AWS Well-Architected Framework

IoT Lens

IoT Lens: AWS Well-Architected Framework

Table of Contents

IoT Lens

Publication date: **November 16, 2023** (*Document revisions*)

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

Introduction

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

In this “Lens” we focus on how to design, deploy, and architect your IoT (Internet of Things) workloads at the edge and in the AWS Cloud. The guidance provided includes both IoT and industrial IoT (IIoT) workloads and the document calls out specific guidance for segments such as consumer, commercial and industrial when relevant. To implement a well-architected IoT application, you must follow well-architected principles, starting from the procurement of connected physical assets (things), operating the asset to the eventual decommissioning of those same assets in a secure, reliable, scalable, sustainable and automated fashion. In addition to AWS Cloud best practices, this document also articulates the impact, considerations, and recommendations for connecting physical assets to the internet.

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

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

Custom lens availability

Custom lenses extend the best practice guidance provided by AWS Well-Architected Tool. AWS WA Tool allows you to create your own custom lenses, or to use lenses created by others that have been shared with you.

To determine if a custom lens is available for the lens described in this whitepaper, reach out to your Technical Account Manager (TAM), Solutions Architect (SA), or AWS Support.

Definitions

The AWS Well-Architected Framework is based on six pillars — operational excellence, security, reliability, performance efficiency, cost optimization, and sustainability. When architecting technology solutions, you must make informed tradeoffs between pillars based upon your business context. For IoT workloads, AWS provides multiple services that allow you to design robust architectures for your applications. Internet of Things (IoT) applications are composed of many devices (or things) that securely connect and interact with complementary edge-based and cloud-based components to deliver business value. IoT applications gather, process, analyze, and act on data generated by connected devices. Industrial Internet of Things (IIoT) are systems that connect and integrates industrial control systems with enterprise systems and the internet, business processes and analytics and is a key enabler for Smart Manufacturing and Industry 4.0. The AWS IoT Lens can be used across all IoT use cases including industrial, consumer (OEM), or and any other workload that has many devices connecting at scale for telemetry and command and control.

This section presents an overview of the AWS services that are used throughout this document to architect IoT workloads. There are seven distinct logical layers to consider when building an IoT workload.

Layers

- Design and manufacturing layer
- Edge layer
- Fleet provisioning layer
- Communication layer
- Ingestion layer
- Analytics layer
- Application layer

Design and manufacturing layer

The design and manufacturing layer consist of product conceptualization, business and technical requirements gathering, prototyping, product layout and design, component sourcing, manufacturing and distribution. Decisions made in each layer impact the next logical layers of the IoT workload described in the following sections. For example, some IoT device creators prefer to have a common firmware image installed and tested by the contract manufacturer. The decisions

made at the design and manufacturing layer will partly determine what steps are required during the provisioning layer.

You may go a step further and provision and install an X.509 certificate and its private key to each device during manufacturing, or include a hardware security module with credentials already pre-provisioned. This decision can impact the provisioning and communications layers, since the type of credential can impact the subsequent selection of network protocols. If the credential never expires it can simplify the communications and provisioning layers at the possible expense of compromised security, data loss, or both.

Edge layer

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

While the AWS IoT Lens is applicable to all IoT systems, industrial IoT deployments often have additional safety, resiliency, and compliance requirements.

Industrial IoT deployments consist of a combination of plant-local Operational Technology (OT), plant-local Information Technology (IT) resources, and remote IT resources, which might be in the public cloud or an enterprise datacenter. The benefit of splitting workloads between local and remote processing is to balance the timeliness and high bandwidth of local resources with the scale and elasticity of remote resources.

The edge deployments are heavily influenced by what AWS calls the three laws of distributed computing: the **law of physics**, which constrain the latency, throughput, and availability of network connectivity; the **law of economics,** which determine the cost-effectiveness of transferring ever-increasing volumes of data; and the **law of the land,** which regulates how data is handled and where it can be stored.

AWS offers the following software and services for the edge layer:

AWS IoT device SDKs include open-source libraries, developer guides with samples, and porting guides so that you can build innovative IoT products or solutions with AWS IoT on your choice of hardware platforms.

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

AWS IoT Greengrass is an IoT edge runtime and cloud service that helps you build, deploy, and manage intelligent IoT device software. It provides you with pre-built components for common capabilities, such as local and cloud MQTT messaging, support for local edge processing including machine learning (ML) inference, logging, monitoring, integration with other AWS services, and local data aggregation, filtering, and transmission to cloud targets. After development is complete, you can seamlessly deploy and remotely manage device software on millions of devices.

AWS IoT SiteWise Edge runs on premises at industrial sites and makes it easy to collect, process, and monitor equipment data locally before sending the data to AWS Cloud destinations. AWS IoT SiteWise Edge can be installed on local hardware, such as third-party industrial gateways and computers, or on AWS Outposts and AWS Snow Family compute devices. It uses AWS IoT Greengrass, an edge runtime that helps build, deploy, and manage applications.

AWS IoT FleetWise Edge is the edge software component for AWS IoT FleetWise. AWS IoT FleetWise Edge allows connected vehicles to collect data and upload it to the AWS IoT FleetWise service. AWS IoT FleetWise helps to transform low-level messages into human-readable values and standardize the data format in the cloud for data analyses. You can also define data collection schemes to control what data to collect in vehicles and when to transfer it to the cloud.

AWS IoT RoboRunner fleet management system gateway (FMSG) is the edge software component that manages all connections between AWS IoT RoboRunner and robot vendor systems. It provides applications for polling robot properties and updating them in AWS IoT RoboRunner, and for enabling multi-robot fleets to interoperate through shared spaces, such as intersections and narrow corridors. Additionally, FMSG includes connectors to vendor fleet management systems.

AWS IoT ExpressLink is connectivity software that powers a range of hardware modules developed and offered by AWS partners, such as Espressif, Infineon, and u-blox. Integrating these wireless modules into the hardware design of your device makes it faster and easier to build IoT products that connect securely with AWS services. These modules provide cloud-connectivity and implement AWS recommended security requirements.

Fleet provisioning layer

The provisioning layer of your IoT workloads consists of mechanisms used to create device identities and the application workflow that provides configuration data to the device. In many

cases, it consists of a public key infrastructure (PKI). The provisioning layer is also involved with ongoing maintenance and eventual decommissioning of devices over time. IoT applications need a robust and automated provisioning layer so that devices can be added and managed by your IoT application in a frictionless way. When you provision IoT devices, you must install a unique cryptographic credential onto them and securely store these credentials.

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

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.

AWS Private Certificate Authority (AWS 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 an 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 services such as AWS IoT Core. With just-in-time-registration, when devices are first connected to your AWS IoT Core endpoint, you can automatically start a workflow that can determine the validity of the certificate identity and determine what permissions it should be granted.

Provisioning devices that don't have certificates: With AWS IoT fleet provisioning, AWS IoT can generate and securely deliver device certificates and private keys to your devices when they connect to AWS IoT for the first time. AWS IoT provides client certificates that are signed by the Amazon root certificate authority (CA). There are two ways to use fleet provisioning:

With **provisioning by claim**, devices can be manufactured with a provisioning claim certificate and unique token (which are special purpose credentials) embedded in them. If these certificates are registered with AWS IoT, the service can exchange them for unique device certificates that the device can use for regular operations.

With **provisioning by trusted user**, a device connects to AWS IoT for the first time when a trusted user, such as an end user or installation technician, uses a mobile app to configure the device in its deployed location.

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.

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

AWS IoT Core for LoRaWAN is a fully managed LoRaWAN Network Server (LNS) that enables you to connect wireless devices that use the LoRaWAN protocol for low-power, long-range wide area network connectivity with the AWS Cloud. This can be useful in use cases such as asset tracking, irrigation management, logistics and transportation management and smart cities.

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

Ingestion layer

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

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

Basic Ingest, can securely send device data to the AWS services supported by AWS IoT rule actions, without incurring messaging costs. Basic Ingest optimizes data flow by removing the publish/subscribe message broker from the ingestion path, making it more cost effective.

Using **AWS IoT Greengrass**, data can be ingested in S3 buckets, Firehose, AWS IoT SiteWise, AWS IoT Analytics, and with custom code to other AWS services.

AWS IoT SiteWise is a managed service that simplifies collecting, organizing, and analyzing industrial equipment data at scale to help you make better, data-driven decisions. You can use AWS IoT SiteWise to monitor operations across facilities, quickly compute common industrial performance metrics, and create applications that analyze industrial equipment data.

AWS IoT FleetWise is a managed service that makes it easy to collect, organize, and transfer vehicle data to the cloud so you can gain insights about your fleets of vehicles. After the data is ingested into AWS, it can be enriched, making it easier for data analytics. You can use data transferred to build applications that quickly detect fleet-wide quality issues, remotely diagnose individual vehicle problems in near real-time, and improve autonomous driving systems.

AWS IoT RoboRunner is a service that provides infrastructure for robotics optimization from a single system view. With AWS IoT RoboRunner, you can build applications that help robots work seamlessly together. This service is designed for industrial customers who buy, deploy, and manage robotic and automated industrial equipment, including automated guided vehicles (AGVs) and autonomous mobile robots (AMRs). AWS IoT RoboRunner provides central and managed data repositories for storing and using data from different robot vendor systems and enterprise management systems. After robots and enterprise management systems are connected to AWS IoT RoboRunner, you can use the sample implementations to create custom applications on top of the centralized data repositories, such as visualization of the robot location and status on a single map view.

Amazon Kinesis and **Amazon Simple Queue Service (Amazon SQS)** can be used in your IoT application to decouple the communication layer from your application layer. 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. 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 from data about what's happening in the local/edge environment. A primary way of realizing contextual insights is by implementing solutions that can process and perform analytics on IoT data.

Storage services

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

Amazon S3 is object-based storage engineered to store and retrieve any amount of data from anywhere on the internet. With Amazon S3, you can build IoT applications that store large amounts of data for a variety of purposes: regulatory, business evolution, metrics, longitudinal studies, analytics, security, machine learning, and organizational enablement. Amazon S3 gives you a broad range of flexibility in the way you manage data for cost optimization, latency, access control and compliance.

Analytics and machine learning services

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

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

AWS IoT Events is a managed service that makes it easy to detect and respond to events from IoT sensors and applications. Events are patterns of data identifying more complicated circumstances than expected.

AWS IoT SiteWise is a managed service that simplifies collecting, organizing, and analyzing industrial equipment data at scale to help you make better, data-driven decisions. You can use

AWS IoT SiteWise to monitor operations across facilities, quickly compute common industrial performance metrics such as overall equipment effectiveness (OEE), and create applications that analyze industrial equipment data.

AWS IoT SiteWise allows you to create no-code, fully managed web applications using **AWS IoT SiteWise Monitor**. With this feature, you can visualize and interact with operational data from devices and equipment connected to AWS IoT services.

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 you pay only for the queries that they run.

Amazon SageMaker is a fully managed service that enables you to quickly build, train, and deploy machine learning models in the cloud and the edge layer. With Amazon SageMaker, IoT architectures can develop a model of historical device telemetry to infer future behavior. Through the integration of AWS IoT Greengrass and Amazon SageMaker, you can automate the full ML lifecycle of collecting IoT data, ML training in the cloud, deploying ML models to the edge for local inference, and then retraining and redeploying in a cycle for continuous improvement of their ML models.

AWS IoT TwinMaker is an AWS IoT service that you can use to build operational digital twins of physical and digital systems. AWS IoT TwinMaker creates digital visualizations using measurements and analysis from a variety of real-world sensors, cameras, and enterprise applications to help you keep track of your physical factory, building, or industrial plant. You can use this real-world data to monitor operations, diagnose and correct errors, and optimize operations.

Amazon Managed Grafana is a fully managed service for open source Grafana developed in collaboration with Grafana Labs. Grafana is a popular open-source analytics environment that enables you to query, visualize, alert on and understand your metrics no matter where they are stored. Grafana has integrations with services such as AWS IoT TwinMaker to make visualizations easier.

Amazon QuickSight is a cloud-scale business intelligence (BI) service that you can use to deliver easy-to-understand insights to the people who you work with, wherever they are. Amazon QuickSight connects to your data in the cloud and combines data from many different sources from the IoT suite.

Application layer

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

Management applications

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

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

AWS IoT Device Management eases the organizing, monitoring, and managing of IoT devices at scale. At scale, customers are managing fleets of devices across multiple physical locations. AWS IoT Device Management enables you to group devices for easier management. You can also enable real-time search indexing against the current state of your devices through Device Management Fleet Indexing. Both Device Groups and Fleet Indexing can be used with Over the Air Updates (OTA) when determining which target devices must be updated to target specific sub-fleets of devices when you want to deploy remote operations (for example, remote reboots, over-the-air (OTA) updates, configuration pushes, and resets.) using jobs. You can also gain privileged and synchronous access (for example, SSH) to your devices for debugging and troubleshooting with Secure Tunneling.

Fleet Hub for AWS IoT Device Management is a fully managed web application that lets domain specialists, such as support technicians and operators, monitor device fleets' health in near real-time, set alerts to notify them of unusual behavior, and take built-in corrective actions (for example, deploy a patch or reboot a device) – all with no code. You can access near real-time state data from devices connected to AWS IoT Core, such as connection status, firmware version, or battery level.

User applications

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

Amazon Cognito lets you add user sign-up, sign-in, and access control to your web and mobile apps quickly and easily. Amazon Cognito scales to millions of users and supports sign-in with social identity providers, such as Apple, Facebook, Google, and Amazon, and enterprise identity providers via SAML 2.0 and OpenID Connect.

Database services

While a data lake can function as a landing zone for all of your unformatted IoT generated data, to support all the formatted views on top of your IoT data, you need to complement your data lake with structured and semi structured data stores. For these purposes, you should use 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.

Amazon Timestream is a fast, scalable, and serverless time series database service for IoT and operational applications that makes it easy to store and analyze trillions of events per day. The purpose-built query engine in Timestream lets you access and analyze recent and historical data together, without needing to specify explicitly in the query whether the data resides in the in-memory or cost-optimized tier. Amazon Timestream has built-in time series analytics functions, helping you identify trends and patterns in your data in near real-time.

Compute services

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

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

General design principles

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

- **Decouple ingestion from processing**: In IoT applications, the ingestion layer must be a highly scalable platform that can handle a high rate of streaming device data. By decoupling the fast rate of ingestion from the processing portion of your application through the use of queues, buffers, and messaging services, your IoT application can scale elastically as needed and 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 situations such as connectivity issues or misconfigured settings, devices might go offline for longer periods of time than anticipated. Design your edge software to handle extended periods of offline connectivity and create metrics in the cloud to track devices that are not connected or communicating on a regular timeframe.
- **Design for 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. As data is ingested into your application, you should first enrich the data with human readable attributes, deserialize, or expand any fields that the device serialized, and then format the data in a data store that is tuned to support your applications read requirements.
- **Handle personalization**: Devices that connect to the edge or cloud using Wi-Fi must receive the SSID name and credentials as one of the first steps performed when setting up the device. This data is usually infeasible to write to the device during manufacturing since it's sensitive and site-specific, or from the cloud since the device isn't connected yet. These factors frequently make personalization data distinct from the device client certificate and private key, which are conceptually upstream, and from cloud-provided firmware and configuration updates, which are conceptually downstream. Supporting personalization can impact design and manufacturing, since it may mean that the device itself requires a user interface for direct data input, or the need to provide a smartphone application to connect the device to the local network.
- **Ensure that devices regularly send status checks**: Even if devices are regularly offline for extended periods of time, ensure that the device firmware contains application logic that sets a regular interval to send device status information to your IoT application. Devices must be active participants in ensuring that your application has the right level of visibility. Sending this

regularly occurring IoT message ensures that your IoT application gets an updated view of the overall status of a device, and can create processes when a device does not communicate within its expected period of time.

- **Use gateways for edge computing, network segmentation, security compliance and bridging administrative domains:** Splitting the workload between local and remote processing helps to balance the timeliness and high bandwidth of local resources with the scale and elasticity of remote resources. Edge gateways can be used to mediate data flows between a low latency local-area network (LAN) and resources on the high-latency wide-area network (WAN), protocols used in each environment and can also mediate between security and administrative domains such as in a Perdue Enterprise Network Architecture (PERA), ANSI/ISA-95 network segmentation. Edge gateways are also used in consumer IoT systems, for example a smart home gateway which collects data from multiple smart home devices.
- **Build security into your IoT solution and apply security at all layers**: IoT implementations can have some very unique challenges not present in traditional IT deployments. For example, deploying a consumer IoT device can introduce a new classification of threats that needs to be addressed and industrial IoT requires more thought around reliability, safety and compliance. Many legacy OT systems are insecure by design and use industrial protocols which don't support authentication, authorization and encryption. In industrial IoT, the convergence of IT and OT systems is creating a mix of technologies that were designed to withstand hostile network environments and ones that were not, which creates risk management difficulties that need to be controlled. So, building security into every part of your IoT solution is essential for minimizing risks to your data, business assets, and reputation. Apply a defense in-depth approach with multiple security controls.

Scenarios

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

Scenarios

- Device provisioning
- Device telemetry
- Device commands

Device provisioning

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

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

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

The 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 other AWS services. One of the ways to provision a device, is through fleet provisioning. AWS IoT can generate and securely deliver device certificates and private keys to your devices when they connect to AWS IoT for the first time. AWS IoT provides client certificates that are signed by the AWS Private Certificate Authority (AWS Private CA). Fleet

provisioning provides two ways to implement this: by trusted user or by claim. Let us look at the process flow for fleet provisioning by claim.

Some devices do not have the capability to accept credentials over a secure transport, and the manufacturing supply chain is not equipped to customize devices at manufacturing time. AWS IoT provides a path for these devices to receive a unique identity when they are deployed.

Device makers must load each device with a shared claim certificate in firmware. This claim certificate should be unique per batch of devices. The firmware containing the claim certificate is loaded by the contract manufacturer without the need to perform any customization. When the device establishes a connection with AWS IoT for the first time, it exchanges the claim certificate for a unique X.509 certificate signed by the AWS certificate authority and a private key. The device should send a unique token, such as a serial number or embedded hardware secret with its provisioning request that the fleet provisioning service can use to verify against an allow list.

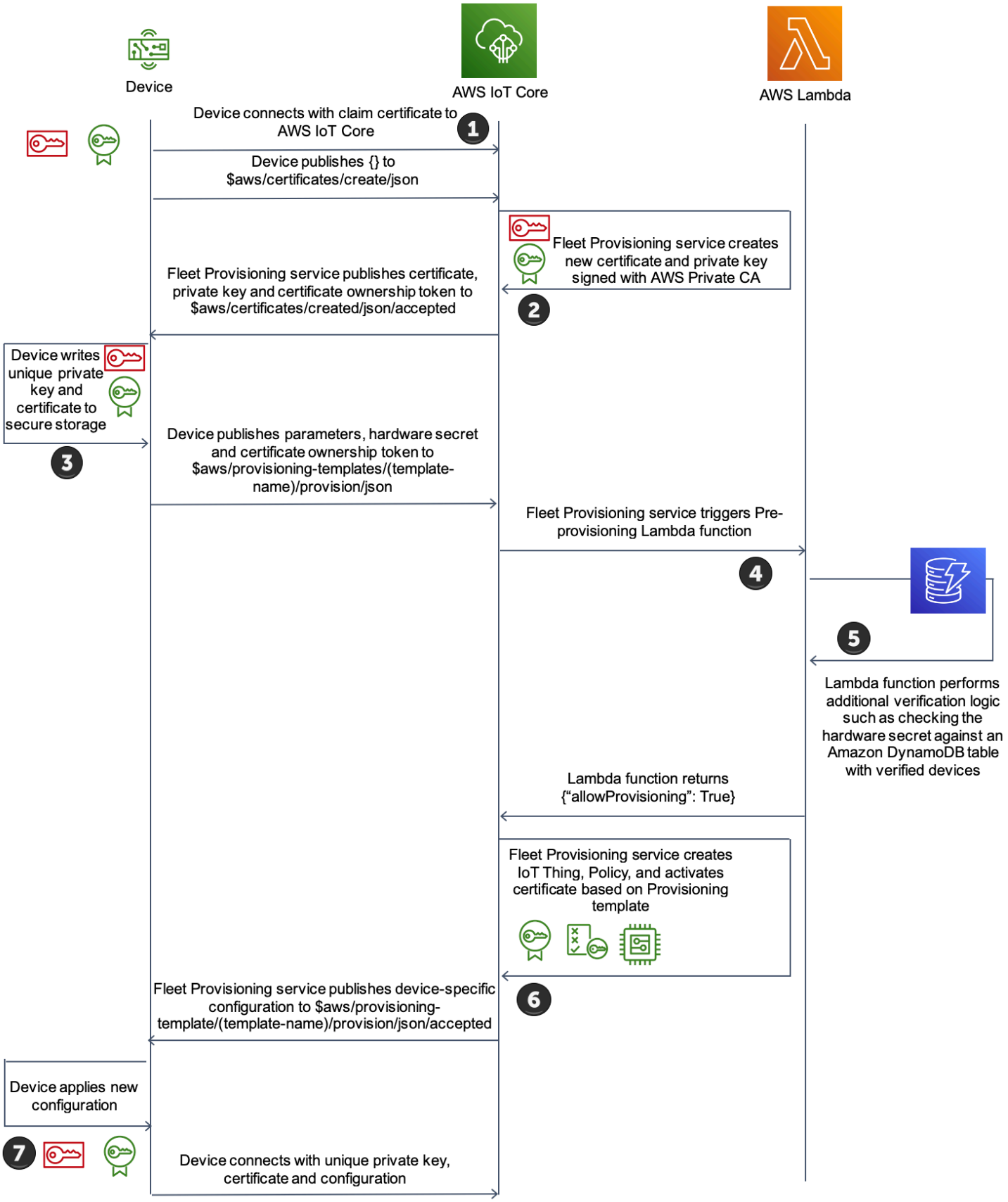

Device
AWS IoT Core
AWS Lambda
Device connects with claim certificate to AWS IoT Core
1
Device publishes {} to $aws/certificates/create/json
Fleet Provisioning service creates new certificate and private key signed with AWS Private CA
2
Fleet Provisioning service publishes certificate, private key and certificate ownership token to $aws/certificates/created/json/accepted
Device writes unique private key and certificate to secure storage
3
Device publishes parameters, hardware secret and certificate ownership token to $aws/provisioning-templates/(template-name)/provision/json
Fleet Provisioning service triggers Pre-provisioning Lambda function
4
5
Lambda function performs additional verification logic such as checking the hardware secret against an Amazon DynamoDB table with verified devices
Lambda function returns {“allowProvisioning”: True}
Fleet Provisioning service creates IoT Thing, Policy, and activates certificate based on Provisioning template
6
Fleet Provisioning service publishes device-specific configuration to $aws/provisioning-template/(template-name)/provision/json/accepted
Device applies new configuration
7
Device connects with unique private key, certificate and configuration

Figure 1: Registration Flow

1. Device connects with claim certificate to AWS IoT Core.
2. The fleet provisioning service creates a new certificate and private key assigned with AWS Private CA.
3. Device writes the unique private key and certificate to secure storage.
4. With the parameters published from the device, the fleet provisioning service starts the pre-provisioning Lambda function.
5. The Lambda function performs additional verification logic, such as checking the hardware secret against a DynamoDB table with verified devices.
6. The fleet provisioning service create IoT thing, policy, and activates certificate based on provisioning template and publishes this to the device.
7. Device applies the new configuration and connects with the unique private key, certificates, and configuration.

Device telemetry

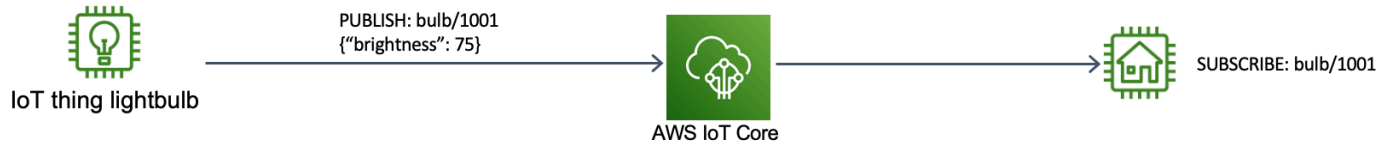
There are many use cases where the value for IoT is in collecting telemetry on how a machine or asset is performing. For example, this data can be used to predict costly, unforeseen equipment failures. Telemetry must be collected from the machine and sent to an IoT application. Another benefit of sending telemetry is the ability of your cloud applications to use this data for analysis and to interpret optimizations that can be made to your firmware over time.

Read-only telemetry data is collected and transmitted to the IoT application. Telemetry data and command and control should be kept on separate MQTT topic namespaces. Telemetry data topics should start with a prefix such as `data`, for example, `data/device/sensortype`. Control plane topics should start with a prefix such as `command`.

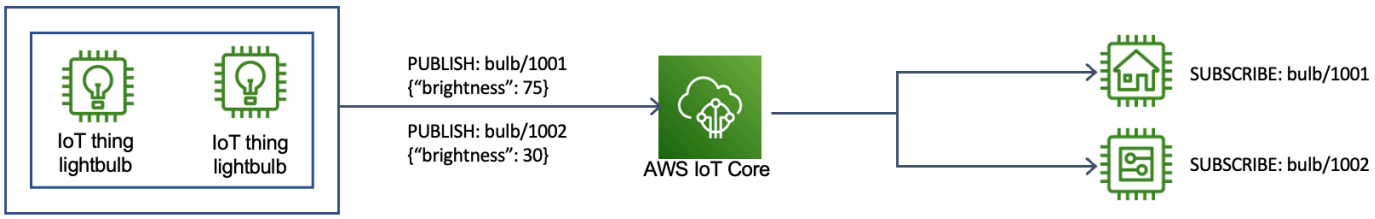
From a logical perspective, we have defined several scenarios for capturing and interacting with device data telemetry.

Options for capturing device data telemetry:

1. One publishing topic and one subscriber. For example, a smart light bulb that publishes its brightness level to a single topic where only a single application can subscribe.

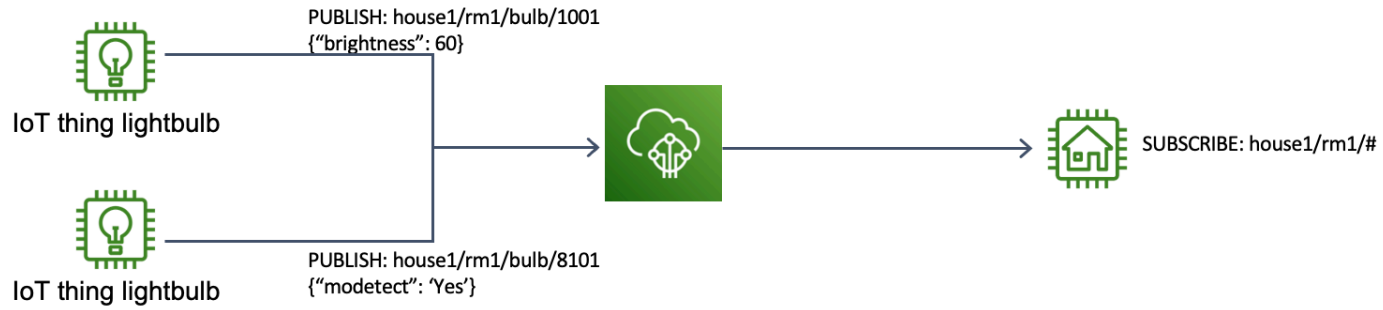


2. One publishing topic with variables and one subscriber. For example, a collection of smart bulbs publishing their brightness on similar but unique topics. Each subscriber can listen to a unique publish message.

