AWS IoT Lens

AWS Well-Architected Framework

November 2018

Contents

Abstract

This paper describes the **IoT Lens** for the AWS Well-Architected Framework. The document covers commonly encountered IoT use cases and identifies key solution elements to ensure that your workloads are architected according to established best practices.

Introduction

The AWS Well-Architected Framework helps you understand the pros and cons of decisions you make while building systems on AWS. By using the Framework, you will learn architectural best practices for designing and operating reliable, secure, efficient, and cost-effective systems in the cloud. It 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) on the AWS Cloud. To implement a well-architected IoT application, you need to 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 covering AWS Cloud best practices, this document also articulates the impact, considerations, and recommendations for connecting physical assets to the Internet.

For brevity, we have only covered details from the Well-Architected Framework that are specific to your IoT workloads. You should still consider best practices and questions that have not been included in this document when designing your architecture. We recommend that you read the AWS Well-Architected Framework whitepaper.

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 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 comprised of many devices (or things) that securely connect and interact with complementary 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 six distinct logical layers you should consider when building an IoT workload:

- Edge layer
- Provisioning layer
- Communications layer
- Ingestion layer
- Analytics layer
- Application layer

Edge Layer

The edge layer of your IoT workloads 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.

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 Greengrass is a software component that 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 Private Key Infrastructure (PKI) used to create unique identities of your devices, the process by which firmware is first installed on devices, and the application workflow that provides configuration data to the device. The provisioning layer also is involved with the 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 need to install X.509 certificates 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). X.509 certificates are an ideal identity mechanism for constrained devices with limited memory and processing capabilities.

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. With the Device Shadow service, you can maintain 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, IoT customers can facilitate creating 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 and aggregating important sensor information from devices 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. You will want to ensure 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

Once your IoT data has reached a central storage location, you can begin to unlock the value of IoT by implementing analytics and machine learning on device behavior. With analytics systems, you can begin to operationalize improvements in your physical hardware 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 can 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 or down to the edge layer. With Amazon SageMaker, IoT architectures can develop a model of historical device telemetry in order to infer future behavior.

Application Layer

One of the key value propositions of using AWS IoT is provided by the ease with which data generated by IoT devices can be consumed by other relevant cloud native capabilities. 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 connectivity state of a device, ensuring device credentials are configured correctly, and querying devices based on their current state must be in place prior to 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. 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 in conjunction with Over the Air Updates (OTA) in determining which target devices need to be updated.

User Applications

In addition to managed applications, other internal and external systems will need different segments of your IoT data for building different applications. To support end-consumer views, business operational dashboards, and other net-new applications you will 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 will 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 lets you 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 a set of general design principles to facilitate good design in the cloud with IoT:

- **Decouple ingestion from processing** In IoT applications, the ingestion layer needs to 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.
- **Ensure devices can regularly send status checks** Even if devices are regularly offline for extended periods of time, the device firmware should contain application logic that sets a regular interval to send device status information to your IoT application. Devices need to be active

participants in ensuring your application has the right level of visibility, so by sending this regularly occurring IoT message, your IoT application can get an updated view of the overall status of a device, but also can create processes when a device does not communicate within its expected period of time.

Scenarios

In this section, we will cover some common scenarios related to IoT applications and how each impacts the architecture of your IoT workload. These scenarios do not include all IoT scenarios but encompass common patterns in IoT. We will present a background on each scenario, general considerations for the design of the system, and a reference architecture of how these scenarios should be implemented.

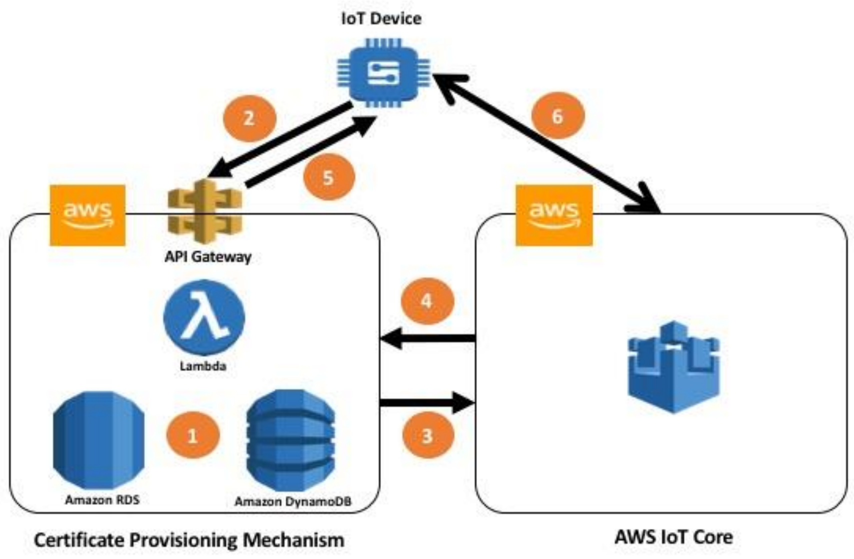
Device Provisioning

In IoT, device provisioning is comprised of several sequential steps. The most important aspect is that each device needs to 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. In IoT, it is common for applications to use device certificates such as X.509 security certificates. By default, AWS IoT Core supports X.509 certificates as device identities. 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 can ensure that one device only 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 certificate provisioning mechanism relies on the fact that during manufacturing the device will receive an initial device certificate, which will be 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 the registration process can be repeated with the new owner's AWS IoT account details.



