Microsoft

Microsoft Azure IoT
Reference Architecture

Table of Contents

Introduction

Connected sensors, devices, and intelligent operations can transform businesses and enable new growth opportunities with a comprehensive set of Microsoft Azure Internet of Things (IoT) services.

This document outlines the core reference architecture for IoT solutions built on the Microsoft Azure platform that enables your organization to connect, store, analyze, and operationalize device data to provide deep insights from your line of business assets. This architecture describes terminology, technology principles, common configuration environments, and composition of Azure IoT services.

The goal of this document is to provide perspective and guidance to Microsoft customers and partners who are delivering IoT solutions using Azure services.

This document is provided "as-is." Information and views expressed in this document, including URL and other Internet Web site references, may change without notice. You bear the risk of using it. Some examples are for illustration only and are fictitious. No real association is intended or inferred. This document does not provide you with any legal rights to any intellectual property in any Microsoft product. You may copy and use this document for your internal, reference purposes.

1. Reference architecture overview

This reference architecture provides guidance for building secure and scalable, device-centric solutions for connecting devices, conducting analysis, and integrating with backend systems. While these solutions may be built on public, private, and hybrid Azure cloud components, the core guidance is focused on public cloud implementations. The concepts of this architecture are also generally applicable to private cloud implementations, at lower scale and with increased deployment and management efforts.

The goal of this architecture is to enable the flow of information between intermittent or continuously connected devices and line-of-business assets (that is, not general-purpose devices such as personal PCs, smartphones, or tablets) and cloud-based backend systems for the purpose of analysis, control, and business process integration. The solution architecture is specifically designed for large-scale IoT environments with devices from industrial serial production with tens of thousands of units, and/or industrial machinery emitting significant amounts of data. The model is suitable for so-called "maker" and hobbyist scenarios where small numbers of special-built devices are operated, but may be costlier than a solution equivalent to running a single website.

The architecture aims to be neutral with regard to particular industries and use-case scenarios and also neutral toward particular approaches for modeling state, metadata, and behavior of devices. However, while the architecture is abstract, the assumption is that realizations of the architecture in particular solutions will be very concrete and aligned with particular industry standards or domain specific designs.

The reference architecture provides flexibility for composability and extensibility to allow for a variety of technology choices driven by the specific solution requirements. The document first introduces foundational principles for the architecture, then presents the conceptual model and components of the architecture, which can be implemented using Azure or third-party services. The remaining sections provide more details about the individual components, design considerations, and technology trade-offs.

Figure 1 shows the high-level conceptual architecture. The architecture is composed of core platform services and application-level components to facilitate the processing needs across three major areas of a typical IoT solution:

- Device connectivity
- Data processing, analytics, and management
- Presentation and business connectivity

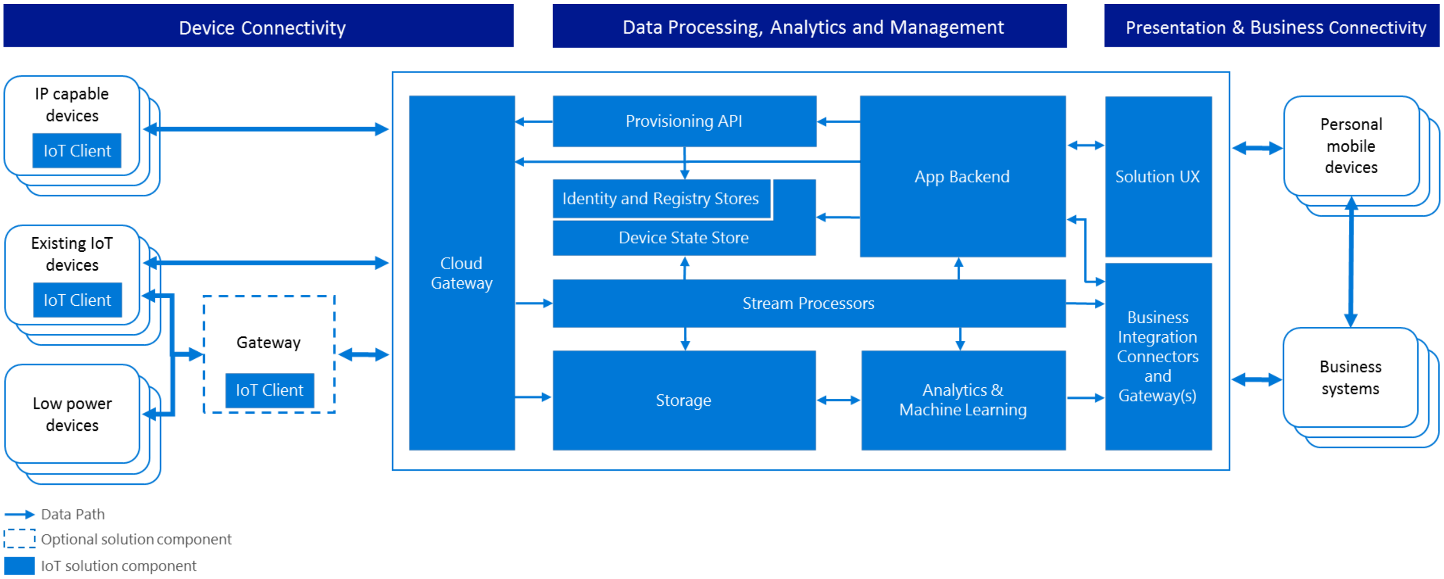


Figure 1 IoT solution architecture

Devices can be connected directly or indirectly via a gateway, and both may implement edge intelligence with different levels of processing capabilities. A cloud gateway provides endpoints for device connectivity and facilitates bidirectional communication with the backend system.

The back end comprises multiple components to provide device registration and discovery, data collection, transformation, and analytics, as well as business logic and visualizations.

The business integration and presentation layer is responsible for the integration of the IoT environment into the business processes of an enterprise. The IoT solution ties into existing line-of-business applications and standard software solutions through adapters or Enterprise Application Integration (EAI) and business-to-business (B2B) gateway capabilities. End users in business-to-business or business-to-consumer scenarios will interact with the IoT solution and the special-purpose IoT devices through this layer. They may use the IoT solution or line-of-business system UIs, including apps on personal mobile devices, such as smartphones and tablets.

