, and cost optimization. When architecting technology solutions, you must make informed tradeoffs between pillars based upon your business context. For IoT workloads, AWS provides multiple services that allow you to design robust architectures for your applications. Internet of Things (IoT) applications are composed of many devices (or things) that securely connect and interact with complementary edge-based and cloud-based components to deliver business value. IoT applications gather, process, analyze, and act on data generated by connected devices. This section presents an overview of the AWS components that are

used throughout this document to architect IoT workloads. There are seven distinct logical layers to consider when building an IoT workload:

- Design and manufacturing layer
- Edge layer
- Provisioning layer
- Communications layer
- Ingestion layer
- Analytics layer
- Application layer

Design and Manufacturing Layer

The design and manufacturing layer consists of product conceptualization, business and technical requirements gathering, prototyping, module and product layout and design, component sourcing, and manufacturing. Decisions made in each phase impact the next logical layers of the IoT workload described below. For example, some IoT device creators prefer to have a common firmware image burned and tested by the contract manufacturer. This decision will partly determine what steps are required during the Provisioning layer.

You may go a step further and burn a unique certificate and privacy key to each device during manufacturing. This decision can impact the Communications layer, since the type of credential can impact the subsequent selection of network protocols. If the credential never expires it can simplify the Communications and Provisioning layers at the possible expense of increased data loss risk due to compromise of the issuing Certificate Authority.

Edge Layer

The edge layer of your IoT workload consists of the physical hardware of your devices, the embedded operating system that manages the processes on your device, and the device firmware, which is the software and instructions programmed onto your IoT devices. The edge is responsible for sensing and acting on other peripheral devices. Common use cases are reading sensors connected to an edge device, or changing the state of a peripheral based on a user action, such as turning on a light when a motion sensor is activated.

AWS IoT Device SDKs simplify using AWS IoT Core with your devices and applications with an API tailored to your programming language or platform.

Amazon FreeRTOS is a real time operating system for microcontrollers that lets you program small, low-power, edge devices while leveraging memory-efficient, secure, embedded libraries.

AWS IoT Greengrass is a software component that extends the Linux Operations System of your IoT devices. AWS IoT Greengrass allows you to run MQTT local routing between devices, data caching, AWS IoT shadow sync, local AWS Lambda functions, and machine learning algorithms.

Provisioning Layer

The provisioning layer of your IoT workloads consists of the Public Key Infrastructure (PKI) used to create unique device identities and the application workflow that provides configuration data to the device. The provisioning layer is also involved with ongoing maintenance and eventual decommissioning of devices over time. IoT applications need a robust and automated provisioning layer so that devices can be added and managed by your IoT application in a frictionless way. When you provision IoT devices, you must install a unique cryptographic credential onto them.

By using **X.509 certificates,** you can implement a provisioning layer that securely creates a trusted identity for your device that can be used to authenticate and authorize against your communication layer. X.509 certificates are issued by a trusted entity called a certificate authority (CA). While X.509 certificates do consume resources on constrained devices due to memory and processing requirements, they are an ideal identity mechanism due to their operational scalability and widespread support by standard network protocols.

AWS Certificate Manager Private CA helps you automate the process of managing the lifecycle of private certificates for IoT devices using APIs. Private certificates, such as X.509 certificates, provide a secure way to give a device a long-term identity that can be created during provisioning and used to identify and authorize device permissions against your IoT application.

AWS IoT Just In Time Registration (JITR) enables you to programmatically register devices to be used with managed IoT platforms such as AWS IoT Core. With Just-In-Time-Registration, when devices are first connected to your AWS IoT Core endpoint, you can automatically trigger a workflow that can determine the validity of the certificate identity and determine what permissions it should be granted.

Communication Layer

The Communication layer handles the connectivity, message routing among remote devices, and routing between devices and the cloud. The Communication layer lets you establish how IoT messages are sent and received by devices, and how devices represent and store their physical state in the cloud.

AWS IoT Core helps you build IoT applications by providing a managed message broker that supports the use of the MQTT protocol to publish and subscribe IoT messages between devices.

The **AWS IoT Device Registry** helps you manage and operate your things. A thing is a representation of a specific device or logical entity in the cloud. Things can also have custom defined static attributes that help you identify, categorize, and search for your assets once deployed.

With the **AWS IoT Device Shadow Service**, you can create a data store that contains the current state of a particular device. The Device Shadow Service maintains a virtual representation of each of your devices you connect to AWS IoT as a distinct device shadow. Each device's shadow is uniquely identified by the name of the corresponding thing.

With **Amazon API Gateway**, your IoT applications can make HTTP requests to control your IoT devices. IoT applications require API interfaces for internal systems, such as dashboards for remote technicians, and external systems, such as a home consumer mobile application. With Amazon API Gateway, you can create common API interfaces without provisioning and managing the underlying infrastructure.

Ingestion Layer

A key business driver for IoT is the ability to aggregate all the disparate data streams created by your devices and transmit the data to your IoT application in a secure and reliable manner. The ingestion layer plays a key role in collecting device data while decoupling the flow of data with the communication between devices.

With **AWS IoT rules engine**, you can build IoT applications such that your devices can interact with AWS services. AWS IoT rules are analyzed and actions are performed based on the MQTT topic stream a message is received on.

Amazon Kinesis is a managed service for streaming data, enabling you to get timely insights and react quickly to new information from IoT devices. Amazon Kinesis integrates directly with the AWS IoT rules engine, creating a seamless way of bridging from a lightweight device protocol of a device using MQTT with your internal IoT applications that use other protocols.

Similar to Kinesis, **Amazon Simple Queue Service (Amazon SQS)** should be used in your IoT application to decouple the communication layer from your application layer. Amazon SQS enables an event-driven, scalable ingestion queue when your application needs to process IoT applications once where message order is not required.

Analytics Layer

One of the benefits of implementing IoT solutions is the ability to gain deep insights and data about what's happening in the local/edge environment. A primary way of realizing contextual insights is by implementing solutions that can process and perform analytics on IoT data.

Storage Services

IoT workloads are often designed to generate large quantities of data. Ensure that this discrete data is transmitted, processed, and consumed securely, while being stored durably.

Amazon S3 is object-based storage engineered to store and retrieve any amount of data from anywhere on the internet. With Amazon S3, you can build IoT applications that store large amounts of data for a variety of purposes: regulatory, business evolution, metrics, longitudinal studies, analytics machine learning, and organizational enablement. Amazon S3 gives you a broad range of flexibility in the way you manage data for not just for cost optimization and latency, but also for access control and compliance.

Analytics and Machine Learning Services

After your IoT data reaches a central storage location, you can begin to unlock the full value of IoT by implementing analytics and machine learning on device behavior. With analytics systems, you can begin to operationalize improvements in your device firmware, as well as your edge and cloud logic, by making data-driven decisions based on your analysis. With analytics and machine learning, IoT systems can implement proactive strategies like predictive maintenance or anomaly detection to improve the efficiencies of the system.

AWS IoT Analytics makes it easy to run sophisticated analytics on volumes on IoT data. AWS IoT Analytics manages the underlying IoT data store, while you build different materialized views of your data using your own analytical queries or Jupyter notebooks.

Amazon Athena is an interactive query service that makes it easy to analyze data in Amazon S3 using standard SQL. Athena is serverless, so there is no infrastructure to manage, and customers pay only for the queries that they run.

Amazon SageMaker is a fully managed platform that enables you to quickly build, train, and deploy machine learning models in the cloud and the edge layer. With Amazon SageMaker, IoT architectures can develop a model of historical device telemetry in order to infer future behavior.

Application Layer

AWS IoT provides several ways to ease the way cloud native applications consume data generated by IoT devices. These connected capabilities include features from serverless computing, relational databases to create materialized views of your IoT data, and management applications to operate, inspect, secure, and manage your IoT operations.

Management Applications

The purpose of management applications is to create scalable ways to operate your devices once they are deployed in the field. Common operational tasks such as inspecting the connectivity state of a device, ensuring device credentials are configured correctly, and querying devices based on their current state must be in place before launch so that your system has the required visibility to troubleshoot applications.

AWS IoT Device Defender is a fully managed service that audits your device fleets, detects abnormal device behavior, alerts you to security issues, and helps you investigate and mitigate commonly encountered IoT security issues.

AWS IoT Device Management eases the organizing, monitoring, and managing of IoT devices at scale. At scale, customers are managing fleets of devices across multiple physical locations. AWS IoT Device Management enables you to group devices for easier management. You can also enable real-time search indexing against the current state of your devices through Device Management Fleet Indexing. Both Device Groups and Fleet Indexing can be used with Over the Air Updates (OTA) when determining which target devices must be updated.

User Applications

In addition to managed applications, other internal and external systems need different segments of your IoT data for building different applications. To support end-consumer views, business operational dashboards, and the other net-new applications you build over time, you will need several other technologies that can receive the required information from your connectivity and ingestion layer and format them to be used by other systems.

Database Services – NoSQL and SQL

While a data lake can function as a landing zone for all of your unformatted IoT generated data, to support all the formatted views on top of your IoT data, you need to complement your data lake with structured and semi structured data stores. For these purposes, you should leverage both NoSQL and SQL databases. These types of databases enable you to create different views of your IoT data for distinct end users of your application.

Amazon DynamoDB is a fast and flexible NoSQL database service for IoT data. With IoT applications, customers often require flexible data models with reliable performance and automatic scaling of throughput capacity.

With **Amazon Aurora** your IoT architecture can store structured data in a performant and cost-effective open source database. When your data needs to be accessible to other IoT applications for predefined SQL queries, relational databases provide you another mechanism for decoupling the device stream of the ingestion layer from your eventual business applications, which need to act on discrete segments of your data.

Compute Services

Frequently, IoT workloads require application code to be executed when the data is generated, ingested, or consumed/realized. Regardless of when compute code needs to be executed, serverless compute is a highly cost-effective choice. Serverless compute can be leveraged from the edge to the core and from core to applications and analytics.

AWS Lambda allows you to run code without provisioning or managing servers. Due to the scale of ingestion for IoT workloads, AWS Lambda is an ideal fit for running stateless, event-driven IoT applications on a managed platform.

General Design Principles

The Well-Architected Framework identifies the following set of design principles in order to facilitate good design in the cloud with IoT:

- **Decouple ingestion from processing**: In IoT applications, the ingestion layer must be a highly scalable platform that can handle a high rate of streaming device data. By decoupling the fast rate of ingestion from the processing portion of your application through the use of queues, buffers, and messaging services, your IoT application can make several decisions without impacting devices, such as the frequency it processes data or the type of data it is interested in.
- **Design for offline behavior**: Due to things like connectivity issues or misconfigured settings, devices may go offline for much more extended periods of time than anticipated. Design your embedded software to handle extended periods of offline connectivity and create metrics in the cloud to track devices that are not communicating on a regular timeframe.
- **Design lean data at the edge and enrich in the cloud**: Given the constrained nature of IoT devices, the initial device schema will be optimized for storage on the physical device and efficient transmissions from the device to your IoT application. For this reason, unformatted device data will often not be enriched with static application information that can be inferred from the cloud. For these reasons, as data is ingested into your application, you should prefer to first enrich the data with human readable attributes, deserialize, or expand any fields that the device serialized, and then format the data in a data store that is tuned to support your applications read requirements.
- **Handle personalization**: Devices that connect to the edge or cloud via Wi-Fi must receive the Access Point name and network password as one of the first steps performed when setting up the device. This data is usually infeasible to write to the device during manufacturing since it's sensitive and site-specific or from the cloud since the device isn't connected yet. These factors frequently make personalization data distinct from the device client certificate and private key, which are conceptually upstream, and from cloud-provided firmware and configuration updates, which are conceptually downstream. Supporting personalization can impact design and manufacturing, since it may mean that the device itself requires a user interface for direct data input, or the need to provide a smartphone application to connect the device to the local network.

- **Ensure that devices regularly send status checks**: Even if devices are regularly offline for extended periods of time, ensure that the device firmware contains application logic that sets a regular interval to send device status information to your IoT application. Devices must be active participants in ensuring that your application has the right level of visibility. Sending this regularly occurring IoT message ensures that your IoT application gets an updated view of the overall status of a device, and can create processes when a device does not communicate within its expected period of time.