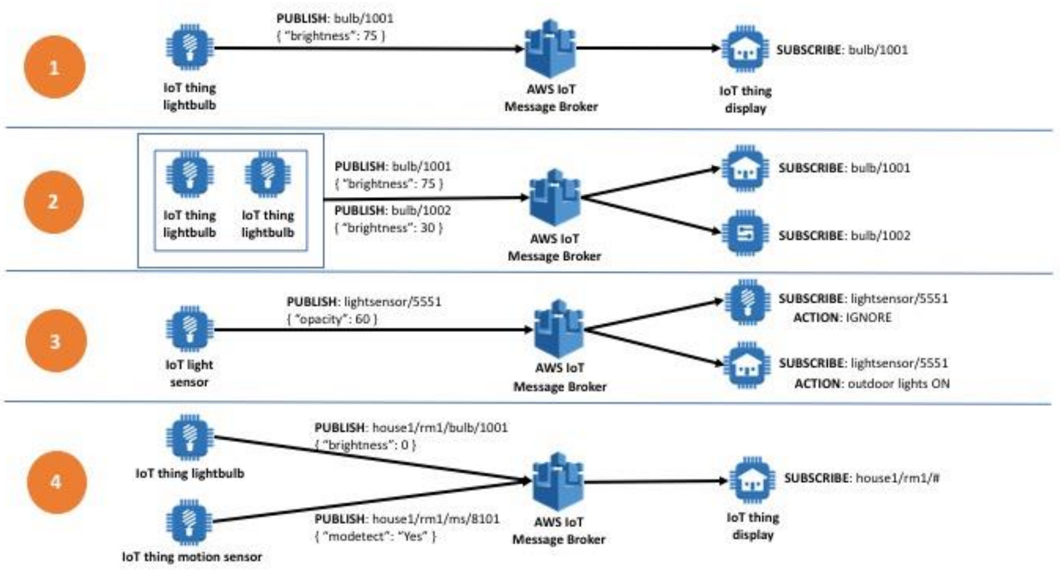
1. Set up the manufacturing device identifier in a relational database or NoSQL table
2. The device connects to API Gateway and requests registration from the CPM. The request is validated.
3. Lambda requests X.509 certificates from your Private Certificate Authority (CA).
4. Your provisioning system 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. This telemetry needs to be collected from the machine and uploaded to an IoT application. The main 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, the MQTT topic for telemetry messages should 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 can define several scenarios for capturing and interacting with device data telemetry.



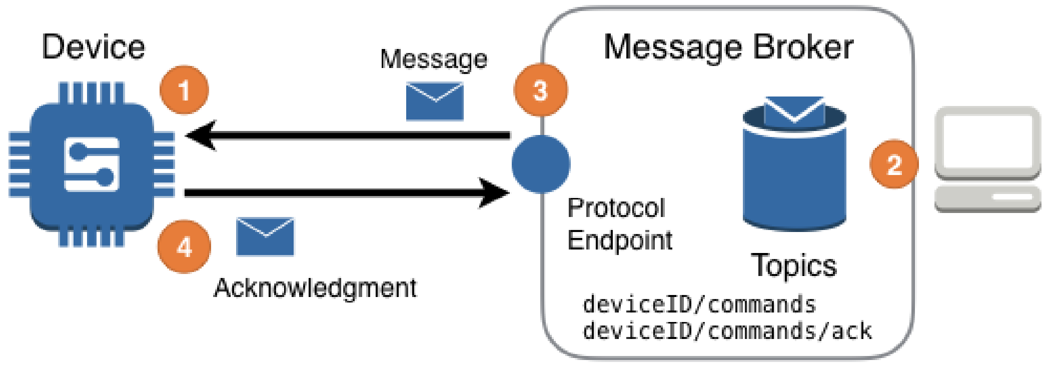
1. One publishing topic and one subscriber. For instance, a smart light bulb that publishes its brightness level to a single topic where only a single application can subscribe.
2. One publishing topic with variables and one subscriber. For example, a collection of smart bulbs publishing their brightness 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 instance, 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

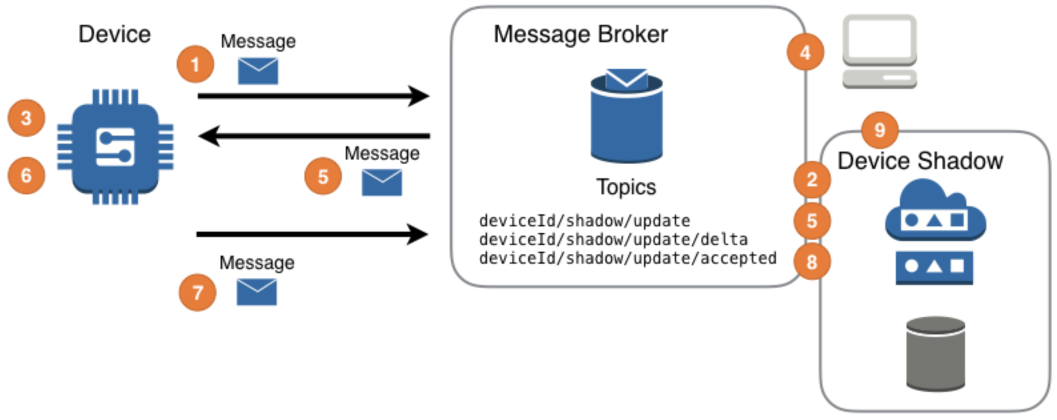
When you are building an IoT application, you will need the ability to interact with your device through commands remotely. Whether you are manufacturing devices in the industrial vertical where controls may only be to request specific data from a piece of equipment or in the smart home vertical where you are using automation to schedule an alarm system remotely, commands are a standard part of an overall IoT application.

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 of when a device has executed the command. 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. The device can then retrieve any missed shadow information by requesting the latest.



AWS IoT Device Shadow Service

IoT solutions that leverage the Device Shadow service in AWS IoT Core can 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 and state changes are communicated. This approach describes Device Shadows service that uses a JSON document to store a device's current state, desired future state, and the difference between current and desired states.