Microsoft provides core platform capabilities, Azure IoT services, and IoT solution components as well as preconfigured solutions that offer a default composition of those elements for common IoT scenarios.

2. Foundational principles and concepts

2.1. Architecture guiding principles

The reference architecture provides a degree of component commonality that allows assembling secure, complex solutions supporting extreme scale, and yet allowing for maximum flexibility with regard to possible solution scenarios. This motivates the following guiding principles across the different areas of the architecture.

Heterogeneity. The proposed reference model must accommodate for a vast variety of scenarios, environments, devices, processing patterns, and standards. It should be able to handle vast hardware and software heterogeneity.

Security. Because IoT solutions represent a powerful connection between the digital and physical worlds, building secure systems is a necessary foundation for building safe systems. This reference model contemplates security and privacy measures across all areas, including device and user identity, authentication and authorization, data protection for data at rest and data in motion, as well as strategies for data attestation.

Hyper-scale deployments. The proposed architecture should support millions of connected devices. It should allow proof-of-concepts and pilot projects that start with a small number of devices to be scaled-out to hyper-scale dimensions.

Flexibility. The heterogeneous needs of the IoT market necessitate open-ended composition of various services and components. The reference model is built upon a principle of composability to allow for a number of extension points and to enable the usage of various first-party or third-party technologies for the individual conceptual components. A high-scale, event-driven architecture with brokered communication is the backbone for a loosely coupled composition of services and processing modules.

2.2. Data concepts

Understanding of data concepts is a critical first step to building device-centric data collection, analysis, and control systems. The role of devices, data models, data streams, and encoding are detailed in the following sections.

2.2.1. Device and data models

Models often describe the schema for the descriptive metadata about the device, like its model and serial number, data schemas for data emitted by the device, and schemas for configuration parameters controlling device behaviors, as well operations and parameters for the control actions a device can execute, or what events it can observe.

There are many different device modeling efforts underway across different industries, and this reference architecture takes a largely neutral stance in order to support many of these ongoing modeling and schematization efforts.

For example, in the case of an industrial automation scenario, the data semantics and structure may be based on the OPC Foundation's information modeling framework.[1] Take note that other implementations such as home automation and automotive applications may use entirely different industry-specific modeling and schema standards.

The architecture adopts a fundamental abstraction of data streams, where device and data models are not required to flow, route, or store information in the core platform components. At the solution layer, structured data will be guarded by data models and schema whenever it is produced or consumed by the components. Developers have the option of using schemas for device-client development, backend analytics, or specific processing logic as required by the solution.

2.2.2. Data streams

The IoT reference architecture adopts a foundational notion of data streams that are composed of data records and represent the data flow through the system. Data streams do not have any prescribed format for the content of records because any structure will depend on the kind of data that is transported. The reference model is strictly *neutral* regarding the structure of the payload and data semantics. It does not prescribe or take any dependency on naming, meaning, or types of any data item or structure that is not immediately required for a fundamental platform function (for example, uniquely identifying a data record or data stream).

[1] https://opcfoundation.org/

Figure 2 shows a logical data flow from a device through the cloud gateway ingestion point and an event stream processor to a storage location.

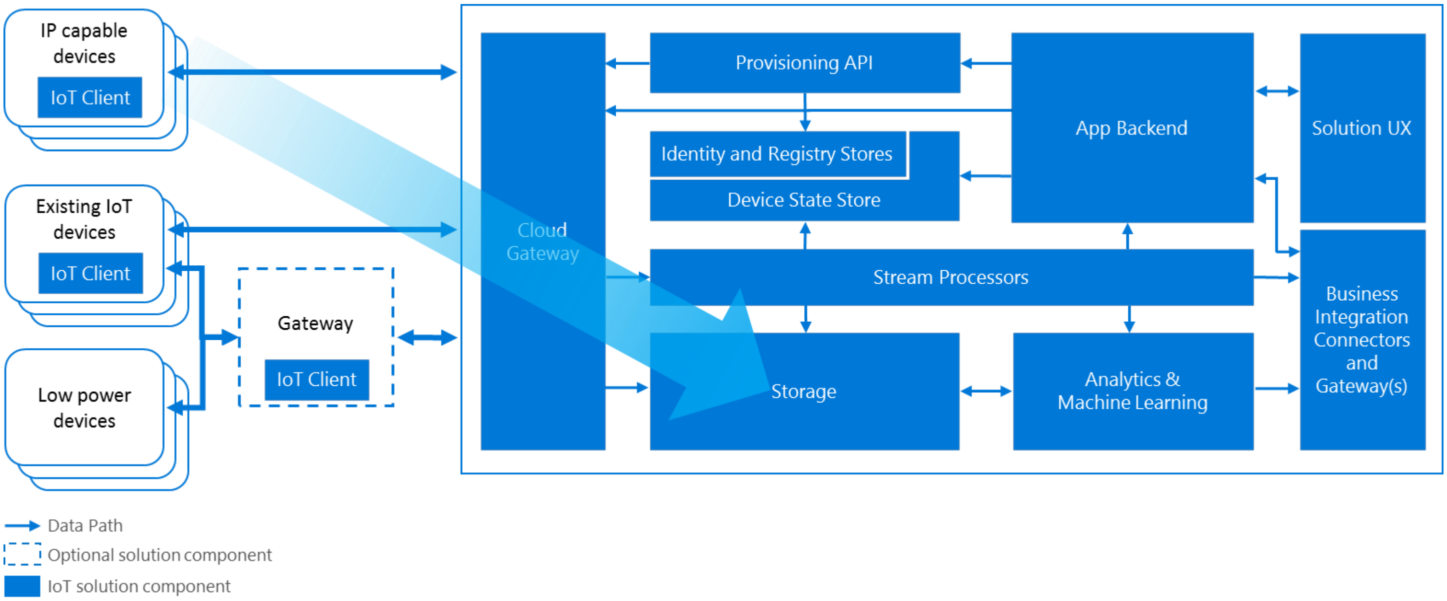


Figure 2 Data flow in the IoT reference architecture

Device telemetry. Device sensor data consists of digital signals—sequences of point-in-time observations made over a defined period that is deemed appropriate for the signal. These signals may require some level of preprocessing on the device, in which case the processing result is relevant to the system, not the individual measurements points. It is also worth mentioning that preprocessed digital signals may be encoded and transferred in many different types of encoding formats, such as MPEG-2[2] Layer 3 audio encoding or H.264 encoding for video. Sound and light signals are examples of sensor data that require onsite analysis in many scenarios.