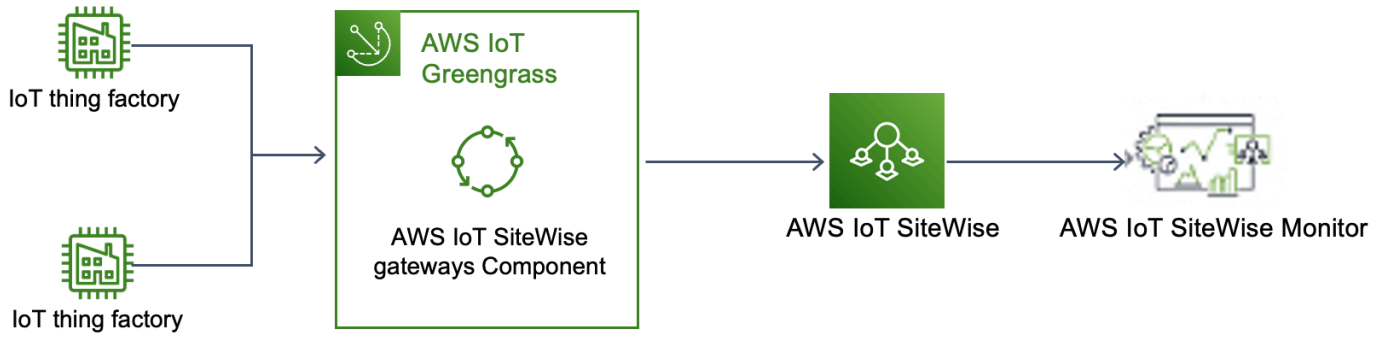


3. Single publishing topic and multiple subscribers. In this case, a light sensor that publishes its values to a topic that all the light bulbs in a house subscribe to.

4. Multiple publishing topics and a single subscriber. For example, a collection of light bulbs with motion sensors. The smart home system subscribes to all of the light bulb topics, inclusive of motion sensors, and creates a composite view of brightness and motion sensor data.



5. AWS IoT SiteWise Edge software running on an edge gateway is used to collect, organize, process, and monitor equipment data on premises and sends it to AWS IoT SiteWise for data storage, organization and visualization.



Other AWS IoT services which can be used for device telemetry data ingestion are AWS IoT SiteWise for industrial data and AWS IoT FleetWise for vehicle data.

Device commands

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

With AWS IoT Core, you can use the bi-directional MQTT protocol to implement command and control of devices. The device subscribes to a specific command MQTT topic. When the device receives a command message, it should verify that the message arrived in the correct order by implementing a sequential ID. The device should then perform the action, and publish a message to the cloud with the results of the command. This ensures that commands are acted upon in order, and the device's current state is always known and maintained in the cloud.

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

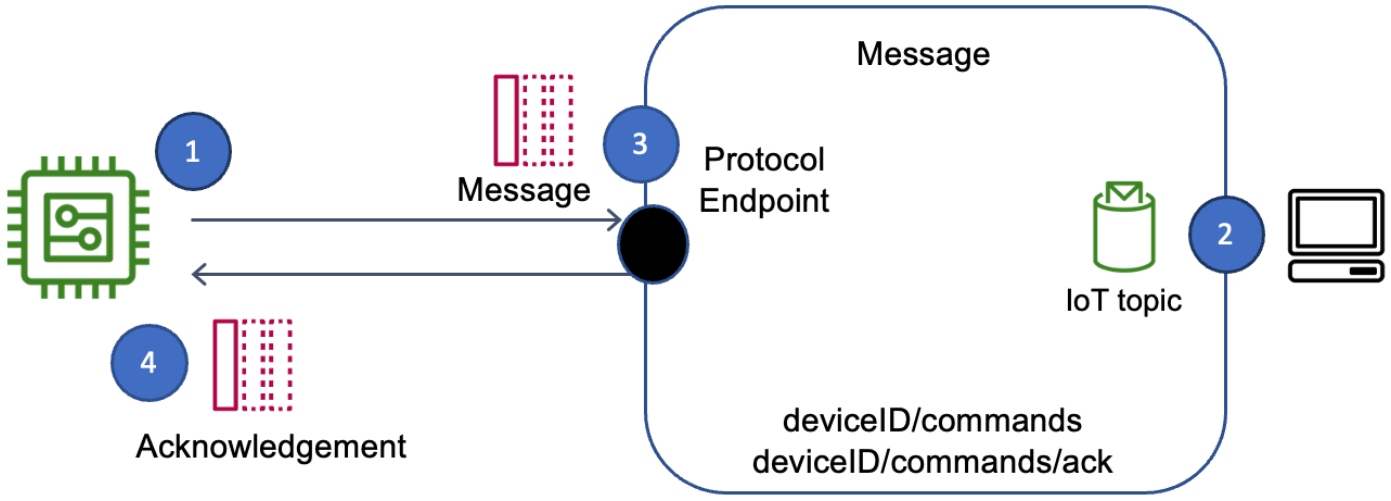


Figure 2: Using a message broker to send commands to a device

AWS IoT Device Shadow service

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

Figure 3: Using AWS IoT Device Shadow service with devices.

1. The device should check its desired state as soon as it comes online by subscribing to the `$aws/things/<<thingName>>/shadow/name/<<shadowName>>/get` topic. A device reports initial device state by publishing that state as a message to the update topic `$aws/things/<<thingName>>/shadow/name/<<shadowName>>/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 `$aws/things/<<thingName>>/shadow/name/<<shadowName>>/update/delta` upon which device-related state change messages will arrive.
4. A component of the solution publishes a desired state message to the topic `$aws/things/<<thingName>>/shadow/name/<<shadowName>>/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 `$aws/things/<<thingName>>/shadow/name/<<shadowName>>/update/delta`, and the Message Broker sends the message to the device.
6. A device receives the delta message and performs the desired state changes.
7. A device publishes an acknowledgment message reflecting the new state to the update topic `$aws/things/<<thingName>>/shadow/name/<<shadowName>>/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 `$aws/things/<<thingName>>/shadow/name/<<shadowName>>/update/accepted` topic.
9. A component of the solution can now request the updated state from the Device Shadow.

AWS IoT Jobs for device commands

In addition to the features described previously for device commands, you can also use AWS IoT Jobs to create a *command pipeline*, where the device infers the command from the payload of the MQTT message, as opposed to the topic. This enables you to perform new kinds of remote operations with minimal device-side code changes. You can control the rate of roll-outs using Jobs, and provide abort, retry, and timeout criteria to further customize the behavior of the job. AWS IoT Jobs integrates with fleet indexing and thing groups, which allows you to search your fleet and target devices in your fleet that meet specific criteria. With *job templates*, you can pre-define all kinds of device commands and create a library of reusable commands with just a few clicks on the target of your choice.

Firmware updates

Supporting firmware updates without human intervention is critical for security, scalability, and delivering new capabilities.

AWS IoT Device Management provides a secure and easy way for you to manage IoT deployments including executing and tracking the status of firmware updates. AWS IoT Device Management uses the MQTT protocol with AWS IoT message broker and AWS IoT Jobs to send firmware update commands to devices, as well as to receive the status of those firmware updates over time. AWS IoT Jobs also integrates with AWS Signer to provide additional security to prevent unauthorized firmware updates and man in the middle attacks. AWS Signer is a fully managed code-signing service to ensure the trust and integrity of your code. With AWS Signer, you can validate code against a digital signature to confirm that the code is unaltered and from a trusted publisher. Firmware images can be signed with a private key in the cloud using the code signing feature, and the device verifies the integrity of that firmware image with the corresponding public key.

To implement firmware updates using AWS IoT Device Management and AWS IoT Jobs, see the following diagram.

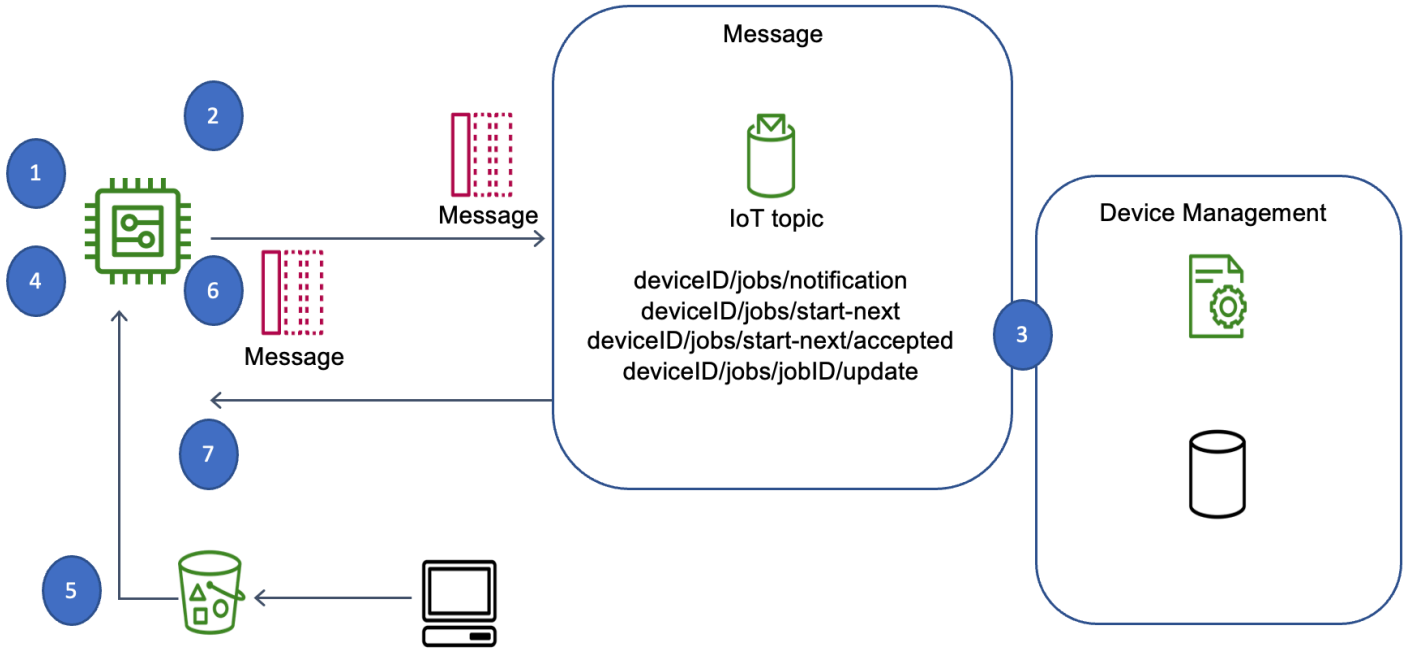


Figure 4: Updating firmware on devices.

1. A device subscribes to the IoT job notification topic `$aws/things/<<thingName>>/jobs/notify-next` upon which IoT job notification messages will arrive.

2. A device publishes a message to `$aws/things/<<thingName>>/jobs/start-next` to start the next job and get the next job, its job document, and other details including any state saved in statusDetails.
3. The AWS IoT Jobs service retrieves the next job document for the specific device and sends this document on the subscribed topic `$aws/things/<<thingName>>/jobs/start-next/accepted` .
4. A device performs the actions specified by the job document using the `$aws/things/<<thingName>>/jobs/jobId/update` MQTT topic to report on the progress of the job.
5. During the upgrade process, a device downloads firmware using a pre-signed URL for Amazon S3. Use code-signing to sign the firmware when uploading to Amazon S3. By code-signing your firmware the end-device can verify the authenticity of the firmware before installing. FreeRTOS devices can download the firmware image directly over MQTT to eliminate the need for a separate HTTPS connection.
6. The device publishes an update status message to the job topic `$aws/things/<<thingName>>/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 running (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.

Pillars

Operational excellence pillar

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

Topics

Design principles

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

- **Plan for device provisioning**: Design your device provisioning process to create your initial device identity in a secure location. Implement a public key infrastructure (PKI) that is responsible for distributing unique certificates to IoT devices. As described above, selection of crypto hardware with a pre-generated private key and certificate eliminates the operational cost of running a PKI. Otherwise, PKI can be done offline with a hardware security module (HSM) during the manufacturing process, or during device bootstrapping. Use technologies that can manage the Certificate Authority (CA) and HSM in the cloud.
- **Implement device bootstrapping**: Devices that support personalization by a technician (in the industrial domain) or user (in the consumer domain) can also undergo provisioning. For example, a smartphone application that interacts with the device over Bluetooth LE and with the cloud over Wi-Fi. You must design the ability for devices to programmatically update their configuration information using a globally distributed bootstrap API. A bootstrapping design ensures that you can programmatically send the device new configuration settings through the cloud. These changes should include settings such as which IoT endpoint to communicate with, how frequently to send an overall status for the device, and any updated security settings such as server certificates. The process of bootstrapping goes beyond initial provisioning and plays a critical role in device operations by providing a programmatic way to update device configuration through the cloud. A bootstrapping API and endpoint must be available for the entire defined life of all devices, and must be able to respond to requests for all versions of firmware that have ever been deployed on a device.
- **Document device communication patterns**: In an IoT application, device behavior is documented by hand at the hardware level. In the cloud, an operations team must formulate how the behavior of a device will scale once deployed to a fleet of devices. A cloud engineer should review the device communication patterns and extrapolate the total expected inbound and outbound traffic of device data and determine the expected infrastructure necessary in the cloud to support the entire fleet of devices. During operational planning, these patterns should be measured using device and cloud-side metrics to ensure that expected usage patterns are met in the system.
- **Implement over-the-air (OTA) updates**: To benefit from long-term investments in hardware, you must be able to continually update the firmware on the devices with new capabilities. In the cloud, you can apply a robust firmware update process that allows you to target specific devices for firmware updates, roll out changes over time, track success and failures of updates, and have the ability to roll back or put a stop to firmware changes based on key performance indicators (KPIs).
- **Implement functional testing on physical assets**: IoT device hardware and firmware must undergo rigorous testing before being deployed in the field. Acceptance and functional testing

are critical for your path to production. The goal of functional testing is to run your hardware components, embedded firmware, and device application software through rigorous testing scenarios, such as intermittent or reduced connectivity or failure of peripheral sensors, while profiling the performance of the hardware. The tests ensure that your IoT device will perform as expected when deployed.

- **Design and build for operations at scale**: Design and build a solution for logging, monitoring, troubleshooting, fleet management, life cycle device and application management at scale.

Best practices

There are three best practice areas for operational excellence:

Topics

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.

Prepare

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

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

When using an IoT platform, you have the opportunity to use additional components and 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 of device data.

IOTOPS 1. What factors drive your operational priorities?

IOTOPS 2. How are you ensuring that newly provisioned devices have the required operation al prerequisites?

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

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

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

Operate

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

implementation time to ensure your ability to inspect, analyze, and act on metrics sent from these remote devices.

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

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

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

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

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

```
{
```

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

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

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

In addition to custom dashboards, AWS IoT provides fleet-level and device-level insights driven from the thing registry and AWS IoT 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 3. How do you evolve your IoT application with minimum impact to downstream IoT devices?

IoT solutions frequently involve a combination of low-power devices, remote locations, low bandwidth, and intermittent network connectivity. Each of those factors poses communications challenges, including upgrading firmware and edge applications. Therefore, it's important for you

to incorporate and implement an IoT update process that minimizes the impact to downstream devices and operations. In addition to reducing downstream impact, devices must be resilient to common challenges that exist in local environments, such as intermittent network connectivity and power loss. Use a combination of grouping IoT devices for deployment and staggering firmware upgrades over a period of time. Monitor the behavior of devices as they are updated in the field, and proceed only after a percentage of devices have upgraded successfully.

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

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

IOTOPS 4. How do you ensure that you are ready to support the operations of devices in your IoT workload?

Operating IoT workloads at scale is different than testing and running prototypes. You need to ensure that your team is prepared and trained to operate a widely distributed IoT data collection application. IoT workloads require your teams to learn new skills and competencies to deliver edge-to-cloud outcomes. Your team needs to be able to pinpoint key operational thresholds that indicate a high level of readiness.

Best practice IOTOPS_4.1 – Train team members supporting your IoT workloads on the lifecycle of IoT applications and your business objectives

Key team members responsible for IoT workloads are trained on major IoT lifecycle events: onboarding, command and control, security, data ingestion, integration, and analytics services. Team members should be able to identify key operational metrics and know how to apply incident response measures. Training team members on the basics of IoT lifecycles and how these align with business objectives provides actionable context on failure scenarios, mitigation strategies, and defining lasting processes that effectively contribute to fewer operational events and less severe impact during events.

Recommendation IOTOPS_4.1.1 – *Build IoT operational expertise by having team members and architects' complete reviews of common IoT architectural patterns, best practices, and educational courses*

- Introduce new team members to IoT lifecycles with onboarding checklists that include at least one educational course.
- Introduce new team members with onboarding checklists that include a step to review, validate, and submit updates to your IoT application architecture documentation and operational monitoring plan.

Recommendation IOTOPS_4.1.2 – *Author runbooks for each component of the architecture and train team members on their use*

- Include guidance for a response procedure for remote devices that are no longer online.
- Apply recovery commands for troubleshooting remote devices that are faulty but still online.

IOTOPS 5. How do you assess whether your IoT application meets your operational goals?

Evaluating your operational goals enables you to fine-tune and identify improvements throughout the lifecycle of your IoT application. Measuring and extracting operational and business value from your IoT application allows you to effectively drive high-value initiatives.

Best practice IOTOPS_5.1 – Enable appropriate responses to events

Key operational data elements are those data points that convey some notion of operational health of your application by classifying events. Detecting operational events early can uncover unforeseen risks in your application and give your operations team a head start to prevent or reduce significant business interruption. By defining a minimum set of logs, metrics, and alarms, your operations team can provide a first line of defense against significant business interruption.

Recommendation IOTOPS_5.1.1 – *Configure logging to capture and store at least error-level events*

- Use AWS IoT service logging options to capture error events in CloudWatch Logs

Recommendation IOTOPS_5.1.2 – *Create a dashboard for your responders to use in investigations of operational events to rapidly pinpoint the period of time when errors are logged*

- Group clusters of error events into buckets of time to quickly identify when surges of errors were captured.
- Drill down into clusters of errors to identify any patterns to signal for triage response.

Recommendation IOTOPS_5.1.3 – *Review the default metrics emitted by your IoT services and configure alarms for metrics that might indicate business interruption*

- For example:
  - Your business deploys a thousand sensors across manufacturing plants and your operations team wants to be alerted if sensors can no longer connect to the cloud and send telemetry.
  - Your IT team administering the AWS account reviews the AWS IoT Core metrics and notes the following metrics to monitor: `Connect.AuthError`, `Connect.ClientError`, `Connect.ClientIDThrottle`, `Connect` `Connect.Throttle`. Activity in any of these metrics constitutes alerting and investigation.
  - Your IT team uses CloudWatch to configure new alarms on these metrics when for any period the metrics' SUM of Count is greater than zero.
  - Your IT team configures an Amazon SNS topic to notify their paging tool when any of the new CloudWatch alarms changes status.
- For more information:
  - Monitor AWS IoT alarms and metrics using Amazon CloudWatch

Recommendation IOTOPS_5.1.4 – *Configure an automated monitoring and alerting tool to detect common symptoms and warnings of operational impact*

- For example:
  - Configure AWS IoT Device Defender to run a daily audit on at least the high and critical checks.
  - Configure an Amazon SNS topic to notify a team email list, paging tool, or operations channel when AWS IoT Device Defender reports non-compliant resources in an audit.
- For more information:
  - AWS IoT Device Defender Audit

IOTOPS 6. How do you govern device fleet provisioning process?

IoT solutions can scale to millions of devices and this requires device fleets to be well planned from the perspectives of provisioning processes and metadata organization. Defining how devices are provisioned must include how the devices are manufactured and how they are registered. Maintain a full chain of security controls over who or what processes can start device provisioning to decrease the likelihood of inviting unintended, or rogue, devices into your fleet.

Best practice IOTOPS_6.1 – Document how devices join your fleet from manufacturing to provisioning

Document the whole device provisioning process to clearly define the responsibilities of different actors at different stages. The end-to-end device provisioning process involves multiple stages owned by different actors. Documenting the plan and processes by which devices onboard and join the fleet affords the appropriate amount of review for potential gaps.

Recommendation IOTOPS_6.1.1 – *Document each step (manual and programmatic) of all the stages for the corresponding actors of that stage and clearly define the sequence*

- Identify the steps at each stage and the corresponding actors.
  - Device assembly by hardware manufacturer.
  - Device registration by service and solution provider.
  - Device activation by the end user of the service or solution provider.
- Clearly define and document the dependencies and specific steps for each actor from device manufacturer to the end user.
- Document whether devices can self-provision or are user-provisioned and how you can ensure that newly provisioned devices are yours.

Recommendation IOTOPS_6.1.2 – *Assign device metadata to enable easy grouping and classification of devices in a fleet*

- The metadata can be used to group the devices in groups to search and force common actions and behaviors.
- For example, you can assign the following metadata at the time of manufacturing:
  - Unique ID

- Manufacturer details
- Model number
- Version or generation
- Manufacturing date

- If a particular model of a device requires a security patch, then you can easily target the patch to all the devices that are part of the corresponding model number group. Similarly, you can apply the patches to devices manufactured in a specific time frame or belonging to a particular version or generation.

Best practice IOTOPS_6.2 – Use programmatic techniques to provision devices at scale

Scaling the onboarding and provisioning of a large device fleet can be a bottleneck if there is even one manual step per device. Programmatic techniques define patterns of behavior for automating the provisioning process such that authenticated and authorized devices can onboard at any time. This practice ensures a well-documented, reliable, and programmatic provisioning mechanism that is consistent across all devices devoid of any human errors.

Recommendation IOTOPS_6.2.1 – *Embed provisioning claims into the devices that are mapped to approval authorities recognized by the provisioning service*

- Generate a provisioning claim and embed it into the device at the time of manufacturing.
- AWS IoT Core can generate and securely deliver certificates and private keys to your devices when they connect to AWS IoT for the first time, using AWS IoT fleet provisioning.

Recommendation IOTOPS_6.2.2 – *Use programmatic bootstrapping mechanisms if you are bringing your own certificates*

- Determine if you will or won't have device information beforehand
- If you don't have device information beforehand, use just-in-time provisioning (JITP).
  - Enable automatic registration and associate a provisioning template with the CA certificate used to sign the device certificate.
  - For example, when a device attempts to connect to AWS IoT by using a certificate signed by a registered CA certificate, AWS IoT loads the template from the certificate and initiates the JITP workflow.
- If you have device information beforehand, use bulk registration.

- Specify a list of single-thing provisioning template values that are stored in a file in an S3 bucket.
- Run the start-thing-registration-task command to register things in bulk. Provide provisioning template, S3 bucket name, a key name, and a role ARN to the command.

Best practice IOTOPS_6.3 – Use device level features to enable re-provisioning

A birth or bootstrap certificate is a low-privilege unique certificate that is associated with each device during the manufacturing process. The certificate should have a policy to restrict devices to only allow connecting to specific endpoints to initiate provisioning process and fetch the final certificate. Before a device is provisioned, it should be limited in functionality to prevent its misuse. Only after a provisioning process is invoked and approved, should the device be allowed to operate on the system as designed.

Recommendation IOTOPS_6.3.1 – *Use a certificate bootstrapping process to establish processes for device assembly, registration, and activation*

- For example, AWS IoT Core offers a fleet provisioning interface to devices for upgrading a birth certificate to long-lived credentials that enable normal runtime operations.

Recommendation IOTOPS_6.3.2 – *Obtain a list of allowed devices from the device manufacturer*

- Check the allow list file to validate that the device has been fully vetted by the supplier.
- Ensure that the list is encrypted, securely stored, and can only be accessed by necessary services and users. Even if the list changes, keep the original list securely stored.
- Ensure that this list is securely transferred from the manufacturer to you, is encrypted, and is not publicly accessible.
- Ensure that any bootstrap certificate used is signed by a certificate authority (CA) you own or trust.

Best practice IOTOPS_6.4 – Use data-driven auditing metrics to detect if any of your IoT devices might have been broadly accessed

Monitor and detect the abnormal usage patterns and possible misuse of devices and automate the quarantine steps. Programmatic methods to detect and quarantine devices from interacting with cloud resources enable teams to operate a fleet in a scalable way while minimizing a dependency on active human monitoring.

Recommendation IOTOPS_6.4.1 – *Use monitoring and logging services to detect anomalous behavior*

Once you detect the compromised device, run programmatic actions to quarantine it.

- Disable the certificate for further investigation and revoke the certificate to prevent the device from any future use.
- Use AWS IoT CloudWatch metrics and logs to monitor for indications of misuse. If you detect misuse, quarantine the device so it does not impact the rest of the platform.
- Use AWS IoT Device Defender to identify security issues and deviations from best practices.

IOTOPS 7. Do you organize the fleet to quickly identify devices?

The ability to quickly identify and interact with specific devices gives you the agility to troubleshoot and potentially isolate devices in case you encounter operational challenges. When operating large-scale device fleets, you need to deploy ways to organize, index, and categorize them. This is useful when targeting new device software with updates and when you need to identify why some devices in your fleet behave differently than others.

Best practice IOTOPS_7.1 – Use static and dynamic device hierarchies to support fleet operations

Using a software registry, devices can be categorized into static groups based on their fixed attributes (such as version or manufacturer) and into dynamic groups based on their changing attributes (such as battery percentage or firmware version). Operationalizing devices in groups can help you manage, control, and search for devices more efficiently.

Recommendation IOTOPS_7.1.1 – *Manage several devices at once by categorizing them into static groups and hierarchy of groups*

- Build a hierarchy of static groups for efficient categorization and indexing of your devices.
- Use provisioning templates to assign devices to static groups as they are provisioned for the first time.
- For example, categorize all sensors of a car under a car group and all the cars under a vehicle group. Child groups inherit policies and permissions attached to their respective parent groups.

Recommendation IOTOPS_7.1.2 – *Build a device index to efficiently search for devices, and aggregate registry data, runtime data, and device connectivity data*

- Use a fleet indexing service to index device and group data.
- Use a device index to search registry metadata, stateful metadata, and device connectivity status metadata.
- Use a group index to search for groups based on group name, description, attributes, and all parent group names.
- For example, if you want to send over-the-air (OTA) updates only to devices that are sufficiently charged, then define a dynamic group for devices with more than 90% battery. Devices will dynamically be added to or removed from the group as their battery percentage crosses the threshold. Send OTA updates to all things under this dynamic group

Best practice IOTOPS_7.2 – Use index and search services to enable rapid identification of target devices

A large IoT deployment can have millions of sensors sending data to the cloud. A separate indexing and search service can make it easy to index and organize the device data, and search for any device by any attribute. Ingesting device data to a search service, for example, Amazon OpenSearch Service (OpenSearch Service), makes it easy to use powerful search, visualization, and analytics capabilities to organize and search for devices. You can ingest your device data and the state to OpenSearch Service seamlessly.

Recommendation IOTOPS_7.2.1 – *Use an indexed data store to get, update, or delete device state*

- Use messaging topics to enable applications and things to get, update, or delete the state information for a Thing (Thing Shadow).
- Ingest the shadow data to Firehose through the AWS IoT Core rules engine.
- Ingest the data from Firehose to OpenSearch Service through built-in destination options.
- Configure search and visualizations on the data directly or through the OpenSearch Dashboards console.
- For more information:
  - AWS IoT Core - Fleet indexing service
  - AWS IoT Core - AWS IoT Device Shadow service
  - What is Amazon OpenSearch Service?

- The Internet of Things on AWS – Official Blog: Archive AWS IoT Device Shadows in Amazon OpenSearch Service
- Analyze device-generated data with AWS IoT and Amazon OpenSearch Service

IOTOPS 8. How do you monitor the status of your IoT devices?

You need to be able to track the status of your devices. This includes operational parameters and connectivity. You need to know whether devices have disconnected intentionally or not. Monitoring the status of your device fleet is important in helping troubleshoot device software operation and connectivity disruptions.