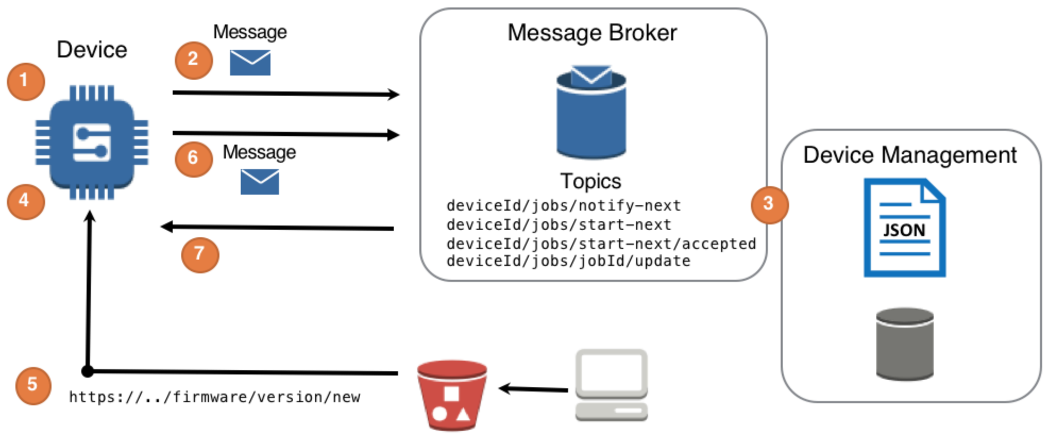
1. A device reports initial device state by publishing that state as a message to the update topic `deviceID/shadow/update`.
2. The Device Shadow reads the message from the topic and records the device state in a persistent data store.
3. A device subscribes to the delta messaging topic `deviceId/shadow/update/delta` 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 should allow devices to upgrade their own firmware. Supporting firmware upgrades without human intervention is both critical for scaling solutions and for delivering a successful long-term IoT offering.

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 in conjunction with AWS IoT message broker and AWS IoT Jobs to send firmware update commands to devices as well as receive the status of those firmware updates over time.

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


Device
Message
Message
Message Broker
Topics
deviceId/jobs/notify-next
deviceId/jobs/start-next
deviceId/jobs/start-next/accepted
deviceId/jobs/jobId/update
Device Management
JSON
https://../firmware/version/new
1
2
3
4
5
6
7

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 then 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 the firmware using a pre-signed URL for Amazon S3.
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 should be documented, tested, and regularly reviewed.

Design Principles

In addition to the overall Well-Architected Operational Excellence design principles, there are several 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. PKI can be done completely offline with a Hardware Security Module (HSM) during the manufacturing process. You can also leverage technologies that can manage the Certificate Authority (CA) and HSM in the cloud.

- **Implement device bootstrapping:** Design the ability for devices to programmatically update configuration information using a globally distributed bootstrap API. A bootstrapping design will ensure you can programmatically send the device new configuration settings by making a change in 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 should extrapolate the total expected inbound and outbound traffic of device data and the expected infrastructure needed 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 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 be able to roll back or put a stop to firmware changes based on KPIs.
- **Implement functional testing on physical assets:** IoT is unique since you must incorporate and test physical assets prior to being deployed out in the field. Functional testing is a critical path to production. The goal of functional testing is to run your physical assets 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 your physical assets perform as expected once they are deployed.

Definition

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

1. Preparation
2. Operations
3. Evolve

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 need to 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 you ask how to ensure 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 be able to scale to meet the needs of an ever-increasing load of connected hardware.

When using an IoT platform, you also 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 you are ready to support the operations of devices of your IoT workload?
IOTOPS 3. How are you provisioning new devices onto your IoT application and ensuring each device has the required prerequisites to operate?

Unlike traditional cloud-native applications, IoT applications rely heavily on actual devices that will be communicating to the cloud. Machines/devices do not have the same type of identity that humans have. They lack user accounts, usernames, and passwords, and often are without user interfaces. Due to these constraints, in IoT 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 well-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 during manufacturing 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, you can 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 if not impossible to troubleshoot locally. This implicit barrier of remote operations on devices elevates the requirements of your IoT application to be able 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. In addition, you must implement a variety of device simulation canaries that continue to test common device interactions directly against your production system. Device canaries will assist in narrowing down the potential areas to investigate when operational metrics are not being met. Device canaries can be used to raise pre-emptive alarms when the canary metrics fall below your expected SLA.

In AWS, you can begin by creating 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 current 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 should 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 several devices at once by categorizing them into groups. Groups can also contain other groups allowing you to build a hierarchy of groups. With organizational structure in you IoT application, you can quickly identify and act on related devices by device group. In addition, by leveraging the cloud, you can automate the addition or removal of devices from groups based on your business logic and the lifecycle of your devices.

In IoT, your devices will 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, you should 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, you can generate dashboards for all Cloud side 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, or errors.

To augment your production device deployments, you should implement IoT simulations on Amazon Elastic Compute Cloud (Amazon EC2) as device canaries across several AWS Regions. These device canaries should be 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 should 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 7. How do you evolve your IoT application with minimum impact to downstream IoT devices?

Since devices pose unique challenges when upgrading firmware, 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 need to be resilient to common challenges that exist in local environments such as intermittent network connectivity and possibly power loss. You should use a combination of grouping IoT devices for deployment and staggering firmware upgrades over a period. As devices are updated in the field, the IoT architecture should monitor the behavior of these upgraded devices and only proceed after a percentage of devices have been upgraded successfully.

When implementing a device upgrade strategy, you should use AWS IoT Device Management for creating deployment groups of devices and delivering over the air updates to specific device groups. During upgrades, you should continue to collect all of the CloudWatch Logs, telemetry, and IoT device job messages and combine that information with the KPIs being used to measure overall application health and the performance of any long-running canaries.
Before and after firmware updates, you should 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 should be 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. AWS IoT Core has several features that can be 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 can be 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 remediations 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 through risk assessments and mitigation strategies.

Design Principles

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

- **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 significantly fewer opportunities to impact any other devices
- **Secure device credentials at rest:** Devices should securely store credential information at rest using mechanisms like a software or hardware based secure element such as a Trusted Platform Module.
- **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. This programmatic process can remove the manual process that would be required to update device identity locally.