State, alerts, and actions. The reference architecture does not prescribe any specific record types for state, alert, or action within a data stream. The "state" of a parameter becomes the "last known value", represented by the last record carrying that specific parameter. Devices may send records with all defined parameters for a particular record, or in many cases only values that have changed since the last message was sent (especially when devices are optimized for network bandwidth). In the latter case, devices might send from time to time a full snapshot of all parameters (also called a key frame) in between differential records for synchronization purposes.

An "alert" is a processing rule triggered by an event record matching certain conditions. An "action" is an operation initiated in response to the receipt of an event record with particular criteria. Actions may be defined in various ways depending on the scenario. To summarize, data streams composed of records are delivered to Azure from devices. These records then either report the state of a device data point, or act as alerts that trigger an action to be taken.

Timestamps. Any structure will depend on the kind of data that is transported, so there is no prescribed format for the content of records. Because there is no common convention on time or timestamps, a specific solution can pick a particular model for time expressions. Examples of this are UTC wall-clock time, a vector clock model, epochs in form of

[2] http://en.wikipedia.org/wiki/Moving_Picture_Experts_Group

offsets from some solution-chosen reference instant, or some other notion of order that is not immediately related to wall-clock.

Data flow. As data is ingested to Azure, it is important to understand how the route of data processing may vary. Intermediate stream processing will consume one shape of data stream, but may transform it and produce a different type of output data stream. Also, data streams may terminate and emerge in different parts of the system based on the type of data processing that occurs. This intermediate processing may be performed "on the fly" as part of stream processors or data pipelines transferring at-rest data.

If multiple data streams need to flow concurrently, for example representing telemetry for different subcomponents of a device, the recommendation is to segregate those by using discriminators in the underlying application protocol for the payload frame. An example would be when using the "subject" field in AMQP[3] or alternatively an application property of the message (for example, "stream-id"). Streams can also be segregated using a convention around a particular payload field when using a single structured data encoding. In alignment with the reference architecture, it is possible for a candidate convention to be adding an optional field, such as "_subject", to each payload record. However, note that this will require "opening" and deserializing the payload prior to routing.

The assumption for each data stream is that all its records are of compatible structure and semantics. Data fields with a given name in one record must match any other correlating field in the record of the data stream, both in type and semantics. However, as mentioned in the previous section the core platform services are payload agnostic and there is no requirement at this level for any particular fields to be present in a message. Completeness and compatibility will be the responsibility of the solution and device developers.

Another example is including a version identifier when versioning records, allowing multiple concurrent streams to be segregated by version. For example, using a "_version" convention in which the value is a version tag can keep the stream together while allowing for differentiation. With a versioning model in place, solution developers can appropriately resolve potential conflicts of record fields in terms of semantics or type.

2.2.3. Device interaction

The reference model adopts the principles of the Service Assisted Communication[4] approach for establishing trustworthy bidirectional communication with devices that are potentially deployed in untrusted physical space. The following principles apply:

- Devices do not accept unsolicited network connections. All connections and routes are established in an *outbound-only* fashion.
- Devices generally *only connect to or establish routes to well-known service gateways* that they are peered with. In case they need to feed information to or receive commands from a multitude of services, devices are peered with a gateway that takes care of routing information downstream, and ensures that commands are only accepted from authorized parties before routing them to the device.
- The communication path between device and service or device and gateway is *secured at the transport and application protocol layers*, mutually authenticating the device to the service or gateway and vice versa. Device applications do not trust the link-layer network.

[3] http://www.amqp.org/

[4] http://blogs.msdn.com/b/clemensv/archive/2014/02/10/service-assisted-communication-for-connected-devices.aspx

- System-level authorization and authentication should be based on *per-device identities*, and access credentials and permissions should be near-instantly revocable in case of device abuse.
- Bidirectional communication for devices that are connected sporadically due to power or connectivity concerns may be facilitated through holding commands and notifications to the devices until they connect to pick those up.
- Application payload data may be separately secured for protected transit through gateways to a particular service.

Note: A common pattern for informing power-constrained devices of important commands while disconnected is through the use of an out-of-band communication channel, such as cellular network protocols and services. For example, an SMS message can be used to "wake up" a device and instruct it to establish an outbound network connection to its "home" gateway. Once connected, the device will receive the outstanding commands and messages.

2.2.4. Communication protocols

There is a very large number of communication protocols available for device scenarios today and the number is rapidly growing. Choosing from among those for use with hyper-scale systems in order to ensure secure operations, while providing the capabilities and assurances promised by the chosen protocols, requires significant expertise in building out distributed systems. Yet, there is a vast number of existing devices for which protocol choices have already been made and these devices must be integrated into solutions.

This reference model discusses preferred communication protocol choices, explains potential trade-offs with these choices, and also explicitly allows for custom protocol extensibility and adaptation at the field gateway or in a cloud-based protocol gateway.

Please note that the communication protocol defines how payloads are moved and carries metadata about the payload that can be used for dispatching/routing and decoding, but commonly does not define the payload shape or format. For example, the communication may be enabled by the AMQP protocol, but the data encoding may be Apache Avro, or JSON, or AMQP's native encoding.

3. Architecture components

3.1. Device connectivity

Devices can be connected directly or indirectly via a field gateway. Both devices and field gateways may implement edge intelligence and analytics capabilities. This enables two things: aggregation and reduction of raw telemetry data before transport to the back end, and local decision-making capability with rules that run either on the device or on the edge.