Best practice IOTOPS_8.1 – Collect lifecycle events from the device fleet

Design a state machine for the device connectivity states, from initialization and first connection, to frequent communication (like keep-alive messages) and final state before going offline. Lifecycle events, such as connection and disconnection, can be used to observe and analyze device behavior over a period of time. Additionally, periodic keep-alive messages can track device connectivity status.

Recommendation IOTOPS_8.1.1 – *Subscribe to lifecycle events and monitor the connections using keep-alive messages:*

- Capture messages from the IoT message broker whenever a device connects or disconnects. Being able to tell the difference between a clean and dirty disconnect is useful in many scenarios where the devices don't maintain a constant connection to the broker.
- Based on the use case and device constraints, have the device send periodic keep-alive messages to AWS IoT Core and monitor, and analyze the keep-alive messages for anomalies.
- Ensure that the frequency of sending keep-alive messages is not causing any network storms (perhaps by introducing some jitter) in the network or causing rate limits.

Best practice IOTOPS_8.2 – Configure your devices to communicate their status periodically.

Implement Last Will and Testament (LWT) messages and periodic device keep-alive messages.

Recommendation IOTOPS_8.2.1 – *Implement a well-designed device connectivity state machine*

- Ensure that the device communicates when it first comes online and just prior to going offline.
- Implement a wait state for lifecycle events. When a disconnect message is received, wait a period of time and verify that the device is still offline before taking action.
- Specify the interval with which each connection should be kept open if no messages are received. AWS IoT drops the connection after that interval unless the device sends a message or a ping.

Recommendation IOTOPS_8.2.2 – *Use device connection and disconnection status for anomaly detection*

- Use connectivity data patterns from devices to detect anomalous behavior in their communication patterns.
- For more information:
  - AWS IoT Now Supports WebSockets, Custom Keepalive Intervals, and Enhanced Console
  - AWS IoT Device Defender Now Provides Statistical Anomaly Detection and Data Visualization
  - AWS Solutions Library: Real-Time IoT Device Monitoring with Managed Service for Apache Flink

Best practice IOTOPS_8.3 – Use device state management services to detect status and connectivity patterns

Edge and cloud-side management services monitor and analyze parameters, such as device connectivity status and latency, to help in providing diagnostics and predicting anomalies.

Recommendation IOTOPS_8.3.1 – *Monitor device state and connectivity patterns*

- Use (or develop as needed) device, gateway, edge, and cloud management tools that allow monitoring the fleet of devices
- Use logging and monitoring features at all processing points—device, gateway, edge, and cloud, to get a complete picture of device connectivity patterns.

  For more information:

  - AWS IoT Core - Managing thing indexing

IOTOPS 9. How do you segment your device operations in your IoT application?

You need to segment your device fleet to pinpoint operational challenges and direct incident response to the appropriate responder. Device fleet segmentation enables you to identify conditions under which devices operate sub optimally and minimize response time to security events.

Best practice IOTOPS_9.1 – Use static and dynamic device attributes to identify devices with anomalous behavior

Anomalies in fleet operations might only surface when analyzing metrics that aggregate across the boundaries of your static and dynamic groups or attributes. For example, devices that are running firmware version 2.0.10 and currently have a battery level over 50%. Static and dynamic groups allow for identifying and pinpointing devices in unique ways to monitor, analyze, and take corrective actions on device behavior.

Recommendation IOTOPS_9.1.1 – *Pinpoint devices with unusual communication patterns*

- Use a combination of static and dynamic groups of devices to perform fleet indexing to group devices and identify behavioral patterns—connectivity status, message transmission, etc.
- Use lifecycle events, device connectivity, and data transmission patterns to detect anomalies and pinpoint unusual behavior using techniques such as statistical anomaly detection (for large fleet of devices).
- Once abnormal behavior has been identified, move rogue and abnormal devices into a different group so that remedial policies can be assigned and implemented on them.

  For more information:

  - AWS IoT Core - Authorization
  - AWS IoT - Device Defender

Key AWS services

Several services can be used to drive operational excellence for your IoT application. The AWS Device Qualification Program for FreeRTOS helps you select hardware components that have been designed and tested for AWS IoT interoperability. Qualified hardware can get you to market faster and reduce operational friction. AWS IoT Core offers features used to manage the initial

onboarding of a device. With AWS IoT Greengrass, you can run any kind of custom compute, such as AWS Lambda functions and Docker containers on the device to respond quickly to local events, interact with local resources, and process data to minimize the cost of transmitting data to the cloud and simplify remote application management. AWS IoT Device Management reduces the operational overhead of performing fleet-wide operations, such as device grouping and searching. In addition, Amazon CloudWatch is used for monitoring IoT metrics, collecting logs, generating alerts, and triggering responses. Other services and features that support the three areas of operational excellence are as follows:

- **Preparation: AWS IoT Device Tester (IDT) for FreeRTOS and AWS IoT Greengrass** is a test automation tool for connected devices to determine if your devices running FreeRTOS or AWS IoT Greengrass can interoperate with AWS IoT Services.
- **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 multi-account registration. 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.
- **Operations: AWS IoT Greengrass** provides many pre-built capabilities on the device to help you focus mostly on their business logic and offers many foundational infrastructure and operation features such as remote application management.
- **Responses**: **AWS IoT Jobs** enables you to proactively push updates to one or more devices such as firmware updates or device configuration. AWS IoT rules engine allows you to inspect IoT messages as they are received by AWS IoT Core and immediately respond to the data, at the most granular level. AWS IoT Analytics and AWS IoT Device Defender enable you to proactively trigger notifications or remediation based on real-time analysis with AWS IoT Analytics, and real-time security and data thresholds with Device Defender.

Resources

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

Documentation and blogs:

- Remote asset health monitoring

- Monitoring AWS IoT connections in near real time
- Managing IoT device certificate rotation
- Configuring near real-time notification for asset-based monitoring with AWS IoT Events
- Use AWS IoT service logging options to capture error events in CloudWatch Logs

Security pillar

The security pillar includes the ability to protect information, systems, and assets while delivering business value. This section provides in-depth, best-practice guidance for architecting secure IoT workloads on AWS.

Topics

- Design principles
- Best practices
- Key AWS services
- Resources

Design principles

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

- **Manage device security lifecycle holistically**: Device security starts at the design phase, and ends with the retirement and destruction of the hardware and data. It is important to take an end-to-end approach to the security lifecycle of your IoT solution to maintain your competitive advantage and retain customer trust.
- **Ensure least privilege permissions**: Devices should all have fine-grained access permissions that limit which topics a device can use for communication. By restricting access, one compromised device will have fewer opportunities to impact any other devices.
- **Secure device credentials at rest**: Devices should securely store credential information at rest using mechanisms such as a dedicated crypto element or secure flash.
- **Implement device identity lifecycle management**: Devices maintain a device identity from creation through end of life. A well-designed identity system will keep track of a device's identity, track the validity of the identity, and proactively extend or revoke IoT permissions over time.

- **Take a holistic view of data security**: IoT deployments involving a large number of remotely deployed devices present a significant attack surface for data theft and privacy loss. Use a model such as the Open Trusted Technology Provider Standard to systemically review your supply chain and solution design for risk and then apply appropriate mitigations.
- **Preserve safety and reliability in critical OT/IIoT environments**: IIoT cybersecurity differs from the IT cybersecurity model because it is not only concerned with data protection, but also with the preservation of safety and reliability of production systems and ensuring environmental health and safety (EHS) in industrial facilities.
- **Implement zero trust principles as per NIST SP 800-207**: Zero trust isn't limited to traditional IT, and extends to IoT, operational technology (OT) and IIoT. A zero-trust model can significantly improve your organization's security posture by eliminating the sole reliance on perimeter-based protection. This doesn't mean getting rid of perimeter security altogether. Where possible, use identity and network capabilities together to protect core assets and apply zero trust principles working backwards from specific use cases with a focus on extracting business value and achieving measurable business outcomes.
- **Establish secure connection with AWS via Site-to-Site VPN or Direct Connect from the industrial edge**

  For IIoT workloads, AWS offers multiple ways and design patterns to establish a secure connection to the AWS environment from the industrial edge. Establish a secure VPN connection to AWS over the internet or set up a dedicated private connection via Direct Connect. Use AWS VPN with Direct Connect to encrypt traffic over Direct Connect.

- **Use VPC Endpoints whenever possible**

  For IIoT workloads, after a secure connection to AWS has been established via VPN over the internet or Direct Connect, use VPC Endpoints whenever possible. VPC Endpoints enables you to privately connect to supported regional services without requiring a public IP address. Endpoints also support endpoint policies, which further allow to control and limit access to only the required resources.

- **Use HTTP over TLS proxy and a firewall for services connecting to AWS over the internet**

  If the VPC Endpoint for the required service is not available, you would have to establish a secure connection over the internet. The best practice in such a scenario is to route these connections via a HTTP connection over a TLS proxy and a firewall. Using a proxy allows the cloud traffic

to be inspected and monitored and enables threat and malware detection. It also allows the security policies to be applied at the network layer. Firewall rules can be established for HTTPS and MQTT traffic to securely connect to AWS IoT services over the public internet.

- **Use secure protocols whenever possible and when using insecure protocols, convert insecure protocols into standardized and secure protocols as close to the source as possible**

  In most environments, prefer to use secure protocols which support encryption. When using secure protocols is not an option, tighten the trust boundaries as described in the next point.

- **Use network segmentation and tighten trust boundaries**

  Follow the micro segmentation approach, that is, build small islands of components within a single network that communicate only with each other and control the network traffic between segments. Select the newer version of industrial protocols which offer security features and configure the highest level of encryption available when using industrial control system (ICS) protocols such as CIP Security, Modbus Secure, and OPC UA. When using secure industrial protocols is not an option, tighten the trust boundary using a protocol converter to translate the insecure protocol to a secure protocol as close to the data source as possible.

  Alternatively, segregate the plant network into smaller cell or area zones by grouping ICS devices into functional areas to limit the scope and area of insecure communications. Use unidirectional gateways and data diodes for one-way data flow and specialized firewall and inspection products that understand ICS protocols to inspect traffic entering and leaving cell/area zones and can detect anomalous behavior in the control network.

- **Securely manage and access edge computing resources**

  Keeping computing resources at the industrial edge up to-date, securely accessing to them for configuration and management, or automatically deploying changes can be challenging. AWS provides options to securely manage edge compute resources (AWS System Manager), IoT resources (IoT Device Management, AWS IoT Greengrass) and also provides a fully managed infrastructure service (AWS Outposts) to make it easy to consistently apply best practices to all resources.

Best practices

There are five best practice areas for security:

Topics

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

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

Identity and access management

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

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

To secure device hardware, you implement solutions such that private keys and sensitive identity are unique to, and only stored on the device in a secure hardware location. Implement hardware or software-based modules that securely store and manage access to the device's private key corresponding to its public key and X.509 certificate. FreeRTOS, AWS IoT Greengrass, and the AWS IoT Device SDKs support this through the use of PKCS#11. In addition to hardware security, IoT devices must be given a valid identity, which will be used for authentication and authorization in your IoT application.

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

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

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

Devices must support rotation and replacement of certificates to ensure continued operation as certificates expire.

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

To support device identity updates, use AWS IoT Jobs, which is a managed service for distributing OTA communication and binaries to your devices. AWS IoT Jobs is used to define a set of remote operations that are sent to and run on one or more devices connected to AWS IoT. AWS IoT Jobs by

default integrates several best practices, including mutual authentication and authorization, device tracking of update progress, and fleet-wide wide metrics for a given update.

Use native provisioning mechanisms to onboard devices when they already have a device certificate (and associated private key) on them. For example, you can use just-in-time provisioning (JITP) or just-in-time registration (JITR) that provisions devices in bulk when they first connect to AWS IoT.

If the devices cannot use X.509 certificates, or you have an existing fleet of devices with a proprietary access control mechanism, that requires use of bearer tokens such as OAuth over JWT or SAML tokens, use custom auth mechanisms. For example, when a device attempts to connect to AWS IoT, it sends a JWT generated by their identity provider in the HTTP header or query string. The signature is validated by AWS IoT custom authorizer and the connection is established.

It is important to use a standard set of naming conventions when designing device name and MQTT topics. For example, use the same client identifier for the device as the IoT Thing Name. This will also allow to include any relevant routing information for the device in the topic namespace.

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

IOTSEC 1. How do you associate IoT identities and permissions with your devices?

Your application is responsible for managing how your devices authenticate and authorize their interactions. By creating a process that ensures devices have identity-based permissions for accessing the IoT platform, you establish the greatest control for managing device interactions.

Best practice IOTSEC_1.1 – Assign unique identities to each IoT device

When a device connects to other devices or cloud services, it must establish trust by authenticating using principals such as X.509 certificates, security tokens, or other credentials. You can find available options from the IoT solution of your choice, and implement device registry and identity stores to associate devices, metadata and user permissions. The solution should enable each device (or Thing) to have a unique name (or ThingName) in the device registry, and it should ensure that each device has an associated unique identity principal, such as an X.509 certificate or security token. Identity principals, such as certificates, should not be shared between devices. When

multiple devices use the same certificate, this might indicate that a device has been compromised. Its identity might have been cloned to further compromise the system.

Recommendation IOTSEC_1.1.1 – *Use X.509 client certificates to authenticate over TLS 1.2*

We recommend that each device be given a unique certificate to enable fine-grained management, including certificate revocation. Devices must support rotation and replacement of certificates to ensure continued operation as certificates expire. For example, AWS IoT Core supports AWS IoT-generated X.509 certificates or your own X.509 certificates for device authentication.

Recommendation IOTSEC_1.1.2 – *Choose the appropriate certificate vending mechanisms for your use case*

We recommend using native provisioning mechanisms to onboard devices when they already have a device certificate (and associated private key) on them. For example, you can use just-in-time provisioning (JITP) or just-in-time registration (JITR) that provisions devices in bulk when they first connect to AWS IoT.

Recommendation IOTSEC_1.1.3 – *Use security bearer tokens for authorizing to the IoT broker*

If the devices cannot use X.509 certificates, or you have an existing fleet of devices with a proprietary access control mechanism, that requires use of bearer tokens such as OAuth over JWT or SAML tokens, use custom auth mechanisms. For example, when a device attempts to connect to AWS IoT, it sends a JWT generated by their identity provider in the HTTP header or query string. The signature is validated by AWS IoT custom authorizer and the connection is established.

Recommendation IOTSEC_1.1.4 – *Use a consistent naming convention that maps your device identity to the MQTT topics*

It is important to use a standard set of naming conventions when designing device name and MQTT topics. For example, use the same client identifier for the device as the IoT Thing Name. This will also allow to include any relevant routing information for the device in the topic namespace.

IOTSEC 2. How do you secure your devices and protect device credentials?

Your IoT devices and identity principals (certificates, private keys, tokens, etc.) must be secured throughout their lifecycle. To ensure device authenticity, your IoT hardware must securely store, manage, and restrict access to the identities that the device uses to authenticate itself with the

cloud. By securing your devices and storing your device credentials safely, you can reduce the risk of unauthorized users misusing device credentials.

Best practice IOTSEC_2.1 – **Use a separate hardware or a secure area on your devices to store credentials.**

A secure element is any hardware feature you can use to protect the device identity at rest. Secure storage at rest helps reduce the risk of unauthorized use of the device identify. Never store or cache device credentials outside of the secure element. Generate private keys on the HSM, and generate the Certificate Signing Requests from the device. If this is not possible, generate and transmit the credentials to the HSM in a secure manufacturing facility with Common Criteria EAL certification. Securely handling a device's identity helps ensure that your hardware and application are resilient to potential security issues that occur in unprotected systems. A secure element provides encryption of private information (such as cryptographic keys) at rest and can be implemented as separate specialized hardware or as part of a system on a chip (SoC).

Recommendation IOTSEC_2.1.1 – *Use tamper-resistant hardware that offloads the security encryption and communication from the IoT application.*

Device credentials always reside in a secure element, which facilitates any usage of the credentials. Using the secure element to facilitate the use of device credentials further limits the risk of unauthorize use. As an example, AWS IoT Greengrass supports using a secure element to store AWS IoT certificates and private keys.

Recommendation IOTSEC_2.1.2 – *Use cryptographic API operations with the secure element hardware for protecting the secrets on the device*

Ensure that any security modules are accessed using the latest security protocols. For example, in FreeRTOS, use the PKCS#11 APIs provided in the corePKCS11 library for protecting secrets.

Recommendation IOTSEC_2.1.3 – *Use the AWS Partner Device Catalog to find AWS Partners that offer hardware security modules*.

If you are getting devices that have not been deployed in the field, AWS recommends reviewing the AWS Partner Device Catalog to find AWS IoT hardware partners that either implement a TPM or a secure element. Use AWS IoT Partners that offer qualified secure elements for storing IoT device identities.

Best practice IOTSEC_2.2 – **Use a trusted platform module (TPM) to implement cryptographic controls**

Generally, a TPM is used to hold, secure, and manage cryptographic keys and certificates for services such as disk encryption, Root of Trust booting, verifying the authenticity of hardware (as well as software), and password management. The TPM has the following characteristics:

1. TPM is a dedicated crypto-processor to help ensure the device boots into a secure and trusted state.
2. The TPM chip contains the manufacturer's keys and software for device encryption.
3. The Trusted Computing Group (TCG) defines hardware-roots-of-trust as part of the trusted platform module (TPM) specification.

A hardware identity refers to an immutable, unique identity for a platform that is inseparable from the platform. A hardware embedded cryptographic key, also referred to as a hardware root of trust, can be an effective device identifier. Vendors such as Microchip, Texas Instruments, and many others have TPM-based hardware solutions.

Implementation guidance

Recommendation IOTSEC_2.2.1 – *Perform cryptographic operations inside the TPM to avoid a third party gaining unauthorized access*

All secret keys from the manufacturer required for secure boot, such as attestation keys, storage keys, and application keys, are stored in the secure enclave of the chip. For example, a device running AWS IoT Greengrass can be used with an Infineon OPTIGA TPM.

Recommendation IOTSEC_2.2.2 – *Use a trusted execution environment (TEE) along with a TPM to act as a baseline defense against rootkits*

TEE is a separate execution environment that provides security services and isolates access to hardware and software security resources from the host operating system and applications. Various hardware architectures support TEE such as:

1. ARM TrustZone divides hardware into secure and non-secure worlds. TrustZone is a separate microprocessor from the non-secure microprocessor core.
2. Intel Boot Guard is a hardware-based mechanism that provides a verified boot, which cryptographically verifies the initial boot block or uses a measuring process for validation.

Recommendation IOTSEC_2.2.3 – *Use physical unclonable function (PUF) technology for cryptographic operations*

A PUF technology is a physical object that provides a physically defined digital fingerprint to serve as a unique identifier for an IoT device. As a different class of security primitive, PUFs normally have a relatively simple structure. It makes them ideal candidates for affordable security solutions for IoT networks. Generally, a hardware root of trust based on PUF is virtually impossible to duplicate, clone, or predict. This makes them suitable for applications such as secure key generation and storage, device authentication, flexible key provisioning, and chip asset management. For example, refer to AWS Partner Device Catalog, that has various device solutions with PUFs such as LPC54018 IoT Solution by NXP.

Best practice IOTSEC_2.3 – Use protected boot and persistent storage encryption

When a device performs a secure boot, it validates that the device is not running unauthorized code from the filesystem. This helps ensure that the boot process starts from a trusted combination of hardware and software, and continues until the host operating system has fully booted and applications are running.

Choose devices with TPM (or TEE) for new deployments. Secure boot also ensures that if even a single bit in the software boot-loader or application firmware is modified after deployment, the modified firmware will not be trusted, and the device will refuse to run this untrusted code.

Full disk encryption ensures that the storage and cryptographic elements are secured in absence of a TPM or secure element. The disk controller needs to ensure that all read accesses to the disk are transparently decrypted at-runtime.

Recommendation IOTSEC_2.3.1 – *Boot devices using a cryptographically verified operating system image*

Use digitally signed binaries that have been verified using an immutable root of trust, such as a master root key (MRK) that's stored securely in a non-modifiable memory, to boot devices.

Recommendation IOTSEC_2.3.2 – *Create separate filesystem partitions for the boot-loader and the applications*

As an example, configure the device boot-loader to use a read-only partition, and applications to use a separate writable partition for separation of concerns and reduce the surface area of the attack.

Recommendation IOTSEC_2.3.3 – *Use encryption utilities provided by the host operating system to encrypt the writable filesystem*

For example, use crypt utilities for Linux such as dm-crypt or GPG, and use BitLocker or Amazon Elastic File System (Amazon EFS) for Microsoft Windows.

Recommendation IOTSEC_2.3.4 – *Use services that enable you to push signed application code from a trusted source to the device*

You can use AWS IoT Jobs to push signed software binaries from the cloud to the device. For microcontrollers using FreeRTOS, ensure that the firmware images are signed before deployment.

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

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

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

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

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

For implementing the decoupled view of telemetry for your users, use a mobile service such as AWS AppSync or Amazon API Gateway. With both of these AWS services, you can create an

abstraction layer that decouples your IoT data stream from your user's device data notification stream. By creating a separate view of your data for your external users in an intermediary datastore, for example. Amazon DynamoDB or Amazon OpenSearch Service, you can use AWS AppSync to receive user-specific notifications based only on the allowed data in your intermediary store. In addition to using external data stores with AWS AppSync, you can define user specific notification topics that can be used to push specific views of your IoT data to your external users.

If an external user needs to communicate directly to an AWS IoT endpoint, ensure that the user identity is either an authorized Amazon Cognito Federated Identity that is associated to an authorized Amazon Cognito role and a fine-grained IoT policy, or uses AWS IoT custom authorizer, where the authorization is managed by your own authorization service. With either approach, associate a fine-grained policy to each user that limits what the user can connect as, publish to, subscribe from, and receive messages from concerning MQTT communication.

Best practice IOTSEC_3.1 – Implement authentication and authorization for users accessing IoT resources

It enables end users with secure access to connected IoT devices and equipment via different channels such as web or mobile devices. Without valid authentication and authorization, devices can be subjected to compromises or malicious attempts.

Recommendation IOTSEC_3.1.1 – *Implement an identity store to authenticate users of your IoT application*

Implement an identity and access management solution for end users. This solution should allow end users with temporary, role-based credentials to access the connected devices. For example, you can use a service like Amazon Cognito to create user pools for authentication. Or, you can use Amazon Cognito integration with SAML or OAuth2.0 compliant identity providers for authentication as well. If you host your own identity store, use AWS IoT custom authorizers to validate tokens (such as JWT or SAML) for authenticating users.

Recommendation IOTSEC_3.1.2 – *Enable users to be authorized with least privileged access*

Authorization is the process of granting permissions to an authenticated identity. You grant permissions to your end users in AWS IoT Core using data plane and control plane IAM policies through the identity broker. Control plane API allows you to perform administrative tasks like creating or updating certificates, things, rules, and so on. Data plane API allows you send data to and receive data from AWS IoT Core. For example, if you are using Amazon Cognito, use federated identities for user authentication. If you are using a different Identity broker than Amazon Cognito,

use AWS IoT custom authorizers to invoke Lambda functions that will create the required IAM policies.

Recommendation IOTSEC_3.1.3 – *Adopt least privilege when assigning user permissions*

Adopt the least privilege principle and assign only the minimum required permissions to user roles. For example, with Amazon Cognito this can be achieved, by setting up role-based access through IAM policies for authenticated (think of consumers, admins) and unauthenticated users. Consumers or unauthenticated users should not be allowed to run destructive actions against IoT services, such as detaching policies, deleting CA, or deleting certificate.

Best practice IOTSEC_3.2 – Decouple access to your IoT infrastructure from the IoT applications

By decoupling the IoT infrastructure from the end-user IoT applications, you can build an additional layer of security and reliability.

Recommendation IOTSEC_3.2.1 – *Use an API layer between the application and IoT layer*

Build an application interface layer to reduce the blast radius of the IoT data plane from end users. Fundamentally, the primary interface to IoT data plane is MQTT topics. Protecting the data plane essentially means protecting the MQTT topics from unwanted communication. For example, use Amazon API Gateway or AWS AppSync to provide a REST or GraphQL API interface between the end-user application and the IoT layer. This helps reduce the blast radius of the IoT data plane from end users.

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

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

In AWS IoT, create fine-grained permissions by using a consistent set of naming conventions in the IoT registry. The first convention is to use the same unique identifier for a device as the MQTT ClientID and AWS IoT thing name. By using the same unique identifier in all these locations, you

can easily create an initial set of IoT permissions that can apply to all of your devices using AWS IoT Thing Policy variables. The second naming convention is to embed the unique identifier of the device into the device certificate. Continuing with this approach, store the unique identifier as the CommonName in the subject name of the certificate to use Certificate Policy Variables to bind IoT permissions to each unique device credential.

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

```
{
    "Version": "2012-10-17",
    "Statement":  [
        {
            "Effect": "Allow",
            "Action":  [
                "iot:Connect"
            ],
            "Resource":  [
                "arn:aws:iot:us-west-2:123456789012:client/
${iot:Certificate.Subject.CommonName}"
            ],
            "Condition": {
                "Bool": {
                    "iot:Connection.Thing.IsAttached": [
                        "true"
                    ]
                }
            }
        },
        {
            "Effect":"Allow",
            "Action": [
                "iot:Publish"
            ],
            "Resource": [
                "arn:aws:iot:us-west-2:123456789012:topic/$aws/things/
${iot:Connection.Thing.ThingName}/shadow/update"
            ]
        }
```

```
    ]
}
```

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

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

```
{
    "Version": "2012-10-17",
    "Statement":  [
        {
            "Effect":"Allow",
            "Action": [
                "iot:Publish"
            ],
            "Resource": [
                "arn:aws:iot:us-west-2:123456789012:topic/$aws/things/edgegateway123-*/
shadow/update"
            ]
        }
    ]
}
```

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

Best practice IOTSEC_4.1 – Assign least privilege access to devices

Permissions (or policies) allow an authenticated identity to perform various control and data plane operations against the IoT Broker, such as creating devices or certificates via the control plane, and connecting, publishing, or subscribing via the data plane.

Recommendation IOTSEC_4.1.1 – *Grant least privileged access to reduce the scope of impact of the potential events*

We recommend using granular device permissions to enable least privileged access, which can help limit the impact of an error or misconfiguration. Define a mechanism so that devices can only communicate with specific authorized resources, such as MQTT topics. If permissions are generated dynamically, ensure that similar practices are followed. For example, create an AWS IoT policy as a JSON document that contains a statement with the following:

1. Effect, which specifies whether the action is allowed or denied.
2. Action, which specifies the action the policy is allowing or denying.
3. Resource, which specifies the resource or resources on which the action is allowed or denied.

Recommendation IOTSEC_4.1.2. – *Consider scaling granular permissions across the IoT fleet*

We recommend reusing permissions across principals rather than hardcoding for better manageability as it helps you avoid create redundant permissions per device. For example, an AWS IoT policy allows access based on various thing attributes such as `ThingName`, `ThingTypeName`, and `Thing Attributes`. Thus, a device can connect with a client ID (such as `foo`), only if the device registry contains the matched device (aka ThingName), such as `arn:aws:iot:us-west-2:123456789012:client/${iot:Connection.Thing.ThingName}` rather than `arn:aws:iot:us-west-2:123456789012:client/foo`.

As another example, an AWS IoT policy also allows access based on various certificate attributes such as `Subject`, `Issuer`, `Subject Alternate Name`, `Issuer Alternate Name`, and `Others`. Thus, a device can only publish to a topic that matches with the Certificate ID associated with the device in the registry like `arn:aws:iot:us-west-2:123456789012:topic/${iot:CertificateId}` rather than `arn:aws:iot:us-west-2:123456789012:topic/xxxxxxxxxxx`

IOTSEC 5: How do you manage device certificates, including installation, validation, revocation, and rotation?