Definition

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

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

In addition to these security best practices in the cloud, IoT applications must also implement hardware security and protection. IoT implementations follow a shared responsibility model where you 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

IoT devices are often a target for attackers because they hold a fundamental identity to your IoT application and, in many cases, they are physical and logical assets that an attacker can attempt to gain direct access to. To provide comprehensive security against attackers, 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 or software based Trusted Platform Module
- Physical unclonable function modules
- Up to date cryptographic libraries and standards such as PKCS11 and TLS1.2

To secure device hardware, you should implement solutions such that private keys and sensitive identity are only known by the device and are stored in a secure location on the hardware. Additionally, access to that information should guard against attackers. AWS recommends that you 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 against your IoT application.

During the lifetime of a device, you will need to be able to manage not only the revocation lifecycle of certificates but also certificates that are set to expire within a given timeframe. 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, you can securely rotate device certificates prior to expiry.

Once you have a well-architected OTA process, the next aspect of certificate lifecycle is to have a central data repository that keeps track of your certificate metadata, such as expiry and revocation. Your IoT application can then automate the process that can use that repository to determine if a certificate is invalid or revoked, or if that certificate is set to expire.

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?
IOTSEC 3. How do you manage the lifecycle of identity and authorization for 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.

AWS recommends 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 should have its own unique thing in the IoT registry, and the thing name should be 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, you may choose to build your own OTA infrastructure, and to that end ensure that the OTA application is well-architected. We recommend you use AWS IoT Jobs, which is a managed platform for distributing OTA communication and binaries to your devices. AWS IoT Jobs can be used to define a set of remote operations that are sent to and executed on one or more devices connected to AWS IoT. AWS recommends using AWS IoT Jobs for OTAs. AWS IoT Jobs by default integrate several best practices, including individual device tracking of update progress and overall fleet wide metrics for a given update.

You may also 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 from there trigger a notification with a list of all 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 4. 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 requires mobile users who must onboard their devices and associate them with their consumer account. In agriculture, you may have a remote technician that needs to diagnose a hardware failure by viewing the latest telemetry from agricultural equipment in near real-time. In cases where you have a user who requires IoT access, it's essential to determine how your application will identify those users, grant the users permissions to communicate with devices, and determine the length of authorization that user may have to interact with particular devices.

Similar to security with devices, the beginning of user permissions begins with identity. Your IoT application should have in place an identity store that keeps track of a user's identity and also how a user authenticates using that identity. The identity store must also maintain a map of policy claims for what devices a user has the privilege to communicate with and what communication is valid. In addition, we recommend decoupling the IoT data store and creating a unique data store view of the device data specifically for users. By creating a separate telemetry data store tailored to user consumption in your application, you can control the access a user has and more easily track user interactions with your IoT data separately from device interactions.

When using AWS to authenticate and authorize IoT application users, you have several options to implement your identity store and how that store associates user permissions. For your own applications, you should use Amazon Cognito for your identity store. Amazon Cognito provides solutions to control access to backend resources from your application. 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, you should use a mobile service like AWS AppSync. With AWS AppSync 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 like 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, you should 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 internal authorization service. With either set of permissions, you should 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 IoT MQTT communication. Lastly, since your user is acting on behalf of a device, such as a tablet or mobile phone, you should model your user's device as a thing in the AWS IoT thing registry. This gives you the ability to take advantage of several IoT features such as thing groups, fine-grained logging, and security profiles for behavior against all of your user interactions against AWS IoT Core.

Unless a user has a static 1:1 mapping to a single device, AWS recommends you create user specific MQTT topics. When your end users can communicate with multiple devices at a time or have constantly changing temporary access to different devices, a fixed user topic allows you to dynamically route data through the cloud while minimizing the effort required to dynamically update or manage security policies per user device. Instead, by creating user specific topics, you can facilitate the routing and mapping of data in the cloud.

While implementing your IoT application, you should plan on how you are going to manage user access to your devices. For a well-architected IoT application, ensure users prove authorization to any device and use authenticated identities with fine-grained permissions. Also, ensure you enforce a time to live on any valid authorization credentials to easily implement revocation for users.

IOTSEC 5. How do you ensure least privilege is applied to any principal that communicates to your IoT application?

After registering a device and establishing its identity, you should also determine what attributes the device will use to not only connect to your IoT application, but to seed any relevant information from the device that will be needed for monitoring, metrics, telemetry, or command and control. Any IoT permission should only grant access to its own resources. By reducing the actions that a device or user can take against your application, you limit the impact that can occur if any single identity is compromised.

In AWS IoT, you can 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, you can store the unique identifier as the SerialNumber in the subject name of the certificate in order to use Certificate Policy Variables to further refine your IoT permissions.

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 SerialNumber as its 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/${iot:Certificate.Subject.SerialNumber}"],
        "Condition":{
            "Bool":{
                "iot:Connection.Thing.IsAttached":["true"]
            }
        }
    },
    {
        "Effect":"Allow",
        "Action":["iot:Publish"],
        "Resource":["arn:aws:iot:us-east-
1:123456789012:topic/$aws/things/${iot:Connection.Thing.ThingName}/sha
dow/update"]
    }
    ]
}
```

In addition to attaching a set of well-defined policies using policy variables to your device identities, you should ensure you attach your device identity (certificate or Amazon Cognito federated identity) to the thing 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, you will still want to 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 in such a way that the MQTT topics for communicating are still secure, but flexible to new devices being added to the edge gateway.

```
{
    "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 policy would be associated with the edge gateway with the identifier, edgegateway123. In addition, the edge gateway permissions in this policy would 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.

IOTSEC 6. How do your devices obtain permissions to other backend applications?

In IoT applications, devices will typically also need to communicate to other backend services using different protocols such as HTTPS. To ensure that devices communicate securely to these other endpoints, you should leverage the already trusted, unique identity associated with their IoT application to authenticate and authorize communication to these other endpoints. By using your initial IoT identity to receive other short term credentials, you can seamlessly enable your device to send data to other applications while using a single identity of truth to manage what the device can access.