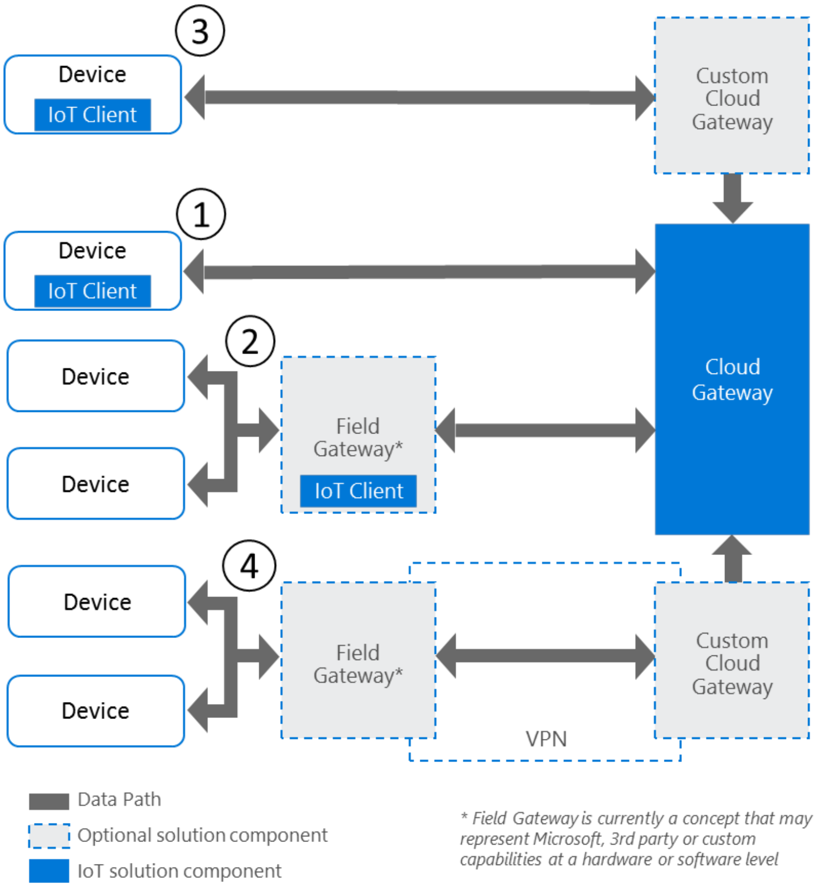
Figure 3 outlines the conceptual representation of the different device connectivity options for IoT solutions. The numbers in the figure correspond to four key connectivity patterns, defined as follows:

1. Direct device connectivity to the cloud gateway:
   For IP capable devices that can establish secure connections via the Internet.
2. Connectivity via a field gateway:
   For devices using industry specific protocols (such as CoAP[5], OPC), short-range communication technologies (such as Bluetooth, ZigBee), as well as for resource-constrained devices not capable of hosting a TLS/SSL stack, or

[5] http://en.wikipedia.org/wiki/Constrained_Application_Protocol

devices not exposed to the Internet. This option is also useful when aggregation of streams and data is executed on a field gateway before transferring to the cloud.

3. Connectivity via a custom cloud gateway:
   For devices that require protocol translation or some form of custom processing before reaching the cloud gateway communication endpoint.
4. Connectivity via a field gateway and a custom cloud gateway:
   Similar to the previous pattern, field gateway scenarios might require some protocol adaption or customizations on the cloud side and therefore can choose to connect to a custom gateway running in the cloud. Some scenarios require integration of field and cloud gateways using isolated network tunnels, either using VPN technology or using an application-level relay service.



** Field Gateway is currently a concept that may represent Microsoft, 3rd party or custom capabilities at a hardware or software level*

Figure 3 Conceptual representation of device connectivity

> *Note:* The use of the term “gateway” (field gateway, custom cloud gateway, and cloud gateway) is only to help set context for the conceptual components that can exist at this level. These do not represent names of Microsoft products. For example, cloud gateway as shown in Figure 3 is not a Microsoft product. It’s a concept that’s realized through Microsoft Azure IoT Hub.

Direct device-to-device communication enables local network control activities and information flow, or collaborative operations where multiple devices perform some sort of coordinated action. Purely local interactions are outside the scope of this architecture and covered by industry standards such as AllJoyn, UPnP/DLNA, and others.

It is important to understand the terminology and key components used to describe device connectivity of the Azure IoT reference architecture. The following sections provide a more detailed description.

3.1.1. Devices

Heterogeneous device support. The goal is to enable secure, efficient, and robust communication between nearly any kind of device and a cloud gateway. This can be done both directly and through gateways, in order to make it possible to implement practical cloud-assisted or cloud-based commercial solutions.

Target devices. The devices in focus are line-of-business assets, from simple temperature sensors to complex factory production lines with hundreds of components with sensors inside them.

The actual purpose for these devices will dictate their technical design as well as the amount of resources needed for their production and scheduled lifetime operation. The combination of these two key factors will define the available operational energy and physical footprint, and thus the available storage, compute, and security capabilities. The reference architecture is generally neutral toward the runtime, platform, operating system, and performed function of the device.

3.1.2. Field gateway

A field gateway is a specialized device-appliance or general-purpose software that acts as a communication enabler and, potentially, as a local device control system and device data processing hub. A field gateway can perform local processing and control functions toward the devices; on the other side it can filter or aggregate the device telemetry and thus reduce the amount of data being transferred to the cloud back end.

A field gateway's scope includes the field gateway itself and all devices that are attached to it. As the name implies, field gateways act outside dedicated data processing facilities and are usually collocated with the devices.

A field gateway is different from a mere traffic router in that it has an active role in managing access and information flow. It is an application-addressed entity and network connection or session terminal. For example, gateways in this context may assist in device provisioning, data filtering, batching and aggregation, buffering of data, protocol translation, and event rules processing. NAT devices or firewalls, in contrast, do not qualify as field gateways since they are not explicit connection or session terminals, but rather route (or deny) connections or sessions made through them.

3.1.3. Cloud gateway

A cloud gateway is the part of the cloud-based architecture that enables remote communication to and from devices or field gateways, which potentially reside at several different sites. A cloud gateway will either be reachable over the public Internet, or a network virtualization overlay (VPN), or private network connections into Azure datacenters, to insulate the cloud gateway and all of its attached devices or field gateways from other network traffic.

It generally manages all aspects of communication, including transport-protocol-level connection management, protection of the communication path, device authentication, and authorization toward the system. It enforces connection and throughput quotas, and collects data used for billing, diagnostics, and other monitoring tasks. The data flow from the device though the cloud gateway is executed through one or multiple application-level messaging protocols.