To protect and encrypt data in transit from an IoT device to the cloud, most IoT broker supports TLS-based mutual authentication using X.509 certificates. Device makers must provision a unique identity, including a unique private key and X.509 certificate, into each device. Certificates are long-lived credentials and managed using a customer-owned Certificate Authority (CA), a third-party CA, or AWS Private CA. Any hosted CA chosen must provide you the ability to validate, activate, deactivate, and rotate certificates.

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

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

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

Another state for certificates is to be near their expiry date but still valid. The client certificate must be valid for at least the service lifetime of the device. It’s up to your IoT application to keep track of devices near their expiry date and perform an OTA process to update their certificate to a

new one with a later expiry, along with logging information about why the certificate rotation was required for audit purposes.

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

Best practice IOTSEC_5.1 – Perform certificate lifecycle management

A certificate lifecycle includes different phases such as creation, activation, rotation, revocation or expiry. An automated workflow can be put in place to identify certificates that needs attention, along with remediation actions.

Recommendation IOTSEC_5.1.1 – *Document your plan for managing certificates*

As explained previously, X.509 certificates help to establish the identity of devices and encrypts the traffic from the edge to cloud. Thus, planning the lifecycle management of device certificates is essential. Enable auditing and monitoring for compromise or expiration of your device certificates. Determine how frequently you need to rotate device certificates, audit cloud or device-related configurations and permissions to ensure that security measures are in place. For example, use AWS IoT Device Defender to monitor the health of the device certificates and different configurations across your fleet. AWS IoT Device Defender can work in conjunction with AWS IoT Jobs to help enable rotate the expired or compromised certificates.

Recommendation IOTSEC_5.1.2 – *Use certificates signed by your trusted intermediate CA for on-boarding devices*

As a best practice, the root CA needs to be locked and protected to secure the chain of trust. The device certificates should be generated from an intermediate CA. Define a process to programmatically manage intermediate CA certificates as well. For example, enable AWS IoT Device Defender Audit to report on your intermediate CAs that are revoked but device certificates are still active or if the CA certificate quality is low. You can thereafter use a security automation workflow using mitigation actions in Device defender to resolve the issues.

Recommendation IOTSEC_5.1.3 – *Secure provisioning claims private keys and disable the certificate in case of misuse and record the event for further investigation*

- Monitor provisioning claims for private keys at all times, including on the device.

For example:

- Use AWS IoT CloudWatch metrics and logs to monitor for indications of misuse. If you detect misuse, disable the provisioning claim certificate so it cannot be used for device provisioning.
- Use AWS IoT Device Defender to identify security issues and deviations from best practices.

- For more information:
  - https://docs.aws.amazon.com/iot/latest/developerguide/vulnerability-analysis-and-management.html
  - https://aws.amazon.com/blogs/iot/just-in-time-registration-of-device-certificates-on-aws-iot/

Detective controls

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

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

IOTSEC 6: How do you analyze application logs and device metrics to detect security issues?

Your device logs and metrics play a critical role in monitoring security behavior of your IoT application. The way you configure your operations, and how anomalies are surfaced in your system will determine how quickly you can react to a security issue. By configuring your IoT logs and metrics appropriately, you can proactively mitigate potential security issues in your IoT application.

In AWS IoT, you can implement detective controls using AWS IoT Device Defender, CloudWatch Logs, AWS IoT Greengrass 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 or with ML detect to automatically learn normal device behaviors. Set a default set of thresholds when device behavior or connectivity behavior deviates from normal activity.

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

Best practice IOTSEC_6.1 – Collect and analyze logs and metrics to capture authorization errors and failures to enable appropriate response

Device logs and metrics can provide your organization with the insight to be operationally efficient with your IoT workloads by identifying security events, anomalies, and issues from device data. Record error-level messages from AWS IoT Core to provide operational visibility to potential security issues.

Recommendation IOTSEC_6.1.1 – *Enable metrics and create alarms that track authorization and error metrics*

- Observe the trends for these AWS IoT metrics: Connect.AuthError, PublishIn.AuthError, PublishOut.AuthError and Subscribe.AuthError.
- Configure CloudWatch alarms for each of the preceding metrics to alarm based on levels higher than normal for your workload.

Best practice IOTSEC 6.2 – Alert when on security events, misconfiguration, and behavior violations are detected

Audit the configuration of your devices and detect and alert when a device behavior differs from the expected behavior. It provides visibility into operational data that can indicate potential security issues active in the device fleet.

Recommendation IOTSEC_6.2.1 – *Enable metrics to detect security events from the data plane*

Create a threat model to detect events from security vulnerabilities or device compromises. You can detect events based on configured rules or machine learning (ML) models. For example, create a security profile in AWS IoT Device Defender, that detects unusual device behavior that might be indicative of a compromise by continuously monitoring high-value security metrics from the device and AWS IoT Core. You can specify normal device behavior for a group of devices by setting up behaviors (rules) for these metrics. AWS IoT Device Defender monitors and evaluates each datapoint reported for these metrics against user-defined behaviors (rules) and alerts you if an anomaly is detected. When you use ML Detect, the feature sets device behaviors automatically with machine learning to monitor device activities.

Recommendation IOTSEC_6.2.2 – *Enable auditing to check misconfigurations*

Audit checks are necessary to determine that device stays configured with required best practices throughout its lifecycle. For instance, its necessary to audit devices regularly on basic checks such as logging, shared certificates and unique device IDs. For example, AWS IoT Device Defender can help you to continuously audit security configurations for compliance with security best practices and your own organizational security policies. Some of the auditing capabilities supported natively are `LOGGING_DISABLED_CHECK`, `IOT_POLICY_OVERLY_PERMISSIVE_CHECK`, `DEVICE_CERTIFICATE_SHARED_CHECK`, and `CONFLICTING_CLIENT_IDS_CHECK`.

Recommendation IOTSEC_6.2.3 – *Ensure alerting on a behavior violation*

Enable alarming or notifications when the device behavior is anomalous based on configured rules or ML models. For example, AWS IoT Device Defender can alert you with the metric datapoint reported by the device when an ML model flags the datapoint as anomalous. This removes the need for you to define accurate behaviors of your devices and helps you get started with monitoring more quickly and easily.

Best practice IOTSEC_6.3 – Alert on non-compliant device configurations and remediate using automation

Enable auditing to continuously assess configurations and metrics on the device. security configurations can be impacted by the passage of time and new threats are constantly emerging. For example, cryptographic algorithms once known to provide secure digital signatures for device certificates can be weakened by advances in the computing and cryptoanalysis methods.

Recommendation IOTSEC_6.3.1 – *Ensure regular auditing for identifying configuration issues*

Audit checks are necessary to determine that device stays configured with required best practices throughout its lifecycle. For instance, its necessary to audit devices regularly on basic checks such as logging, shared certificates and unique device IDs. For example, AWS IoT Device Defender can help you to continuously audit security configurations for compliance with security best practices and your own organizational security policies. Some of the auditing capabilities supported natively are `LOGGING_DISABLED_CHECK`, `IOT_POLICY_OVERLY_PERMISSIVE_CHECK`, `DEVICE_CERTIFICATE_SHARED_CHECK`, and `CONFLICTING_CLIENT_IDS_CHECK`.

Recommendation IOTSEC_6.3.2 – *Use automation to remediate issues*

Investigate issues by providing contextual and historical information about the device such as device metadata, device statistics, and historical alerts for the device. For example, you can use AWS IoT Device Defender built-in mitigation actions to perform mitigation steps on Audit and Detect alarms such as adding things to a thing group, replacing default policy version and updating device certificate. Or you can enable a mitigation action to re-enable logging and publish the finding to Amazon SNS should the LOGGING_DISABLED_CHECK find that logging is not enabled.

Infrastructure protection

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

One way to approach the device security lifecycle is through supply chain analysis. The IoT supply chain includes the actors, processes and assets that participate in the realization (for example, development, design, maintenance, patch management) of any IoT device. For example, even a modestly sized IoT device manufacturer or solution integrator has a large number of suppliers that make up its supply chain, whether directly or indirectly. To maximize solution lifetime and reliability, ensure that you are receiving authentic components.

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

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

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

AWS offers flexible ways and design patterns to establish a secure connection to the AWS environment from the edge. When choosing a secure connection to the AWS environment, take into consideration the use case requirements such as latency and data locality to ensure that the chosen connection solution meets the performance and compliance requirements. Use AWS Systems Manager to carry out routine management tasks on edge computing resources, Secure Tunneling for AWS IoT Device Management to access IoT devices behind restricted firewalls at remote sites for troubleshooting, configuration updates, and other operational tasks and AWS IoT Greengrass for secure remote application management. Take advantage of on-premises managed infrastructure solutions such as AWS Outposts, AWS Storage Gateway, AWS Snow Family to simplify management and monitoring.

Data protection

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

During the design process, determine how hardware, firmware, and data are handled at device end-of-life. Store long-term historical data in the cloud. Store a portion of current sensor readings

locally on a device, namely only the data required to perform local operations. By only storing the minimum data required on the device, the risk of unintended access is limited.

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

In IIoT environments, to allow one-way data flow, access controls can be applied at the connectivity layer using security appliances such as firewalls and data diodes.

IOTSEC 7: How do you ensure that device data is protected at rest and in transit?

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

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

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

Protect your data at rest by defining your requirements and implementing controls, including encryption, to reduce the risk of unauthorized access or loss. Protect your data in transit by defining your requirements and implementing controls, including encryption, reduces the risk of

unauthorized access or exposure. By providing the appropriate level of protection for your data in transit, you protect the confidentiality and integrity of your IoT data.

Follow these best practices and check if your workload is well architected.

Best practice IOTSEC_7.1 – Use encryption to protect IoT data in transit and at rest

For data at rest, the Storage Networking Industry Association (SNIA) defines storage security as "Technical controls, which may include integrity, confidentiality and availability controls that protect storage resources and data from unauthorized users and uses." Thus, it's required to protect the confidentiality of sensitive data, such as the device identity, secrets, or user data, by encrypting it at rest. For data in transit, use a secure transport mechanism such as TLS to protect the confidentiality and integrity of all data transmitted to and from your devices.

Recommendation IOTSEC_7.1.1 – *Require the use of device SDKs or client libraries for the device to communicate to cloud*

Configure the IoT devices to communicate to cloud endpoints only over TLS. For example, use AWS IoT Greengrass or FreeRTOS SDKs to secure connectivity from your devices to AWS IoT Core over TLS 1.2.

Recommendation IOTSEC_7.1.2 – *Encrypt data at rest or secrets on IoT devices*

As explained previously in section IOTSEC_2.3.3, take advantage of encryption utilities provided by the host operating system to encrypt the data stored at rest in the local filesystem. In addition, take advantage of Secure Elements and TPMs. TEEs can add storage protections as well.

Best practice IOTSEC_7.2 – Use data classification strategies to categorize data access based on levels of sensitivity

Data classification and governance is the customer's responsibility.

1. Identify and classify data based on sensitivity collected throughout your IoT workload and learn their corresponding business use case.
2. Identify and act on opportunities to stop collecting unused data, or adjusting data granularity and retention time.
3. Consider a defense in depth approach and reduce human access to device data.

See the following for more details:

- AWS IoT Greengrass Developer Guide: Manage data streams on the AWS IoT Greengrass core

- The Internet of Things on AWS – Official Blog: Designing dataflows for multi-schema messages in AWS IoT Analytics

Recommendation IOTSEC_7.2.1 – *Implement data classification strategies for all data stored on devices or in the cloud, as well as all data sent over the network. Process data based on the level of sensitivity (for example, highly classified, or personally identifiable data.)*

Before architecting an IoT application, data classification, governance, and controls must be designed and documented to reflect how the data can be persisted on the edge or in the cloud, and how data should be encrypted throughout its lifecycle. For example:

- By using AWS IoT Greengrass stream manager, you can define policies for storage type, size, and data retention on a per-stream basis. For highly classified data, you can define a separate data stream.
- By using AWS IoT Analytics, you can create different workflows for storing classified data. For highly classified data, you can define a separate pipeline and data store.

Best practice IOTSEC_7.3 – Protect your IoT data in compliance with regulatory requirements

Data governance is the rules, processes, and behavior that affect the way in which data is used, particularly as it regards openness, participation, accountability, effectiveness, and coherence. Data governance practices for IoT is important as it enables protecting classified data and complying with regulatory obligations. It helps to determine what data needs protection, or which data needs access control.

See the following for more information:

- AWS Cloud Enterprise Strategy Blog: Using a Cloud Center of Excellence (CCOE) to Transform the Entire Enterprise

Recommendation IOTSEC_7.3.1 – *Define specific roles for personnel responsible for implementing IoT data governance*

For example, there might be a need for new roles to monitor security, from both the functional and policy perspectives, to control data when it moves from IoT environments to the cloud.

Recommendation IOTSEC_7.3.2 – *Define data governance policies to monitor compliance with approved standards*

For example, you might define a policy that requires security credentials to never be hardcoded, even on edge devices. Thus, only services such as AWS Secrets Manager can retrieve secrets in an encrypted manner.

Recommendation IOTSEC_7.3.3 *– Define clear responsibilities to drive the IoT data governance process*

Multiple administrative roles can exist for a single system. For instance, you might define roles for users who can replace defective devices, and separate roles for users who can apply security patches and upgrade device firmware. Note that roles and responsibilities might change over the lifecycle of your IoT systems.

Incident response

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

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

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

- Test incident response procedures on a periodic basis.

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

Implement a monitoring solution in the OT, IoT, and IIoT environments to create an industrial network traffic baseline and monitor anomalies and adherence to the baseline. Collect security logs and analyze them in real-time using dedicated tools, for example, security information and event management (SIEM) class solutions such as within a security operation center (SOC). AWS works with a number of OT Intrusion Detection System (IDS) and SIEM partners that can be found on AWS Marketplace.

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

IOTSEC 8: How do you plan the security lifecycle of your IoT devices?

The security lifecycle of your IoT devices includes everything, from how you choose your suppliers, contract manufacturers, and other outsourced relationships to how you manage security in your third-party firmware and manage security events over time. With visibility into the full spectrum of actors and activities in your hardware and software supply chain, you can be better prepared to respond to compliance questions, detect and mitigate events, and avoid common security risks related to third-party components.

Best practice IOTSEC_8.1 – Build incident response mechanisms to address security events at scale

There are several formalized incident management methodologies in common use. The processes involved in monitoring and managing incident response can be extended to IoT devices. For

instance, AWS IoT Device Management capabilities provide fleet analysis and activity tracking to identify potential issues, in addition to mechanisms to enable an effective response.

Recommendation IOTSEC_8.1.1 – *Ensure that IoT devices are searchable by using a device management solution*

Devices should be grouped by dynamic attributes, such as connectivity status, firmware version, application status, and device health.

Recommendation IOTSEC_8.1.2 – *Quarantine any device that deviates from expected behavior*

Inspect the device for potential compromise of the configurations, firmware or applications using device logs or metrics. If a compromise is detected, the device can be diagnosed remotely provided that capability exists. For example, configure AWS IoT Secure Tunneling to remotely diagnose a fleet of devices.

If remote diagnosis is not sufficient or available, the other option is to push a security patch, application or firmware upgrade to quarantine the device. When sending code to devices, the best practice is to sign the code file. This allows devices to detect if the code has been modified in transit. For example, With code signing for AWS IoT, you can sign code that you create for IoT devices supported by FreeRTOS and AWS IoT device management. In addition, the signed code can be valid for a limited amount of time to avoid further manipulation.

Recommendation IOTSEC_8.1.3 – *Over-the-air (OTA) updates should be configured and staged for deployment activation during regular maintenance*

Whether it's a security patch or a firmware update, an update to a configuration file on a device, or a factory reset, you need to know which devices in your fleet have received and processed any of your updates, either successfully or unsuccessfully. In addition, a staged rollout is recommended to reduce the blast radius along with rollout and abort criterias for a failsafe solution. For example, you can use AWS IoT Jobs for OTA updates of security patch and device configurations in a staged manner with required rollout and abort configurations.

Practice IOTSEC_8.2 – Require timely vulnerability notifications and software updates from your providers

Components in a device bill of materials (BOM), such as secure elements for certificate storage or a trusted platform module (TPM), can make use of updatable software components. Some of this software might be contained in the Board Support Package (BSP) assembled for your device. You

can help to mitigate device-side security issues quickly by knowing where the security-sensitive software components are within your device software stack, and by understanding what to expect from component suppliers with regard to security notifications and updates.

Recommendation IOTSEC_8.2.1 – *Ensure that your IoT device manufacturer provides security-related notifications to you, and provides software updates in a timely manner to reduce the associated risks of operating hardware or software with known security vulnerabilities*

Ask your suppliers about their product conformance to the Common Criteria for Information Technology Security Evaluation. In addition, consider using the AWS Partner Device Catalog where you can find devices and hardware to help you explore, build, and go to market with your IoT solutions.

Key AWS services

The essential AWS security services in IoT are the AWS IoT registry, AWS IoT Device Defender, AWS Identity and Access Management (IAM), and Amazon Cognito. In combination, these services along with other AWS security services allow you to securely control access to IoT devices, AWS services, IoT applications, and resources for your users. The following services and features support IoT security:

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

Asset inventory: AWS IoT Device Management can be used as an inventory for IoT devices and AWS Systems Manager Inventory can be used to provide visibility into on premises computing resources and edge gateways.

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

Detective controls: AWS IoT Device Defender records device communication and cloud side metrics from AWS IoT Core. AWS IoT Device Defender can automate security responses by sending notifications through Amazon Simple Notification Service (Amazon SNS) to internal systems

or administrators. AWS CloudTrail logs provide administrative actions of your IoT application. Amazon CloudWatch is a monitoring service with integration with AWS IoT Core and can trigger CloudWatch Events to automate security responses. CloudWatch captures detailed logs related to connectivity and security events between IoT edge components and cloud services.

Infrastructure protection: AWS IoT Core is a cloud service that lets connected devices easily and securely interact with cloud applications and other devices. The AWS IoT rules engine in AWS IoT Core uses IAM permissions to communicate with other downstream AWS services. AWS has created a wide selection of industry leading IoT silicon vendors, device manufacturers, and gateway partners who have integrated AWS IoT Greengrass into their software and hardware offerings. You have the option to store your device private key on a hardware secure element and store sensitive device information at the edge with AWS IoT Greengrass Secrets Manager and encrypt secrets using private keys for root of trust security.

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

Patch management: Implement patch management to fix device vulnerabilities and define appropriate update mechanisms for software and firmware updates using AWS IoT Device Management Jobs service and AWS Systems Manager Patch Manager. Perform deployment of patches only after testing the patches in a test environment before implementing them in production and verify the integrity of the software before starting to run it ensuring that it comes from a reliable source (signed by the vendor) and that it is obtained in a secure manner.

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. AWS Security Hub can be used to aggregate security alerts from various AWS services and partner products to help you analyze your security trends and identify the highest priority security issues. AWS Security Hub provides you with a comprehensive view of your security state within AWS and your compliance with security standards and best practices and enables automated remediation. Security Hub has out-of-the-box integrations with ticketing, chat, Security Information and Event Management (SIEM), Security Orchestration Automation and Response (SOAR), threat investigation, Governance

Risk and Compliance (GRC), and incident management tools to provide users with a complete security operations workflow.

Business continuity and recovery

To backup IoT data at the edge and in the cloud, you can use AWS IoT Greengrass stream manager to locally buffer data and send data to local storage destinations and other life cycle management features available in AWS IoT Greengrass to support your data resiliency and backup needs. AWS Backup can be used to centrally manage and automate backups across AWS services and on premise IoT systems.

Resources

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

Documentation and blogs

- AWS IoT Device Management
- IoT Security Identity
- AWS IoT Device Defender
- IoT Authentication Model
- MQTT on port 443
- Assessing OT and IIoT cybersecurity risk
- Detect Anomalies with Device Defender
- Implement security monitoring across OT, IIoT and Cloud
- AWS IoT secure tunneling
- AWS Systems Manager
- Plant network to Amazon VPC connectivity options
- Ten security golden rules for Industrial IoT solutions
- AWS Security Incident Response Guide
- AWS Backup

Whitepapers

- MQTT Topic Design

- Security Best Practices in Manufacturing OT
- Securing Internet of Things (IoT) with AWS
- Device Manufacturing and Provisioning with X.509 Certificates in AWS IoT Core

Reliability pillar

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

Topics

- Design principles
- Best practices
- Key AWS services
- Resources

Design principles

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

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

connectivity, ability to roll back firmware updates, ability to fail over to a different networking protocol or communicate locally for critical message delivery, running redundant sensors or edge gateways to be resilient to hardware failures, and the ability to perform a factory reset.

Best practices

There are three best practice areas for reliability:

Topics

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 a denial of service event. The system should be designed to detect the failure and automatically heal itself.

Foundations

IoT devices must continue to operate in some capacity in the face of network or cloud errors. Design device firmware to handle intermittent connectivity or loss in connectivity in a way that is sensitive to memory and power constraints. IoT cloud applications must also be designed to handle remote devices that frequently transition between being online and offline to maintain data coherency and scale horizontally over time. Monitor overall IoT utilization and create a mechanism to automatically increase capacity to ensure that your application can manage peak IoT traffic.

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

IOTREL 01. How do you handle AWS service limits for peaks in your IoT application?

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

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

Soft limits in AWS IoT traditionally correlate to account-level limits that are independent of a single device. For any account-level limits, you should calculate your IoT usage for a single device and then multiply that usage by the number of devices to determine the base IoT limits that your application will require for your initial product launch. AWS recommends that you have a ramp-up period where your limit increases align closely to your current production peak usage with an additional buffer. To ensure that the IoT application is not under provisioned:

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

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

needed. Plan for sustained ramp-up periods for onboarding devices so that the IoT limit increases align with regular day-to-day data plane usage.

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

IOTREL 02. What is the strategy for managing ingestion and processing throughput of IoT data to other applications?

Although IoT applications have communication that is only routed between other devices, there will be messages that are processed and stored in your application. In these cases, the rest of your IoT application must be prepared to respond to incoming data. All internal services that are dependent upon that data need a way to seamlessly scale the ingestion and processing of the data. In a well-architected IoT application, internal systems are decoupled from the connectivity layer of the IoT platform through the ingestion layer. The ingestion layer is composed of queues and streams that enable durable short-term storage while allowing compute resources to process data independent of the rate of ingestion.

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

IOTREL 03. How do you implement your IoT workload to withstand component and system faults?

Understanding and predicting the fault scenarios in the system helps you to architect for failure conditions and use service features to handle them. Therefore, the handling of such predicted system faults and recovering from them should be architected into the system.

Best practice IOTREL_3.1 – Use the services provided by your vendors for integration and error handling to withstand component failure

An IoT design consists of device software, connectivity and control services, and analytics services. Test the entire IoT environment for resiliency, starting with device firmware, data flow, the cloud services used, and error handling. Vendors have services integrated with each other to provide a simplified integration and fault handling.

Recommendation IOTREL_3.1.1 – *Understand and apply the standard libraries available to manage your device firmware*

- Devices can be built on FreeRTOS, which provides connectivity, messaging, power management and device management libraries that are tested for reliability and designed for ease of use.

Recommendation IOTREL_3.1.2 – *Use log levels appropriate to the lifecycle stage of your workload*

- AWS IoT logs can be set up per region and per account with the logging level set to DEBUG during product development phase to provide insights on data flow and resources used. This data can be used to improve the IoT system security and performance.
- AWS IoT Secure Tunneling can be used to test and debug devices that are behind a restrictive firewall in the field.

IOTREL 04. How do you ensure that all IoT messages are processed?

Data sent from devices should be processed and stored without excessive loss. Services that queue and deliver IoT data to compute and database services should be used to ensure the processing of data. IoT devices send lots of data in small sizes without order, and the cloud application should be able to handle this.

Best practice IOTREL_4.1 – Dynamically scale cloud resources with utilization

The elastic nature of the cloud can be used to increase and decrease resources on demand. Use the ability to increase and decrease cloud resources based on data, number of messages and size of messages and number of devices.

Recommendation IOTREL_4.1.1 – *Know the mechanisms that can be used to monitor cloud resource usage and methods to scale the resources*

- Use Amazon CloudWatch Logs to trigger based on rate of data flow to auto-scale cloud resources as needed.
- Use AWS IoT Rules engine error actions to provision additional cloud resources and message retries as needed.
- Examine IoT logs for errors in communicating to resources and provision resources based on that data.
- Use AWS Lambda to automatically scale your application by running code in response to each event.
- Use automatic scaling where possible. Kinesis Data Streams and Amazon DynamoDB are two services that provide automatic scaling.

IOTREL 05. How do you ensure that your IoT device operates with intermittent connectivity to the cloud?

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

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

- Store important messages durably offline and, once reconnected, send those messages to AWS IoT Core.
- Implement exponential retry and back-off logic when connection attempts fail.
- If necessary, have a separate failover network channel to deliver critical messages to AWS IoT. This can include failing over from Wi-Fi to standby cellular network, or failing over to a wireless personal area network protocol (such as Bluetooth LE) to send messages to a connected device or gateway.
- Have a method to set the current time using an NTP client or low-drift real-time clock. A device should wait until it has synchronized its time before attempting a connection with AWS IoT

Core. If this isn't possible, the system provides a way for a user to set the device's time so that subsequent connections can succeed.
- Send error codes and overall diagnostics messages to AWS IoT Core.
- Configure a AWS IoT Greengrass group to write logs to the local file system and to CloudWatch Log

Connection to the cloud can be intermittent and devices should be designed to handle this. Choose devices with firmware designed for intermittent cloud connection and that have the ability to store data on the device if you cannot afford to lose the data.

Best practice IOTREL_5.1 – Synchronize device states upon connection to the cloud

IoT devices are not always connected to the cloud. Design a mechanism to synchronize device states every time the device has access to the cloud. Synchronizing the device state to the cloud allows the application to get and update device state easily, as the application doesn't have to wait for the device to come online.

Recommendation IOTREL_5.1.1 – *Use a digital devices state representation to synchronize device state using the below capabilities:*

- AWS provides device shadow capabilities that can be used to synchronize device state when the device connects to the cloud. The AWS IoT Device Shadow service maintains a shadow for each device that you connect to AWS IoT and is supported by the AWS IoT Device SDK, AWS IoT Greengrass core, and FreeRTOS.
- Synchronizing device shadows – Device SDKs and the AWS IoT Core take care of synchronizing property values between the connected device and its device shadow in AWS IoT Core.
- AWS IoT Greengrass – AWS IoT Greengrass core software provides local shadow synchronization of devices and these shadows can be configured to sync with cloud.
- FreeRTOS – The FreeRTOS device shadow API operations define functions to create, update, and delete AWS IoT Device Shadows.

Best practice IOTREL_5.2 – Use device hardware with sufficient capacity to meet your data retention requirements while disconnected

Store important messages durably offline and, once reconnected, send those messages to the cloud. Device hardware should have capabilities to store data locally for a finite period of time to prevent any loss of information.