With AWS IoT, devices use their unique identity, x.509 certificate, or Amazon Cognito federated identity, to communicate using MQTT to the AWS IoT broker. Depending on the type of protocol and endpoint, your application may use IoT as a credentials provider, which will allow you to retrieve temporary AWS credentials associated to an IAM role or create your authorization endpoint using services like Amazon API Gateway with your authorization store when you need to communicate to internal applications not fronted by IAM.

By using IoT as credentials provider, you can attach scoped down IAM policies to a role that your device can assume. Also, the credentials provider supports the following policy variables:

- `credentials-iot:ThingName`
- `credentials-iot:ThingTypeName`
- `credentials-iot:AwsCertificateId`

Policy variables are used to further restrict permissions for each device based on their identity. As an example, you could use credentials-iot:ThingName to restrict uploads to Amazon S3 to include the thing name in the path of an S3 bucket. The primary goal in your final policy is to ensure that your permissions restrict a device to only its needed additional backend resources.

In cases where your IoT device cannot support a one-time HTTP call, you can still achieve a similar credential exchange by using your current IoT communication channel to receive temporary permissions. For example, if using AWS IoT, you would publish an MQTT command on a “permissions” topic for your specific device, then in your IoT application retrieve temporary credentials that can be published down to the device using a different MQTT topic.

By using your initial secure device identity, you create a mechanism that extends the root of trust for that device to other services. This approach also reduces the hardware footprint required for your device by reducing the number of unique identities it must persist locally.

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. Attackers will probe for known vulnerabilities in your IoT application and will look to compromise individual devices to gain further access into other devices, applications and cloud resources. With the dual micro and macro view of security, you will need to enable several levels of logging, monitoring, and alerting to detect issues at the device level and 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 7: How are you analyzing application logs and metrics?
IOTSEC 8: How are you analyzing IoT device logs, metrics, and settings?

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

You should augment Device Defender metrics with the CloudWatch Metrics and CloudWatch Logs generated by AWS IoT Core. These service level logs provide important insight into activity pertaining to updates to a device’s dynamic attributes like its shadow, successful processing by the AWS IoT rules engine, and overall AWS IoT Core protocol usage. All CloudWatch Logs should be transferred to an internal, central logging system to correlate log information across all of your applications.

IOTSEC 9: 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. Due to its sensitivity, it's vital in IoT applications to be able to manage invalid identities such as certificates centrally. An invalid certificate can be in one of finite set of states: revoked, expired, or inactive. As part of a well-architected application, you should have a process for capturing all invalid certificates and based on the state of the certificate, trigger an automated response.

A well-architected IoT application should first establish a certificate revocation list (CRL) that keeps track of all revoked device certificates or certificate authorities (CAs). You should use your own trusted CA for on-boarding devices and sync your CRL on a regular basis to your IoT application. Your IoT application can then use its CRL to proactively revoke identities that are no longer valid.

With AWS, you do not need to manage your entire PKI on-premises. You can use AWS Certificate Manager Private Certificate Authority or an APN Partner that can offload the certificate signing process by providing preconfigured secure elements. AWS Certificate Manager has the capability to export revoked certs 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. Certificates are long-lived identities but must have an expiration date indicating when they are no longer valid and when they should be exchanged for new certificates. It is up to your IoT application to keep track of devices near their expiry and perform an OTA process to update their certificate with a new certificate with a later expiry.

AWS recommends enabling AWS IoT Device Defender audits related to the certificate and CA expiry. Device Defender will produce an audit log of certificates that are set to expire in 30 days. This list can then be used to programmatically update a device before the certificate is no longer valid. You may also choose to build your own expiry store to keep track of certificate expiry dates and programmatically query, identify, and trigger an OTA.

Infrastructure Protection

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

IOTSEC 10: How do you manage the permission of IAM roles that are used by the AWS IoT rules engine?

One of the benefits of using AWS IoT is that the communication between devices and other AWS services is made simpler by using the AWS IoT rules engine. In the rules engine, you will configure a set of rules and a set of actions, and with each rule, you will delegate an IAM role that the rule can assume to communicate to the downstream AWS service. Concerning IAM roles assumed by the rules engine, you should also enforce least-privileged access such that each IoT rule does not share an IAM role.

Data Protection

Before architecting an IoT application, data classification, governance, and controls must be designed and documented to reflect not only how data can be persisted in the cloud, but also how data should be encrypted on a device, and between the devices and the cloud. Unlike traditional cloud applications, data sensitivity and governance extend to the IoT devices that can be deployed in remote locations outside of your network boundary. These techniques are important since they support objectives such as protecting personally identifiable data transmitted from devices and complying with regulatory obligations such as PCI, GDPR, and HIPAA.

IOTSEC 10: How do you classify your data?
IOTSEC 11: How do you manage data protection on device?
IOTSEC 12: How do you protect data transmitted from device to your IoT application?

All traffic to and from AWS IoT must be encrypted over 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 encrypt not only the certificates of the device but the data collected and processed on the device.

For embedded development, AWS has several services that abstract away 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 AWS functionality. In addition, Amazon FreeRTOS contains a set of security APIs that allow you to create embedded applications that securely communicate to AWS IoT. For Linux-based gateways, AWS has AWS Greengrass which extends cloud functionality to edge gateway. AWS Greengrass implements several security features into the Greengrass core component including x.509 certificates between the Greengrass core and connecting devices, IAM policies and roles to securely allow Greengrass applications to communicate to cloud applications, and Greengrass subscriptions which are used to determine how and if data can be routed between downstream end devices and Greengrass core.

Incident Response

Implementing incident responses in IoT requires planning on how you will deal with larger scale IoT events such as network outages, and how you will respond to events that can happen on a smaller scale to a single device or a handful of devices. The architecture of your IoT application will play a large role in determining how quickly you will be able to diagnose incidents, correlate the data across the incident, and then subsequently apply any runbooks to the affected devices in an automated, reliable fashion.