In order to support event-driven architectures and the common communication patterns outlined in section 2.2.3, a cloud gateway typically offers a brokered communication model. Telemetry and other messages from devices are input into the cloud and the message exchange is brokered by the gateway. Data is durably buffered, which not only decouples the sender from the receiver, but also enables multiple consumers of the data. Traffic from the service back end to devices (such as notifications and commands) is commonly implemented through an “inbox” pattern. Even when a device is offline, messages sent to it will be durably persisted in a store or queue (representing the inbox for a device) and delivered once the device connects. Due to possible time-delayed consumption of events, providing a time-to-live (TTL) value is important, especially for time-sensitive commands, such as "open car or home door" or "start car or machine." The inbox pattern will store the messages in the durable store for the given TTL duration, after which the messages will expire.

Brokering the communication through the described patterns allows decoupling the edge from the cloud components with respect to run-time dependencies, speed of processing, and behavior contracts. It also enables the composability of publishers and consumers as needed to build efficient, high-scale, event-driven solutions.

Technology options

Azure IoT Hub. The role of the main cloud gateway technology in Azure is taken on by Azure IoT Hub, which is a high-scale service enabling secure bidirectional communication from variety of devices. Azure IoT Hub connects millions of devices and supports high-volume telemetry ingestion to a cloud back end as well as command and control traffic to devices. Azure IoT Hub follows the principles outlined in section 7 and supports multiple consumers for cloud ingestion as well as the inbox pattern for devices. Azure IoT Hub provides support for the AMQP 1.0 with optional WebSocket[6] support, MQTT 3.1.1[7], and native HTTP 1.1 over TLS protocols.

Azure Event Hubs. Azure Event Hubs is a high-scale ingestion-only service for collecting telemetry data from concurrent sources at very high throughput rates. Event Hubs could also be used in IoT scenarios, in addition to IoT Hub, for secondary telemetry streams (that is, non-device telemetry), or collecting data from other system sources (such as weather feeds or social streams). Event Hubs doesn’t offer per-device identity or command and control capabilities, so it might be suited only for additional data streams that could be correlated with device telemetry on the back end, but not as a primary gateway for connecting devices. Azure Event Hubs provides support for the AMQP 1.0 with optional WebSocket support, and native HTTPS protocols.

Support for additional protocols beyond AMQP, MQTT and HTTP can be implemented using a protocol gateway adaptation model. Examples of protocols that can use this model are CoAP or OPC TCP.[8]

3.1.4. Custom cloud gateway

A custom cloud gateway enables protocol adaptation and/or some form of custom processing before reaching the cloud gateway communication endpoints. This can include the respective protocol implementation required by devices (or field gateways) while forwarding messages to the cloud gateway for further processing and transmitting command and control messages from the cloud gateway back to the devices. In addition, custom processing such as message transformations or compression/decompression can also be implemented as part of a custom gateway. However, this needs to be evaluated carefully because, in general, it’s beneficial to ingest the data to the cloud gateway as fast as possible and then perform transformations on the cloud back end decoupled from the ingestion.

[6] http://en.wikipedia.org/wiki/WebSockets
[7] http://mqtt.org/
[8] http://en.wikipedia.org/wiki/OPC_Unified_Architecture

Custom gateways help connect a variety of devices with custom or proprietary requirements and normalize the edge traffic on the cloud end. Solution-specific custom gateways will commonly act as a pass-through facility and can implement a custom authentication or rely on the authentication and authorization capabilities of the cloud gateway.

> *Note:* In general, custom gateways can be deployed on the edge as well, and in some cases there might be multiple gateways between a device and the cloud gateway. In the context of this reference model, a custom gateway deployed on the edge would act as a field gateway.

Technology options

Custom gateways are typically built and operated to fulfil specific solution requirements. They may, and often will, lean on shared open-source code that is built in collaboration with the system integrator (SI) and independent software vendor (ISV) community.

Azure IoT protocol gateway. Azure IoT protocol gateway is an open-source framework for custom gateways and protocol adaptation. The Azure IoT protocol gateway facilitates high-scale, bidirectional communications between devices and Azure IoT Hub. It includes a protocol adapter for MQTT that showcases the techniques for implementing custom protocols and enables customizations of the MQTT protocol behavior, if required. The protocol gateway also allows for additional processing such as custom authentication, message transformations, compression/decompression, or encryption/decryption.

3.1.5. IoT client

Cloud-communication with devices or field gateways must occur through secure channels to the cloud gateway endpoints (or cloud-hosted custom gateways).

In addition to a secure communication channel, the device usually needs to deliver telemetry data to the cloud gateway and allow for receiving messages and executing actions or dispatching those to appropriate handlers in the client. As described earlier in the section 2.2.3, all device (or gateway) connections and routes should be established in an outbound-only fashion.

There are three key patterns for client connectivity being used in IoT systems:

- Direct connectivity from the device app/software layer
- Connectivity through agents
- Using client components integrated in the app/software layer of the device or gateway

Direct connectivity. In this case the communication to a cloud gateway endpoint is natively coded in the device (or field gateway) software layer using the desired protocols. This requires knowledge of the required protocols and message exchange patterns, but provides full control over the implementation down to the bits on the wire.

Agents. An agent is a software component installed on a device (or field gateway) that performs actions on behalf of another program or managing component. In the IoT space agents are typically controlled and act for components running on the cloud back end. For example, in the case of a command sent to a device, the agent will receive the command and can execute it directly on the device.

Agents could be proprietary agents, specifically written for a particular software solution, or standard-based agents implementing particular standards such as OMA LWM2M. In both cases it’s convenient for device developers to integrate and rely on the encapsulated capabilities of the agents; however, there are some limitations. Typically, agents represent a closed system, constrained to the capabilities provided by the agent for a set of supported platforms. Portability to other platforms or customizations and extensions beyond the provided functionality are typically not possible.

Client components. Client components provide a set of capabilities that can be integrated in the software code running on the device to simplify the connectivity to a back end. They are typically provided as libraries or SDKs that can be linked or compiled into the software layer of the device. For example, if a cloud back end sends a command to a device, the client components will simplify receiving the command, though the execution will be performed in the scope of the app/software layer.