Recommendation IOTREL_5.2.1 – *You can leverage the device edge software capabilities for storing data locally.*

- Using AWS IoT Greengrass for device software can help collect, process, and export data streams, including when devices are offline.
  - Messages collected on the device are queued and processed in FIFO order.
  - By default, the AWS IoT Greengrass Core stores unprocessed messages destined for AWS Cloud targets in memory.
  - Configure AWS IoT Greengrass to cache messages to the local file system so that they persist across core restarts.
  - AWS IoT Greengrass stream manager makes it easier and more reliable to transfer high-volume IoT data to the AWS Cloud.
  - Configure AWS IoT Greengrass core
  - Manage data streams on AWS IoT Greengrass Core
  - AWS IoT Greengrass Developer Guide
  - Run Lambda on AWS IoT Greengrass Core for preprocessing
- The ETL with AWS IoT Extract, Transform, Load with AWS IoT Greengrass Solution Accelerator helps to quickly set up an edge device with AWS IoT Greengrass to perform extract, transform, and load (ETL) functions on data gathered from local devices before being sent to AWS.
  - ETL with AWS IoT Greengrass solution accelerator
- Consider using AWS IoT SiteWise for data coming from disparate industrial equipment
  - The AWS IoT SiteWise connector sends local equipment data in AWS IoT SiteWise. You can use this connector to collect data from multiple OPC Unified Architecture (UA) servers and publish it to AWS IoT SiteWise.
  - AWS IoT SiteWise connector with AWS IoT Greengrass can cache data locally in the event of intermittent network connectivity.
  - You can configure the maximum disk buffer size used for caching data. If the cache size exceeds the maximum disk buffer size, the connector discards the earliest data from the queue.
  - AWS IoT Greengrass: AWS IoT SiteWise connectors

Best practice IOTREL_5.3 – Down sample data to reduce storage requirements and network utilization

Data should be down sampled where possible to reduce storage in the device and lower transmission costs and reduce network pressure.

Recommendation IOTREL_5.3.1 – *Use device edge software capabilities for down sampling*

- Using AWS IoT Greengrass for device software to down sample data.
  - Local Lambda functions can be used on AWS IoT Greengrass to down sample the data before sending it to the cloud.
- ETL with AWS IoT Extract, Transform, Load with AWS IoT Greengrass Solution Accelerator helps to quickly set up an edge device with AWS IoT Greengrass to perform extract, transform, and load (ETL) functions on data gathered from local devices before being sent to AWS.

Best practice IOTREL_5.4 – Use an exponential backoff with jitter and retry logic to connect remote devices to the cloud

Consider implementing a retry mechanism for IoT device software. The retry mechanism should have exponential backoff with a randomization factor built in to avoid retries from multiple devices occurring simultaneously. Implementing retry logic with exponential backoff with jitter allows the IoT devices to more evenly distribute their traffic and prevent them from creating unnecessary peak traffic.

Recommendation IOTREL_5.4.1 – *Implement logic in the cloud to notify the device operator if a device has not connected for an extended period of time*

- AWS IoT Events can be used to monitor devices remotely.
- Remote monitoring using AWS IoT Events

Recommendation IOTREL_5.4.2 – *Use device edge software and the SDK to leverage built-in exponential back off logic*

- Exponential backoff logic is included in the AWS SDK, including the AWS IoT Device SDK, and edge software, such as AWS IoT Greengrass Core and FreeRTOS.
- AWS SDK handles the exponential back off
- AWS IoT Device SDK for C uses “IOT_MQTT_RETRY_MS_CEILING” for setting maximum retry interval limit.

Recommendation IOTREL_5.4.3 – *Establish alternate network channels to meet requirements*

- Have a separate failover network channel to deliver critical messages to AWS IoT. Failover channels can include Wi-Fi, cellular networks, or a wireless personal network.

- For low latency workload, use AWS Wavelength for 5G devices and AWS Local Zones to keep your cloud services closer to the user.

IOTREL 06. How do you control the frequency of message delivery to the device?

Devices can be restricted in message processing capacity and messages from the cloud might need to be throttled. The cloud-side message delivery rate might need to be architected based on the type of devices that are connected.

Best practice IOTREL_6.1 – Target messages to relevant devices

Devices receive information from shadow updates, or from messages published to topics they subscribe to. Some data are relevant only to specific devices. In those cases, design your workload to send messages to relevant devices only, and to remove any data that is not relevant to those devices.

Recommendation IOTREL_6.1.1 – *Preprocess data to support the specific needs of the device*

- Use AWS Lambda to pre-process the data and hone-in specifically to attributes and variables that are needed by the device to act upon

Best practice IOTREL_6.2 – Implement retry and backoff logic to support throttling by device type

Retry and backoff logic should be implemented in a controlled manner so that when you need to alter throttling settings per device type, you can easily do it. Using data storage of any chosen kind gives you flexibility on what data to publish down to the device.

Recommendation IOTREL_6.2.1 – *Use storage mechanisms that enable retry mechanisms*

- Using DynamoDB, you can hold data in key value format where device ID is the key. Retry logic can be applied to only certain device IDs.

- Using Amazon Relational Database Service (Amazon RDS), you have the flexibility to use a variety of database engines. The retry messages can have new real-time data augmented with historic data from previous device interactions stored in Amazon RDS.
- AWS IoT Events provides state machines with built-in timers to hold back data and retry based on timers.

Change management

IOTREL 07. How do you ensure that you can reliably update device firmware from your IoT application?

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

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

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

Devices will need new features over time for better user experience and the firmware will need to be updated remotely. Devices should be designed to receive and update their firmware and the IoT application should be designed to send firmware updates and monitor the success of such an update send.

Best practice IOTREL_7.1 – Use a mechanism to deploy and monitor firmware updates

When performing over-the-air (OTA) updates to remote devices' firmware, we should always ensure that the updates are controlled and reversible to avoid functional impact of the device to the user, or the device entering a non-recoverable state. Use tools that allow you to deploy and track management tasks in your device fleet.

Recommendation IOTREL_7.1.1 – *Use a cloud-based update orchestrator to deploy your firmware*

- You can use AWS IoT Jobs to send remote actions to one or many devices at once, control the deployment of jobs to your devices, and track the current and past status of job executions for each device.
- Using FreeRTOS OTA using AWS IoT Jobs: By using AWS IoT Jobs for FreeRTOS, you have reliability and security provided out of the box where OTA update job will send firmware to your end device over secure MQTT or HTTPS and system reserved topics are provided to keep track on the status of the job schedule.
- Using custom IoT jobs with AWS IoT connected devices: By using AWS IoT Jobs with one or more devices connected to AWS IoT gives you the ability to track the full roll out of the update.

Best practice IOTREL_7.2 – Implement firmware rollback capabilities in devices

Augment hardware with software to hold two versions of firmware and the ability to switch between them. Devices can rapidly roll back to older firmware if the new firmware has issues.

Recommendation IOTREL_7.2.1 – *Leverage an RTOS with functionality to roll back device firmware*

By combining OTA agents provided by FreeRTOS or using AWS IoT Device SDK, you can create flexibility to hold two versions of firmware with the hardware that is capable of storing it.

Best practice IOTREL_7.3 – Implement support for incremental updates to target device groups

It's a good practice to test new firmware on a small group of devices. Using a smaller group of devices for firmware updates helps ensure that the firmware as well as the upgrade process is well tested before the entire fleet is updated.

Recommendation IOTREL_7.3.1 – *Use a cloud orchestrator in conjunction with device settings augmentation. Cloud services can help you control and manage jobs in tandem with the devices running the jobs.*

- The AWS IoT Jobs API provides a granular level of control from the cloud to the device for carrying out firmware update incrementally and roll back as needed.
- A job document created as part of AWS IoT job details the remote operations the device needs to perform. This includes shutting down rollouts based on timeouts, number of updates per device among other things. Devices can use this information to reject or accept firmware updates.

Best practice IOTREL_7.4 – Implement dynamic configuration management for devices

Deploying software changes to devices constitutes a high-risk operation due to the recovery cost associated with remotely deployed devices. When possible, prefer mechanisms for making changes using command-and-control channels to reduce the risk that comes with software deployments and firmware upgrades. This approach enables you to push some changes to devices while minimizing the risk of entering fault states that require on-premises recovery actions. Configuration changes reduces the amount of bandwidth compared to firmware updates.

Recommendation IOTREL_7.4.1 – *Use cloud tools to command-and-control devices. Changing configuration of devices is less error prone and easier to trace back than updating firmware.*

- Use Secure Tunneling or AWS Systems Manager to facilitate patching of the operating system instead of pushing a new image to be loaded on the device.
- Use Device Shadows to command-and-control devices rather than sending commands directly to device.
- Use AWS IoT Device Defender and AWS IoT Device Management jobs to rotate expiring device certificates instead of pushing a new image with updated certificates.
- Secure Tunneling
- Device Shadows
- Device Defender

Failure management

IOTREL 08. How do you implement cloud-side mechanisms to control and modify the message frequency to the device?

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

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

Beginning with the rules engine, your application logic should initially process messages from a queue and validate that the schema of that message is correct. Your application logic should catch and log any known errors and optionally move those messages to their own dead letter queue (DLQ) for further analysis. Have a catch-all IoT rule that uses Amazon 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.

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

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

Devices can be restricted in message processing capacity and messages from the cloud might need to be throttled. The cloud-side message delivery rate might need to be architected based on the type of devices that are connected to control the frequency of message delivery to the device.

IOTREL 09. How do you plan for disaster recovery (DR) in your IoT workloads?

When companies run their core production operations and cybersecurity functions in the cloud, it is important to design resilience at the edge and cloud in IoT systems. IoT implementations must allow for loss of internet connectivity, local data storage and processing.

Best practice IOTREL_9.1 – Design server software to initiate communication only with devices that are online

Communication should be server initiated with devices that are online rather than client-server requests. This enables you to design client software to accept commands from the server.

Recommendation IOTREL_9.1.1 – *Design client software to accept commands from the server*

- FreeRTOS provides pub/sub and shadow library to connected devices.
- AWS IoT Core provides device shadow capability to persist device states.
- AWS IoT Device Registry contains a list of devices connected to AWS IoT Core. AWS IoT Device Registry lets you manage devices by grouping them.

Best practice IOTREL_9.2 – Implement multi-region support for IoT applications and devices

Cloud service providers have the same service in multiple regions. This architecture enables you to divert device data to a regional endpoint that is in not down. Data consumers should be enabled in all regions that consume the diverted device data.

Recommendation IOTREL_9.2.1 – *Architect device software to reach multiple regions in case one is not available*

- AWS IoT is available in multiple Regions with different endpoints. If an endpoint is not available, divert device traffic to a different endpoint.
- AWS IoT configurable endpoints can be used with Amazon Route 53 to divert IoT traffic to a new regional endpoint.
- AWS IoT Configurable Endpoints

Recommendation IOTREL_9.2.2 – *Enable device authentication certificates in multiple regions*

- AWS IoT provides devices with authentication certificates to verify on connection. Deploy the device certificates in the Regions where the device will connect.
- Setup the cloud side IoT data consumers to accept and process data in multiple regions.
- AWS IoT device registration

Recommendation IOTREL_9.2.3 – *Use device services in all the regions the device connects to*

- AWS IoT Rules Engine diverts device data to use multiple services. Set up AWS IoT Rules Engine in the respective Regions to divert traffic to the appropriate services.
- Rules for AWS IoT

Best practice IOTREL_9.3 – Use edge devices to store and analyze data

Edge storage can provide additional storage for device data. Data can be stored at the edge during large-scale network events and streamed later, when network is available.

Recommendation IOTREL_9.3.1 – *Use an edge device as a connection point to store and analyze data*

- AWS IoT Greengrass can be used for local processing for serverless functions, containers, messaging, storage, and machine learning inference.
- Data can be stored in AWS IoT Greengrass and sent to the network when it's available.
- AWS IoT Greengrass Features

IOTREL 10. How do you provision reliable storage for IoT data that has been sent to the cloud?

IoT devices send a lot of small messages with no guarantee of delivery order. This data might not be immediately useful, but the data volume is typically low enough to economically store against a future need. It will be beneficial to store the data so that the data can processed in order. Stored data can be reprocessed as new requirements are developed.

Best practice IOTREL_10.1 – Store data before processing

Ensure that the data from the devices is stored before processing. As new requirements and capabilities are added, stored data can be analyzed to meet the new requirements.

Recommendation IOTREL_10.1.1 – *Use IoT Core Rules Engine to send data to Firehose to batch and store data on Amazon Simple Storage Service (Amazon S3)*

- IoT Rules Engine can send data to Firehose to batch and store data on Amazon Simple Storage Service (Amazon S3). Intelligent tiering can be enabled on Amazon S3 to reduce storage costs.
- Understand the latency to access data and choose the Region to store the data in based on device location.
- If data will be processed in Amazon EC2 instances, consider using the highly available and low-latency Amazon Elastic Block Store (Amazon EBS).
- NoSQL data can be stored in Amazon DynamoDB, which is a key-value and document database that delivers single-digit millisecond performance at any scale.

Best practice IOTREL_10.2 – Have mechanisms in place to compensate when the primary storage location is unavailable

There should be recovery plans for failures in storing and accessing device data in the cloud. Understand the recovery point objective (RPO) and recovery time objective (RTO) needed by your application to access data to be used for analysis.

Recommendation IOTREL_10.2.1 – *Know how to monitor and take action on cloud storage failures for IoT data*

- AWS Health Dashboard provides notification and remediation guidance when AWS is experiencing events that might impact you. Storage and access of data can be modified based on the notification.
- Use Amazon CloudWatch Logs to trigger on events on writing and reading data and take appropriate error handling action.
  - Use AWS IoT rules engine error actions to provision data storage to other locations if primary storage is unavailable.

IOTREL 11. How do you ensure that your device accurately determines UTC?

A secure device should have a valid certificate. IoT devices use a server certificate to communicate to the cloud and the certificate presented uses time for certificate validity. Having reliable and accurate time is compulsory to be able to validate certificates. Because IoT data is not ordered, including an accurate timestamp with the data will enhance your analytic capabilities.

Best practice IOTREL_11.1 – Use NTP to maintain time synchronization on devices

IoT devices need to have a client to keep track of time—either using Real Time Clock (RTC) or Network Time Protocol (NTP) to set the RTC on boot. Failure to provide accurate time to an IoT device could prevent it from being able to connect to the cloud.

Recommendation IOTREL_11.1.1 – *Prefer NTP to RTC when NTP synchronization is available*

Many computers have an RTC peripheral that helps in keeping time. Consider that RTC is prone to clock drift of about one second a day, which can result in the device going offline because of certificate invalidity.

Recommendation IOTREL_11.1.2 – *Use Network Time Protocol for connected applications*

- Select a safe, reliable NTP pool to use, and a one that addresses your security design.
- Many operating systems include an NTP client to sync with an NTP server
- If the IoT device is using GNU/Linux, it's likely to include the ntpd daemon
- You can import an NTP client to your platform if using FreeRTOS
- The device's software needs to include an NTP client and should wait until it has synchronized with an NTP server before attempting a connection with AWS IoT Core
- The system should provide a way for a user to set the device's time so that subsequent connections can succeed.
- Use NTP to synchronize RTC on the device to prevent the device from deviating from UTC
- Chrony is a different implementation of NTP than what ntpd uses and it's able to synchronize the system clock faster and with better accuracy than `ntpd`. Chrony can be set up as a client and server.
  - https://chrony.tuxfamily.org/

Best practice IOTREL_11.2 – Provide devices access to NTP servers

An NTP server should be available for clients to use for local time. NTP servers are required by NTP clients to synchronize device time and function properly

Recommendation IOTREL_11.2.2 – *Provide access to NTP services*

- ntp.org – can be used to synchronize your computer clocks.

- Amazon Time Sync Service – a time synchronization service delivered over NTP, which uses a fleet of redundant satellite-connected and atomic clocks in each Region to deliver a highly accurate reference clock. This is natively accessible from Amazon EC2 instances and this can be pushed to edge devices.
- Chrony is a different implementation of NTP than what `ntpd` uses and it's able to synchronize the system clock faster and with better accuracy than `ntpd`. Chrony can be set up as a server and client.

Key AWS services

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

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

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

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

Resilience at the edge: AWS IoT Greengrass offers several features to help support data resiliency and backup needs with features which allow devices to communicate over the local network even after loss in internet connectivity, allowing the core to receive messages sent while the core is offline and using stream manager to process data locally until the connection is restored and send data to cloud or local storage destinations.

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
- Resilience in AWS IoT Greengrass

Performance efficiency pillar

The performance efficiency pillar focuses on using 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 resources to meet the requirements and the maintenance of that efficiency as demand changes and technologies evolve.

Topics

- Design principles
- Best practices
- Key AWS services
- Resources

Design principles

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

- **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.
- **Decouple ingestion and processing:** Decouple the connectivity portion of IoT applications from the ingestion and processing portion in IoT. By decoupling the ingestion layer, your IoT application can handle data in aggregate and can scale more seamlessly by processing multiple IoT messages at once.

- **Use event-driven architectures:** IoT systems publish events from devices and permeate those events to other subsystems in your IoT application. Design mechanisms that cater to event-driven architectures include using queues, message handling, idempotency, dead-letter queues, and state machines.

Best practices

There are four best practice areas for performance efficiency:

Topics

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

Selection

Well-Architected IoT solutions are made up of multiple systems and components such as devices, connectivity, databases, data processing, and analytics. In AWS, there are several IoT services, database offerings, and analytics solutions that enable you to quickly build solutions that are well architected while allowing you to focus on business objectives. AWS recommends that you use a mix of managed AWS services that best fit your workload. The following questions focus on these considerations for performance efficiency.

IOTPERF 01. How do you ensure your IoT application's performance and have the capabilit ies to measure it?

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

Devices

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

In addition to selecting the right hardware partner, you might choose to use a number of software components to run your application logic on the device, including FreeRTOS and AWS IoT Greengrass. You can orchestrate native OS processes on specific hardware to improve performance and you also can run containerized workloads for isolation.

Defining and analyzing key performance metrics for your IoT applications helps you to understand the performance characteristics for your application. Logging and end-to-end application monitoring are key to measuring, evaluating, and optimizing the performance of your IoT applications.

Best practice IOTPERF_1.1 – Analyze the runtime performance of your application

Application performance in production can be different from what you observe in a controlled test environment. Actively analyzing the performance of your application based on device health, network latency, and payload size provides insight on how to obtain performance improvements. By using different types of metrics, the health of each device in a multi-device setting can be obtained.

Recommendation IOTPERF_1.1.1 – *Analyze connection patterns, sensor data and set up a device security profile to detect anomalies*

- Measuring changes in connection patterns of devices might indicate some devices having a jittery network connection.
- Comparing device-side timestamps from multiple devices to arrival times on the cloud-side might indicate local network latency or additional hops in device path.
- AWS IoT Device Defender Detect
- Anomaly detection using AWS IoT
- Detect anomalies in device metrics

Best practice IOTPERF_1.2 – Add timestamps to each message published

Timestamps (ideally in UTC time) help in determining delays that might occur during the transmission of a message from the device to the application. Timestamps can be associated with the message and to fields contained in the message. If a timestamp is included, the sent timestamp, along with the sensor or event data, is recorded on the cloud-side.

Recommendation IOTPERF_1.2.1 – *Add timestamps on the server side*

- If the devices lack the capability to add timestamps to the messages, consider using server-side features to enrich the messages with timestamps that correspond to receiving the message.
- For example, AWS IoT Rules SQL language provides a timestamp() function to generate a timestamp when the message is received.

Recommendation IOTPERF_1.2.2 – *Have a reliable time source on the device*

- Without a reliable time source, the timestamp can only be used relative to the specific device. For example:
  - Devices should use the Network Time Protocol (NTP) to obtain a reliable time when connected.
  - Real Time Clock (RTC) devices can be used to maintain an accurate time while the device lacks network connectivity.

- Depending on the application, timestamps can be added at the message level or at the single payload field level. Delta encoding can be used to reduce the size of the message when multiple timestamps are included. Choosing the right approach is a trade-off between accuracy, energy efficiency, and payload size.
- Developer Guide – timestamp()

- Time series compression algorithms
- Delta encoding

Best practice IOTPERF_1.3 – Load test your IoT applications

Applications can be complex and have multiple dependencies. Testing the application under load helps identify problems before going into production. Load testing your IoT applications ensures that you understand the cloud-side performance characteristics and failure modes of your IoT architecture. Testing helps you understand how your application architecture operates under load, identify any performance bottlenecks, and apply mitigating strategies prior to releasing changes to your production systems.

Recommendation IOTPERF_1.3.1 – *Simulate the real device behavior*

- A device simulator should implement the device behavior as closely as possible. Test not only message publishing, but also connections, reconnections, subscriptions, enrollment and other environmental disruptive events. Start testing at a lower load, and progressively increase to 100%.
  - Start the load test at a low percent of your estimated total device fleet, for example, 10%.
  - Evaluate the performance of your application using operational dashboards created to measure end-to-end delivery of device telemetry data and automated device commands.
  - Make any necessary changes to the application architecture to achieve desired performance goals.
  - Iterate these steps increasing the load until you get to 100%.
- IoT Device Simulator
- From testing to scaling

Review

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

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

Monitoring

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

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

There are several key types of performance monitoring related to IoT deployments including device, cloud performance, and storage/analytics. Create appropriate performance metrics using data collected from logs with telemetry and command data. Start with basic performance tracking and build on those metrics as your business core competencies expand.

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

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

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

IoT connectivity

IOTPERF 02. How do you ensure your IoT device's performance and have mechanisms to measure it?

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

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

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

Defining and analyzing key performance metrics for your hardware devices helps you to understand the performance characteristics of the devices and how they relate to the application performances. Capturing device logs and device metrics are key to measuring, evaluating, and optimizing the performance of your IoT devices.

Best practice IOTPERF_2.1 – Capture device diagnostic data into the IoT platform

As the number of devices increases, watch out for performance bottlenecks when all the devices connect to the cloud-side. These devices could generate a large aggregate amount of data, and obtaining device diagnostics is critical for ensuring the understanding of the area of improvement. Using different types of device diagnostics, the immediate health of a device and those in proximity to that device can be obtained.

Recommendation IOTPERF_2.1.1 – *Deploy an agent to the device to start capturing the relevant diagnostic data*

- For microprocessor-based applications, consider deploying the AWS Systems Manager Agent (SSM Agent) so that you can continuously monitor your device's performance metrics.
- There are sample agents provided to use on the device-side (device or gateway). If device-side diagnostic metrics cannot be obtained, then it is possible to obtain limited cloud-side metrics. Below are some sample metrics:
  - TCP connections
    - TCP_connections
    - Connections
    - Local_interface
  - Listening TCP/UDP ports
    - Listening_TCP/UDP_ports
    - Interface
  - Network statistics
    - Bytes_in/out
    - Packets_in/out
    - Network_statistics

- Use custom metrics to define and monitor metrics that are unique to your fleet or use case.

Best practice IOTPERF_2.2 – Measure, evaluate, and optimize firmware updates

Firmware updates are critical to ensure that the IoT devices remain performant over time, but might not always have the expected impact. As you deploy firmware updates to your devices, monitoring your KPIs will ensure that the updates do not have any unintended impacts to the performance of your hardware devices or to your IoT applications.

Recommendation IOTPERF_2.2.1 – ***Implement canary deployment for device firmware***

- Deploy new firmware to a limited set of devices and monitor the impact on performance before rolling the update out to the entire fleet. Abort deployment if degradation is detected.
  - Use AWS IoT Jobs to manage OTA updates and configure it to deploy to a limited set of devices.

- After the update, evaluate end-to-end performance of the system using your previously identified KPIs.
- If performance characteristics appear to have been impacted after the firmware release, use AWS IoT Secure Tunneling, a feature of AWS IoT Device Management, to remotely troubleshoot the device.
- Release additional firmware updates to remediate identified issues.

- For more information:
  - The Internet of Things on AWS – Official Blog – Using Continuous Jobs with AWS IoT Device Management
  - The Internet of Things on AWS – Official Blog – Using Device Jobs for Over-the-Air Updates
  - The Internet of Things on AWS – Official Blog – Introducing Secure Tunneling for AWS IoT Device Management, a new secure way to troubleshoot IoT devices

Best practice IOTPERF_2.3 – Limit the number of messages that devices receive and filter out

Firmware updates are critical, and filtering messages at the edge might subject the devices to unnecessary load. This result could be counterproductive from a power and memory consumption perspective. Sending only messages that the device makes use of reduces the load on the resources and ensures better performances.

Recommendation IOTPERF_2.3.1 – *Structure the topics using the scope/verb approach.*

In this way, you can subscribe to all messages for a given scope (for example, a device) or refine the subscription on a given scope and verb.

Resources

Related documents

- Designing MQTT Topics for AWS IoT Core

IOTPERF 03. How do you ensure that your application operates within the limits set by the AWS service?

Databases

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

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

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

For storing raw, unformatted event data, use AWS IoT Analytics. AWS IoT Analytics filters, transforms, and enriches IoT data before storing it in a time series data store for analysis. Use Amazon SageMaker to build, train, and deploy machine learning models, based off of your IoT data, in the cloud and on the edge using AWS IoT services, such as machine learning inference in AWS IoT Greengrass. Consider storing your raw formatted time series data in a data warehouse solution such as Amazon Redshift. Unformatted data can be imported to Amazon Redshift using Amazon S3 and Amazon Data Firehose. By archiving unformatted data in a scalable, managed data storage solution, you can begin to gain business insights, explore your data, and identify trends and patterns over time.

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

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

IOTPERF 04. How do you bootstrap devices to use the endpoint with least latency?

In IoT, bootstrapping refers to the process of assigning identity to the device and enabling communications with an endpoint. Devices in a global fleet should be provisioned in the regional data center nearest to its physical location for minimum latency. Each device should get its regional endpoint and certificate no later than the time of bootstrapping. Each device is provisioned nearest to its physical location and gets the certificate and IoT endpoint at the time of bootstrapping. This ensures best possible latency for bidirectional communications.

Compute

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

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

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

Analytics

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

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

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

IOTPERF 05. How do you ensure that your applications operate within the limits set by the AWS service?

Being aware of the soft and hard quotas of the AWS service and continuously monitoring the key performance indicators enables you to anticipate when actions must be taken to request increases in the service quotas and re-evaluate your architecture. Ensuring that your application operates within the quotas of the services that you are building on is key to providing the optimal performance to your users.

Best practice IOTPERF_5.1 – Monitor and manage your IoT service quotas using available tools and metrics

Monitoring enables you to be aware of which service quotas you might be reaching, allowing you to change your application to cope with the hard quotas or to request the increase of a soft quota with sufficient lead time.

Recommendation IOTPERF_5.1.1 – *Familiarize yourself with the service quotas of the different IoT services*

- Pay attention to which limits are soft quotas and which are hard quotas as they require different approaches.
- For example:
  - A hard quota, such a control plane request rate, would require changes in the application behavior to avoid the event repeating too often. Workarounds for hard quotas might require different design decisions, such as using multiple accounts. It's good to know the hard and soft quotas in advance so that you can make these design decisions as early as possible in the development process.
  - Soft quotas should be monitored to anticipate the need for additional capacity and provide sufficient notice so that a request for a limit increase can be made well ahead of time.
    - For example:
      - For AWS IoT Core, alert on `RulesMessageThrottles`, `Connect.ClientIDThrottle`, `Connect.Throttle`, `PublishI`
      - For AWS IoT Analytics, alert on `ActionExecutionThrottled`, `PipelineConcurrentExecutionCount`
      - For AWS IoT Device Management, monitor active continuous jobs, and active snapshot jobs in Service Quotas
      - For AWS IoT SiteWise, monitor the quotas in Service Quotas.
- For more information:
  - AWS IoT Core
  - AWS IoT Device Defender
  - AWS IoT Device Management
  - AWS IoT Events
  - AWS IoT Greengrass
  - AWS IoT SiteWise
  - AWS IoT Things Graph
  - AWS IoT Analytics
  - AWS IoT 1-Click
  - AWS IoT Analytics CloudWatch
  - AWS IoT Core monitoring with Amazon CloudWatch
  - Service Quotas for AWS IoT SiteWise
  - Service Quotas for AWS IoT Device Management

- Amazon CloudWatch dashboards
- AWS IoT Monitoring tools
- Logging AWS IoT API calls with AWS CloudTrails

Tradeoffs

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

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

IOTPERF 06. How do you optimize the ingestion of telemetry data?