For IoT applications, AWS recommends 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.

IOTSEC 13: How do you prepare to respond to an incident that affects an individual device?
IOTSEC14: How do you prepare to respond to an incident that affects a large number of devices?

Implement a strategy in which devices can be quickly identified for remediation by your InfoSec team. Ensure the InfoSec team has runbooks that consider firmware related considerations for device updates. Create automated processes that can proactively resolve common security patches through applying default updates to vulnerable devices as they come online.

Key AWS Services

The AWS services that are essential to security in IoT are AWS Identity and Access Management (IAM), x.509 device certificates, and Amazon Cognito Identity Pools, and Amazon Cognito User Pools. 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 four areas of security:

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) enables 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. AWS Private CA provides a software-based approach to creating and managing device certificates. Use AWS

IoT thing groups to group devices and 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 monitors any administrative actions of your IoT application. Amazon CloudWatch is a monitoring service with a default integration to AWS IoT Core and CloudWatch 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 managed 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 requires IAM permissions in order to communicate with other downstream AWS services.

Data protection: AWS IoT includes encryption capabilities for devices to the cloud over TLS 1.2 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 an AWS edge offering such as Amazon FreeRTOS or AWS Greengrass to support secure communication on the device.

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 & 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 how we handle change.

Design Principles

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

- **Simulate device behavior at production scale**: You can create a production-scale test environment that would closely mirror their 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, your simulation tests should ramp up over a period of time starting with 10% of overall traffic for a single test and incrementing over time (i.e. 25%, 50%, then 100% of day one device traffic). During simulation tests, you should monitor performance and review logs to ensure the entire solution behaves as expected.
- **Buffer message delivery from the IoT rules engine with streams or queues**: Leverage services like Amazon Kinesis and Amazon SQS as the initial buffer for 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**: Along with designing for resiliency in the AWS Cloud for your IoT application, it's essential to plan for resiliency on the device itself. A resilient device includes but is not limited to: having a robust retry logic for intermittent connectivity, the ability to rollback firmware updates, and ability to fail over to a different networking protocol or communicate locally for critical message delivery.

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, ideally, automatically heal itself.

Best Practices

Foundations

Unlike traditional web traffic, IoT device traffic is composed of data sent from similar application firmware and normally operates in a homogeneous fashion. In addition, these devices must continue to operate in some capacity in the face of network errors or IoT cloud errors that may occur. Devices are less forgiving to non-scalable cloud architectures since over a period of time the device retry logic will continue to attempt to connect to your IoT application, and due to memory and hardware constraints, all error handling on a device must be provisioned into the firmware beforehand. In order to prepare for a growing number of connected devices, an IoT application must be able to horizontally scale over time. In addition, you must monitor overall IoT utilization and create a mechanism to automatically increase capacity to ensure your application can manage peak IoT traffic.

To prevent devices from creating unnecessary peak traffic, device side logic must have enough variability in how it operates to prevent the entire fleet of devices from attempting the same operations at identically the same time. For example, if an IoT application was composed of alarm systems and all alarm systems sent an activation event exactly at 9am local time, your IoT application would be inundated with an immediate spike from your entire fleet at the same moment. Instead, by incorporating a randomization factor into those scheduled activities, such as timed events, retry logic, and after firmware updates, IoT devices can more evenly distribute their peak traffic within a window of time.

The following questions focus on these considerations for reliability.

IOTREL 1. How do you handle IoT limits for peak workloads?
IOTREL 2. How are you regulating IoT messages to and from your IoT devices?

AWS IoT provides a set of soft limits 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 are normally device side operations such as doing an MQTT Connect, MQTT Publish, or MQTT Subscribe. Data plane operations are the primary driver of your device connectivity and therefore it's important to review the IoT limits and ensure 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 will be to ensure that you architect around any hard limits because exceeding limits that are not adjustable will result in application errors such as throttling and client errors. Hard limits are normally related to throughput on a single IoT connection. If you find your application exceeds a hard limit, we recommend re-architecting 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 prior to 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 the IoT application is not under provisioned, you should:

- 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 you need a limit increase
- Set alarms for all thresholds in IoT including MQTT connect, publish, subscribe, receive, and rule engine actions.
- Ensure you request a limit increase in a timely fashion, typically weeks in advance.

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, IoT 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. It is also the case that these control plane calls are normally in the order of magnitude of 10's of requests per second. It is important for you to work backward from peak expected registration usage in order to determine any limit increases for control plane operations. You should also 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 in order to control data rates to these APIs.

IOTREL 3. How are you regulating ingestion throughput of IoT data to internal IoT applications?

IOTREL 4. What is your strategy for managing ingestion and processing of IoT data to other applications?

Although IoT applications will 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 will need to 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 layers is comprised of internal queues or streams, which enable durable short-term storage while allowing compute resources to process data independent of the rate of ingestion.

In order to accomplish this queuing layer in AWS, we recommend that your data be rested using streaming layers (as Amazon Kinesis Data Streams, Amazon Kinesis Data Firehose, or Amazon Simple Queue Service) before performing any compute operations.

For all of these intermediate streaming points, you should ensure these resources are over-provisioned to handle peak capacity. With AWS IoT rules, you can configure a rule that sends data to any number of temporary ingestion services including Kinesis Data Streams, Kinesis Data Firehose, or Amazon SQS. This naturally creates the queueing layer needed for upstream applications to process data resiliently.

IOTREL 5. How do you handle device reliability when communicating with AWS IoT Core?

A unique part of implementing reliability in IoT is handling reliability on the IoT device itself. Devices will be deployed in remote locations and must deal with intermittent connectivity or loss in connectivity due to a variety of external factors that will be out of the control of your IoT application. 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? AWS recommends you 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 for long periods of time. As noted above, there will be environmental variables such as loss of coverage due to the location, which will be out of your control. The only way to defend against this occurrence is to implement robust operations in your firmware that can:

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

Change Management

IOTREL 6. How do you roll forward and roll back changes to your IoT devices?
IOTREL 7. How do you manage to update your IoT application?