Compared to agents, client components require integration effort into the device software, but they provide the greatest flexibility for extensibility and portability.

Technology options

Azure IoT device SDKs. The Azure IoT device SDKs represent a set of client components that can be used on devices or gateways to simplify the connectivity to Azure IoT Hub. The device SDKs can be used to implement an IoT Client (shown in Figure 3) that facilitates the connectivity to the cloud. They provide a consistent client development experience across a broad number of platforms without having to confront device developers with the complexity of distributed systems messaging. The libraries enable the connectivity of a heterogeneous range of devices and field gateways to an Azure-based IoT solution. They simplify common connectivity tasks by abstracting details of the underlying protocols and message processing patterns. The libraries can be used directly in a device application or to create a separate agent running on the device that establishes connectivity with the cloud gateway and facilitates the communication between the device and the IoT solution back end.

The Azure IoT device SDKs are an open-source framework that is aligned with the Azure IoT platform capabilities. While these libraries simplify the connectivity to Azure IoT Hub, they are optional and not required if device developers choose to connect to the IoT Hub endpoints using existing frameworks and supported protocol standards.

3.2. Device identity store

Device identity authority. The device identity store is the authority for all device identity information. It also stores and allows for validation of cryptographic secrets for the purposes of device client authentication (see Figure 4, on the following page). The identity store does not provide any indexing or search facility beyond direct lookup by the device identifier; that functional role is taken on by the device registry (see next section for details). Identity and registry stores are primarily separated for security reasons; lookups on the registry should not allow disclosing cryptographic material. Further, limiting the identity store to a minimal set of system-controlled attributes helps to provide fast and responsive operations, while on the other hand the schema of the registry store is determined by the solution requirements.

The cloud gateway relies on the information in the identity store for the purposes of device authentication and management. The identity store could be contained in the cloud gateway, or alternatively the cloud gateway could use separate device identities externally.

Provisioning. Device provisioning uses the identity store to create identities for new devices in the scope of the system or to remove devices from the system. Devices can also be enabled or disabled. When they are disabled, they have no access to the system, but all access rules, keys, and metadata stay in place.

Changes on the device identity store should be made through the Provisioning API, described in section 3.4.

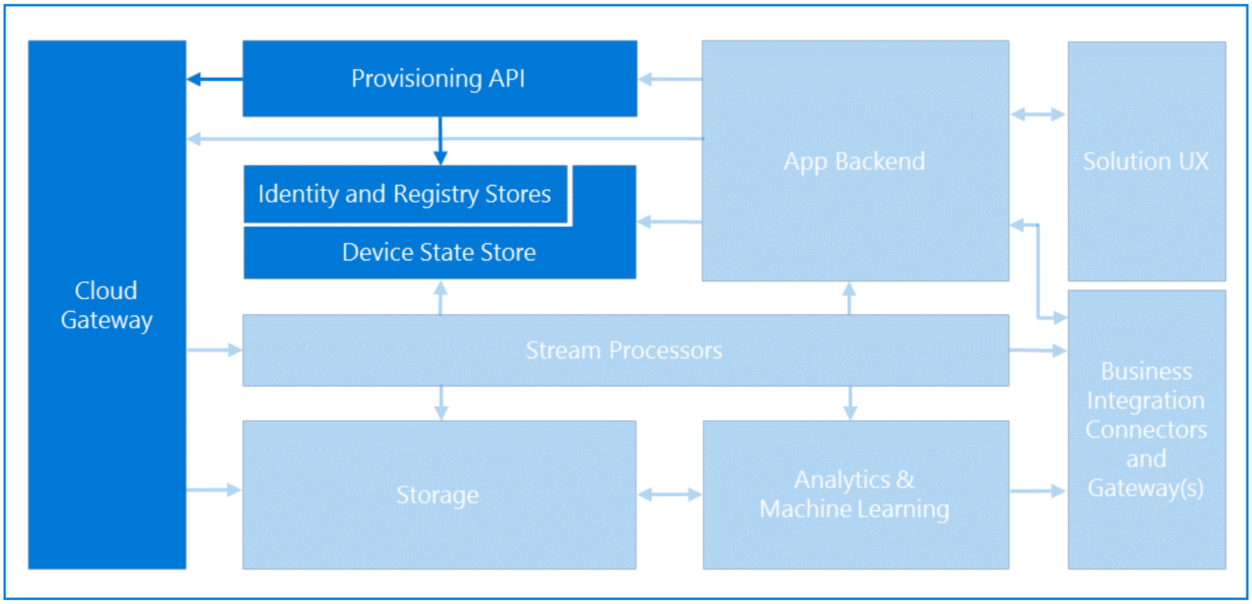

Provisioning API
Identity and Registry Stores
Device State Store
App Backend
Solution UX
Cloud Gateway
Stream Processors
Business Integration Connectors and Gateway(s)
Storage
Analytics & Machine Learning

Figure 4 Device provisioning and device identity, registry, and state stores

Technology options

Azure IoT Hub includes a built-in device identity store that is the authority for registered devices and provides per-device security credentials.

When a custom cloud gateway is used, it can also rely on the IoT Hub identity store and its authentication and authorization capabilities. In case there are specific solution requirements that necessitate a custom implementation of the identity store, it will be a separate component that will primarily enforce identifier uniqueness, store all required security keys for the device, and will potentially hold an “enabled/disabled” status. If it includes transmitted passphrases, those should be stored in the form of salted hashes. Please keep in mind that a custom implementation of the identity store needs to be secured appropriately, because it stores credential information.

The identity store should only permit access to privileged parts of the system as necessary; custom gateways will look up the required authentication material from this store.