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

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

Evaluating and optimizing your IoT application for its specific needs, whether telemetry data ingestion or controlling devices in the field, ensures that you get the best outcomes in balancing performances and cost. Separating the way that your application handles data collected through sensors or device probes from command-and-control flows helps achieve better performance.

Best practice IOTPERF_6.1 – Identify the ingestion mechanisms that best fit your use case

Identify which data ingestion method best fits with your use case to obtain the best performance and operational complexity tradeoff. Multiple mechanisms might be needed. This method provides the optimal ingestion path for the data generated by your devices to obtain the best trade-offs between performance and cost.

Recommendation IOTPERF_6.1.1 – *Evaluate ingestion mechanism for telemetry data*

- Determine if the communication pattern is unidirectional (device to backend) or bi-directional. For example:
  - HTTPS should be considered if your device is acting as an aggregator and needs to send more than 100 messages per second instead of opening multiple MQTT connections. Use multiple threads and multiple HTTP connections to maximize the throughput for high delay networks as HTTP calls are synchronous.
- Consider the APIs provided by the destination for your data and adopt them if you can securely access them. For example:
  - AWS IoT Analytics provides an HTTP API that is capable of batching several messages and is suitable for high-rate data ingestion when the data is consumed in near-real-time fashion and a service-integrated data storage, data processing and data retention and replay are desired.
  - AWS IoT SiteWise provides an HTTP API to ingest operational data from industrial applications which needs to be stored for a limited period of time and processed as a time series with hierarchical aggregation capabilities.
  - Real-time video (for example, video surveillance cameras) has specific characteristics that makes it more suitable to ingest in a dedicated service, such as Amazon Kinesis Video Streams.
- Consider the need for data to be buffered locally while the device is disconnected and the transmission resumed as soon as the connection is re-established. For example:
  - AWS IoT Greengrass stream manager provides a managed stream service with local persistence, local processing pipelines and exporters to Amazon Kinesis Data Streams, AWS IoT Analytics, AWS IoT SiteWise and Amazon S3.
- Consider the latency, throughput, and ordering characteristics of the data you want to ingest. For example:

  - For applications with a high ingestion rate (high-frequency sensor data) and where message ordering is important, Amazon Kinesis Data Streams provides stream-oriented processing capabilities and the ability to act as temporary storage.
  - For applications that do not have any real time requirements (such as logging, large images) and when the devices have the possibility to store data locally, uploading data directly to Amazon S3 can be both performant and cost efficient.
- AWS IoT Core supported protocols
- AWS IoT Analytics
- AWS IoT SiteWise
- Amazon Kinesis Data Streams
- Amazon Kinesis Video Streams
- Amazon S3
- Amazon S3 pre-signed URLs
- AWS IoT Greengrass stream manager

Best practice IOTPERF_6.2 – Evaluate network connectivity and data freshness requirements

Evaluating these requirements enable you to make the right assumptions on the local data storage and data transmission needed to satisfy the requirements of your workload. It also provides a clear understanding of the requirements of the workload and allows you to determine the hardware and software needs of the devices and the platform.

Recommendation IOTPERF_6.2.1 – *Choose the right Quality of Service (QoS) for publishing the messages*

- QoS 0 should be the default choice for all telemetry data that can cope with message loss and where data freshness is more important than reliability.
- QoS 1 provides reliable message transmission at the expense of increased latency, ordered ingestion in case of retries, and local memory consumption. It requires a local buffer for all unacknowledged messages.
- QoS 2 provides once and only once delivery of messages but increases the latency.

Recommendation IOTPERF_6.2.2 – *Right size the offline persistent storage to ensure that your application objective can be obtained without wasting resources*

- The AWS IoT Greengrass message spooler can be configured with an offline message queue for messages that need to be sent to the AWS IoT Core. The size and type of storage should be configured according to the needs of the workload.

Resources

Related documents

- MQTT QoS
- Publish/subscribe AWS IoT Core MQTT messages

Best practice IOTPERF_6.3 – Optimize data sent from devices to backend services

Optimizing the amount of data sent by the devices at the edge allows the backend to better meet the processing targets set by the business. Detailed data generated at the edge might have little value for your application in its raw form.

Recommendation IOTPERF_6.3.1 – *Aggregate or compress data at the edge*

- You can aggregate data points at the edge before sending it to the cloud, such as performing statistical aggregation, frequency histograms, signal processing.
- For example, if you are using AWS IoT Greengrass, you can implement data processing at the edge with a combination of streams and Lambda functions.
- Run Lambda functions on the AWS IoT Greengrass core
- Industrial OEE workshop
- AWS IoT Greengrass stream manager

Key AWS services

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

Devices: AWS hardware partners provide production ready IoT devices that can be used as part of you IoT application. FreeRTOS is an operating system with software libraries for microcontrollers.

AWS IoT Greengrass allows you to run local compute, messaging, data caching, synchronization, 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 initiated from AWS IoT Core or services such as Amazon Kinesis and Amazon SQS.

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

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

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

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

Resources

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

Documentation and blogs

- AWS Lambda Getting Started
- DynamoDB Getting Started
- AWS IoT Analytics User Guide
- FreeRTOS Getting Started

- AWS IoT Greengrass Getting Started
- AWS IoT Blog

Cost optimization pillar

The cost optimization pillar includes the continual process of refinement and improvement of a system over its entire lifecycle. From the initial design of your first proof of concept to the ongoing operation of production workloads, adopting the practices in this paper will enable you to build and operate cost-aware systems that achieve business outcomes and minimize costs, allowing your business to maximize its return on investment.

Topics

Design principles

In addition to the overall Well-Architected Framework cost optimization design principles, there are three design principle for cost optimization for IoT:

- **Manage manufacturing cost tradeoffs:** Business partnering criteria, hardware component selection, firmware complexity, and distribution requirements all play a role in manufacturing cost. Minimizing that cost helps determine whether a product can be brought to market successfully over multiple product generations. However, taking shortcuts in the selection of your components and manufacturer can increase downstream costs. For example, partnering with a reputable manufacturer helps minimize downstream hardware failure and customer support cost. Selecting a dedicated crypto component can increase bill of material (BOM) cost, but reduce downstream manufacturing and provisioning complexity, since the part may already come with an onboard private key and certificate.
- **Avoid unnecessary data access, storage, and transmission:** Identify and classify data collected throughout your IoT environment and learn their corresponding business use-case. Then identify and execute on opportunities to stop collecting unused data or adjusting their granularity and retention time.

- **Process data at the edge whenever possible:** Processing data at the edge can help to save costs. Process large volumes of data locally, upload only relevant insights and high-value data to the cloud.

Best practices

There are five best practice areas for cost optimization:

Topics

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

Practice Cloud Financial Management

There are no Cloud Financial Management best practices specific to the IoT Lens.

Expenditure and usage awareness

There are no expenditure and usage awareness best practices specific to the IoT Lens.

Cost-effective resources

Given the scale of devices and data that can be generated by an IoT application, using the appropriate AWS services for your system is key to cost savings. In addition to the overall cost for your IoT solution, IoT architects often look at connectivity through the lens of bill of materials

(BOM) costs. For BOM calculations, you must predict and monitor what the long-term costs will be for managing the connectivity to your IoT application throughout the lifetime of that device. Leveraging AWS services will help you calculate initial BOM costs, make use of cost-effective services that are event driven, and update your architecture to continue to lower your overall lifetime cost for connectivity.

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

IOTCOST 01. How do you choose cost-efficient tools for data aggregation of your IoT workloads?

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

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

Raw data from devices can also be processed at the edge using AWS IoT Greengrass thus eliminating the need to send all the data to the cloud for storage and processing. This can result in lower network cost and lower cost in cloud services. You can dynamically change or update that logic, as well as frequency of transmission using AWS IoT Greengrass since it's not hardcoded and can be adjusted as needed by the use case. This gives you added flexibility for cost optimization.

In addition, observe the following general practice recommendations:

Methods and tools for how data is acquired, validated, categorized, and stored impacts the overall cost of your application. Focusing on tools that can automatically vary in scale and cost with demand and support your data with a minimum of administrative overhead can help you achieve lowest cost for your application. By considering the data pipeline from origination to archival, you can make informed decisions and examine tradeoffs among technical and business requirements to identify the most effective solution.

Best practice IOTCOST_1.1 – Use a data lake for raw telemetry data

A data lake brings different data sources together and provides a common management framework for browsing, viewing, and extracting the sources. An effective data lake enables IoT cost management by storing data in the right format for the right use case. With a data lake, storage and interaction characteristics can be aligned to a specific dataset format and required interfaces.

Recommendation IOTCOST_1.1.1 – *Categorize telemetry types and map to storage capabilities*

- For each telemetry stream, identify key features of telemetry using the 4Vs of big data—velocity, volume, veracity, and variety.
- Map each stream into the appropriate storage capability. For example, a stream that sends an MQTT message with a JSON payload every second would be an ideal candidate for being batched, compressed then stored in Amazon S3.
- For more information:
  - AWS storage types
  - AWS re:Invent 2018: Building with AWS Databases: Match Your Workload to the Right Database (DAT301)

Best practice IOTCOST_1.2 – Provide a self-service interface for end users to search, extract, manage, and update IoT data

With inexpensive cloud computing resources, pay-as-you-go pricing, and strong identity and encryption controls, your organization should allow groups to define and share data models in the format that makes the most sense for them. Self-service interfaces will encourage experimentation and speed up change by removing barriers for teams to gain access to the data they need to make decisions.

Recommendation IOTCOST_1.2.1 – *Use an architecture that allows various end users to easily find, obtain, enhance, and share data*

Recommendation IOTCOST_1.2.2 – *Use a subscriber model, which allows teams to subscribe to data feeds and receive notification of updates, to reduce the need for active polling and re-synching with data sources*

For more information:

- Creating a data lake from a JDBC source in AWS Lake Formation
- AWS Data Lake Quick Start
- AWS Data Exchange offers subscriptions to third-party data sources with notification on updates

Best practice IOTCOST_1.3 – Track and manage the utilization of data sources

Applications and users evolve over time, and IoT solutions can generate large volumes of data quickly. As your application matures, it's important for cost management of your IoT workload to track that data collected is still being used. Consistent tracking and review of data utilization provides an objective basis for cost optimization decisions.

Recommendation IOTCOST_1.3.1 – *Track and manage the utilization of data sources to identify hot and cold spots to assess value of data*

- Track access rates and storage trends for your data lake sources.
- Use automated guidance tools, such as AWS Cost Explorer and AWS Trusted Advisor, to identify under-utilized or resizable components of your workload. AWS Cost Explorer has a forecast feature that predicts how much you will use AWS services over the forecast time period you selected.
- Use AWS Budgets and AWS Cost Anomaly Detection to prevent surprise bills.
- For more information:
  - Monitoring Amazon S3 metrics with Amazon CloudWatch
  - Find cost of your S3 buckets using AWS Cost Explorer
  - Forecast your spend using Cost explorer
  - Best practices for AWS Budget

Best practice IOTCOST_1.4 – Aggregate data at the edge where possible

Data aggregation is an architectural decision that can have impacts on data fidelity. Aggregations should be thoroughly reviewed with engineering and architectural teams before implementation. If the device can aggregate data before sending for processing using methods such as combining

messages or removing duplicate or repeating values, that can reduce the amount of processing, the number of associated resources, and associated expense.

Recommendation IOTCOST_1.4.1 – *Examine device telemetry for opportunities to batch and aggregate data*

- A common mechanism includes combining multiple status updates to a final status, or combining a series of measurements generated by the device into a single message.
- For example, 10 KB of device telemetry data might be packaged as one 10-KB message, two 5-KB messages, or 10 1-KB messages. Each packaging format has implications outside of cost such as network traffic (10 1-KB messages will each add their own header messaging as opposed to a single 10-KB message with one header) and the impact of a lost or delayed message. Optimizing message size should consider how a lost message impacts the functional or non-functional characteristics of the system.

Recommendation IOTCOST_1.4.2 – *Use cost calculators to model different approaches for message size and count*

- The AWS Pricing Calculator can estimate IoT costs for specific message sizes, traffic, and operations.

IOTCOST 02. How do you optimize the cost of raw telemetry data?

Raw telemetry is an original source for analytics but can also be a major component of cost. Analyze the flow and usage of your telemetry to identify the right service and handling process required. Select storage and processing mechanisms that match the needs of your specific telemetry case.

Best practice IOTCOST_2.1 – Use lifecycle policies to archive your data

When selecting an automated lifecycle policy for data, 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. Use your organization's data classification strategies to define a lifecycle policy to take raw telemetry measurements through various services. Setting milestones by time sets expectations and encourages aggregation and production of data over mere collection.

Recommendation IOTCOST_2.1.1 – *Evaluate your organization's data retention and handling requirements and configure your AWS services to support them*

- Check your organization's data management policy for requirements on retention, deletion, and encryption and align your retention policies and tools with those guidelines.
- S3 Lifecycle policies or S3 Intelligent-Tiering can move the data to the most cost-effective Amazon S3 storage class or Amazon S3 Glacier for long-term storage.

Best practice IOTCOST_2.2 – Evaluate storage characteristics for your use case and align with the right services

Not all data needs to be stored in the same way, and data storage needs change through a data item's lifecycle. A growing fleet of devices can exponentially scale its messaging rate and device operation traffic. This scaling of message volumes can also mean an increase in storage costs.

Recommendation IOTCOST_2.2.1 – *Evaluate velocity and the volume of data coming from IoT devices when selecting storage services*

- For data at high scale of devices, time, or other characteristics—Consider a data warehouse such as Amazon Redshift or Amazon S3 with Amazon Athena. The data partitioning and tiering features of AWS storage services can help reduce storage costs.
- For data at lower scale of time, devices, or other characteristics—Consider Amazon DynamoDB, Amazon OpenSearch Service (OpenSearch Service), or Amazon Aurora for short-term historical data. Use your data lifecycle policies to optimize what is kept in the short-term storage.

Best practice IOTCOST_2.3 – Store raw archival data on cost effective services

Using the right storage solution for a specific data type will align costs with usage.

Recommendation IOTCOST_2.3.1 – *Use an object store for archival storage*

- Use an object store, such as Amazon S3, for raw archival storage. Object stores are immutable and often more efficient and cost-effective than block storage, especially for data which doesn't require editing.
- Avoid costs by using a schema-on-read service, such as Amazon Athena, to query the data in its native form. Using Athena can help reduce the need for large-scale storage arrays or always-on databases to read raw archival data.

IOTCOST 03. How do you optimize the cost of interactions between devices and your IoT cloud solution?

Interactions to and from devices can be a significant driver of your workload's overall cost. Understanding and optimizing interactions between devices and cloud solution can be a significant factor of cost management.

Best practice IOTCOST_3.1 – Select services to optimize cost

Understand how services use and charge for messaging, as well as operating modes that offer cost benefits. Understanding service billing characteristics can help you identify ways to optimize messaging, which could result in considerable cost savings at scale.

Recommendation IOTCOST_3.1.1 – *Select services to optimize cost*

- Review your IoT architecture to find communication patterns and scenarios that could map to service discount features.
- With AWS IoT Core Rules Engine Basic Ingest, you can publish directly to a rule without messaging charges.
- Use your registry of things only for primarily immutable data, such as serialNumber.
- For your device's shadow, minimize the frequency of reads and writes to reduce the total metered operation and your operating costs.
- For more information:
  - AWS IoT Rules Engine Basic Ingest
  - AWS IoT Pricing

Best practice IOTCOST_3.2 – Implement and configure telemetry to reduce data transfer costs

Matching the precision of telemetry data, such as the number of decimal places, to the precision of the required calculation can help address both the overall message size and the precision of calculations.

Recommendation IOTCOST_3.2.1 – *Reduce string lengths and decimal precision where feasible*

- For example, strings sent regularly such as "POWER" or "CHARGE" could be reduced to the strings "P" or "C". Similarly, decimal values such as "21.25" or "71.86" could be reduced to

"21" or "72" if the additional precision is not required for the application. This is common in room temperature readings where precision beyond a whole number is rarely required. Across many millions of messages, the savings from removing a few characters can make a significant difference in message size and cost.

Best practice IOTCOST_3.3 – Use shadow only for slow changing data

Shadow is used in IoT applications as a persistence mechanism of device state. The shadow maintains data that remains consistent across multiple points in time. Device shadow operations can be billed and metered differently than publish/subscribe messages. Reducing the shadow update frequency from the device can reduce the number of billed operations while maintaining an acceptable level of data freshness.

Recommendation IOTCOST_3.3.1 – *Avoid using shadows as a guaranteed-delivery mechanism or for continuously fluctuating data*

- As a workload scales up, the cost of frequent shadow updates could exceed the value of the data.
- Consider MQTT Last Will and Testament (LWT) as a mitigation for the risk of loss of device communication instead of using shadow.
- Use the AWS Pricing Calculator to compare device shadow interactions versus telemetry messages understand cost implications.
- For more information:
  - Last Will and Testament (LWT) Lifecycle Event

Best practice IOTCOST_3.4 – Group and tag IoT devices and messages for cost allocation

An IoT billing group enables you to tag devices by categories related to your IoT application. You should create tags that represent business categories, such as cost centers. Visibility into devices and messages by category makes cost dimensions easier to understand and visualize.

Recommendation IOTCOST_3.4.1 – *Use AWS IoT Core Billing Groups to tag IoT devices for cost allocation*

- Add tracking elements to messages and devices to help trace source such as product model and serial number.
- Ensure that your system can group and organize data by both technical and business entity.
- For more information:

- Tagging IoT Billing Groups

Best practice IOTCOST_3.5 – Implement and configure device messaging to reduce data transfer costs

Charges for different cloud and data transporter providers can vary based on specifics of message size and frequency. IoT workloads can cross multiple communication, such as cell networks, and each layer can have its own metering and pricing standards.

Recommendation IOTCOST_3.5.1 – *Evaluate tradeoffs between message size and number of messages*

- Frequency optimization is performed with payload optimization to both accurately assess the network load and identify adequate trade-offs between frequency and payload size.
- For example, your devices might send one message per second. If you could aggregate those messages on the device and send five observations in a single message every five seconds, that could drastically reduce your message count and cost.

Recommendation IOTCOST_3.5.2 – *Evaluate cost of streaming services versus IoT messaging services*

- Use the AWS Cost Calculator to compare the cost of using messaging services like Kinesis and API Gateway to offload components of your IoT workload.

Manage demand and supplying resources

Optimally matching supply to demand delivers the lowest cost for a system. However, given the susceptibility of IoT workloads to data bursts, solutions must be dynamically scalable and consider peak capacity when provisioning resources. With the event driven flow of data, you can choose to automatically provision your AWS resources to match your peak capacity and then scale up and down during known low periods of traffic.

IOTCOST 04. How do you optimize cost by matching the supply of resources with device demand?

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

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

Optimize over time

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

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

IOTCOST 05. How do you optimize payload size between devices and your IoT platform to save cost?

IoT applications must balance the networking throughput that can be realized by end devices with the most efficient way that data should be processed by your IoT application. We recommend that IoT deployments initially optimize data transfer based on the device constraints. Begin by sending discrete data events from the device to the cloud, making minimal use of batching multiple events in a single message. Later, if necessary, you can use serialization frameworks to compress the messages prior to sending them to your IoT platform.

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

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

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

IOTCOST 06. How do you optimize the costs of storing the current state of your IoT device?

Well-Architected IoT applications have a virtual representation of the device in the cloud. This virtual representation is composed of a managed data store or specialized IoT application data store. In both cases, your end devices must be programmed in a way that efficiently transmits device state changes to your IoT application. For example, your device should only send its full device state if your firmware logic dictates that the full device state might be out of sync and would be best reconciled by sending all current settings. As individual state changes occur, the device should optimize the frequency it transmits those changes to the cloud.

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

Cost optimization processes for device shadows and registry focus on managing how many operations are performed and the size of each operation. If your operation is cost sensitive to shadow and registry operations, you should look for ways to optimize shadow operations. For example, for the shadow you could aggregate several reported fields together into one shadow

message update instead of sending each reported change independently. Grouping shadow updates together reduces the overall cost of the shadow by consolidating updates to the service.

Key AWS services

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

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

Resources

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

Documentation and blogs

- AWS IoT Blogs

Sustainability pillar

The sustainability pillar includes the ability to continually improve sustainability impacts by reducing energy consumption and increasing efficiency across all components of a workload by maximizing the benefits from the provisioned resources and minimizing the total resources required.

The term *sustainability* encompasses a wide range of factors such as economic viability, social equity, biodiversity, resilience, environmental impact, and more. We will focus specifically on how to design your IoT solutions to improve sustainability by lowering their carbon footprint while simultaneously reducing operational costs.

Reducing carbon footprint is a key aspect of mitigating climate change, which is a pressing global environmental challenge. One way to achieve this is to use resources in a responsible and efficient manner, minimize waste and maximize the use of renewable resources. As described later in this paper, the actions taken to improve sustainability usually have a positive effect on operational costs, not just through resource efficiency, but also by optimizing operational processes.

While IoT solutions span the range from sensors and devices at the edge to applications in the cloud, this paper focuses largely on sustainability at the edge. Sustainability *of* and *through* the cloud is covered in the Sustainability Pillar of the AWS Well-Architected Framework.

It is important to recognize that the carbon footprint of an IoT device is distributed across its entire lifecycle, consisting of the design and build phase, the operational (in-use) phase, and the disposal phase.

- The *design and build phase* deals with design, component sourcing, manufacturing, and other supply chain activities. This phase usually accounts for most of the carbon footprint associated with a device—referred to as the *embodied carbon*.
- The *operational phase* of the device lifecycle has a significant carbon footprint impact as well. The Reaching Net-Zero Carbon by 2040: Decarbonizing and Neutralizing the Use Phase of Connected Devices whitepaper shows that this phase accounts for 10-15% of the entire lifecycle carbon footprint for re-chargeable battery powered devices and 60-80% for plugged-in devices.
- The *disposal phase* of IoT devices refers to the stage when a device is no longer needed or usable in operation.

Implementing the best practices outlined here involves trade-offs between cost, reliability, performance, and carbon footprint. Your organization should examine these best practices, both holistically and individually, and agree on how to best meet your sustainability goals for each use case.

Terminology

> **Note**
>
> The terms *energy efficiency* and *power efficiency* are used interchangeably in this document.

Truck roll, or *site visit*, refers to the act of sending a technician to a site to resolve an issue. This has impacts both from an operational cost perspective (technician labor, fuel costs, maintenance

running to hundreds of dollars per device) as well as carbon footprint (emissions, wear and tear). Consequently, designing IoT devices to minimize the need for truck rolls is a key imperative.

Battery life refers to low long a device can run on a single charge.

Battery lifetime refers to how long the battery can be in operation before needing replacement.

Design principles

In addition to the overall Well-Architected Framework design principles, there are additional design principles for IoT.

Topics

- Buffer or spool messages
- Optimize the frequency of messages for your use case
- Use gateways to offload and pre-process your data at the edge
- Perform analytics at the edge
- Monitor and manage your fleet operations to maximize sustainability

Right-size your hardware

The choices made in hardware component selection can greatly influence the operational phase impact of devices and therefore must be carefully considered. The use case should dictate the required peripherals and communication modules on the device. When choosing the processor, amount of RAM, and amount of flash storage, the designer must focus on resource optimization; over-provisioning hardware can lead to choosing a processor that has a large power draw but stays mostly idle or has an excess of memory that does not get used, increasing the overall carbon footprint of the device unnecessarily. Under-sizing hardware can lead to long runtimes or a lack of extensibility necessary to insure the longevity of the device.

Your design will typically fall into one of three broad classes of devices – Microcontroller (MCU) class devices, Microprocessor (MPU) class devices, and devices that use hardware acceleration for machine learning use cases (which we refer to as *inference class* devices).

Considerations for general purpose IoT devices

For general purpose IoT devices, the application and use case will determine the choice between microcontroller (MCU) and microprocessor (MPU). MCU devices are typically designed for low-power, resource-constrained applications. MPU class devices are designed for applications that require higher computational power, multitasking, and higher-level services provided by more capable operating systems.

For MCU devices, there are several features that can contribute to improved energy efficiency and sustainability.

- Support for multiple low power modes, allowing power consumption to be reduced when the processor is not active.

  This should include a mode that retains volatile memory content, allowing for quick restoration of application state when required.
- A Real-Time Clock (RTC) to wake up the device from a low power mode only when needed.

- On-chip or on-board crypto accelerators to enable secure communication while minimizing energy consumption.
- Connectivity module low-power and sleep modes to reduce unnecessary power drain when not in use.
- If required, support for a floating point unit (FPU) for more efficient and faster processing of numerical calculations, contributing to improved energy efficiency.

Similar criteria also apply to the selection of MPUs, though there are differences. Some MPUs use a combination of high-performance and low-power cores, such as those using the Arm big.LITTLE architecture. In such architectures, ensure that tasks are assigned to the appropriate core based on the workload. This reduces power consumption while maintaining sufficient processing capabilities. The FreeRTOS operating system can take advantage of this architecture through Asymmetric Multiprocessing (AMP). Each core runs an instance of FreeRTOS and communicates through shared memory space and interprocess communication (IPC). The smaller core can be assigned to run non-intensive applications and the large core can be put into low-power mode until needed for more compute intensive tasks.

Choose the right CPU

To optimize energy consumption, choose a CPU that meets the performance requirements of your application without exceeding them. Keep in mind that a more powerful processor might have higher performance per watt compared to a less powerful processor, but can perform more work per unit of power consumed, resulting in lower energy consumption. Newer CPU architectures or higher-end processors may have better power efficiency at higher workloads due to advancements in technology or microarchitecture, resulting in a more energy-efficient solution overall. Use performance per watt as a guideline, and test CPUs against your actual workload to make a final choice.

Choose a CPU that is designed for power efficiency. These CPUs feature sleep and idle states to reduce power consumption during periods of inactivity. These states allow the CPU to minimize power usage by either reducing the clock frequency, halting execution, or shutting down certain components when they are not required.

The instruction set architecture (ISA) of a CPU, which defines its instruction set and programming model, can also impact power consumption. Some features, such as complex out-of-order execution and speculative execution, can increase power consumption due to the increased hardware complexity and activity. RISC (Reduced Instruction Set Computing) architectures, such

as ARM and RISC-V, are generally known for their power efficiency compared to CISC (Complex Instruction Set Computing) architectures, such as x86.

Choose a processor to minimize the energy used by your workload

Workload characteristics impact the power effectiveness of a processor. Some workloads, such as data-intensive or compute-intensive tasks, may require more processing power to achieve a desired level of performance. In such cases, a more powerful processor with higher power consumption may be able to complete the workload faster, resulting in shorter overall runtime, and potentially lower total energy consumption than a less powerful processor running the same workload for a longer duration. If a workload is optimized for a particular processor architecture or has specific requirements that can be better met by a more powerful processor, then using that processor may result in lower energy consumption overall, despite its higher power consumption. Test with your expected workload to help narrow down a processor choice.

Choose a processor with advanced power management features

Look for CPUs with advanced power management features, such as dynamic voltage and frequency scaling (DVFS), clock gating, and idle state management. These features can help dynamically adjust the CPU's performance and power consumption based on workload requirements and system conditions, leading to improved power efficiency.

The Thermal Design Power (TDP) of a CPU specifies the amount of heat it is expected to dissipate under maximum or typical load conditions. Higher temperatures impact power efficiency negatively due to increased leakage currents and resistance. Look for CPUs with a low TDP, as these typically consume less power and generate less heat, which can help the overall power efficiency of a system. In addition, low TDP CPUs can help reduce the need for active cooling systems and the associated power consumption.

Finally, analyze the entire system, including components such as memory, storage, and peripherals, as they can also impact the overall power consumption. Optimizing the system-level power management, including power states, sleep modes, and power management settings, can contribute to improved power efficiency.