Scenarios

This section addresses common scenarios related to IoT applications, with a focus on how each scenario impacts the architecture of your IoT workload. These examples are not exhaustive, but they encompass common patterns in IoT. We present a background on each scenario, general considerations for the design of the system, and a reference architecture of how the scenarios should be implemented.

Device Provisioning

In IoT, device provisioning is composed of several sequential steps. The most important aspect is that each device must be given a unique identity and then subsequently authenticated by your IoT application using that identity.

As such, the first step to provisioning a device is to install an identity. The decisions you make in device design and manufacturing determines if the device has a production-ready firmware image and/or unique client credential by the time it reaches the customer. Your decisions determine whether there are additional provisioning-time steps that must be performed before a production device identify can be installed.

Use X.509 client certificates in IoT for your applications — they tend to be more secure and easier to manage at scale than static passwords. In AWS IoT Core, the device is registered using its certificate along with a unique thing identifier. The registered device is then associated with an IoT policy. An IoT policy gives you the ability to create fine-grained permissions per device. Fine-grained permissions ensure that only one device has permissions to interact with its own MQTT topics and messages.

This registration process ensures that a device is recognized as an IoT asset and that the data it generates can be consumed through AWS IoT to the rest of the AWS ecosystem. To provision a device, you must enable automatic registration and associate

a provisioning template or an AWS Lambda function with the initial device provisioning event.

This registration mechanism relies on the device receiving a unique certificate during provisioning (which can happen either during or after manufacturing) which is used to authenticate to the IoT application, in this case AWS IoT. One advantage of this approach is that the device can be transferred to another entity, and be re-provisioned, allowing the registration process to be repeated with the new owner's AWS IoT account details.

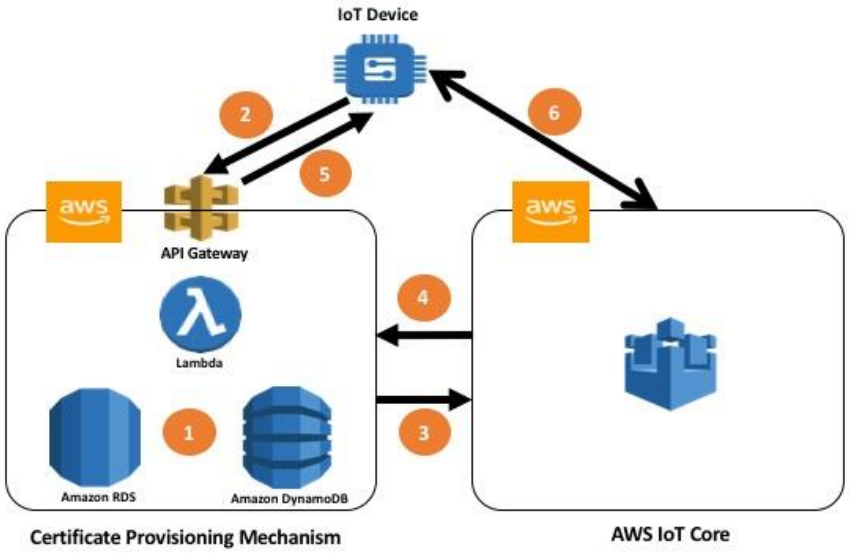

Figure 1: Registration Flow

1. Set up the manufacturing device identifier in a database.
2. The device connects to API Gateway and requests registration from the CPM. The request is validated.
3. Lambda requests X.509 certificates from your Private Certificate Authority (CA).
4. Your provisioning system registered your CA with AWS IoT Core.
5. API Gateway passes the device credentials to the device.
6. The device initiates the registration workflow with AWS IoT Core.

Device Telemetry

There are many uses cases (such as industrial IoT) where the value for IoT is in collecting telemetry on how a machine is performing. For example, this data can be used to enable predictive maintenance, preventing costly unforeseen equipment failures. Telemetry must be collected from the machine and uploaded to an IoT application. Another benefit of sending telemetry is the ability of your cloud applications to use this data for analysis and to interpret optimizations that can be made to your firmware over time.

Telemetry data is read-only that is collected and transmitted to the IoT application. Since telemetry data is passive, ensure the MQTT topic for telemetry messages does not overlap with any topics that relate to IoT commands. For example, a telemetry topic could be `data/device/sensortype` where any MQTT topic that begins with “data” is considered a telemetry topic.

From a logical perspective, we have defined several scenarios for capturing and interacting with device data telemetry.

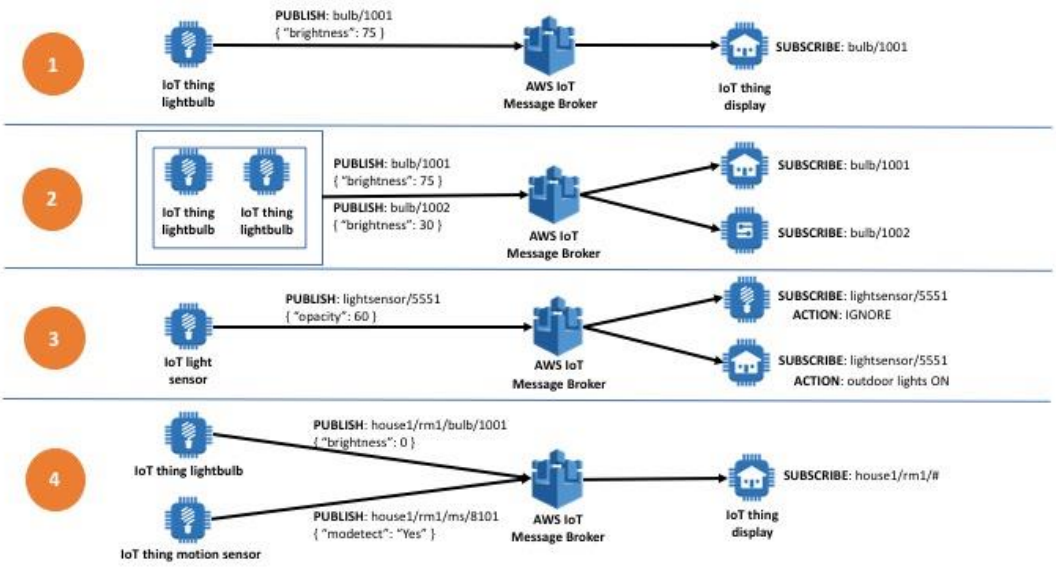

Figure 2: Options for capturing telemetry

1. One publishing topic and one subscriber. For example, a smart light bulb that publishes its brightness level to a single topic where only a single application can subscribe.
2. One publishing topic with variables and one subscriber. For example, a collection of smart bulbs publishing their brightness on similar but unique topics. Each subscriber can listen to a unique publish message.
3. Single publishing topic and multiple subscribers. In this case, a light sensor that publishes its values to a topic that all the light bulbs in a house subscribe to.
4. Multiple publishing topics and a single subscriber. For example, a collection of light bulbs with motion sensors. The smart home system subscribes to all of the light bulb topics, inclusive of motion sensors, and creates a composite view of brightness and motion sensor data.

Device Commands

When you are building an IoT application, you need the ability to interact with your device through commands remotely. An example in the industrial vertical is to use remote commands to request specific data from a piece of equipment. An example usage in the smart home vertical is to use remote commands to schedule an alarm system remotely.

With AWS IoT Core, you can implement commands using MQTT topics, or the AWS IoT Device Shadow, to send commands to a device and receive an acknowledgment when a device has executed the command. Use the Device Shadow over MQTT topics for implementing commands. The Device Shadow has several benefits over using standard MQTT topics, such as a clientToken, to track the origin of a request, version numbers for managing conflict resolution, and the ability to store commands in the cloud in the event that a device is offline and unable to receive the command when it is issued. The device's shadow is commonly used in cases where a command needs to be persisted in the cloud even if the device is currently not online. When the device is back online, the device requests the latest shadow information and executes the command.

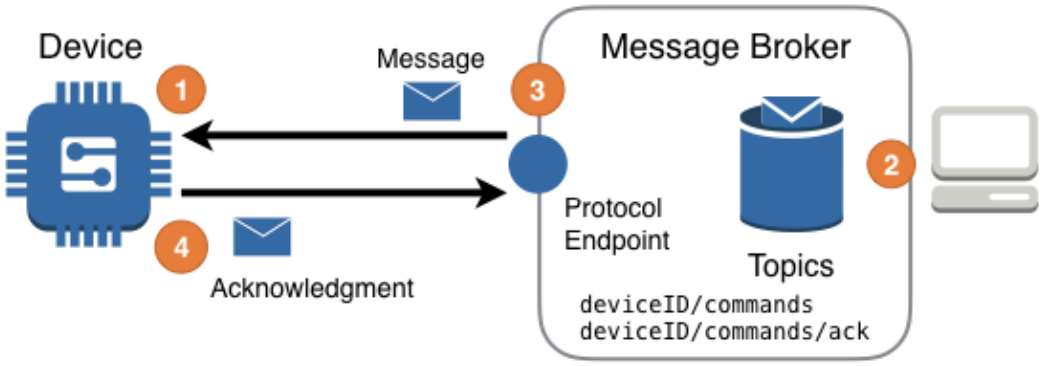

Figure 3: Using a message broker to send commands to a device.

AWS IoT Device Shadow Service

IoT solutions that use the Device Shadow service in AWS IoT Core manage command requests in a reliable, scalable, and straightforward fashion. The Device Shadow service follows a prescriptive approach to both the management of device-related state and how the state changes are communicated. This approach describes how the Device Shadows service uses a JSON document to store a device's current state, desired future state, and the difference between current and desired states.

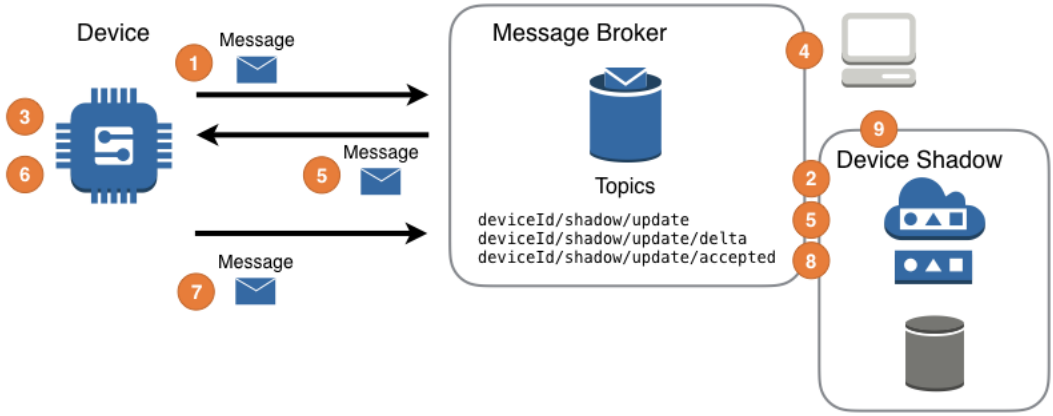

Figure 4: Using Device Shadow with devices.

1. A device reports initial device state by publishing that state as a message to the update topic `deviceID/shadow/update`.
2. The Device Shadow reads the message from the topic and records the device state in a persistent data store.

3. A device subscribes to the delta messaging topic `deviceId/shadow/update/delta` upon which device-related state change messages will arrive.
4. A component of the solution publishes a desired state message to the topic `deviceID/shadow/update` and the Device Shadow tracking this device records the desired device state in a persistent data store.
5. The Device Shadow publishes a delta message to the topic `deviceId/shadow/update/delta`, and the Message Broker sends the message to the device.
6. A device receives the delta message and performs the desired state changes.
7. A device publishes an acknowledgment message reflecting the new state to the update topic `deviceID/shadow/update` and the Device Shadow tracking this device records the new state in a persistent data store.
8. The Device Shadow publishes a message to the `deviceId/shadow/update/accepted` topic.
9. A component of the solution can now request the updated state from the Device Shadow.

Firmware Updates

All IoT solutions must allow device firmware updates. Supporting firmware upgrades without human intervention is critical for security, scalability, and delivering new capabilities.