If not using Azure IoT Hub, external implementations can be realized on top of Azure DocumentDB, Azure Tables, Azure SQL Database, or third-party solutions:

- **Azure DocumentDB:** In Azure DocumentDB,[9] each device is represented by a document. The system-level device identifier directly corresponds to the “id” of the document. All further properties are held alongside the “id” in the document.
- **Azure Tables:** In Azure Tables, the identity store maps to a table. Each device is represented by a row. The system-level device identifier is held in a combination of PartitionKey and RowKey, which together provide uniqueness. All further properties are held in columns; complex data can be stored as JSON, if needed. The

[9] http://azure.microsoft.com/en-us/documentation/articles/documentdb-introduction/

concrete split of the identifier information across those fields is application specific and should follow the scale guidance for the service.[10]

- **SQL Database:** In SQL, the identity store also maps to a table and each device is represented by a row. The system-level device identifier is held in a clustered index primary key column. All further properties are stored in columns; complex data or data requiring extensibility can be stored as JSON, if needed.
- **Third-party options:** Third-party solutions available through the Azure Marketplace or directly deployed on Azure compute nodes can be used as well. For example, in Cassandra, each device can be represented by a row in a column family. The store will be partitioned and indexed for fast access as needed.

3.3. Device registry store

Definition and function. The device registry is an “index” database existing alongside the identity store, which contains discovery and reference data related to provisioned devices. While the identity store only contains system-controlled attributes and cryptographic material that is immediately available, the registry will store other device-related metadata information for the solution (see Figure 4, in the preceding section).

The registry does not impose any particular schema model or structure for device metadata, but it is possible to define a schema model, or select some vertical industry-standard schema model for device metadata. During provisioning, each device is registered with a metadata record, which can contain structured metadata and can include links to externally held operational data.

Device registry versus identity store. The device registry is an index, while the identity store represents the authoritative list of device identities. The record in the identity store determines whether or not a device is active in the system. For security reasons, the device registry must not store any key or other cryptographic information related to the device.

Metadata. The distinction between metadata describing the device itself and operational data reflecting the state of the device is important because it directly impacts how the registry information can be used, cached, and distributed throughout the system. Metadata is typically slow-changing data, while the operational data is expected to be fast-changing.

For example, the geo-position of a traffic-light pole is metadata, but the current geo-position of a vehicle is considered operational data. The vehicle’s identification number, model, and make will be metadata. Discovery of all traffic lights on a particular stretch of road can be performed as a registry query, while finding all vehicles currently driving on a particular stretch of road would be an analysis task inside the solution over operational data. The metadata in the registry can help as reference data for finding all vehicles of a particular model on the road, however.

Changes on the device registry should be made through the Provisioning API.

Technology options

The device registry storing descriptive information about the device should provide rich or free-form index capabilities with the goal of providing fast lookups.

[10] http://msdn.microsoft.com/en-us/library/azure/hh508997.aspx

The registry store can be implemented on top of one of the following technologies:

- **DocumentDB:** In Azure DocumentDB, each device is represented by a document. The system-level device identifier directly corresponds to the "id" of the document. All further properties are held alongside the "id." DocumentDB is well suited for the registry function because it accepts arbitrarily structured data and automatically creates indexes (unless disabled for specific attributes). This allows for fast and flexible lookups[11] across the registered devices, which is the purpose of the registry.
- **SQL Database:** In SQL, the registry maps to a table and each device is represented by a row. The system-level device identifier is held in a clustered index primary key column. All further properties are stored in columns; complex data or data needing extensibility can be stored as JSON or XML, if needed. Based on query patterns the appropriate columns will need to be indexed.
- **Third-party options:** In addition to managed Azure services, third-party data services available through the Azure Marketplace or directly deployed on Azure compute nodes can be used as well. In this case the actual schema depends on the application used but the structure is going to be similar to the one used for SQL Database or DocumentDB. Partitioning and indexing will be applied as needed for fast access based on device properties. For example, Cassandra's column family could have the device identifier as partition key and could define additional indexes on other properties of the device.

3.4. Device provisioning

Definition. Provisioning represents the step of the device life cycle that is undertaken to be made known to the system. The Provisioning API is the common external interface for how changes are made on the device identity store and the device registry. It is an abstract interface with common gestures, and there is an implementation of that abstract interface for the identity and registry stores. Higher level workflows can implement the same or similar interface and delegate to the Provisioning API as appropriate in the workflow implementation.

Provisioning workflow. A solution's provisioning workflow takes care of processing individual and bulk requests for registering new devices and updating or removing existing devices. It will also handle the activation, and potentially the temporary access suspension and eventual access resumption. This may also include interactions with external systems such as a mobile operator's M2M API to enable or disable network SIMs, or with business systems such as billing, support, or customer relationship management solutions.

Technology options

As an alternative to traditional programing techniques, Azure API Apps[12] can be used for the implementation of the Provisioning API. API Apps provides a platform for building, hosting, and distributing APIs in the cloud and on-premises. API Apps integrates seamlessly with Azure Logic Apps,[13] which can be used for the implementation of an overarching provisioning workflow across the IoT solution and external business systems.

The provisioning interface is a simple set of gestures for managing the device life cycle. The provisioning interface (API) should be implemented as the primary API over the identity and registry stores and optionally other internal solution components if required. It is not only used from the solution UI (for example, the device administration portal), but also

[11] http://azure.microsoft.com/en-us/documentation/articles/documentdb-sql-query/
[12] https://azure.microsoft.com/en-us/documentation/articles/app-service-api-apps-why-best-platform/
[13] https://azure.microsoft.com/en-us/documentation/articles/app-service-logic-what-are-logic-apps/

serves as the interface for a higher level workflow that can also interact with external entities such as a mobile operator's M2M API for managing SIM cards or a backend business system for activating a billing account associated with the device.

The following table provides an overview of the typical functionality exposed through the provisioning API. For brevity, it does not list success or error codes.

Operation	Arguments	Return
Register Registers the device in the system.	**id (string)** Device identifier to be used. Might be optional in which case the identity is service assigned. **keytokens** A single key or a map of named keys to be registered for the device in the identity store. Might be optional, in which case the keys are service assigned. **metadata** Structured data object containing descriptive metadata for the device.	**id (string)** Device identifier recorded for the device. **keytokens** A map of named keys or tokens assigned to the device. This is the only time these particular keys surface through an API call. **metadata** Structured data object containing descriptive metadata for the device.
Unregister Removes the device from the system.	**id (string)** Device identifier.	
Activate Activates the device from a previously deactivated state (which includes granting access).	**id (string)** Device identifier. **metadata** Structured data object containing descriptive metadata for the device. This metadata is merged into the existing metadata set.	**id (string)** Device identifier recorded for the device. **metadata** Structured data object containing descriptive metadata for the device.
Deactivate Deactivates the device from a previously activated state (which includes revoking access).	**id (string)** Device identifier. **metadata** Structured data object containing descriptive metadata for the device. This metadata is merged into the existing metadata set.	**id (string)** Device identifier recorded for the device. **metadata** Structured data object containing descriptive metadata for the device.