For IoT applications, it's 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 be capturing 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 will 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 "Default" version for your devices to fall back to in the event of an error.
- Implement a secure update strategy using concepts, such as secure boot and multiple non-volatile storage partitions, to deploy software images and implement a fallback.
- 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.

By automating and treating as much of your infrastructure as code on AWS, you are able to monitor and apply changes to your IoT application easily. A well-architected solution will also version all of the device firmware artifacts and ensure updates can be verified, installed, or rolled back when needed.

Failure Management

IOTREL 8. How do you back up IoT data?
IOTREL 9. How does your IoT application withstand failures?

Since 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 will have the ability to log and retry errors in data processing. An IoT application will also 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 gives you the benefit of capturing, monitoring, alerting, and eventually retrying messages that could not be delivered to their primary IoT action. We recommend an IoT error action is configured with a different AWS service from the primary action. We also recommend using durable storage for error actions such as Amazon SQS or Kinesis.

Starting with the rules engine, your application logic should initially process any ingestion messages from a queue and validate that the schema of that message is correct. Your application logic should be able to catch any known errors and can immediately proceed to log any known errors in the payload or format and move those messages to their own DLQ for further analysis. You should also 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.

IOTPERF 10. What functional testing has been done to verify different levels of hardware failure modes for your physical assets?

IoT implementations need to 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").

Also, there is a distinction between device failure and sensor failure. For instance, a temperature sensor failure should never, by itself, affect the entire IoT implementation and operation. Until the sensor can be replaced, is there another sensor on another device in the vicinity that can be enabled to replace the one that failed?

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, the best practice is to use Fleet Indexing with connectivity status to track the change in connectivity status. If the device is offline for extended periods of time, you can trigger an alert that the device may require remediation.

Key AWS Services

The AWS service that is key to ensuring reliability is Amazon CloudWatch, which monitors run-time metrics. 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 lets you 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 IT and 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 performance efficiency design principles, there are 4 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 batch**: IoT applications should decouple the connectivity portion of IoT with 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.
- **Design event driven architectures**: IoT systems are distinctly event-driven architectures. IoT systems publish events from devices and permeate those events to other subsystems in your IoT application. You should 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 (devices, connectivity, databases, compute, and analytics)
2. Review
3. Monitoring
4. Tradeoffs

Take a data-driven approach to 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 will ensure that you are aware of any deviation from expected performance and allow you to take action on it. Finally, your architecture can make tradeoffs to improve performance, such as using compression or caching, or relaxing consistency requirements.

Best Practices

Selection

Well-Architected IoT systems are made up of multiple solutions that must manage different aspects of common IoT scenarios, 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 their business objectives. With a large array of options, from running your own IoT applications on Amazon EC2, to leveraging AWS managed services, AWS recommends 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 that creates a long-term view of how you will operate and run your IoT applications. At its core, 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 greatly based on the hardware footprint of the device. For devices that use AWS IoT Core or simply want to communicate securely to their own cloud backend, AWS recommends that devices 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 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, a secure, scalable connectivity platform should be implemented that can support long-term growth of your devices over time. In addition to the sheer volume of devices, an IoT platform must be able to horizontally scale the communication workflows between devices and the cloud, whether that is simple ingestion of telemetry or command and response communication between devices. In AWS, you have two primary choices for IoT solutions in the cloud for connectivity.

You can build your own IoT application on Amazon EC2, which will allow you the flexibility to choose all aspects of your IoT application, such as the IoT protocol to implement and the security standards to employ. However, by building your own application on Amazon EC2, you take on the undifferentiated heavy lifting that is not core to building unique value into your IoT offering. AWS recommends you use AWS IoT Core for your IoT platform.

AWS IoT Core supports HTTP, WebSockets, and MQTT, a lightweight communication protocol specifically 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

In an IoT application, you will have multiple databases, each selected for their specific attributes, considering 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 several other criteria you should 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, or 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 naturally align with the nature of IoT applications. For structured data you should use Amazon Aurora. Amazon Aurora provides 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, you should use DynamoDB. DynamoDB is 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, AWS recommends using AWS IoT Analytics and Amazon S3. AWS IoT Analytics filters, transforms, and enriches IoT data before storing it in a time-series data store for analysis. You should also review 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 by exploring your data and spotting trends or patterns over time.

In addition to storing and leveraging the historical trends of your IoT data, you will need to have a system that stores the current state of the device and gives you the ability to query against the current state of all of your devices not only for internal use cases but for enabling external views into your IoT data.

The AWS IoT Shadow service can be 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, you can leverage the managed capabilities of Fleet Indexing, which will provide a Lucene-based index incorporating your IoT registry and shadow metadata. If there is a need to provide a similar Lucene-based searching or filtering capability to a large number of external users, such as for a consumer application, you should 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 your database for your IoT device state?

Compute

Since IoT applications lend themselves to a high flow of ingestion that requires continuous processing over the stream of messages, an architecture must choose the appropriate compute services that most align with the steady enrichment of stream processing and can also support running business applications during and prior to data storage.

The most common compute services used in IoT are AWS Lambda and Amazon EC2. Lambda functions can be invoked from the moment telemetry data reaches AWS IoT Core and can be invoked on AWS Greengrass. It is worth noting that Lambda can be used at different points throughout IoT. Where you elect to trigger your business logic via Lambda is influenced by when you want to process a specific data event .

EC2 instances can 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 IoT events?

Analytics

The primary business case for implementing IoT solutions is to gain important contextual data about how devices are performing and being used in the field. Based on that data, businesses can make more informed decisions about what new products or features to prioritize or how to more efficiently operate workflows within their organization. For these reasons, 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 to 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 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 historical analysis of your IoT application?