AWS IoT Device Management provides a secure and easy way for you to manage IoT deployments including executing and tracking the status of firmware updates. AWS IoT Device Management uses the MQTT protocol with AWS IoT message broker and AWS IoT Jobs to send firmware update commands to devices, as well as to receive the status of those firmware updates over time.

An IoT solution must implement firmware updates using AWS IoT Jobs shown in the following diagram to deliver this functionality.

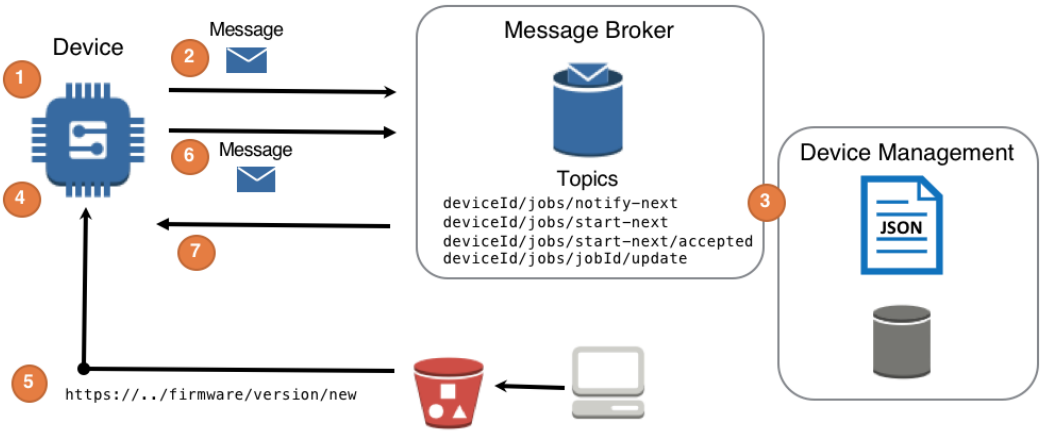

Figure 5: Updating firmware on devices.

1. A device subscribes to the IoT job notification topic `deviceId/jobs/notify-next` upon which IoT job notification messages will arrive.
2. A device publishes a message to `deviceId/jobs/start-next` to start the next job and get the next job, its job document, and other details including any state saved in statusDetails.
3. The AWS IoT Jobs service retrieves the next job document for the specific device and sends this document on the subscribed topic `deviceId/jobs/start-next/accepted`.
4. A device performs the actions specified by the job document using the `deviceId/jobs/jobId/update` MQTT topic to report on the progress of the job.
5. During the upgrade process, a device downloads firmware using a presigned URL for Amazon S3. Use code-signing to sign the firmware when uploading to Amazon S3. By code-signing your firmware the end-device can verify the authenticity of the firmware before installing. Amazon FreeRTOS devices can download the firmware image directly over MQTT to eliminate the need for a separate HTTPS connection.
6. The device publishes an update status message to the job topic `deviceId/jobs/jobId/update` reporting success or failure.
7. Because this job's execution status has changed to final state, the next IoT job available for execution (if any) will change.

The Pillars of the Well-Architected Framework

This section describes each of the pillars and includes definitions, best practices, questions, considerations, and essential AWS services that are relevant when architecting solutions for AWS IoT.

Operational Excellence Pillar

The **Operational Excellence** pillar includes operational practices and procedures used to manage production workloads. Operational excellence comprises how planned changes are executed, as well as responses to unexpected operational events. Change execution and responses should be automated. All processes and procedures of operational excellence must be documented, tested, and regularly reviewed.

Design Principles

In addition to the overall Well-Architected Framework operational excellence design principles, there are five design principles for operational excellence for IoT in the cloud:

- **Plan for device provisioning**: Design your device provisioning process to create your initial device identity in a secure location. Implement a public key infrastructure (PKI) that is responsible for distributing unique certificates to IoT devices. As described above, selection of crypto hardware with a pre-generated private key and certificate eliminates the operational cost of running a PKI. Otherwise, PKI can be done offline with a Hardware Security Module (HSM) during the manufacturing process, or during device bootstrapping. Use technologies that can manage the Certificate Authority (CA) and HSM in the cloud.

- **Implement device bootstrapping**: Devices that support personalization by a technician (in the industrial vertical) or user (in the consumer vertical) can also undergo provisioning. For example, a smartphone application that interacts with the device over Bluetooth LE and with the cloud over Wi-Fi. You must design the ability for devices to programmatically update their configuration information using a globally distributed bootstrap API. A bootstrapping design ensures that you can programmatically send the device new configuration settings through the cloud. These changes should include settings such as which IoT endpoint to communicate with, how frequently to send an overall status for the device, and any updated security settings such as server certificates. The process of bootstrapping goes beyond initial provisioning and plays a critical role in device operations by providing a programmatic way to update device configuration through the cloud.
- **Document device communication patterns**: In an IoT application, device behavior is documented by hand at the hardware level. In the cloud, an operations team must formulate how the behavior of a device will scale once deployed to a fleet of devices. A cloud engineer should review the device communication patterns and extrapolate the total expected inbound and outbound traffic of device data and determine the expected infrastructure necessary in the cloud to support the entire fleet of devices. During operational planning, these patterns should be measured using device and cloud-side metrics to ensure that expected usage patterns are met in the system.
- **Implement over the air (OTA) updates**: In order to benefit from long-term investments in hardware, you must be able to continuously update the firmware on the devices with new capabilities. In the cloud, you can apply a robust firmware update process that allows you to target specific devices for firmware updates, roll out changes over time, track success and failures of updates, and have the ability to roll back or put a stop to firmware changes based on KPIs.
- **Implement functional testing on physical assets**: IoT device hardware and firmware must undergo rigorous testing before being deployed in the field. Acceptance and functional testing are critical for your path to production. The goal of functional testing is to run your hardware components, embedded firmware, and device application software through rigorous testing scenarios, such as intermittent or reduced connectivity or failure of peripheral sensors, while profiling the performance of the hardware. The tests ensure that your IoT device will perform as expected when deployed.

Definition

There are three best practice areas for operational excellence in the cloud:

1. Preparation
2. Operation
3. Evolution

In addition to what is covered by the Well-Architected Framework concerning process, runbooks, and game days, there are specific areas you should review to drive operational excellence within IoT applications.

Best Practices

Preparation

For IoT applications, the need to procure, provision, test, and deploy hardware in various environments means that the preparation for operational excellence must be expanded to cover aspects of your deployment that will primarily run on physical devices and will not run in the cloud. Operational metrics must be defined to measure and improve business outcomes and then determine if devices should generate and send any of those metrics to your IoT application. You also must plan for operational excellence by creating a streamlined process of functional testing that allows you to simulate how devices may behave in their various environments.

It is essential that you ask how to ensure that your IoT workloads are resilient to failures, how devices can self-recover from issues without human intervention, and how your cloud-based IoT application will scale to meet the needs of an ever-increasing load of connected hardware.

When using an IoT platform, you have the opportunity to use additional components/tools for handling IoT operations. These tools include services that allow you to monitor and inspect device behavior, capture connectivity metrics, provision devices using unique identities, and perform long-term analysis on top of device data.

IOTOPS 1. What factors drive your operational priorities?

IOTOPS 2. How do you ensure that you are ready to support the operations of devices of your IoT workload?

IOTOPS 3. How are you ensuring that newly provisioned devices have the required operational prerequisites?

Logical security for IoT and data centers is similar in that both involve predominantly machine-to-machine authentication. However, they differ in that IoT devices are frequently deployed to environments that cannot be assumed to be physically secure. IoT applications also commonly require sensitive data to traverse the internet. Due to these considerations, it is vital for you to have an architecture that determines how devices will securely gain an identity, continuously prove their identity, be seeded with the appropriate level of metadata, be organized and categorized for monitoring, and enabled with the right set of permissions.

For successful and scalable IoT applications, the management processes should be automated, data-driven, and based on previous, current, and expected device behavior. IoT applications must support incremental rollout and rollback strategies. By having this as part of the operational efficiency plan, you will be equipped to launch a fault-tolerant, efficient IoT application.

In AWS IoT, you can use multiple features to provision your individual device identities signed by your CA to the cloud. This path involves provisioning devices with identities and then using just-in-time-provisioning (JITP), just-in-time-registration (JITR), or Bring Your Own Certificate (BYOC) to securely register your device certificates to the cloud. Using AWS services including Route 53, Amazon API Gateway, Lambda, and DynamoDB, will create a simple API interface to extend the provisioning process with device bootstrapping.

Operate

In IoT, operational health goes beyond the operational health of the cloud application and extends to the ability to measure, monitor, troubleshoot, and remediate devices that are part of your application, but are remotely deployed in locations that may be difficult or impossible to troubleshoot locally. This requirement of remote operations must be

considered at design and implementation time in order to ensure your ability to inspect, analyze, and act on metrics sent from these remote devices.

In IoT, you must establish the right baseline metrics of behavior for your devices, be able to aggregate and infer issues that are occurring across devices, and have a robust remediation plan that is not only executed in the cloud, but also part of your device firmware. You must implement a variety of device simulation canaries that continue to test common device interactions directly against your production system. Device canaries assist in narrowing down the potential areas to investigate when operational metrics are not met. Device canaries can be used to raise preemptive alarms when the canary metrics fall below your expected SLA.

In AWS, you can create an AWS IoT thing for each physical device in the device registry of AWS IoT Core. By creating a thing in the registry, you can associate metadata to devices, group devices, and configure security permissions for devices. An AWS IoT thing should be used to store static data in the thing registry while storing dynamic device data in the thing's associated device shadow. A device's shadow is a JSON document that is used to store and retrieve state information for a device.

Along with creating a virtual representation of your device in the device registry, as part of the operational process, you must create thing types that encapsulate similar static attributes that define your IoT devices. A thing type is analogous to the product classification for a device. The combination of thing, thing type, and device shadow can act as your first entry point for storing important metadata that will be used for IoT operations.

In AWS IoT, thing groups allow you to manage devices by category. Groups can also contain other groups — allowing you to build hierarchies. With organizational structure in your IoT application, you can quickly identify and act on related devices by device group. Leveraging the cloud allows you to automate the addition or removal of devices from groups based on your business logic and the lifecycle of your devices.

In IoT, your devices create telemetry or diagnostic messages that are not stored in the registry or the device's shadow. Instead these messages are delivered to AWS IoT using a number of MQTT topics. To make this data actionable, use the AWS IoT rules engine to route error messages to your automated remediation process and add diagnostic information to IoT messages. An example of how you would route a message that contained an error status code to a custom workflow is below. The rules engine inspects the status of a message and if it is an error, it starts the Step Function workflow to remediate the device based off the error message detail payload.

```
{
    "sql": "SELECT * FROM 'command/iot/response WHERE code =
'eror'",
    "ruleDisabled": false,
    "description": "Error Handling Workflow",
    "awsIotSqlVersion": "2016-03-23",
    "actions": [{
        "stepFunctions": {
            "executionNamePrefix": "errorExecution",
            "stateMachineName": "errorStateMachine",
            "roleArn":
"arn:aws:iam::123456789012:role/aws_iot_step_functions"
        }
    }]
}
```

To support operational insights to your cloud application, generate dashboards for all metrics collected from the device broker of AWS IoT Core. These metrics are available through CloudWatch Metrics. In addition, CloudWatch Logs contain information such as total successful messages inbound, messages outbound, connectivity success, and errors.

To augment your production device deployments, implement IoT simulations on Amazon Elastic Compute Cloud (Amazon EC2) as device canaries across several AWS Regions. These device canaries are responsible for mirroring several of your business use cases, such as simulating error conditions like long-running transactions, sending telemetry, and implementing control operations. The device simulation framework must output extensive metrics, including but not limited to successes, errors, latency, and device ordering and then transmit all the metrics to your operations system.

In addition to custom dashboards, AWS IoT provides fleet-level and device-level insights driven from the Thing Registry and Device Shadow service through search capabilities such as AWS IoT Fleet Indexing. The ability to search across your fleet eases the operational overhead of diagnosing IoT issues, whether they occur at the device-level or fleet-wide level.