Use accelerators for machine learning inference

Performing machine learning (ML) inference on the IoT device can greatly reduce the amount of data transmitted to the cloud. ML inference is a computationally intensive process that might not be energy optimal when run on a CPU without the right instruction set. For ML applications, the

use of a CPU with additional vector operations and specialized acceleration hardware can lower the energy consumption of IoT devices.

Inference-class devices can come in various forms such as GPUs (Graphics Processing Units), NPUs (Neural Processing Units), Digital Signal Processors (DSPs), and FPGA (Field-Programmable Gate Array) devices or CPUs with vector manipulation operators.

If there are no performance or latency constraints in the workload it might be preferable to run inference on standard MCU devices, for cost and energy savings. For further optimization, resources such as TinyEngine and other frameworks can be used to run machine learning models directly on the microcontroller.

When lower latency than what is achievable by the MCUs is desired, specialized hardware accelerators for ML tasks, such as built-in Digital Signal Processors (DSPs) or ML accelerators, can provide efficient ML inferencing at the edge with lower power consumption.

DSPs are designed to perform mathematical functions very quickly without using the host MCU's clock cycles. They are more power efficient than an MCU. In a typical DSP-accelerated ML application, such as wake-word detection in a smart speaker, the DSP first processes an analog signal such as an audio or voice signal and then wakes up the host MCU from a deep-sleep mode via an interrupt. The processor can thus remain in a low-power mode while performing inference, and only wakes up when necessary for further processing or connection to the cloud.

MPUs are suitable for edge ML applications that require higher processing capabilities, such as running more complex ML models or handling larger input datasets. MPUs may also have built-in hardware accelerators for ML tasks, such as NPUs that improve ML inferencing performance.

NPUs are optimized for artificial neural networks. If your application involves inference using deep neural networks, such as image recognition, natural language processing, or recommendation systems, an NPU can provide inference acceleration and an order of magnitude or more energy efficiency compared to general-purpose CPUs or GPUs.

GPUs are specialized processors designed for graphics-intensive tasks, but can offer high performance for ML inference. If you are already using a deep learning framework or software that is optimized for GPUs, it may be more convenient to continue using GPUs. GPUs are not very power efficient and should only be selected for the highest intensity workloads.

Choose storage that supports device longevity

There are several low-power memory storage options that are suitable for IoT devices due to their energy-efficient characteristics. When choosing storage, the use case and workload dictates much

of the decision. For example, for a device that is deployed to a harsh environment that is not easy to access, choose storage that is durable and resilient. Also consider the wear and tear on flash memory due to the workload, and its impact on device longevity. Provision the amount of storage beyond current requirements for the application to allow for additional use cases in the future, thereby extending the device's useful life. In summary, the appropriate type and amount of storage for the device application should be carefully chosen as it can have an impact on the operational phase of the device as well as the overall cost.

File system choices can impact processor and memory requirements for the device, in turn impacting the power draw and ultimately sustainability. Choosing a lightweight file system such as littleFS or Reliance Edge for embedded systems can help reduce power consumption and increase the life-time of the storage through features such as wear leveling, power-efficient file updates, power-safe operations, small footprint, and customizable configuration options.

For IoT use cases that demand higher power edge gateways, various disk drives or solid state drives may also be used. Using techniques like RAID, though more power intensive, can improve the performance and resiliency of gateways or servers and in turn improve the overall sustainability of the system by reducing the need for truck rolls.

A key consideration for storage selection is to support dual partitioning of persistent storage for over-the-air update (OTA) capability. In addition, the flash subsystem must be capable of independent erase and write operations for each partition (or image block).

When deciding on storage options for a use case, satisfying just the requirements of the application will not always be the most sustainable practice. Think about the performance of the storage over the life of the device, whether the device has enough storage to support buffering/analysis of data to reduce the need to transmit, and if the storage supports extensibility and longevity of devices to reduce the operational impact and carbon footprint of upgrades and replacements.

Choose a power source with high efficiency

Efficiency of the power source is a crucial factor to consider in IoT device design. It is recommended to choose power supplies with at least 90% efficiency across all load conditions, such as 80PLUS Titanium-rated power supplies. This can help minimize wasted energy and reduce the overall power consumption of the IoT device, thus lowering the device's carbon footprint.

Dimension and manage batteries to maximize battery life

If the IoT device is powered by a battery or secondary battery, it is important to consider the minimum useful lifetime of the battery under normal deployment conditions. This impacts how frequently a technician visit is required for battery replacement, which in turn affects operational costs and potential waste from battery replacements. In addition, implementing techniques such as low-power sleep modes can extend battery life and reduce required battery dimensions. This consideration can help reduce operational costs and minimize the environmental impact of your IoT devices.

Choose an operating system that is appropriate for the type of device's features and functionality

The operating system can greatly impact how power efficient an IoT device is.

Try to choose an operating system that is appropriate for the type of device's features and functionality. For example, choosing Android or Linux for a simple temperature sensor might be more convenient for the developer, but is not sustainable practice due to its impact on the device's resource dimensions and consequent carbon footprint.

Consider features in the operating system that support low power modes, power management, and operational efficiency. Operating systems must support low power modes that allow the system to conserve power when idle or when certain subsystems are not in use. For example, an embedded system might use a sleep mode to conserve power when the user is not interacting with the system, or it might shut down peripherals or subsystems that are not currently in use. The operating system should expose the hardware's low power modes such that applications can take advantage of these modes.

Some operating systems support tick-less operation, where the operating system avoids unnecessary timer interrupts, further reducing power consumption. For example, FreeRTOS stops the periodic tick interrupt during periods when there are no application tasks that are able to run. Stopping the tick interrupt allows the microcontroller to remain in a deep power saving mode until an event occurs or the kernel is ready to run a task.

Dynamic voltage and frequency scaling (DVFS) operations adjust the CPU performance and frequency based on the application workload's demands. Operating systems that support DVFS can decrease the CPU's voltage in real time during decreased workloads, reducing power consumption.

Use an event-driven architecture in your IoT devices

Using an event-driven architecture in IoT device firmware reduces processing and communication overhead, and helps reduce energy consumption. There are a number of mechanisms that can be used in IoT devices to realize an event-driven architecture.

Design device software to minimize the amount of energy IoT devices consume while checking for events or data. By using interrupts instead of continuous polling, the CPU can sleep until an event occurs, reducing the time the device spends actively checking for data or events.

Another mechanism is Pub/Sub topics (such as those used by the MQTT protocol). Devices that subscribe to specific topics receive notifications when new messages are available, allowing for appropriate processing to be triggered.

Asynchronous callbacks are another technique used to realize an event-driven architecture. In this model, the device firmware registers callback functions to be run when specific events occur. This enables efficient processing of events without the need for polling or continual querying.

Real time operating systems such as FreeRTOS are built for event-driven firmware by employing the use of priority interrupts, pointers to callback functions, and multi-threading.

Choose a power efficient programming language

Programming languages have varying levels of efficiency when it comes to power consumption. Some languages, such as C and C++, are known for their low-level, close-to-hardware nature, which allows for fine-grained control over system resources and can result in more power-efficient code. On the other hand, interpreted languages like Python or JavaScript, may have higher levels of abstraction and runtime overhead, which can result in several times higher power consumption, as outlined in this AWS blog.

Modern compilers often provide optimization options that can reduce code size, decrease unnecessary computations, and optimize register usage, which can lead to more power-efficient code. For code sections that are run at a high frequency (or are time sensitive), enabling compiler optimizations, such as loop unrolling, function inlining, and dead code elimination, can result in more efficient code execution and lower power consumption.

Optimize ML models for the edge

Leveraging edge computing for machine learning (ML) inference and analytics can offer several benefits compared to processing data in the cloud. Pre-processing and real-time data analysis and

decision making can be done locally, reducing the need for frequent data transfers to the cloud, thereby reducing messaging costs, computing power requirements, and the energy consumption associated with data transmission as well as processing in the cloud.

Machine learning is a computationally expensive task. Choose machine learning frameworks and algorithms that are optimized for low-power consumption and can run on hardware accelerators. Models can be developed in the cloud, and then optimized for edge devices with frameworks such as the Open Neural Network Exchange, ONNX, before being deployed to the device.

The size and complexity of ML models can impact their suitability for edge deployment. Techniques such as model quantization, compression, and pruning can help reduce the size of ML models, making them more suitable for deployment on edge devices.

ML models should be observable while deployed on edge devices in order to detect if the model is being affected by changing data or model drift. When negative model performance is detected, updating the model over-the-air increases the device's useful lifetime, reducing the need for replacement.

Use over-the-air device management

Over-the-air device management refers to operations to update, secure, and configure your IoT devices from the cloud. Devices that go offline unexpectedly or have critical security vulnerabilities often require expensive site visits or shipment of a device to a refurbishment center. To reduce the carbon footprint and operational costs across your device fleet, your device must support one or more capabilities for over-the-air device management.

Over-the-air (OTA) updates

OTA updates enable IoT devices to receive and install software or configuration updates without the need for physical access to the devices. This mechanism is supported across a variety of embedded operating systems, including embedded Linux and FreeRTOS, and can be powered by the AWS IoT Jobs agent and cloud service.

The OTA update mechanism for IoT devices must be resilient, reliable, and secure in order to prevent a failed update from requiring a truck roll to fix the device. There are several techniques that device makers can use to build a resilient OTA mechanism.

OTA updates may encounter errors or failures during the update process, such as network errors, data corruption, or other issues. The device's firmware should have error handling and recovery mechanisms in place to detect and recover from potential errors, such as checksum

verification, error correction codes (ECC), redundancy, or other suitable techniques to ensure data integrity and reliability.

Atomic updates are updates that are either fully applied or fully rolled back, without leaving the system in an inconsistent state. This can be achieved by storing a new version of the firmware on an inactive partition and then swapping it with the active firmware partition upon successful completion. This approach also supports safe rollback in case the updated firmware encounters any errors.

Once an OTA update is installed and running correctly, the device must prevent it from being rolled back to a previous vulnerable version. This can be achieved through mechanisms such as secure bootloaders, cryptographic signatures, or other techniques that prevent the device from reverting to earlier, potentially less secure, firmware or software versions.

OTA updates can also be used for certificate rotation when there is a security incident or to renew an expiring certificate. A certificate rotation mechanism extends the device's lifetime without the need to use long-lived certificates and allows the device to benefit from improvements in security algorithms and ciphers that might not have been available at the time the device was manufactured.

Devices on networks such as LoRaWAN and NB-IoT face additional challenges. These networks are constrained in bandwidth, making OTA updates power inefficient for large file transfers. Sending large files over-the-air is also problematic for battery-powered devices connected to more power-hungry Wi-Fi and cellular networks. To overcome this issue, instead of sending the entire firmware image to a device, send only the portions of the image that have actually changed, reducing communication and processing required, and reducing power consumption. The Delta Over the Air Updates feature supported by FreeRTOS uses this approach.

Remote access

To reduce the need to send people on site in case of a malfunctioning system, it is recommended to provide a remote access capability on the device. When IoT devices are deployed in the field, remote access provides a way to troubleshoot, change the configuration, access files such as logs, and perform other operational tasks even if the device is behind a firewall or private network. Users can update devices through its command line interface or access the device's package manager to add new software via Secure Shell (SSH) or Remote Desktop Protocol (RDP).

Use AWS IoT Secure Tunneling to establish bidirectional communication to remote devices over a secure connection that is managed by AWS IoT. Secure tunneling does not require updates to your

existing inbound firewall rules, so you can keep the same security level provided by firewall rules at a remote site without adding operational overhead.

AWS Systems Manager is another AWS service that you can use to view and control your edge devices. Systems Manager enables you to view operational data, automate operation tasks, and maintain security and compliance through remote device access.

Choose a communication technology that is optimal for your use case

The communications layer deals with connectivity to a network, message routing among remote devices, and routing between devices and the cloud. Communications, whether over a wired or wireless network, can be a significant consumer of power and compute for an IoT device. Care must be taken to minimize this power draw when designing the hardware of a device as well as the application. Choose an optimal connectivity type from options available at the device's operating location to support data transfer with minimal power, optimal connected time and minimal retries.

Wireless connectivity

For wireless connectivity there are many design choices that need to be made. For nearly all cases, the most power efficient applications are those that minimize the amount of time that the radio or network interface is on. For example, an LTE-M module's active transmission mode has thousands of times higher power consumption than its Power Save Mode (PSM). Typically, the transmission phase uses the most power and can be particularly penalizing to battery powered devices.

There are a few best practices applicable to all Low Power Wide Area (LPWA) use cases:

- Reduce the amount of time the radio is on.
- Use the appropriate technology for the use case. Using the wrong technology type can lead to data retransmission and reduced power efficiency.
- Reduce the amount of data transmitted.
- Use LTE network session resumption mechanisms as much as possible to reduce lengthy handshakes.
- Use advanced error handling techniques and advanced buffering techniques to properly manage degraded network conditions.
- Optimize application and network settings to improve the chances of successful communication.
- Monitor your device fleet via network information to continuously optimize device applications.

The following sections provide details on specific wireless technologies, along with guidance from the above list that is specific to each technology.

Cellular technologies

There are several different generations of cellular technology that can be used in IoT applications. Given this range of choices, it is important to select the right technology for your use case in order to optimize power consumption and cost, as different technologies will have different power implications. These technologies are discussed below.

LTE Categories above Cat. 1, 5G Sub-6GHZ, and 5G mmWave

These categories of cellular technologies are high power and high bandwidth technologies that are suited for use cases such as wireless cameras that require high bandwidth mobility. Using these types of radio access technologies is not appropriate for low power use cases. These higher bandwidth technologies leverage many different features such as carrier aggregation and various frequency deployments to deliver high bandwidth. Applications should buffer and segment any data that is streaming to the device such as audio and video. The application should also keep the radio off or the radio should be allowed to enter into an idle state (RRC Idle or RRC Inactive) if the use case does not dictate that the device remains available.

LTE Cat 1, Cat 1-bis, and 5G RedCap

The technologies in this category fall just above LTE-M and Narrowband IoT in terms of data upload and download throughput supported. These technologies are full-featured LTE technologies and support use cases very broadly from fleet asset tracking to voice in use cases like alarm panels. The technologies in this category should be chosen when connectivity in transit is required and devices require data throughput speeds that are more than 1 Mbps but less than roughly 5Mbps. 5G RedCap will offer significantly improved data throughput in this category.

LTE-M

LTE-M is a category of LTE that falls solidly into the set of LPWA technologies. It uses less power than LTE Cat. 1 and provides less throughput as well. LTE-M uses a feature known as coverage enhancement to provide better coverage to devices by means of repeating signaling. LTE-M should be used for devices that require long battery life or deep penetrating coverage. LTE-M also supports mobility, and use cases like asset tracking, which do not have tight deadlines for real time location streaming can take advantage of low power and low cost LTE-M radios.

Narrowband IoT

Narrowband IoT (NB-IoT) is another category of LTE that is an LPWA technology. It uses similar coverage enhancement techniques as LTE-M but improves upon overall power consumption by using a narrower bandwidth. Consequently, NB-IoT devices support lower data throughput compared to LTE-M. NB-IoT does not support connected mode mobility and is much less suited to mobile use cases. If a device must be mobile the device's application must manage the mobility and should expect session resumptions at each cell. NB-IoT is well suited to stationary use cases that require long battery life, enhanced coverage, and use optimized software to reduce overall data transmission.

2G

2G network deployments are still a source of connectivity for global IoT deployments and are used widely in some legacy device deployments. Many new LPWA radio modules still support 2G along with LTE-M and Narrowband-IoT to offer as close to global coverage as possible, offering a single SKU solution that is a more sustainable choice than building multiple SKUs. Only choose 2G as a primary technology if there is no other coverage type in the deployment area; otherwise leverage the newer technologies for better power optimization features.

3G

In most cases, carriers have sunset or have plans to sunset 3G networks, and 2G networks that exist will likely outlive their 3G counterparts. It is not recommended to design new 3G products or to expect 3G fallback for LTE devices as the phase-out of 3G will impact the longevity of the device and ultimately the overall sustainability of the solution due to replacements.

Application note for LPWA cellular networks

Applications should not be considered immediately portable between LPWA technology types and the higher bandwidth technology types. Applications leveraging LTE-M or Narrowband IoT should have advanced error processing to accommodate for slow throughput and high retransmission rates. The application should also use advanced methods for reduction of the size of packets sent. This includes reducing the size of TLS handshakes using pre-shared keys as well as using CBOR or some other application layer mechanism to reduce the data payload size. Developers should make the application aware of when the device enters into coverage enhancement modes. Coverage enhancement modes can greatly reduce the data throughput rate and greatly increase various errors within the TLS stack, IP stack, and the application due to timeouts, retransmissions, etc. In order to maintain an LPWA device and improve its longevity, developers should take note of the

elements called out in this section in order to reduce the overall time the radio is on and sending or receiving data.

Adopt power conservation practices appropriate to your wireless technology

Cellular

In addition to the broad use cases mentioned previously, there are a number of considerations to take into account to make an energy efficient choice for cellular connectivity. For devices that communicate using LTE modems, select network operators that support eDRX (Extended Discontinuous Reception) and PSM (Power Save Mode). These are two complementary features that can be used together to optimize power consumption in LTE devices.

In PSM, LTE devices enter a low-power sleep mode for extended periods of time, waking up at pre-defined intervals to reconnect to the network without a full negotiation, greatly reducing power consumption. With PSM, while the device is asleep the radio can be completely off, in contrast to eDRX, which continues a paging cycle. This means that if a page or network-initiated downlink request were to be made, the device in PSM mode would not receive anything. This can result in an increased disconnected window compared to eDRX.

eDRX allows the device to define more precisely the interval between listening periods, which can reduce power consumption while still maintaining connectivity. This is useful for devices that must remain connected to the network for longer periods of time, such as mobile devices. This increased paging window means that latency between a request to the device is increased but the device is nonetheless available.

PSM is more power efficient than eDRX. However, there is an inflection point at which it may make more sense for a device to use one versus the other. While building an application, you should test your particular hardware for overall power consumption during a typical window of sleep and active states. Use both PSM and eDRX at the predefined eDRX paging windows and then choose the methodology that best optimizes your device's power usage during its typical usage. eDRX is also a more recent technology and might not be supported by all network providers.

Wi-Fi

Wi-Fi is a widely used connectivity option for applications that require high speed local connectivity within a limited range. These include verticals such as Smart Home, Industrial IoT, Healthcare, and Retail.

Wi-Fi has several power-saving features that can help to reduce power consumption in wireless devices. Some of the most common power-saving features in Wi-Fi include:

- **Power Save Mode (PSM):** This feature allows Wi-Fi devices to enter a low-power sleep mode when not in use, reducing power consumption. During PSM, the device periodically wakes up to check for incoming data, and then returns to sleep mode. u-APSD allows devices to selectively wake up from PSM to receive only the specific data packets that are intended for them. This allows devices to remain in PSM for longer periods of time and further reduces power consumption.
- **Wake-on-Wireless-LAN (WoWLAN):** This feature allows devices to wake up from sleep mode when a wireless LAN signal is detected, rather than requiring a physical button press or other intervention.
- **Dynamic Frequency Selection (DFS):** This feature allows devices to avoid using certain Wi-Fi channels that may be subject to interference, thus reducing power consumption.
- **Transmit Beamforming (TxBF):** This feature allows devices to optimize their transmission patterns based on the location of the receiving device, which can improve signal quality and reduce power consumption.
- **802.11n Power Save:** This feature is designed specifically for 802.11n networks and allows devices to enter a low-power sleep mode during periods of low network activity.

The power consumption in idle mode can be reduced through features such as frame bursting and low power listen, while the power consumption in sleep mode can be reduced through features such as deep sleep and wake-on-WiFi.

LoRaWAN

LoRaWAN is suitable for long-range, low power applications such as smart agriculture, asset tracking, and smart cities.

To maximize power savings on a LoRaWAN gateway or device, you can consider the following configurations:

- Choose the appropriate device class. LoRaWAN defines class A, B, and C device types. These device types have different receive window configurations that significantly impact the power consumption of a LoRaWAN device.
- Where possible use FUOTA (Firmware Update Over The Air) over multicast for efficient file transfer to many devices receiving the same firmware.
- Choose hardware components, such as processors and radios that have low power consumption to minimize energy usage.

- Position the gateway antenna for maximum coverage and signal strength, to reduce the need for high transmit power.
- Set the gateway to use a lower transmit power that still provides good coverage in the target area. This reduces energy usage and can increase the lifetime of the gateway.
- Use Listen Before Talk (LBT) and Adaptive Data Rate (ADR) - these features can help minimize the number of retransmissions and collisions, which can reduce energy usage.
- Configure the gateway to use the appropriate packet size and data rate for the application. Large packet sizes and high data rates can increase energy usage.
- Configure the gateway to use deep sleep mode when it is not actively transmitting or receiving data. This can significantly reduce energy usage.

Amazon Sidewalk

Amazon Sidewalk allows endpoint devices to use the pre-existing Amazon Sidewalk network, enabling service providers and utilities to deploy monitoring networks without the operational expense, carbon footprint, and overhead of deploying and managing gateways. Amazon Sidewalk makes use of existing Amazon Echo and Ring devices to create a low bandwidth network that devices can use to communicate with the AWS Cloud. The Amazon Sidewalk network is built such that devices can roam onto any existing Sidewalk gateway and the gateway can use its network connection to route data to cloud services. This creates a low power, best-effort community network.

Amazon Sidewalk compliant devices inherently support a number of features that help optimize power consumption. Sidewalk uses power-efficient wireless communication protocols such as Bluetooth LE. It also uses adaptive transmission power, which adjusts the transmit power of connected devices based on the distance between devices, reducing power consumption and extending battery life. Devices connected to Amazon Sidewalk also enter a low-power sleep mode when not in use, which also extends battery life.

Choose a lightweight protocol for messaging

IoT devices can use a number of application layer protocols to communicate with each other as well as the cloud. The most common protocol for IoT devices is MQTT.

MQTT is designed to be a lightweight and power efficient protocol for IoT applications that require low power consumption and low bandwidth usage. It uses a publish-subscribe model, which allows devices to exchange small messages with each other using a minimal amount of network bandwidth.

HTTP, on the other hand, is a more heavyweight protocol that is used for web-based applications. While HTTP is not specifically designed for IoT applications, it is widely used for sending and receiving data over the internet. HTTP uses a request-response model, which requires devices to send larger amounts of data back and forth over the network. This can result in higher power consumption and shorter battery life for IoT devices.

Reduce the amount of data transmitted

Data transmission is one of the most power-intensive operations for IoT devices, especially when using wireless communication protocols such as Wi-Fi, Bluetooth, or cellular networks. The overarching principles to be followed for messaging are to reduce the amount of data transmitted and reduce the distance traveled by that data.

The amount of data transmitted between an IoT device and the cloud can be reduced using message compression, using binary protocols, reducing the frequency at which messages are sent, and using transport and application layer protocols that are efficient.

IoT devices often collect large amounts of sensor data. By storing the data on the device, it can be processed and analyzed in real-time, without the need to transmit it to the cloud for analysis. This can help reduce transmission power and costs, particularly in cases where cellular or satellite connectivity is used. Storing sensor data on the device can provide an additional level of redundancy and fault tolerance, which can reduce the need for human intervention or site visits in case of extended connectivity loss.

Given that most IoT devices are resource constrained, apply compression to data generated by IoT devices, including sensor data, image or video data, and log data. There are many compression techniques available—choose one that fits your device capabilities and the use case.

Compression can not only reduce network bandwidth requirements, but can also improve data storage efficiency by reducing the amount of storage space required for data. This can be particularly useful in scenarios where large amounts of data are generated, such as in video surveillance or industrial IoT applications.

Message formats should be chosen such that they reduce processing as well as storage requirements both locally and in the cloud. It is possible that a single message format or encoding scheme may not be the most optimal for your solution.

One commonly used format is Protocol Buffers (protobuf). Protobuf is a language and platform neutral, extensible mechanism for serializing structured data. It allows efficient binary encoding of device messages with low communication overhead and low CPU usage. Protobuf data structures

can change over time with backward compatibility, reducing operational overhead. AWS IoT Core supports the ingestion and translation of protobuf messages natively, making it easy for applications to use this feature.

The Concise Binary Object Representation (CBOR) is an IETF defined, self-describing data format designed for constrained devices. It allows for small code size and small message sizes. Each item is preceded by a tag that defines its type, allowing for extensibility without the need for version negotiation. CBOR is based on the JSON data model, which uses numbers, strings, arrays, maps (called objects in JSON), and values such as false, true, and null.

Between the two options, protobuf is the recommended choice for achieving speed and efficient use of resources, while CBOR is preferred for its flexibility and extensibility.

If using MQTT, it is advisable to use MQTT5 topic aliases for frequently used topics. This reduces topic lengths, resulting in reduced transmission time, lower power consumption, decreased communication costs, and improved latency.

Reduce the distance traveled by data

There are multiple techniques that can be applied to reduce the distance traveled by data between IoT devices and the cloud. These include moving processing to the edge, filtering data at the edge and using intermediate elements such as gateways to reduce the number of connections to the cloud as well as perform local caching functions.

You should also ensure that your IoT solution uses Regions that reduce the distance that network traffic must travel. If appropriate for your use case, select a Region with close physical proximity to the deployed devices to minimize the distance data needs to travel across the network. The right choice of cloud services and architecture can reduce the network resources required to support your workload.

Optimize log verbosity

Use debug levels to increase/decrease metric and log verbosity based on the context. In normal operation, send the minimum set of metrics and logs required to identify system health. If runtime issues require additional verbosity, a more detailed log level could be set either dynamically by the device software, or by sending a new configuration to the devices that require it.

Buffer or spool messages

Message buffering and spooling can contribute to power consumption optimization in IoT devices in several ways.

Instead of immediately sending each message as it is generated, buffering allows for the aggregation of multiple messages into batches, if appropriate for the use case. By sending data in larger chunks or batches, the device can stay in a low-power state for longer periods, reducing the frequency of transitioning between active and sleep modes. This also allows the device to minimize network activity—instead of establishing a connection and sending data for every individual message, the device can consolidate multiple messages and initiate network communication less frequently. This reduction in network activity results in power savings, since establishing and maintaining network connections can be energy-intensive. It also reduces the amount of processing required, which lowers the power consumption.

In scenarios where the use of spooling is appropriate, it can be used to write data to disk in batches, rather than writing each data point as it is generated. This reduces the number of disk access operations required, which can help reduce power consumption.

Along with this, IoT devices can optimally time their data transmissions. Devices can assess factors like network availability, signal strength, or time-based considerations to select when to send accumulated messages. This approach helps to minimize energy consumption by avoiding unnecessary transmissions during periods of poor network conditions or low power availability (such as low battery charge levels).

Optimize the frequency of messages for your use case

To optimize the performance of a device, it is essential to adjust the input sampling time and ensure that it is sending messages and checking for updates at an optimal rate. The optimal rate should be determined by the use case and not necessarily by the rate at which inputs or sensor values change. Alternatively, an update on change (Change of Value) approach can be used, in which case an interpolation technique should be selected, and the device must be configured with a parameter that determines what a change is. Interpolation helps fill in the gaps in the time series data to provide a more complete and continuous representation of the data.

By restricting the application related data that is transmitted to only what is required by the application, there are also benefits in terms of reduced data storage as well as the amount of processing required in the cloud, which contribute to a reduced carbon footprint.

Use gateways to offload and pre-process your data at the edge

The decision to connect a device directly to the cloud or via a gateway will depend on a variety of factors, including the specific requirements of the application and the characteristics of the sensor and network. It can be more power-efficient to use a gateway to receive and preprocess

data locally, reducing the need for higher power radios on the device, reducing long-haul communications traffic and extending battery life.