In verticals like IoT, there is a large percentage of time spent on the components to implement and scale your IoT application that is not directly impacting your business outcomes. For example, managing IoT protocols, securing device identities, transferring telemetry between devices and the cloud. Although these aspects of IoT are important, they do not directly lead to differentiating value in your IoT application. Given the pace of innovation of IoT services, AWS releases new features and services based on the common challenges of IoT. With the added visibility of collecting telemetry and diagnostic data from your devices, you should perform a regular review of your data to see if new AWS IoT services can either solve a current IoT gap you have in your architecture or can replace components of your architecture that are not considered core business differentiators.

Monitoring

IOTPERF 7. How frequently do you run end to end simulation tests of your IoT application?

IoT applications can be simulated using either production devices set up as test devices (i.e. with specific test MQTT namespace) or using simulated devices. All incoming data captured using the IoT rules engine is then processed using the same flows that will be used during production.

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

IOTPERF 9. How can performance monitoring be used in IoT implementation?

There are several key types of performance monitoring related to IoT deployments: device, cloud performance, and storage/analytics. Best practice is to create appropriate performance metrics using data collected from logs together with telemetry and command data. AWS recommends that you start with basic performance tracking and build on these 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. Also, create CloudWatch alarms based on your application custom metrics to gain quick insight into how your IoT application is behaving.

You should set up fine-grained logs to track specific thing groups. During IoT solution development you should enable DEBUG logging to gain 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 implementation, you should change the logging to ERROR and WARN.

In addition to cloud instrumentation, you need to run instrumentation on devices prior to deployment to ensure devices make most efficient use of their local resources and that firmware code does not lead to unwanted scenarios like memory leaks. Best practice is to 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 typically 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 needs to be given to architecting highly efficient IoT implementations where data and analytics get pushed by devices to the cloud, and when supported machine learning algorithms are pulled down on the device gateways from the cloud.

In addition, individual devices will be constrained by the throughput that is supported over a given network. Whether a device is directly connected to a LAN or is operating over cellular, the frequency with which data is sent from and received to the device must be balanced with the transport layer and the ability of the device to optionally store, aggregate, and then send data to the cloud.

IOTPERF 9. How quickly is the data 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 determine the most efficient point to process IoT data. In network constrained environments where the hardware is not limited, you can make use of edge solutions such as AWS Greengrass to operate and process data completely offline. In cases where both the network and hardware are constrained, you should look for opportunities to compress message payloads when possible by compressing the data into a binary format or by 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, you can 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 overall performance and operational health. The following services are important in the areas of 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 for microcontrollers. AWS Greengrass is software that lets you 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 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

- AWS Lambda Getting Started
- DynamoDB Getting Started
- AWS IoT Analytics User Guide
- Amazon FreeRTOS Getting Started
- AWS 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 very 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, thus allowing your business to maximize its return on investment.

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

As with the other pillars, 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 "just in case" rather than spend time benchmarking for the most cost-optimal deployment. This often leads to drastically over-provisioned and under-optimized deployments. The following sections provide techniques and strategic guidance for the initial and ongoing cost optimization of your deployment.

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 an overall cost for your IoT solution, often IoT architectures look at connectivity through the lens of a bill of material (BOM) costs. For BOM calculations, you need to be able to predict and monitor what the long-term costs will be to manage connectivity to your IoT application throughout the lifetime of that device. By leveraging AWS services, you can calculate initial BOM costs, making use of cost-effective services that are event driven, and over time 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. Not only can aggregation save on compute resources, but aggregating data into storage can enable solutions where data is compressed and archived before being persisted. This strategy decreases the overall storage footprint without losing data or compromising query ability of the data.

COST 1. How do you select an approach for batch, enriched, and aggregate data delivered from 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 first creating an AWS IoT rule that sends the data to Kinesis Data Streams, Kinesis Data Firehose, AWS IoT Analytics, or Amazon SQS. From these services, you can begin to process multiple data events at once.

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

Matching Supply and Demand

Optimally matching supply to demand delivers the lowest cost for a system, but given the torrent nature of IoT workloads, solutions need to be dynamically scalable and must 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 these considerations for cost optimization.

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

If you utilize serverless technologies such as AWS Lambda and API Gateway, you only pay for when your application utilizes those services on a per request basis. In addition, AWS IoT Core, AWS IoT Device Management, AWS IoT Device Defender, AWS Greengrass, and AWS IoT Analytics are all managed services that are pay per usage and do not charge you for idle compute capacity. The benefit of managed services is that AWS takes care of the automatic provisioning of your resources. If you utilize managed services, you are still responsible for monitoring and alerting when you need limit increases for AWS services.

When architecting to match supply against demand, you will want to proactively plan your expected usage over time and what limits you are most likely to exceed and at what time. You must then work backwards to factor in those limit increases into your future planning.

Expenditure Awareness

There are no expenditure awareness price optimizations unique to IoT applications that belong to this sub-section.

Optimizing Over Time

In addition to new features released by AWS over time, in IoT, you can analyze how your devices are performing and make changes to how your devices communicate to your IoT application to further optimize on cost.

Since managed services from AWS can often optimize an application, it is important to be aware of new IoT managed services and features as they become available.

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

COST 4. 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. This would translate to sending discrete data events from the device to the cloud, making minimal use of batching multiple events in a single message. Other device side optimizations would involve using 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 128KB. For this reason, ideally, each MQTT payload size should be as close to possible to any 5 KB. For example, a payload that is currently sized at 6KB is not as efficient cost-wise as a payload that is 10KB since 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 2KB message every 1 second can be achieved at a lower IoT message cost by sending two 2KB messages every other second.

Note this approach does come with tradeoffs that should be considered before implementation. By adding complexity or delay in your devices, you may unexpectedly increase the cost to process this more complex data in the cloud. A cost optimization exercise around IoT payloads should only happen after your solution has been in production and you can make a data-driven approach to the cost impact of changing the way data is sent to AWS IoT Core.

COST 5. 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 normally 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. By grouping shadow updates together when possible you reduce the overall cost of the shadow by consolidating updates to the service.

Key AWS Services

The key AWS feature that supports 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 four 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

- 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 will help 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
- 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
November 2018	First publication.