Evolve

IOTOPS 4. How do you evolve your IoT application with minimum impact to downstream IoT devices?

IoT solutions frequently involve a combination of low-power devices, remote locations, low bandwidth, and intermittent network connectivity. Each of those factors poses communications challenges, including upgrading firmware. Therefore, it's important for you to incorporate and implement an IoT update process that minimizes the impact to downstream devices and operations. In addition to reducing downstream impact, devices must be resilient to common challenges that exist in local environments, such as intermittent network connectivity and power loss. Use a combination of grouping IoT devices for deployment and staggering firmware upgrades over a period of time. Monitor the behavior of devices as they are updated in the field, and proceed only after a percentage of devices have upgraded successfully.

Use AWS IoT Device Management for creating deployment groups of devices and delivering over the air updates (OTA) to specific device groups. During upgrades, continue to collect all of the CloudWatch Logs, telemetry, and IoT device job messages and combine that information with the KPIs used to measure overall application health and the performance of any long-running canaries.

Before and after firmware updates, perform a retrospective analysis of operations metrics with participants spanning the business to determine opportunities and methods for improvement. Services like AWS IoT Analytics and AWS IoT Device Defender are used to track anomalies in overall device behavior, and to measure deviations in performance that may indicate an issue in the updated firmware.

Key AWS Services

Several services can be used to drive operational excellence for your IoT application. The AWS Device Qualification Program helps you select hardware components that have been designed and tested for AWS IoT interoperability. Qualified hardware can get you to market faster and reduce operational friction. AWS IoT Core offers features used to manage the initial onboarding of a device. AWS IoT Device Management reduces the operational overhead of performing fleet-wide operations, such as device grouping and searching. In addition, Amazon CloudWatch is used for monitoring IoT metrics, collecting logs, generating alerts, and triggering responses. Other services and features that support the three areas of operational excellence are as follows:

- **Preparation**: **AWS IoT Core** supports provisioning and onboarding your devices in the field, including registering the device identity using just-in-time provisioning, just-in-time registration, or Bring Your Own Certificate. Devices can then be associated with their metadata and device state using the device registry and the Device Shadow.
- **Operations**: **AWS IoT thing groups and Fleet Indexing** allow you to quickly develop an organizational structure for your devices and search across the current metadata of your devices to perform recurring device operations. Amazon CloudWatch allows you to monitor the operational health of your devices and your application.
- **Responses**: **AWS IoT Jobs** enables you to proactively push updates to one or more devices such as firmware updates or device configuration. AWS IoT rules engine allows you to inspect IoT messages as they are received by AWS IoT Core and immediately respond to the data, at the most granular level. AWS IoT Analytics and AWS IoT Device Defender enable you to proactively trigger notifications or remediation based on real-time analysis with AWS IoT Analytics, and real-time security and data thresholds with Device Defender.

Security Pillar

The **Security** pillar includes the ability to protect information, systems, and assets while delivering business value.

Design Principles

In addition to the overall Well-Architected Framework security design principles, there are specific design principles for IoT security:

- **Manage device security lifecycle holistically**: Data security starts at the design phase, and ends with the retirement and destruction of the hardware and data. It is important to take an end-to-end approach to the security lifecycle of your IoT solution in order to maintain your competitive advantage and retain customer trust.
- **Ensure least privilege permissions**: Devices should all have fine-grained access permissions that limit which topics a device can use for communication. By restricting access, one compromised device will have fewer opportunities to impact any other devices.

- **Secure device credentials at rest**: Devices should securely store credential information at rest using mechanisms such as a dedicated crypto element or secure flash.
- **Implement device identity lifecycle management**: Devices maintain a device identity from creation through end of life. A well-designed identity system will keep track of a device's identity, track the validity of the identity, and proactively extend or revoke IoT permissions over time.
- **Take a holistic view of data security**: IoT deployments involving a large number of remotely deployed devices present a significant attack surface for data theft and privacy loss. Use a model such as the Open Trusted Technology Provider Standard to systemically review your supply chain and solution design for risk and then apply appropriate mitigations.

Definition

There are five best practice areas for security in the cloud:

1. Identity and access management (IAM)
2. Detective controls
3. Infrastructure protection
4. Data protection
5. Incident response

Infrastructure and data protection encompass the IoT device hardware, as well as the end to end solution. IoT implementations require expanding your security model to ensure that devices implement hardware security best practices and your IoT applications follow security best practices for factors such as adequately scoped device permissions and detective controls.

The security pillar focuses on protecting information and systems. Key topics include confidentiality and integrity of data, identifying and managing who can do what with privilege management, protecting systems, and establishing controls to detect security events.

Best Practices

Identity and Access Management (IAM)

IoT devices are often a target because they are provisioned with a trusted identity, may store or have access to strategic customer or business data (such as the firmware itself), may be remotely accessible over the internet, and may be vulnerable to direct physical tampering. To provide protection against unauthorized access, you need to always begin with implementing security at the device level. From a hardware perspective, there are several mechanisms that you can implement to reduce the attack surface of tampering with sensitive information on the device such as:

- Hardware crypto modules
- Software-supported solutions including secure flash
- Physical function modules that cannot be cloned
- Up-to-date cryptographic libraries and standards including PKCS #11 and TLS 1.2

To secure device hardware, you implement solutions such that private keys and sensitive identity are unique to, and only stored on the device in a secure hardware location. Implement hardware or software-based modules that securely store and manage access to the private keys used to communicate with AWS IoT. In addition to hardware security, IoT devices must be given a valid identity, which will be used for authentication and authorization in your IoT application.

During the lifetime of a device, you will need to be able to manage certificate renewal and revocation. To handle any changes to certificate information on a device, you must first have the ability to update a device in the field. The ability to perform firmware updates on hardware is a vital underpinning to a well-architected IoT application. Through OTA updates, securely rotate device certificates before expiry including certificate authorities.

IOTSEC 1. How do you securely store device certificates and private keys for devices?

IOTSEC 2. How do you associate AWS IoT identities with your devices?

For example, with AWS IoT, you first provision X.509 certificate and then separately create the IoT permissions for connecting to IoT, publishing and subscribing to messages, and receiving updates. This separation of identity and permissions provides flexibility in managing your device security. During the configuration of permissions, you can ensure that any device has the right level of identity as well as the right level of access control by creating an IoT policy that restricts access to MQTT actions for each device.

Ensure that each device has its own unique X.509 certificate in AWS IoT and that devices should never share certificates (one certificate for one device rule). In addition to using a single certificate per device, when using AWS IoT, each device must have its own unique thing in the IoT registry, and the thing name is used as the basis for the MQTT ClientID for MQTT connect.

By creating this association where a single certificate is paired with its own thing in AWS IoT Core, you can ensure that one compromised certificate cannot inadvertently assume an identity of another device. It also alleviates troubleshooting and remediation when the MQTT ClientID and the thing name match since you can correlate any ClientID log message to the thing that is associated with that particular piece of communication.

To support device identity updates, use AWS IoT Jobs, which is a managed platform for distributing OTA communication and binaries to your devices. AWS IoT Jobs is used to define a set of remote operations that are sent to and executed on one or more devices connected to AWS IoT. AWS IoT Jobs by default integrate several best practices, including mutual authentication and authorization, device tracking of update progress, and fleet-wide wide metrics for a given update.

Enable AWS IoT Device Defender audits to track device configuration, device policies, and checking for expiring certificates in an automated fashion. For example, Device Defender can run audits on a scheduled basis and trigger a notification for expiring certificates. With the combination of receiving notifications of any revoked certificates or pending expiry certificates, you can automatically schedule an OTA that can proactively rotate the certificate.

IOTSEC 3. How do you authenticate and authorize user access to your IoT application?

Although many applications focus on the thing aspect of IoT, in almost all verticals of IoT, there is also a human component that needs the ability to communicate to and receive notifications from devices. For example, consumer IoT generally requires users to onboard their devices by associating them with an online account. Industrial IoT typically entails the ability to analyze hardware telemetry in near real time. In either case, it's essential to determine how your application will identify, authenticate, and authorize users that require the ability to interact with particular devices.

Controlling user access to your IoT assets begins with identity. Your IoT application must have in place a store (typically a database) that keeps track of a user's identity and also how a user authenticates using that identity. The identity store may include additional user attributes that can be used at authorization time (for example, user group membership).

IoT device telemetry data is an example of a securable asset. By treating it as such, you can control the access each user has and audit individual user interactions.

When using AWS to authenticate and authorize IoT application users, you have several options to implement your identity store and how that store maintains user attributes. For your own applications, use Amazon Cognito for your identity store. Amazon Cognito provides a standard mechanism to express identity, and to authenticate users, in a way that can be directly consumed by your app and other AWS services in order to make authorization decisions. When using AWS IoT, you can choose from several identity and authorization services including Amazon Cognito Identity Pools, AWS IoT policies, and AWS IoT custom authorizer.

For implementing the decoupled view of telemetry for your users, use a mobile service such as AWS AppSync or Amazon API Gateway. With both of these AWS services, you can create an abstraction layer that decouples your IoT data stream from your user’s device data notification stream. By creating a separate view of your data for your external users in an intermediary datastore, for example. Amazon DynamoDB or Amazon ElasticSearch Service, you can use AWS AppSync to receive user-specific notifications based only on the allowed data in your intermediary store. In addition to using external data stores with AWS AppSync, you can define user specific notification topics that can be used to push specific views of your IoT data to your external users.

If an external user needs to communicate directly to an AWS IoT endpoint, ensure that the user identity is either an authorized Amazon Cognito Federated Identity that is associated to an authorized Amazon Cognito role and a fine-grained IoT policy, or uses AWS IoT custom authorizer, where the authorization is managed by your own authorization service. With either approach, associate a fine-grained policy to each user

that limits what the user can connect as, publish to, subscribe from, and receive messages from concerning MQTT communication.

IOTSEC 4. How do you ensure that least privilege is applied to principals that communicates to your IoT application?

After registering a device and establishing its identity, it may be necessary to seed additional device information needed for monitoring, metrics, telemetry, or command and control. Each resource requires its own assignment of access control rules. By reducing the actions that a device or user can take against your application, and ensuring that each resource is secured separately, you limit the impact that can occur if any single identity or resource is used inadvertently.

In AWS IoT, create fine-grained permissions by using a consistent set of naming conventions in the IoT registry. The first convention is to use the same unique identifier for a device as the MQTT ClientID and AWS IoT thing name. By using the same unique identifier in all these locations, you can easily create an initial set of IoT permissions that can apply to all of your devices using AWS IoT Thing Policy variables. The second naming convention is to embed the unique identifier of the device into the device certificate. Continuing with this approach, store the unique identifier as the CommonName in the subject name of the certificate in order to use Certificate Policy Variables to bind IoT permissions to each unique device credential.

By using policy variables, you can create a few IoT policies that can be applied to all of your device certificates while maintaining least privilege. For example, the IoT policy below would restrict any device to connect only using the unique identifier of the device (which is stored in the common name) as its MQTT ClientID and only if the certificate is attached to the device. This policy also restricts a device to only publish on its individual shadow:

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": ["iot:Connect"],
        "Resource": ["arn:aws:iot:us-east-
1:123456789012:client/${iot:Certificate.Subject.CommonName}"],
        "Condition":{
            "Bool":{
```

```
                "iot:Connection.Thing.IsAttached":["true"]
            }
        }
    },
    {
        "Effect":"Allow",
        "Action":["iot:Publish"],
        "Resource":["arn:aws:iot:us-east-
1:123456789012:topic/$aws/things/${iot:Connection.Thing.ThingName}/
shadow/update"]
    }
    ]
}
```

Attach your device identity (certificate or Amazon Cognito Federated Identity) to the thing in the AWS IoT registry using AttachThingPrincipal.

Although these scenarios apply to a single device communicating with its own set of topics and device shadows, there are scenarios where a single device needs to act upon the state or topics of other devices. For example, you may be operating an edge appliance in an industrial setting, creating a home gateway to manage coordinating automation in the home, or allowing a user to gain access to a different set of devices based on their specific role. For these use cases, leverage a known entity, such as a group identifier or the identity of the edge gateway as the prefix for all of the devices that communicate to the gateway. By making all of the endpoint devices use the same prefix, you can make use of wildcards, "*", in your IoT policies. This approach balances MQTT topic security with manageability.