If the device generates a large amount of data, it may be more efficient to use a gateway to pre-process and filter the data before sending it to the cloud, reducing long-haul network traffic. In addition, if the long-haul network connection is unreliable, it may be more practical to use a gateway with AWS IoT Greengrass to buffer data and ensure that it is delivered reliably to the cloud.

Perform analytics at the edge

Data preparation is usually required before analytics can be performed at the edge. There are various types of preparation and transformation that can be done at the edge to improve the sustainability of the solution:

- **Data filtering:** IoT edge devices can filter out irrelevant data or noise from sensor data streams. For example, temperature sensors may monitor and report temperature at a high rate, but reflecting those changes as they come may not be useful to a specific application or use case. By filtering out the unnecessary data points, edge devices can reduce the amount of data that needs to be transmitted and processed, reducing communication related power and cloud processing costs.
- **Data aggregation:** Devices can aggregate sensor data from multiple sources and over time to reduce the amount of data sent to the cloud. This can involve combining data from multiple sensors to create a single data stream, or aggregating data from multiple devices to provide a system-level view of performance or behavior. This can reduce the number of connection requests as well as messages sent to the cloud.
- **Data enrichment:** Sensor data can be enriched with additional contextual information, such as location or time data. This can enable more accurate analysis and insights, as well as provide additional context for cloud applications or systems. This reduces the need to enrich data in the cloud. Local processing and data enrichment should be considered when the cost of doing the same operation in the cloud out-weighs the impact of larger payloads and additional local computation and resources.
- **Data normalization:** Devices can normalize sensor data from different devices or sources to ensure consistency and compatibility with cloud applications or systems. This can involve converting data formats, units of measurement, or other data attributes to a common standard, reducing the need to do this in the cloud.

Devices can perform real-time analytics, predictive maintenance, anomaly detection, optimization, and security monitoring by analyzing sensor data and triggering alerts or actions based on predefined rules or machine learning models, without sending the source data to the cloud.

To perform analytics in an efficient and sustainable manner on the edge, use lightweight algorithms to improve performance and resource efficiency on IoT devices. Examples include algorithms such as decision trees or regression models that are less computationally intensive than deep learning models. In addition, optimizing data structures can improve the performance and efficiency of analytics algorithms on IoT devices. Examples include data structures such as binary trees or hash tables that require less memory and processing power.

Monitor and manage your fleet operations to maximize sustainability

It is recommended that your solution be monitored and managed efficiently across its lifecycle to reduce the carbon footprint of your operations. Common operational tasks such as inspecting the connectivity state of a device, ensuring device credentials are configured correctly, and querying devices based on their current state must be in place before launch so that your system has the required visibility to troubleshoot applications and reduce the need for site visits. From a sustainability point of view, inaccurate information about device status can result in delayed or erroneous actions such as unnecessary truck rolls or mis-directed corrective actions. This can lead to decreased efficiency and increased costs.

Having an accurate view of the state of all devices is therefore an imperative for operating at scale. This is best done through the use of tools, services, and automation. Here are some of the key ways to monitor IoT operations using AWS IoT services:

- Use AWS IoT Device Management services to manage the entire lifecycle of your IoT devices, including firmware updates in the field. This helps in the remote management of devices, reducing the need for site visits and lowering the carbon footprint of your operations. For additional observability into your device's resource allocation, you can install the AWS X-Ray daemon to see how edge applications perform and where CPU utilization or other optimizations can be made to decrease power consumption or better allocate device resources. Using Fleet Hub for AWS IoT Device Management, you can build standalone web applications for monitoring the health of your device fleets, as well as performing common fleet-wide tasks such as investigating and remediating operational and security issues.
- Monitor resource utilization periodically and set up an alerting system to notify appropriate stakeholders if the utilization goes above a specific threshold. For simple threshold conditions, detection can be done on the device itself and alerts sent to the cloud. During the development

phase, cloud-based monitoring can be used to determine the appropriate threshold values for your workload. For more sophisticated anomaly detection during operation, the ML detect feature of AWS IoT Device Defender can be used to learn behaviors and generate alerts based on anomalies unique to your fleet. It is recommended to configure automatic actions such as optimizing resource allocation or disabling processes to avoid system downtime caused by anomalous conditions. Regularly reviewing monitoring data and threshold adjustments will ensure accurate and timely notifications that reflect the current performance of the IoT device.

- AWS IoT Core provides a set of metrics that you can use to monitor the performance and health of your IoT devices and applications. These metrics include device connections, message delivery, and rule engine metrics. You can view these metrics using Amazon CloudWatch or the AWS IoT Core console. In addition, you can monitor device metrics such as CPU and memory utilization, battery levels and the like using the Fleet Metrics feature of AWS IoT Core. These metrics can be used to proactively identify devices that are degrading or will soon need attention, averting service impact and the need for emergency truck rolls.

Best practices

The following domain areas identify opportunities where specific best practices can be employed to improve the sustainability impact of your IoT solution:

Best practice areas

- Region selection
- Alignment to demand
- Software and architecture – Software optimization
- Software and architecture – Cloud
- Data management
- Hardware and services - Hardware optimization
- Hardware and services – Power management
- Process and culture – User guidance

These best practices should be used in conjunction with the design principles detailed previously in this document.

Region selection

There are currently no Well-Architected best practices associated with Region selection in the IoT Lens. Refer to Reduce the distance traveled by data as well as the general Sustainability Pillar guidance in the AWS Well-Architected Framework.

Alignment to demand

There are currently no Well-Architected best practices associated with alignment to demand in the IoT Lens. Refer to the general Sustainability Pillar guidance in the AWS Well-Architected Framework.

Software and architecture – Software optimization

IOTSUS 1. How do you optimize software to reduce the device's carbon footprint?

Best Practice IOTSUS_1.1 – Remove unnecessary modules, libraries, and processes

Ensure that the operating system only runs processes that are necessary for the functionality of the IoT device. Unnecessary libraries, modules, and processes should be avoided as they can contribute to a larger device footprint, increase patching requirements, and create a larger attack surface and more processes for the CPU to run. Streamlining the operating system and only including essential processes can lead to a more efficient and secure IoT device design. The Yocto project is an open source project that allows developers to build custom Linux images that include only the modules needed to support functionality of the device. Device software such as AWS IoT Greengrass or the AWS IoT Device Client can be built into Yocto Projects using the meta-aws Yocto layer.

Best Practice IOTSUS_1.2 – Use AWS IoT features to optimize network usage and power consumption

AWS IoT Core offers some features that can be used to reduce network usage and power consumption.

AWS IoT Device Shadows are virtual representations of IoT devices in the cloud. They enable decoupled bidirectional communication between the device and applications running in the cloud. Applications can obtain device state from the shadow rather than the device, reducing traffic between the device and cloud, and allowing the application to continue operation even if the

device is disconnected intermittently. When a device comes back online, it can check if there were any changes requested by the application while it was offline, and take action as needed. Thus, the device doesn't have to stay online constantly, which can save power.

Also consider the use of MQTT retained messages, message expiry and session expiry features. Retained messages and device shadows both retain data from a device, but have different capabilities and suitability. MQTT5 message expiry can be used to ensure that devices only receive time relevant messages, thus reducing processing load. The session expiry feature can be used by MQTT clients to set application specific session expiry limits, ensuring that the broker does not need to retain resources beyond what is needed.

Best Practice IOTSUS_1.3 – Use a hardware watchdog to restart your device automatically

IoT devices should have a hardware watchdog mechanism, which can reduce downtime by automatically restarting the device when it becomes unresponsive. In many cases, restarting the device can put it in a state where it can be remotely managed, minimizing the impact of failures and reducing the need for site visits.

Best Practice IOTSUS_1.4 – Use communication clients that reduce connection traffic

Clients communicating with the cloud must not only be functionally correct, but also implement resilient and scalable system behavior. It is recommended that clients support exponential backoff with jitter when handling connection retries to cloud endpoints. This minimizes the number of connection attempts when dealing with a congested network, thus reducing the work done by each client as well as reducing the network traffic - both help reduce power consumption. It also helps smooth out the spikes in the number of connection requests from clients in the case of widespread events such as connectivity loss or power loss, reducing the number of unsuccessful connection attempts.

Clients should also define a threshold at which point it is more effective to enter low power modes during a backoff period. This will allow the device to conserve power whilst waiting for an appropriate communication opportunity.

Clients that support MQTTv5 should support reason codes and use that information to determine if and when they should reconnect. For example, if the server disconnect is due to a wrongly configured policy, reconnecting can be deferred until the policy has been updated. Since the client does not have a means of knowing when the policy has been updated, use a retry interval that has a reasonable upper bound, to avoid the need for manual intervention when the policy has been fixed.

Software and architecture – Cloud

IOTSUS 2. How do I use the cloud to minimize my carbon footprint?

Studies by 451 Research have shown that AWS' infrastructure is 3.6 times more energy efficient than the median of U.S. enterprise data centers surveyed and up to five times more energy efficient than the average in Europe. 451 Research also found that AWS can lower customers' workload carbon footprints by nearly 80% compared to surveyed enterprise data centers, and up to 96% once AWS is powered with 100% renewable energy by 2025. While this makes it much easier to lower your carbon footprint for workloads in the cloud, it does not reduce the need for optimization of cloud workloads.

It is recommended that customers use the AWS customer carbon footprint tool to view estimates of the carbon emissions associated with their usage of AWS products and services. This tool provides data visualizations to help customers understand their historical carbon emissions, evaluate emission trends as their use of AWS evolves, approximate the estimated carbon emissions they have avoided by using AWS instead of an on-premises data center, and review forecasted emissions. The forecasted emissions are based on current usage, and show how a customer's carbon footprint will change through its actions as well as those brought about as Amazon moves to use 100% renewable energy by 2025.

More detailed guidance on sustainability practices in the cloud can be found in documentation relating to those services as well as in the Sustainability Pillar of the AWS Well-Architected Framework.

Best Practice IOTSUS_2.1 – Use the Basic Ingest feature in AWS IoT Core

The ingestion layer is where all data coming from disparate devices and data sources is aggregated and made available for cloud application consumption. To ensure that you are using the right ingestion approach to achieve both performance efficiency and sustainability, there are a number of considerations to keep in mind.

When ingesting data to AWS IoT Core, consider whether to use the Basic Ingest feature or not. With the Basic Ingest feature, you can securely send device data to the AWS services supported by AWS IoT rule actions, without incurring messaging costs. Basic Ingest optimizes data flow by removing the publish/subscribe message broker from the ingestion path, making it more cost effective as

well as more resource efficient. You can use this approach if your application does not require multiple subscribers for the data being ingested.

For all other ingestion mechanisms (such as the Amazon Kinesis family of services), refer to the AWS IoT Lens for guidance on which service is appropriate for which use case. At this time, there are no additional sustainability best practices for these services.

Best Practice IOTSUS_2.2 – Choose an appropriate QoS level

Use MQTT when you have IoT devices or other resource-constrained environments that need to communicate efficiently and reliably with a publish-subscribe messaging pattern. MQTT supports different quality of service (QoS) levels for message delivery. Higher QoS levels involve additional network overhead for acknowledgment and retransmission, which can increase power consumption. Consider using lower QoS levels (such as QoS 0) if the reliability of message delivery is not critical for your use case.

Data management

There are currently no Well-Architected best practices associated with data management in the IoT Lens. Refer to Reduce the amount of data transmitted as well as the general Sustainability Pillar guidance in the AWS Well-Architected Framework.

Hardware and services - Hardware optimization

IOTSUS 3. How do you select the right hardware components?

Best Practice IOTSUS_3.1 – Source sustainable components

The design and manufacturing layer consist of product conceptualization, business and technical requirements gathering, prototyping, product layout and design, component sourcing, manufacturing and distribution.

Several factors can impact sustainability in various stages of the design process. These include choices related to materials, packaging, and product design, which can significantly influence the carbon footprint of the final product.

Sustainable supply-chain practices may involve sourcing from suppliers that demonstrate environmentally responsible practices, use of recycled or renewable materials, or products with

lower environmental impact throughout their lifecycle. An example of this would be to use products that are Climate Pledge Friendly certified.

Best Practice IOTSUS_3.2 – Consider the manufacturing and distribution footprint of your device

The manufacturing and distribution phase has significant sustainability implications, including the size of the factory, energy usage, and transportation distance for shipping products. Choosing manufacturing facilities with low environmental impacts, optimizing transportation routes to minimize emissions, and using energy-efficient manufacturing processes can all contribute to improved sustainability in the supply chain. One example of energy efficient manufacturing is the use of low temperature solder during the pick-and-place operation to reduce the amount of energy expended to heat the solder during component placement on the Printed Circuit Board (PCB). Another consideration is designing IoT devices and packaging in smaller form factors that allow for easier shipping in large volumes and consuming less raw materials and harmful chemicals.

By considering these factors, organizations can make more sustainable supply-chain decisions and contribute to positive environmental outcomes through thoughtful product design.

Best Practice IOTSUS_3.3 – Use benchmarks to help you make a processor choice

Processor and IoT benchmarks can help you assess and narrow down which processor is appropriate for your use case. Here are some key characteristics of benchmarks that you should consider:

- IoT devices have unique requirements, such as low-power operation, real-time processing and connectivity, and therefore benchmarks should include workloads that closely mimic the actual workloads that IoT devices are expected to handle, such as sensor data processing, edge filtering, and running communication protocols.
- Benchmarks should take into account the constraints associated with IoT devices and assess the characteristics of CPUs in such resource-constrained scenarios. Look for benchmarks that provide relevant performance metrics considering these resource limitations.
- Benchmarks should include energy efficiency metrics to assess how efficiently CPUs can process workloads while minimizing energy consumption. The use of energy-efficient CPUs can extend battery life, reduce energy costs, and minimize environmental impact. Examples of metrics to consider include Performance per Watt and Thermal Design Power (TDP).
- Benchmarks should include test cases that evaluate the real-time processing capabilities of CPUs, including latency and responsiveness.

- IoT devices require communication and connectivity capabilities to interact with other devices, cloud services, or data centers. Benchmarks should include test cases that evaluate the communication and connectivity performance and efficiency of CPUs.

The Embedded Microprocessor Benchmark Consortium (EEMBC) is an organization that develops benchmarks specifically for embedded systems, including IoT devices. EEMBC benchmarks, such as the EEMBC IoTMark and EEMBC ULPBench, are used in the industry for evaluating the performance of CPUs in IoT applications. The benchmarks reflect real-world IoT workloads, and include IoT-specific characteristics such as low-power operation, real-time processing, and connectivity. They also provide performance metrics, including energy efficiency, communication efficiency, and system impact, which align with sustainability evaluation criteria.

Best Practice IOTSUS_3.4 – Optimize your device based on real-world testing

It is recommended that final selection of your device hardware be based on evaluating one or more hardware choices under close-to-actual operating conditions. Processors, peripherals, and other components must be chosen to optimize power draw during execution as well as during device idle states. Other criteria as discussed throughout this document can be used to make a final selection based on the results of your testing.

Once the hardware has been finalized, examine whether the observed performance matches the expected performance. Profiling of your code on the target hardware under real workloads can help identify power-hungry sections of the code and help you optimize them for efficiency. Examining application and OS use of the device's power saving features and modes may also be required to achieve optimal efficiency.

Best Practice IOTSUS_3.5 – Use sensors with built-in event detection capabilities

Sensor components are the foundation of IoT, bridging the physical and digital worlds and providing real-time data on environmental conditions.

Sensor components should be designed to operate with minimal power consumption by optimizing data transmission. Some sensor components have built-in data processing capabilities to generate events that are directly usable by the host device's application. Sample rates should be configured to balance between capturing enough data for accuracy and conserving power to reduce battery drain, while meeting the needs of the use case. This can involve techniques such as adjusting the sampling rate based on sensor data variability, prioritizing critical data over less important data, or using event-triggered or adaptive sampling approaches to reduce unnecessary data collection.

One example of this concept is the use of inertial measurement unit (IMU) sensors for fall detection. IMU sensors consist of accelerometers, gyroscopes, and magnetometers that can measure acceleration, orientation, and other motion-related data. These sensors can often detect fall events by processing the raw sensor data locally on the sensor itself, using embedded algorithms or machine learning models, and generating an interrupt to the host processor with an alert when a fall is detected. This allows the host processor to perform general purpose actions or operate in a low-power mode until the interrupt is detected and the host processor wakes up to process the event.

Best Practice IOTSUS_3.6 – Use hardware acceleration for video encoding/decoding

Power efficient IoT devices that require processing and streaming video to the cloud, such as doorbell or security cameras, should use video encoding to reduce data transmission and file size. Like machine learning, video encoding requires a large number of clock cycles for operations such as bit-shifting and matrix processing.

H.265 (also known as HEVC or High Efficiency Video Coding) and H.264 (also known as AVC or Advanced Video Coding) are the most popular video encoding standards for edge devices. H.265 is designed to provide better video quality at lower bitrates compared to H.264, which means that it can achieve the same video quality with less data, resulting in reduced bandwidth requirements, lower power consumption and lower communication costs during video playback or transmission.

Choosing a microcontroller or microprocessor with dedicated video encoding hardware is strongly recommended to improve performance and reduce power consumption in video processing tasks. Some system designs offer a dedicated video encoding coprocessor that runs a single video encoding algorithm, saving the host processor for other general-purpose tasks.

With advancements in technology, it is likely that more efficient video encoding algorithms will be developed. Choosing a hardware accelerator with an FPGA or other updatable logic can extend your device's life if a more efficient encoding algorithm is required in the future.

Best Practice IOTSUS_3.7 – Use HSMs to accelerate cryptographic operations and save power

Use of secure hardware, such as Trusted Platform Modules (TPMs), Hardware Security Modules (HSMs), Secure Elements, and Secure Enclaves (SEs) like Arm TrustZone (collectively referred to as HSMs in this document) is highly recommended. Including hardware-based crypto acceleration in the device can significantly speed up cryptographic operations, reduce energy consumption, and enhance security, making it a beneficial addition to the device's design. An HSM typically performs ECDSA (Elliptic Curve Digital Signature Algorithm) signature operations several times faster than software on a general-purpose microcontroller, allowing the host microcontroller to spend more

time in a low-power mode while the HSM performs complex cryptographic operations. If the device supports cellular connectivity, using the SIM card in place of a dedicated secure element can reduce the overall Bill of Materials (BOM).

Best Practice IOTSUS_3.8 – Use low-power location tracking

In wireless devices such as asset trackers, location can be the most important measurement for the device as well as the most computationally expensive to obtain. For GPS based devices, it is recommended to use chipsets that support assisted-GPS (A-GPS), which reduces power consumption. Another option is to use location services like AWS IoT Core Device Location, which uses cloud based location solvers, such as Wi-Fi scan, Cellular scan, Global Navigation Satellite System (GNSS) scan, or reverse IP look-up to determine the geo-coordinates of IoT devices for use cases such as map visualization, historical route tracking, and Geo-fencing. This in many cases reduces the device power consumption required to resolve location.

HSMs can reduce the time and resources needed for maintenance, improving the sustainability of the system. Use device certificates with long expiration dates that are designed to only be rotated as needed to maintain the security posture of the device. Rotating device certificates can be computationally expensive and requires additional communication with the cloud.

Hardware and services – Power management

IOTSUS 4. How do you minimize power usage and waste?

Best Practice IOTSUS_4.1 – Use energy harvesting technologies to power your device

One approach to improve sustainability is to use energy harvesting technologies to provide some or all of the power needs of a device, reducing reliance on grid-based power sources. IoT devices can use renewable energy sources such as solar energy, thermal energy, vibration and mechanical energy, radio frequency energy, wind energy, and piezoelectric energy. The energy thus captured is usually stored in batteries or supercapacitors to ensure continuous availability of power.

This approach offers several advantages. It can lead to significant cost savings by reducing the need for wired power connections, reducing installation time and infrastructure expenses, lowering device operating costs and maintenance expenses. It also means that devices can be deployed in remote locations, expanding the potential use cases and allowing for applications such as remote environmental monitoring or forest fire detection.

Best Practice IOTSUS_4.2 – Manage Idle state power

Tickless operation is a power management technique used in operating systems to minimize power consumption by reducing the frequency of system interrupts, or *ticks*, while the system is idle. In embedded operating systems like FreeRTOS, it is common to use the idle hook function to place the microcontroller CPU in a low power mode. This responsibility is left to the embedded application developer, and must not be overlooked.

For power critical applications, consider factors such as the latency and power requirements of entering and exiting low power modes, and choose the low power mode that provides the best trade-off between power savings and responsiveness. In addition, configuring the right wake-up sources or events can further help minimize power consumption.

Best Practice IOTSUS_4.3 – Manage power based on application context

Allow applications or software running on the edge device to make decisions about changing the hardware state, such as CPU frequency, voltage, or other hardware settings, based on the specific requirements of the application and the available resources. For example, an application running on an asset tracker in motion may need to perform tasks with high computational requirements, such as collecting sensor data, determining location, processing data, or transmitting data to a remote server. When the tracker is at rest, the application can determine that these tasks can be performed less often, and can use power management techniques to reduce the power draw.

Benefits of this approach include the ability to optimize power consumption based on the specific needs of the application, enabling longer battery life, reduced energy consumption, and improved performance. However, it also requires careful consideration of the trade-offs between power savings and performance.

In addition to these approaches, techniques such as dynamic power management, where the device adjusts its power consumption in real time based on the available energy, can be employed. Many microcontrollers and processors used in IoT devices come with low-power libraries and Application Programming Interfaces (APIs) that provide optimized functions for power management. Leveraging these libraries and APIs can help in the realization of dynamic power management.

Process and culture – User guidance

IOTSUS 5. How can you help users lower the carbon footprint of their devices?

Best Practice IOTSUS_5.1 – Make installation easy

It is important to educate users on the proper installation and use of devices to avoid errors or misuse that could necessitate a site visit from a technician, leading to additional cost and environmental impact. Providing user-friendly documentation or mobile applications that give a user detailed step-by-step guidance is highly recommended.

Best Practice IOTSUS_5.2 – Make proper disposal easy

Disposing of IoT devices in an environmentally responsible manner includes recycling electronic components, properly disposing of batteries, and adhering to local regulations and guidelines for electronic waste disposal. It is important for IoT device manufacturers, users, and stakeholders to work collaboratively to develop and implement sustainable and responsible practices for the disposal of IoT devices to minimize their environmental impact.

Promoting repairability and supporting repair and transfer of ownership options for IoT devices is also recommended. It should be easy for users to update, upgrade, and/or repair devices back to a working state.

Provide users with detailed instructions on how to perform a factory reset, wipe data, and disassociate devices from the current user account. Also consider how your device will be disposed of at the end of its life. Provide users with clear instructions on how to properly dispose of the device, recycle components, and isolate any harmful materials. Also consider creating incentives for users to do this.

Best Practice IOTSUS_5.3 – Identify when devices in the field can/should be retired.

As circumstances change in your deployed solution (sites shut down, for instance) devices may remain active even though not needed. To minimize the impact of such cases, unused assets should be decommissioned. Refer to the guidance in the AWS Sustainability Pillar.

Key AWS services

Ingest IoT data at scale: Use AWS IoT Core is a managed service used to manage device connectivity, device security to the cloud, message ingestion, message routing, and device state. Use Amazon Kinesis Video Streams, a fully managed AWS service with device side producer SDKs, to stream live video from devices to the AWS Cloud, or build applications for real-time video processing or batch-oriented video analytics. Amazon Kinesis Data Streams can be used to collect and process large streams of data records in real time.

Remote device operations: Use AWS IoT Jobs to instruct devices to download and install applications, run firmware updates, reboot, rotate certificates, or perform remote troubleshooting operations.

Seamless remote access: AWS IoT Secure Tunneling allows you to establish bidirectional communication to remote devices over a secure connection that is managed by AWS IoT, without requiring changes to your existing inbound firewall rules.

Monitor and manage devices: AWS Systems Manager can act as the operations hub for your AWS applications and resources and a secure end-to-end management solution for your devices as well as hybrid and multi-cloud environments, enabling secure operations at scale.

Monitor and mitigate device risks: AWS IoT Device Defender is a security service that allows you to audit the configuration of your devices, monitor connected devices to detect abnormal behavior, and mitigate security risks. It gives you the ability to enforce consistent security policies across your AWS IoT device fleet and respond quickly when devices are compromised.

Develop edge applications: AWS IoT Greengrass is an open source IoT edge runtime and cloud service that helps you build, deploy, and manage IoT applications on your edge devices.

Resources

Refer to the following resources to learn more about our best practices related to sustainability.

Documentation and blogs

- Optimizing your AWS Infrastructure for Sustainability
- Amazon Sustainability - The Cloud
- AWS enables sustainability solutions
- Amazon Sustainability - Driving Climate Solutions
- Greenhouse Gas Protocol
- Deploying and managing an IoT workload on AWS
- How to build smart applications using Protocol Buffers with AWS IoT Core
- Enabling device maintenance across multiple time zones using AWS IoT Jobs
- Connect to remote devices using AWS IoT Secure Tunneling
- How to remote access devices from a web browser using secure tunneling
- Build resilient IoT device applications that remain active using the AWS IoT Device SDKs

- Enhancing IoT device security using Hardware Security Modules and AWS IoT Device SDK
- Enable compliance and mitigate IoT risks with automated incident response
- Introducing new MQTTv5 features for AWS IoT Core to help build flexible architecture patterns
- Schedule remote operations using AWS IoT Device Management Jobs
- Design considerations for cost-effective video surveillance platforms with AWS IoT for Smart Homes
- Build machine learning at the edge applications using Amazon SageMaker Edge Manager and AWS IoT Greengrass V2
- Best practices for ingesting data from devices using AWS IoT Core and/or Amazon Kinesis
- Automate application deployment to IoT devices using AWS IoT Device Management
- Optimize image classification on AWS IoT Greengrass using ONNX Runtime

Whitepapers

- Reaching Net-Zero Carbon by 2040: Decarbonizing and Neutralizing the Use Phase of Connected Devices

Conclusion

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

Contributors

The following individuals and organizations contributed to this document:

- Olawale Oladehin, Solutions Architect Specialist, IoT, Amazon Web Services
- Dan Griffin, Software Development Engineer, IoT, Amazon Web Services
- Catalin Vieru, Solutions Architect Specialist, IoT, Amazon Web Services
- Brett Francis, Product Solutions Architect, IoT, Amazon Web Services
- Craig Williams, Partner Solutions Architect, IoT, Amazon Web Services
- Philip Fitzsimons, Sr. Manager Well-Architected, Amazon Web Services
- Harish Rajagopalan, Senior Solutions Architect, IoT, Amazon Web Services
- Ryan Dsouza, Principal Solutions Architect, IoT, Amazon Web Services
- Ashok Bhaskar, Senior Partner Solutions Architect, IoT, Amazon Web Services
- David Walters, Principal Partner Solutions Architect, IoT, Amazon Web Services
- Jordan Alexander, Partner Solutions Architect, IoT, Amazon Web Services
- Massimiliano Angelino, EMEA Prototyping Lead Architect, Amazon Web Services
- Gaurav Gupta, Principal Partner Solutions Architect, IoT/Connectivity, Amazon Web Services

Document revisions

To be notified about updates to this whitepaper, subscribe to the RSS feed.

Change	Description	Date
Major update	Sustainability pillar added.	November 16, 2023
Major update	Updated to include Lens Checklist, industrial IoT (IIoT), and new AWS IoT services and features.	March 31, 2023
Minor update	Updated link.	March 10, 2021
Whitepaper updated	Updated to include additiona l guidance on IoT SDK usage, bootstrapping, device lifecycle management, and IoT	December 23, 2019
Initial publication	IoT Lens first published.	November 1, 2018

AWS Glossary

For the latest AWS terminology, see the AWS glossary in the *AWS Glossary Reference*.

Notices

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