```
{
    "Version": "2012-10-17",
    "Statement": [
    {
        "Effect":"Allow",
        "Action":["iot:Publish"],
        "Resource":["arn:aws:iot:us-east-
1:123456789012:topic/$aws/things/edgegateway123-*/shadow/update"]
    }
    ]
}
```

In the preceding example, the IoT operator would associate the policy with the edge gateway with the identifier, edgegateway123. The permissions in this policy would then allow the edge appliance to publish to other Device Shadows that are managed by the edge gateway. This is accomplished by enforcing that any connected devices to the gateway all have a thing name that is prefixed with the identifier of the gateway. For example, a downstream motion sensor would have the identifier, edgegateway123-motionsensor1, and therefore can now be managed by the edge gateway while still restricting permissions.

Detective Controls

Due to the scale of data, metrics, and logs in IoT applications, aggregating and monitoring is an essential part of a well-architected IoT application. Unauthorized users will probe for bugs in your IoT application and will look to take advantage of individual devices to gain further access into other devices, applications, and cloud resources. In order to operate an entire IoT solution, you will need to manage detective controls not only for an individual device but also for the entire fleet of devices in your application. You will need to enable several levels of logging, monitoring, and alerting to detect issues at the device level as well as the fleet-wide level.

In a well-architected IoT application, each layer of the IoT application generates metrics and logs. At a minimum, your architecture should have metrics and logs related to the physical device, the connectivity behavior of your device, message input and output rates per device, provisioning activities, authorization attempts, and internal routing events of device data from one application to another.

IOTSEC 5: How are you analyzing application logs and metrics across cloud and devices?

In AWS IoT, you can implement detective controls using AWS IoT Device Defender, CloudWatch Logs, and CloudWatch Metrics. AWS IoT Device Defender processes logs and metrics related to device behavior and connectivity behaviors of your devices. AWS IoT Device Defender also lets you continuously monitor security metrics from devices and AWS IoT Core for deviations from what you have defined as appropriate behavior for each device.

Set a default set of thresholds when device behavior or connectivity behavior deviates from normal activity.

Augment Device Defender metrics with the Amazon CloudWatch Metrics, Amazon CloudWatch Logs generated by AWS IoT Core, and Amazon GuardDuty. These

service-level logs provide important insight into activity about not only activities related to AWS IoT Platform services and AWS IoT Core protocol usage, but also provide insight into the downstream applications running in AWS that are critical components of your end to end IoT application. All Amazon CloudWatch Logs should be analyzed centrally to correlate log information across all sources.

IOTSEC 6: How are you managing invalid identities in your IoT application?

Security identities are the focal point of device trust and authorization to your IoT application. It's vital to be able to manage invalid identities, such as certificates, centrally. An invalid certificate can be revoked, expired, or made inactive. As part of a well-architected application, you must have a process for capturing all invalid certificates, and an automated response based on the state of the certificate trigger. In addition to the ability of capturing the events of an invalid certificate, your devices should also have a secondary means of establishing secure communications to your IoT platform. By enabling a bootstrapping pattern as described previously, where two forms of identity are used for a device, you can create a reliable fallback mechanism for detecting invalid certificates and providing a mechanism for a device or an administrator to establish trusted, secure communication for remediation.

A well-architected IoT application establishes a certificate revocation list (CRL) that tracks all revoked device certificates or certificate authorities (CAs). Use your own trusted CA for on-boarding devices and synchronize your CRL on a regular basis to your IoT application. Your IoT application must reject connections from identities that are no longer valid.

With AWS, you do not need to manage your entire PKI on-premises. Use AWS Certificate Manager (ACM) Private Certificate Authority to host your CA in the cloud. Or, you can work with an APN Partner to add preconfigured secure elements to your IoT device hardware specification. ACM has the capability to export revoked certificates to a file in an S3 bucket. That same file can be used to programmatically revoke certificates against AWS IoT Core.

Another state for certificates is to be near their expiry date but still valid. The client certificate must be valid for at least the service lifetime of the device. It’s up to your IoT application to keep track of devices near their expiry date and perform an OTA process to update their certificate to a new one with a later expiry, along with logging information about why the certificate rotation was required for audit purposes.

Enable AWS IoT Device Defender audits related to the certificate and CA expiry. Device Defender produces an audit log of certificates that are set to expire within 30 days. Use this list to programmatically update devices before certificates are no longer valid. You may also choose to build your own expiry store to manage certificate expiry dates and programmatically query, identify, and trigger an OTA for device certificate replacement or renewal.

Infrastructure Protection

Design time is the ideal phase for considering security requirements for infrastructure protection across the entire lifecycle of your device and solution. By considering your devices as an extension of your infrastructure, you can take into account how the entire device lifecycle impacts your design for infrastructure protection. From a cost standpoint, changes made in the design phase are less expensive than changes made later. From an effectiveness standpoint, data loss mitigations implemented at design time are likely to be more comprehensive than mitigations retrofitted. Therefore, planning the device and solution security lifecycle at design time reduces business risk and provides an opportunity to perform upfront infrastructure security analysis before launch.

One way to approach the device security lifecycle is through supply chain analysis. For example, even a modestly sized IoT device manufacturer or solution integrator has a large number of suppliers that make up its supply chain, whether directly or indirectly. To maximize solution lifetime and reliability, ensure that you are receiving authentic components.

Software is also part of the supply chain. The production firmware image for a device includes drivers and libraries from many sources including silicon partners, open-source aggregation sites such as GitHub and SourceForge, previous first-party products, and new code developed by internal engineering.

To understand the downstream maintenance and support for first-party firmware and software, you must analyze each software provider in the supply chain to determine if it offers support and how it delivers patches. This analysis is especially important for connected devices: software bugs are inevitable, and represent a risk to your customers because a vulnerable device can be exploited remotely. Your IoT device manufacturer or solution engineering team must learn about and patch bugs in a timely manner to reduce these risks.

IOTSEC 7. How are you vetting your suppliers, contract manufacturers, and other outsource relationships?

IOTSEC 8. How are you planning the security lifecycle of your IoT devices?

IOTSEC 9. How are you ensuring timely notification of security bugs in your third-party firmware and software components?

Although there is no cloud infrastructure to manage when using AWS IoT services, there are integration points where AWS IoT Core interacts on your behalf with other AWS services. For example, the AWS IoT rules engine consists of rules that are analyzed that can trigger downstream actions to other AWS services based on the MQTT topic stream. Since AWS IoT communicates to your other AWS resources, you must ensure that the right service role permissions are configured for your application.

Data Protection

Before architecting an IoT application, data classification, governance, and controls must be designed and documented to reflect how the data can be persisted in the cloud, and how data should be encrypted, whether on a device or between the devices and the cloud. Unlike traditional cloud applications, data sensitivity and governance extend to the IoT devices that are deployed in remote locations outside of your network boundary. These techniques are important because they support protecting personally identifiable data transmitted from devices and complying with regulatory obligations.

During the design process, determine how hardware, firmware, and data are handled at device end-of-life. Store long-term historical data in the cloud. Store a portion of current sensor readings locally on a device, namely only the data required to perform local operations. By only storing the minimum data required on the device, the risk of unintended access is limited.

In addition to reducing data storage locally, there are other mitigations that must be implemented at the end of life of a device. First, the device should offer a reset option which can reset the hardware and firmware to a default factory version. Second, your IoT application can run periodic scans for the last logon time of every device. Devices that have been offline for too long a period of time, or are associated with inactive customer accounts, can be revoked. Third, encrypt sensitive data that must be persisted on the device using a key that is unique to that particular device.

IOTSEC 10: How do you classify, manage, and protect your data in transit and at rest?

All traffic to and from AWS IoT must be encrypted using Transport Layer Security (TLS). In AWS IoT, security mechanisms protect data as it moves between AWS IoT and other devices or AWS services. In addition to AWS IoT, you must implement device-level security to protect not only the device's private key but also the data collected and processed on the device.

For embedded development, AWS has several services that abstract components of the application layer while incorporating AWS security best practices by default on the edge. For microcontrollers, AWS recommends using Amazon FreeRTOS. Amazon FreeRTOS extends the FreeRTOS kernel with libraries for Bluetooth LE, TCP/IP, and other protocols. In addition, Amazon FreeRTOS contains a set of security APIs that allow you to create embedded applications that securely communicate with AWS IoT.

For Linux-based edge gateways, AWS IoT Greengrass can be used to extend cloud functionality to the edge of your network. AWS IoT Greengrass implements several security features, including mutual X.509 certificate-based authentication with connected devices, AWS IAM policies and roles to manage communication permissions between AWS IoT Greengrass and cloud applications, and subscriptions, which are used to determine how and if data can be routed between connected devices and Greengrass core.

Incident Response

Being prepared for incident response in IoT requires planning on how you will deal with two types of incidents in your IoT workload. The first incident is an attack against an individual IoT device in an attempt to disrupt the performance or impact the device's behavior. The second incident is a larger scale IoT event, such as network outages and DDoS attack. In both scenarios, the architecture of your IoT application plays a large role in determining how quickly you will be able to diagnose incidents, correlate the data across the incident, and then subsequently apply runbooks to the affected devices in an automated, reliable fashion.

For IoT applications, follow the following best practices for incident responses:

- IoT devices are organized in different groups based on device attributes such as location and hardware version.
- IoT devices are searchable by dynamic attributes, such as connectivity status, firmware version, application status, and device health.

- OTA updates can be staged for devices and deployed over a period of time. Deployment rollouts are monitored and can be automatically aborted if devices fail to maintain the appropriate KPIs.
- Any update process is resilient to errors, and devices can recover and roll back from a failed software update.
- Detailed logging, metrics, and device telemetry are available that contain contextual information about how a device is currently performing and has performed over a period of time.
- Fleet-wide metrics monitor the overall health of your fleet and alert when operational KPIs are not met for a period of time.
- Any individual device that deviates from expected behavior can be quarantined, inspected, and analyzed for potential compromise of the firmware and applications.

IOTSEC 11: How do you prepare to respond to an incident that impacts a single device or a fleet of devices?

Implement a strategy in which your InfoSec team can quickly identify the devices that need remediation. Ensure that the InfoSec team has runbooks that consider firmware versioning and patching for device updates. Create automated processes that proactively apply security patches to vulnerable devices as they come online.

At a minimum, your security team should be able to detect an incident on a specific device based on the device logs and current device behavior. After an incident is identified, the next phase is to quarantine the application. To implement this with AWS IoT services, you can use AWS IoT Things Groups with more restrictive IoT policies along with enabling custom group logging for those devices. This allows you to only enable features that relate to troubleshooting, as well as gather more data to understand root cause and remediation. Lastly, after an incident has been resolved, you must be able to deploy a firmware update to the device to return it to a known state.

Key AWS Services

The essential AWS security services in IoT are the AWS IoT registry, AWS IoT Device Defender, AWS Identity and Access Management (IAM), and Amazon Cognito. In combination, these services allow you to securely control access to IoT devices, AWS

services, and resources for your users. The following services and features support the five areas of security:

Design: The AWS Device Qualification Program provides IoT endpoint and edge hardware that has been pre-tested for interoperability with AWS IoT. Tests include mutual authentication and OTA support for remote patching.

AWS Identity and Access Management (IAM): Device credentials (X.509 certificates, IAM, Amazon Cognito identity pools and Amazon Cognito user pools, or custom authorization tokens) enable you to securely control device and external user access to AWS resources. AWS IoT policies add the ability to implement fine grained access to IoT devices. ACM Private CA provides a cloud-based approach to creating and managing device certificates. Use AWS IoT thing groups to manage IoT permissions at the group level instead of individually.

Detective controls: AWS IoT Device Defender records device communication and cloud side metrics from AWS IoT Core. AWS IoT Device Defender can automate security responses by sending notifications through Amazon Simple Notification Service (Amazon SNS) to internal systems or administrators. AWS CloudTrail logs administrative actions of your IoT application. Amazon CloudWatch is a monitoring service with integration with AWS IoT Core and can trigger CloudWatch Events to automate security responses. CloudWatch captures detailed logs related to connectivity and security events between IoT edge components and cloud services.

Infrastructure protection: AWS IoT Core is a cloud service that lets connected devices easily and securely interact with cloud applications and other devices. The AWS IoT rules engine in AWS IoT Core uses IAM permissions to communicate with other downstream AWS services.

Data protection: AWS IoT includes encryption capabilities for devices over TLS to protect your data in transit. AWS IoT integrates directly with services, such as Amazon S3 and Amazon DynamoDB, which support encryption at rest. In addition, AWS Key Management Service (AWS KMS) supports the ability for you to create and control keys used for encryption. On devices, you can use AWS edge offerings such as Amazon FreeRTOS, AWS IoT Greengrass, or the AWS IoT Embedded C SDK to support secure communication.

Incident response: AWS IoT Device Defender allows you to create security profiles that can be used to detect deviations from normal device behavior and trigger automated responses including AWS Lambda. AWS IoT Device Management should be

used to group devices that need remediation and then using AWS IoT Jobs to deploy fixes to devices.

Resources

Refer to the following resources to learn more about our best practices for security:

Documentation and Blogs

- IoT Security Identity
- AWS IoT Device Defender
- IoT Authentication Model
- MQTT on port 443
- Detect Anomalies with Device Defender

Whitepapers

- MQTT Topic Design

Reliability Pillar

The reliability pillar focuses on the ability to prevent and quickly recover from failures to meet business and customer demand. Key topics include foundational elements around setup, cross-project requirements, recovery planning, and change management.

Design Principles

In addition to the overall Well-Architected Framework design principles, there are three design principles for reliability for IoT in the cloud:

- **Simulate device behavior at production scale**: Create a production-scale test environment that closely mirrors your production deployment. Leverage a multi-step simulation plan that allows you to test your applications with a more significant load before your go-live date. During development, ramp up your simulation tests over a period of time starting with 10% of overall traffic for a single test and incrementing over time (that is, 25%, 50%, then 100% of day one device traffic). During simulation tests, monitor performance and review logs to ensure that the entire solution behaves as expected.
- **Buffer message delivery from the IoT rules engine with streams or queues**: Leverage-managed services enable high throughput telemetry. By injecting a queuing layer behind high throughput topics, IoT applications can manage failures, aggregate messaging, and scale other downstream services.
- **Design for failure and resiliency**: It's essential to plan for resiliency on the device itself. Depending on your use case, resiliency may entail robust retry logic for intermittent connectivity, ability to roll back firmware updates, ability to fail over to a different networking protocol or communicate locally for critical message delivery, running redundant sensors or edge gateways to be resilient to hardware failures, and the ability to perform a factory reset.

Definition

There are three best practice areas for reliability in the cloud:

1. Foundations
2. Change management
3. Failure management

To achieve reliability, a system must have a well-planned foundation and monitoring in place, with mechanisms for handling changes in demand, requirements, or potentially defending an unauthorized denial of service attack. The system should be designed to detect the failure and automatically heal itself.

Best Practices

Foundations

IoT devices must continue to operate in some capacity in the face of network or cloud errors. Design device firmware to handle intermittent connectivity or loss in connectivity in a way that is sensitive to memory and power constraints. IoT cloud applications must

also be designed to handle remote devices that frequently transition between being online and offline to maintain data coherency and scale horizontally over time. Monitor overall IoT utilization and create a mechanism to automatically increase capacity to ensure that your application can manage peak IoT traffic.

To prevent devices from creating unnecessary peak traffic, device firmware must be implemented that prevents the entire fleet of devices from attempting the same operations at the same time. For example, if an IoT application is composed of alarm systems and all the alarm systems send an activation event at 9am local time, the IoT application is inundated with an immediate spike from your entire fleet. Instead, you should incorporate a randomization factor into those scheduled activities, such as timed events and exponential back off, to permit the IoT devices to more evenly distribute their peak traffic within a window of time.

The following questions focus on the considerations for reliability.

IOTREL 1. How do you handle AWS service limits for peaks in your IoT application?

AWS IoT provides a set of soft and hard limits for different dimensions of usage. AWS IoT outlines all of the data plane limits on the IoT limits page. Data plane operations (for example, MQTT Connect, MQTT Publish, and MQTT Subscribe) are the primary driver of your device connectivity. Therefore, it's important to review the IoT limits and ensure that your application adheres to any soft limits related to the data plane, while not exceeding any hard limits that are imposed by the data plane.

The most important part of your IoT scaling approach is to ensure that you architect around any hard limits because exceeding limits that are not adjustable results in application errors, such as throttling and client errors. Hard limits are related to throughput on a single IoT connection. If you find your application exceeds a hard limit, we recommend redesigning your application to avoid those scenarios. This can be done in several ways, such as restructuring your MQTT topics, or implementing cloud-side logic to aggregate or filter messages before delivering the messages to the interested devices.

Soft limits in AWS IoT traditionally correlate to account-level limits that are independent of a single device. For any account-level limits, you should calculate your IoT usage for a single device and then multiply that usage by the number of devices to determine the base IoT limits that your application will require for your initial product launch. AWS

recommends that you have a ramp-up period where your limit increases align closely to your current production peak usage with an additional buffer. To ensure that the IoT application is not under provisioned:

- Consult published AWS IoT CloudWatch metrics for all of the limits.
- Monitor CloudWatch metrics in AWS IoT Core.
- Alert on CloudWatch throttle metrics, which would signal if you need a limit increase.
- Set alarms for all thresholds in IoT, including MQTT connect, publish, subscribe, receive, and rule engine actions.
- Ensure that you request a limit increase in a timely fashion, before reaching 100% capacity.

In addition to data plane limits, the AWS IoT service has a control plane for administrative APIs. The control plane manages the process of creating and storing IoT policies and principals, creating the thing in the registry, and associating IoT principals including certificates and Amazon Cognito federated identities. Because bootstrapping and device registration is critical to the overall process, it's important to plan control plane operations and limits. Control plane API calls are based on throughput measured in requests per second. Control plane calls are normally in the order of magnitude of tens of requests per second. It’s important for you to work backward from peak expected registration usage to determine if any limit increases for control plane operations are needed. Plan for sustained ramp-up periods for onboarding devices so that the IoT limit increases align with regular day-to-day data plane usage.

To protect against a burst in control plane requests, your architecture should limit the access to these APIs to only authorized users or internal applications. Implement back-off and retry logic, and queue inbound requests to control data rates to these APIs.

IOTREL 2. What is your strategy for managing ingestion and processing throughput of IoT data to other applications?

Although IoT applications have communication that is only routed between other devices, there will be messages that are processed and stored in your application. In these cases, the rest of your IoT application must be prepared to respond to incoming data. All internal services that are dependent upon that data need a way to seamlessly scale the ingestion and processing of the data. In a well-architected IoT application,

internal systems are decoupled from the connectivity layer of the IoT platform through the ingestion layer. The ingestion layer is composed of queues and streams that enable durable short-term storage while allowing compute resources to process data independent of the rate of ingestion.

In order to optimize throughput, use AWS IoT rules to route inbound device data to services such as Amazon Kinesis Data Streams, Amazon Kinesis Data Firehose, or Amazon Simple Queue Service before performing any compute operations. Ensure that all the intermediate streaming points are provisioned to handle peak capacity. This approach creates the queueing layer necessary for upstream applications to process data resiliently.

IOTREL 3. How do you handle device reliability when communicating with the cloud?

IoT solution reliability must also encompass the device itself. Devices are deployed in remote locations and deal with intermittent connectivity, or loss in connectivity, due to a variety of external factors that are out of your IoT application's control. For example, if an ISP is interrupted for several hours, how will the device behave and respond to these long periods of potential network outage? Implement a minimum set of embedded operations on the device to make it more resilient to the nuances of managing connectivity and communication to AWS IoT Core.

Your IoT device must be able to operate without internet connectivity. You must implement robust operations in your firmware provide the following capabilities:

- Store important messages durably offline and, once reconnected, send those messages to AWS IoT Core.
- Implement exponential retry and back-off logic when connection attempts fail.
- If necessary, have a separate failover network channel to deliver critical messages to AWS IoT. This can include failing over from Wi-Fi to standby cellular network, or failing over to a wireless personal area network protocol (such as Bluetooth LE) to send messages to a connected device or gateway.
- Have a method to set the current time using an NTP client or low-drift real-time clock. A device should wait until it has synchronized its time before attempting a connection with AWS IoT Core. If this isn't possible, the system provides a way for a user to set the device's time so that subsequent connections can succeed.

- Send error codes and overall diagnostics messages to AWS IoT Core.

Change Management

IOTREL 4. How do you roll out and roll back changes to your IoT application?

It is important to implement the capability to revert to a previous version of your device firmware or your cloud application in the event of a failed rollout. If your application is well-architected, you will capture metrics from the device, as well as metrics generated by AWS IoT Core and AWS IoT Device Defender. You will also be alerted when your device canaries deviate from expected behavior after any cloud-side changes. Based on any deviations in your operational metrics, you need the ability to:

- Version all of the device firmware using Amazon S3.
- Version the manifest or execution steps for your device firmware.
- Implement a known-safe default firmware version for your devices to fall back to in the event of an error.
- Implement an update strategy using cryptographic code-signing, version checking, and multiple non-volatile storage partitions, to deploy software images and rollback.
- Version all IoT rules engine configurations in CloudFormation.
- Version all downstream AWS Cloud resources using CloudFormation.
- Implement a rollback strategy for reverting cloud side changes using CloudFormation and other infrastructure as code tools.

Treating your infrastructure as code on AWS allows you to automate monitoring and change management for your IoT application. Version all of the device firmware artifacts and ensure that updates can be verified, installed, or rolled back when necessary.

Failure Management

IOTREL 5. How does your IoT application withstand failure?

Because IoT is an event-driven workload, your application code must be resilient to handling known and unknown errors that can occur as events are permeated through your application. A well-architected IoT application has the ability to log and retry errors in data processing. An IoT application will archive all data in its raw format. By archiving

all data, valid and invalid, an architecture can more accurately restore data to a given point in time.

With the IoT rules engine, an application can enable an IoT error action. If a problem occurs when invoking an action, the rules engine will invoke the error action. This allows you to capture, monitor, alert, and eventually retry messages that could not be delivered to their primary IoT action. We recommend that an IoT error action is configured with a different AWS service from the primary action. Use durable storage for error actions such as Amazon SQS or Amazon Kinesis.

Beginning with the rules engine, your application logic should initially process messages from a queue and validate that the schema of that message is correct. Your application logic should catch and log any known errors and optionally move those messages to their own DLQ for further analysis. Have a catch-all IoT rule that uses Amazon Kinesis Data Firehose and AWS IoT Analytics channels to transfer all raw and unformatted messages into long-term storage in Amazon S3, AWS IoT Analytics data stores, and Amazon Redshift for data warehousing.

IOTREL 6. How do you verify different levels of hardware failure modes for your physical assets?

IoT implementations must allow for multiple types of failure at the device level. Failures can be due to hardware, software, connectivity, or unexpected adverse conditions. One way to plan for thing failure is to deploy devices in pairs, if possible, or to deploy dual sensors across a fleet of devices deployed over the same coverage area (meshing).

Regardless of the underlying cause for device failures, if the device can communicate to your cloud application, it should send diagnostic information about the hardware failure to AWS IoT Core using a diagnostics topic. If the device loses connectivity because of the hardware failure, use Fleet Indexing with connectivity status to track the change in connectivity status. If the device is offline for extended periods of time, trigger an alert that the device may require remediation.

Key AWS Services

Use Amazon CloudWatch to monitor runtime metrics and ensure reliability. Other services and features that support the three areas of reliability are as follows:

Foundations: AWS IoT Core enables you to scale your IoT application without having to manage the underlying infrastructure. You can scale AWS IoT Core by requesting account level limit increases.

Change management: AWS IoT Device Management enables you to update devices in the field while using Amazon S3 to version all firmware, software, and update manifests for devices. AWS CloudFormation lets you document your IoT infrastructure as code and provision cloud resources using a CloudFormation template.

Failure management: Amazon S3 allows you to durably archive telemetry from devices. The AWS IoT rules engine Error action enables you to fall back to other AWS services when a primary AWS service is returning errors.

Resources

Refer to the following resources to learn more about our best practices related to reliability:

Documentation and Blogs

- Using Device Time to Validate AWS IoT Server Certificates
- AWS IoT Core Limits
- IoT Error Action
- Fleet Indexing
- IoT Atlas

Performance Efficiency Pillar

The **Performance Efficiency** pillar focuses on using computing resources efficiently. Key topics include selecting the right resource types and sizes based on workload requirements, monitoring performance, and making informed decisions to maintain efficiency as business and technology needs evolve. The performance efficiency pillar focuses on the efficient use of computing resources to meet the requirements and the maintenance of that efficiency as demand changes and technologies evolve.

Design Principles

In addition to the overall Well-Architected Framework performance efficiency design principles, there are three design principles for performance efficiency for IoT in the cloud:

- **Use managed services**: AWS provides several managed services across databases, compute, and storage which can assist your architecture in increasing the overall reliability and performance.
- **Process data in batches**: Decouple the connectivity portion of IoT applications from the ingestion and processing portion in IoT. By decoupling the ingestion layer, your IoT application can handle data in aggregate and can scale more seamlessly by processing multiple IoT messages at once.
- **Use event driven architectures**: IoT systems publish events from devices and permeate those events to other subsystems in your IoT application. Design mechanisms that cater to event-driven architectures, such as leveraging queues, message handling, idempotency, dead-letter queues, and state machines.

Definition

There are four best practice areas for Performance Efficiency in the cloud:

1. Selection
2. Review
3. Monitoring
4. Tradeoffs

Use a data-driven approach when selecting a high-performance architecture. Gather data on all aspects of the architecture, from the high-level design to the selection and configuration of resource types. By reviewing your choices on a cyclical basis, you will ensure that you are taking advantage of the continually evolving AWS platform. Monitoring ensures that you are aware of any deviation from expected performance and allows you to act. Your architecture can make tradeoffs to improve performance, such as using compression or caching, or relaxing consistency requirements.

Best Practices

Selection

Well-Architected IoT solutions are made up of multiple systems and components such as devices, connectivity, databases, data processing, and analytics. In AWS, there are several IoT services, database offerings, and analytics solutions that enable you to quickly build solutions that are well-architected while allowing you to focus on business objectives. AWS recommends that you leverage a mix of managed AWS services that

best fit your workload. The following questions focus on these considerations for performance efficiency.

IOTPERF 1. How do you select the best performing IoT architecture?

When you select the implementation for your architecture, use a data-driven approach based on the long-term view of your operation. IoT applications align naturally to event driven architectures. Your architecture will combine services that integrate with event-driven patterns such as notifications, publishing and subscribing to data, stream processing, and event-driven compute. In the following sections we look at the five main IoT resource types that you should consider (devices, connectivity, databases, compute, and analytics).

Devices

The optimal embedded software for a particular system will vary based on the hardware footprint of the device. For example, network security protocols, while necessary for preserving data privacy and integrity, can have a relatively large RAM footprint. For intranet and internet connections, use TLS with a combination of a strong cipher suite and minimal footprint. AWS IoT supports Elliptic Curve Cryptography (ECC) for devices connecting to AWS IoT using TLS. A secure software and hardware platform on device should take precedence during the selection criteria for your devices. AWS also has a number of IoT partners that provide hardware solutions that can securely integrate to AWS IoT.

In addition to selecting the right hardware partner, you may choose to use a number of software components to run your application logic on the device, including Amazon FreeRTOS and AWS IoT Greengrass.

IOTPERF 2. How do you select your hardware and operating system for IoT devices?

IoT Connectivity

Before firmware is developed to communicate to the cloud, implement a secure, scalable connectivity platform to support the long-term growth of your devices over time. Based on the anticipated volume of devices, an IoT platform must be able to scale the communication workflows between devices and the cloud, whether that is simple ingestion of telemetry or command and response communication between devices.

You can build your IoT application using AWS services such as EC2, but you take on the undifferentiated heavy lifting for building unique value into your IoT offering. Therefore, AWS recommends that you use AWS IoT Core for your IoT platform.

AWS IoT Core supports HTTP, WebSockets, and MQTT, a lightweight communication protocol designed to tolerate intermittent connections, minimize the code footprint on devices, and reduce network bandwidth requirements.

IOTPERF 3. How do you select your primary IoT platform?

Databases

You will have multiple databases in your IoT application, each selected for attributes such as the write frequency of data to the database, the read frequency of data from the database, and how the data is structured and queried. There are other criteria to consider when selecting a database offering:

- Volume of data and retention period.
- Intrinsic data organization and structure.
- Users and applications consuming the data (either raw or processed) and their geographical location/dispersion.
- Advanced analytics needs, such as machine learning or real-time visualizations.
- Data synchronization across other teams, organizations, and business units.
- Security of the data at the row, table, and database levels.
- Interactions with other related data-driven events such as enterprise applications, drill-through dashboards, or systems of interaction.

AWS has several database offerings that support IoT solutions. For structured data, you should use Amazon Aurora, a highly scalable relational interface to organizational data. For semi structured data that requires low latency for queries and will be used by multiple consumers, use DynamoDB, a fully managed, multi-region, multi-master database that provides consistent single-digit millisecond latency, and offers built-in security, backup and restore, and in-memory caching.

For storing raw, unformatted event data, use AWS IoT Analytics. AWS IoT Analytics filters, transforms, and enriches IoT data before storing it in a time series data store for analysis. Use Amazon SageMaker to build, train, and deploy machine learning models,

based off of your IoT data, in the cloud and on the edge using AWS IoT Services such as Greengrass Machine Learning Inference. Consider storing your raw formatted time series data in a data warehouse solution such as Amazon Redshift. Unformatted data can be imported to Amazon Redshift via Amazon S3 and Amazon Kinesis Data Firehose. By archiving unformatted data in a scalable, managed data storage solution, you can begin to gain business insights, explore your data, and identify trends and patterns over time.

In addition to storing and leveraging the historical trends of your IoT data, you must have a system that stores the current state of the device and provides the ability to query against the current state of all of your devices. This supports internal analytics and customer facing views into your IoT data.

The AWS IoT Shadow service is an effective mechanism to store a virtual representation of your device in the cloud. AWS IoT device shadow is best suited for managing the current state of each device. In addition, for internal teams that need to query against the shadow for operational needs, leverage the managed capabilities of Fleet Indexing, which provides a searchable index incorporating your IoT registry and shadow metadata. If there is a need to provide index based searching or filtering capability to a large number of external users, such as for a consumer application, dynamically archive the shadow state using a combination of the IoT rules engine, Kinesis Data Firehose, and Amazon ElasticSearch Service to store your data in a format that allows fine grained query access for external users.

IOTPERF 4. How do you select the database for your IoT device state?

Compute

IoT applications lend themselves to a high flow of ingestion that requires continuous processing over the stream of messages. Therefore, an architecture must choose compute services that support the steady enrichment of stream processing and the execution of business applications during and prior to data storage.

The most common compute service used in IoT is AWS Lambda, which allows actions to be invoked when telemetry data reaches AWS IoT Core or AWS IoT Greengrass. AWS Lambda can be used at different points throughout IoT. The location where you elect to trigger your business logic with AWS Lambda is influenced by the time that you want to process a specific data event.

Amazon EC2 instances can also be used for a variety of IoT use cases. They can be used for managed relational databases systems and for a variety of applications, such as web, reporting, or to host existing on-premises solutions.

IOTPERF 5. How do you select your compute solutions for processing AWS IoT events?

Analytics

The primary business case for implementing IoT solutions is to respond more quickly to how devices are performing and being used in the field. By acting directly on incoming telemetry, businesses can make more informed decisions about which new products or features to prioritize, or how to more efficiently operate workflows within their organization. Analytics services must be selected in such a way that gives you varying views on your data based on the type of analysis you are performing. AWS provides several services that align with different analytics workflows including time-series analytics, real-time metrics, and archival and data lake use cases.

With IoT data, your application can generate time-series analytics on top of the steaming data messages. You can calculate metrics over time windows and then stream values to other AWS services.

In addition, IoT applications that use AWS IoT Analytics can implement a managed AWS Data Pipeline consisting of data transformation, enrichment, and filtering before storing data in a time series data store. Additionally, with AWS IoT Analytics, visualizations and analytics can be performed natively using QuickSight and Jupyter Notebooks.

Review

IOTPERF 6. How do you evolve your architecture based on the historical analysis of your IoT application?

When building complex IoT solutions, you can devote a large percentage of time on efforts that do not directly impact your business outcome. For example, managing IoT protocols, securing device identities, and transferring telemetry between devices and the cloud. Although these aspects of IoT are important, they do not directly lead to differentiating value. The pace of innovation in IoT can also be a challenge.

AWS regularly releases new features and services based on the common challenges of IoT. Perform a regular review of your data to see if new AWS IoT services can solve a current IoT gap in your architecture, or if they can replace components of your architecture that are not core business differentiators. Leverage services built to aggregate your IoT data, store your data, and then later visualize your data to implement historical analysis. You can leverage a combination of sending timestamp information from your IoT device with leveraging services like AWS IoT Analytics and time-based indexing to archive your data with associated timestamp information. Data in AWS IoT Analytics can be stored in your own Amazon S3 bucket along with additional IT or OT operational and efficiency logs from your devices. By combining this archival state of data in IoT with visualization tools, you can make data driven decisions about how new AWS services can provide additional value and measure how the services improve efficiency across your fleet.

Monitoring

IOTPERF 7. How are you running end to end simulation tests of your IoT application?

IoT applications can be simulated using production devices set up as test devices (with a specific test MQTT namespace), or by using simulated devices. All incoming data captured using the IoT rules engine is processed using the same workflows that are used for production.

The frequency of end-to-end simulations must be driven by your specific release cycle or device adoption. You should test failure pathways (code that is only executed during a failure) to ensure that the solution is resilient to errors. You should also continuously run device canaries against your production and pre-production accounts. The device canaries act as key indicators of the system performance during simulation tests. Outputs of the tests should be documented and remediation plans should be drafted. User acceptance tests should be performed.

IOTPERF 9. How are you using performance monitoring in your IoT implementation?

There are several key types of performance monitoring related to IoT deployments including device, cloud performance, and storage/analytics. Create appropriate

performance metrics using data collected from logs with telemetry and command data. Start with basic performance tracking and build on the metrics as your business core competencies expand.

Leverage CloudWatch Logs metric filters to transform your IoT application standard output into custom metrics through regex (regular expressions) pattern matching. Create CloudWatch alarms based on your application's custom metrics to gain quick insight into your IoT application's behavior.

Set up fine-grained logs to track specific thing groups. During IoT solution development, enable DEBUG logging for a clear understanding of the progress of events about each IoT message as it passes from your devices through the message broker and the rules engine. In production, change the logging to ERROR and WARN.

In addition to cloud instrumentation, you must run instrumentation on devices prior to deployment to ensure that devices make the most efficient use of their local resources, and that firmware code does not lead to unwanted scenarios like memory leaks. Deploy code that is highly optimized for constrained devices and monitor the health of your devices using device diagnostic messages published to AWS IoT from your embedded application.

Tradeoffs

IoT solutions drive rich analytics capabilities across vast areas of crucial enterprise functions, such as operations, customer care, finance, sales, and marketing. At the same time, they can be used as efficient egress points for edge gateways. Careful consideration must be given to architecting highly efficient IoT implementations where data and analytics are pushed to the cloud by devices, and where machine learning algorithms are pulled down on the device gateways from the cloud.

Individual devices will be constrained by the throughput supported over a given network. The frequency with which data is exchanged must be balanced with the transport layer and the ability of the device to optionally store, aggregate, and then send data to the cloud. Send data from devices to the cloud at timing intervals that align to the time required by backend applications to process and take action on the data. For example, if you need to see data at a one-second increment, your device must send data at a more frequent time interval than one second. Conversely, if your application only reads data at an hourly rate, you can make a trade-off in performance by aggregating data points at the edge and sending the data every half hour.

IOTPERF 9. How are you ensuring that data from your IoT devices is ready to be consumed by business and operational systems?

IOTPERF 10. How frequently is data transmitted from devices to your IoT application?

The speed with which enterprise applications, business, and operations need to gain visibility into IoT telemetry data determines the most efficient point to process IoT data. In network constrained environments where the hardware is not limited, use edge solutions such as AWS IoT Greengrass to operate and process data offline from the cloud. In cases where both the network and hardware are constrained, look for opportunities to compress message payloads by using binary formatting and grouping similar messages together into a single request.

For visualizations, Amazon Kinesis Data Analytics enables you to quickly author SQL code that continuously reads, processes, and stores data in near-real-time. Using standard SQL queries on the streaming data allows you to construct applications that transform and provide insights into your data. With Kinesis Data Analytics, you can expose IoT data for streaming analytics.

Key AWS Services

The key AWS service for performance efficiency is Amazon CloudWatch, which integrates with several IoT services including AWS IoT Core, AWS IoT Device Defender, AWS IoT Device Management, AWS Lambda, and DynamoDB. Amazon CloudWatch provides visibility into your application’s overall performance and operational health. The following services also support performance efficiency:

Selection

Devices: AWS hardware partners provide production ready IoT devices that can be used as part of you IoT application. Amazon FreeRTOS is an operating system with software libraries for microcontrollers. AWS IoT Greengrass allows you to run local compute, messaging, data caching, sync, and ML at the edge.

Connectivity: AWS IoT Core is a managed IoT platform that supports MQTT, a lightweight publish and subscribe protocol for device communication.

Database: Amazon DynamoDB is a fully managed NoSQL datastore that supports single digit millisecond latency requests to support quick retrieval of different views of your IoT data.

Compute: AWS Lambda is an event driven compute service that lets you run application code without provisioning servers. Lambda integrates natively with IoT events triggered from AWS IoT Core or upstream services such as Amazon Kinesis and Amazon SQS.

Analytics: AWS IoT Analytics is a managed service that operationalizes device level analytics while providing a time series data store for your IoT telemetry.

Review: The AWS IoT Blog section on the AWS website is a resource for learning about what is newly launched as part of AWS IoT.

Monitoring: Amazon CloudWatch Metrics and Amazon CloudWatch Logs provide metrics, logs, filters, alarms, and notifications that you can integrate with your existing monitoring solution. These metrics can be augmented with device telemetry to monitor your application.

Tradeoff: AWS IoT Greengrass and Amazon Kinesis are services that allow you to aggregate and batch data at different locations of your IoT application, providing you more efficient compute performance.

Resources

Refer to the following resources to learn more about our best practices related to performance efficiency:

Documentation and Blogs

- AWS Lambda Getting Started
- DynamoDB Getting Started
- AWS IoT Analytics User Guide
- Amazon FreeRTOS Getting Started
- AWS IoT Greengrass Getting Started
- AWS IoT Blog

Cost Optimization Pillar

The **Cost Optimization** pillar includes the continual process of refinement and improvement of a system over its entire lifecycle. From the initial design of your first proof of concept to the ongoing operation of production workloads, adopting the

practices in this paper will enable you to build and operate cost-aware systems that achieve business outcomes and minimize costs, allowing your business to maximize its return on investment.

Design Principles

In addition to the overall Well-Architected Framework cost optimization design principles, there is one design principle for cost optimization for IoT in the cloud:

- **Manage manufacturing cost tradeoffs**: Business partnering criteria, hardware component selection, firmware complexity, and distribution requirements all play a role in manufacturing cost. Minimizing that cost helps determine whether a product can be brought to market successfully over multiple product generations. However, taking shortcuts in the selection of your components and manufacturer can increase downstream costs. For example, partnering with a reputable manufacturer helps minimize downstream hardware failure and customer support cost. Selecting a dedicated crypto component can increase bill of material (BOM) cost, but reduce downstream manufacturing and provisioning complexity, since the part may already come with an onboard private key and certificate.

Definition

There are four best practice areas for Cost Optimization in the cloud:

1. Cost-effective resources
2. Matching supply and demand
3. Expenditure awareness
4. Optimizing over time

There are tradeoffs to consider. For example, do you want to optimize for speed to market or cost? In some cases, it's best to optimize for speed—going to market quickly, shipping new features, or meeting a deadline—rather than investing in upfront cost optimization. Design decisions are sometimes guided by haste as opposed to empirical data, as the temptation always exists to overcompensate rather than spend time benchmarking for a cost-optimal deployment. This leads to over-provisioned and under-optimized deployments. The following sections provide techniques and strategic guidance for your deployment’s initial and ongoing cost optimization.

Best Practices

Cost-Effective Resources

Given the scale of devices and data that can be generated by an IoT application, using the appropriate AWS services for your system is key to cost savings. In addition to the overall cost for your IoT solution, IoT architects often look at connectivity through the lens of BOM costs. For BOM calculations, you must predict and monitor what the long-term costs will be for managing the connectivity to your IoT application throughout the lifetime of that device. Leveraging AWS services will help you calculate initial BOM costs, make use of cost-effective services that are event driven, and update your architecture to continue to lower your overall lifetime cost for connectivity.

The most straightforward way to increase the cost-effectiveness of your resources is to group IoT events into batches and process data collectively. By processing events in groups, you are able to lower the overall compute time for each individual message. Aggregation can help you save on compute resources and enable solutions when data is compressed and archived before being persisted. This strategy decreases the overall storage footprint without losing data or compromising the query ability of the data.

COST 1. How do you select an approach for batch, enriched, and aggregate data delivered from your IoT platform to other services?

AWS IoT is best suited for streaming data for either immediate consumption or historical analyses. There are several ways to batch data from AWS IoT Core to other AWS services and the differentiating factor is driven by batching raw data (as is) or enriching the data and then batching it. Enriching, transforming, and filtering IoT telemetry data during (or immediately after) ingestion is best performed by creating an AWS IoT rule that sends the data to Kinesis Data Streams, Kinesis Data Firehose, AWS IoT Analytics, or Amazon SQS. These services allow you to process multiple data events at once.

When dealing with raw device data from this batch pipeline, you can use AWS IoT Analytics and Amazon Kinesis Data Firehose to transfer data to S3 buckets and Amazon Redshift. To lower storage costs in Amazon S3, an application can leverage lifecycle policies that archive data to lower cost storage, such as Amazon S3 Glacier.

Matching Supply and Demand

Optimally matching supply to demand delivers the lowest cost for a system. However, given the susceptibility of IoT workloads to data bursts, solutions must be dynamically

scalable and consider peak capacity when provisioning resources. With the event driven flow of data, you can choose to automatically provision your AWS resources to match your peak capacity and then scale up and down during known low periods of traffic.

The following questions focus on the considerations for cost optimization:

COST 2. How do you match the supply of resources with device demand?

Serverless technologies, such as AWS Lambda and API Gateway, help you create a more scalable and resilient architecture, and you only pay for when your application utilizes those services. AWS IoT Core, AWS IoT Device Management, AWS IoT Device Defender, AWS IoT Greengrass, and AWS IoT Analytics are also managed services that are pay per usage and do not charge you for idle compute capacity. The benefit of managed services is that AWS manages the automatic provisioning of your resources. If you utilize managed services, you are responsible for monitoring and setting alerts for limit increases for AWS services.

When architecting to match supply against demand, proactively plan your expected usage over time, and the limits that you are most likely to exceed. Factor those limit increases into your future planning.

Optimizing Over Time

Evaluating new AWS features allows you to optimize cost by analyzing how your devices are performing and make changes to how your devices communicate with your IoT.

To optimize the cost of your solution through changes to device firmware, you should review the pricing components of AWS services, such as AWS IoT, determine where you are below billing metering thresholds for a given service, and then weigh the trade-offs between cost and performance.

COST 3. How do you optimize payload size between devices and your IoT platform?

IoT applications must balance the networking throughput that can be realized by end devices with the most efficient way that data should be processed by your IoT

application. We recommend that IoT deployments initially optimize data transfer based on the device constraints. Begin by sending discrete data events from the device to the cloud, making minimal use of batching multiple events in a single message. Later, if necessary, you can use serialization frameworks to compress the messages prior to sending it to your IoT platform.

From a cost perspective, the MQTT payload size is a critical cost optimization element for AWS IoT Core. An IoT message is billed in 5-KB increments, up to 128 KB. For this reason, each MQTT payload size should be as close to possible to any 5 KB. For example, a payload that is currently sized at 6 KB is not as cost efficient as a payload that is 10 KB because the overall costs of publishing that message is identical despite one message being larger than the other.

In order to take advantage of the payload size, look for opportunities to either compress data or aggregate data into messages:

- You should shorten values while keeping them legible. If 5 digits of precision are sufficient then you should not use 12 digits in the payload.
- If you do not require IoT rules engine payload inspection, you can use serialization frameworks to compress payloads to smaller sizes.
- You can send data less frequently and aggregate messages together within the billable increments. For example, sending a single 2-KB message every second can be achieved at a lower IoT message cost by sending two 2-KB messages every other second.

This approach has tradeoffs that should be considered before implementation. Adding complexity or delay in your devices may unexpectedly increase processing costs. A cost optimization exercise for IoT payloads should only happen after your solution has been in production and you can use a data-driven approach to determine the cost impact of changing the way data is sent to AWS IoT Core.

COST 4. How do you optimize the costs of storing the current state of your IoT device?

Well-Architected IoT applications have a virtual representation of the device in the cloud. This virtual representation is composed of a managed data store or specialized IoT application data store. In both cases, your end devices must be programmed in a way that efficiently transmits device state changes to your IoT application. For example,

your device should only send its full device state if your firmware logic dictates that the full device state may be out of sync and would be best reconciled by sending all current settings. As individual state changes occur, the device should optimize the frequency it transmits those changes to the cloud.

In AWS IoT, device shadow and registry operations are metered in 1 KB increments and billing is per million access/modify operations. The shadow stores the desired or actual state of each device and the registry is used to name and manage devices.

Cost optimization processes for device shadows and registry focus on managing how many operations are performed and the size of each operation. If your operation is cost sensitive to shadow and registry operations, you should look for ways to optimize shadow operations. For example, for the shadow you could aggregate several reported fields together into one shadow message update instead of sending each reported change independently. Grouping shadow updates together reduces the overall cost of the shadow by consolidating updates to the service.

Key AWS Services

The key AWS feature supporting cost optimization is cost allocation tags, which help you to understand the costs of a system. The following services and features are important in the three areas of cost optimization:

- **Cost-effective resources**: Amazon Kinesis, AWS IoT Analytics, and Amazon S3 are AWS services that enable you to process multiple IoT messages in a single request in order to improve the cost effectiveness of compute resources.
- **Matching supply and demand**: AWS IoT Core is a managed IoT platform for managing connectivity, device security to the cloud, messaging routing, and device state.
- **Optimizing over time**: The AWS IoT Blog section on the AWS website is a resource for learning about what is newly launched as part of AWS IoT.

Resources

Refer to the following resources to learn more about AWS best practices for cost optimization.

Documentation and Blogs

- AWS IoT Blogs

Conclusion

The AWS Well-Architected Framework provides architectural best practices across the pillars for designing and operating reliable, secure, efficient, and cost-effective systems in the cloud for IoT applications. The framework provides a set of questions that you can use to review an existing or proposed IoT architecture, and also a set of AWS best practices for each pillar. Using the framework in your architecture helps you produce stable and efficient systems, which allows you to focus on your functional requirements.

Contributors

The following individuals and organizations contributed to this document:

- Olawale Oladehin, Solutions Architect Specialist, IoT
- Dan Griffin, Software Development Engineer, IoT
- Catalin Vieru, Solutions Architect Specialist, IoT
- Brett Francis, Product Solutions Architect, IoT
- Craig Williams, Partner Solutions Architect, IoT
- Philip Fitzsimons, Sr. Manager Well-Architected, Amazon Web Services

Document Revisions

Date	Description
December 2019	Updated to include additional guidance on IoT SDK usage, bootstrapping, device lifecycle management, and IoT
November 2018	First publication