Update Updates the device metadata in the registry.	**id (string)** Device identifier. **metadata** Structured data object containing descriptive metadata for the device. This metadata is merged into the existing metadata set.	**id (string)** Device identifier recorded for the device. **metadata** Structured data object containing descriptive metadata for the device.
ResetCredentials Full reset of all device credentials and tokens.	**id (string)** Device identifier. **keytokens** A single key or a map of named keys to be registered for the device in the identity store. Might be optional, in which case keys are service assigned.	**id (string)** Device identifier recorded for the device. **keytokens** A map of named keys or tokens assigned to the device. This is the only time these particular keys surface through an API call.

Security keys can be generated outside of the API and passed in as parameters or can be created and assigned by the service as part of the provisioning API call.

Generating a security token can be performed in the Provisioning API using the required signing key. The token issued to a device will be limited in scope to a particular endpoint (for example, a device endpoint in the case of IoT Hub or Event Hub publisher policy). The data returned by the *Register* and *ResetCredentials* operation contains the required security tokens that must be transferred to the devices. Alternatively, security tokens could be generated on the device or externally and passed to the devices.

For custom gateways, the required credentials can be generated externally and passed into the API for storage, or the API can be extended to create the keys.

3.5. Device state store

Definition. Operational data related to the devices resides in the device state store. The device state store is separate from the device registry. Any device's registry record will commonly point to the device's state store.

While storing the raw information from the device is often desirable, the state store is an optional architectural element. A simple implementation can include writing the device data stream to a storage by default, but the store itself and the data flow to it can be removed or changed.

The default shape for the device state store is that it retains two kinds of information. One part is the raw stream of incoming events from the device. The other is a "last known values" record that is a projection of the last observed values captured from the device.

Technology options

For a minimal implementation, the incoming raw data or message projections can be stored in Azure Blobs or Tables, which helps to optimize for cost. The incoming messages can be forwarded "as is," or can be modified to some other output shape. The data could be consumed directly from the store or additional transformations can be performed using Azure Data Factory,[14] which will allow for secondary transformation, aggregations, and data movement as needed for the solution.

In addition to the raw data, the last known values for a device are stored separately as a record that gets constantly overwritten. This record can be a separate Azure blob or table object, or can be added as a separate document to the device registry for fast querying capabilities. Aggregated or calculated values can be added to this or a separate record too.

In some cases, the operational data will be segregated in different stores based on access patterns. For example, data for audit purposes such as state changes, history of operations, and commands will be accessed and processed independent of other operational data. Archival data, moved to a separate storage after a certain retention period, is another example. Logical separation of data for groups of devices (such as geographically) or maybe for subcomponents of complex devices is possible as well. The partitioning design will follow the access needs and operational requirements for the data. An implementation choice should be made for each of these stores individually driven by the workload patterns:

- **Azure Data Lake:** Azure Data Lake is a distributed data store allowing to persist vast amounts of relational and nonrelational data without transformation or schema definition. It can handle high volumes of small writes at low latency, is optimized for massive throughput, and well suited for event streams in IoT scenarios.
- **Azure Blob storage:** Blobs can be used to store raw device data. Containers and blob names can be used to represent a certain structure of the storage space that will be designed based on the solution requirements. Using append blobs instead of standard blobs can be considered when appropriate.
- **Azure Tables:** Device records and aggregated or computed values can be stored in Azure Table. Azure Tables are also suited for logging, auditing information, or other types of device data that will be accessed by partitions or directly at the entity level, because there are no secondary indexes. The partitioning strategy and use of PartitionKey and RowKey is solution specific and should be designed in accordance with the scale guidance for Azure Tables.[15]
- **Azure DocumentDB:** Datasets that can benefit from flexible, schema-agnostics indexing capabilities and rich SQL query interface can be stored in DocumentDB, which combines the management of schema-free, no-SQL documents with complex SQL language queries.
- **SQL Database:** For datasets that require relational storage and query capabilities. SQL Database also provides advanced features for data management, protection and security, and business continuity.
- **Azure Search:** Azure Search[16] can be used in scenarios requiring full-text search scoped over the data content and advanced search behavior.
- **Azure Cache:** In addition to durable storage options, device state can also be held in Azure Cache for fast lookups of device state and operational data.

[14] https://azure.microsoft.com/en-us/documentation/articles/data-factory-introduction/
[15] http://msdn.microsoft.com/en-us/library/azure/hh508997.aspx
[16] https://azure.microsoft.com/en-us/documentation/articles/search-what-is-azure-search/

- **Third-party options:** In addition to the Azure services just mentioned, third-party options hosted in Azure can be used as well. A few examples of third-party data services include MongoDB, Cassandra, Elastic Search, and time series databases such as OpenTSDB and InfluxDB. If an Actor Model as described in section 3.8 is used, parts of the device state will be based on the persistence options that the Actor framework provides. For example, for Akka and Akka.net this can be Cassandra, LevelDB, or any other storage system they support.

The different storage options will require careful design of the partitioning strategy and capacity unit management (with the exception of Azure Data Lake).

3.6. Data flow and stream processing

Facilitating data flow. After ingress through the cloud gateway, the flow of data through the system is facilitated by data pumps and analytics tasks (shown as Stream Processors in Figure). Data pumps are typically moving or routing data without any transformation, while analytics tasks perform complex event processing.

As described in the data concepts section above, multiple data streams can flow concurrently. In addition, since the cloud gateway provides brokered communication and supports multiple consumers, the same data can be consumed by different stream processors for different purposes. For example, a stream processor may listen only for special types of events, while another one could perform complex event processing in parallel. Those processors can determine the path of data and route without any reshaping or perform complex event processing tasks such as data aggregation, data enrichment through correlation with reference data, as well as analytics tasks such as detection of threshold limits or anomalies and generation of alerts.

Figure shows the core flows facilitated by stream processors of an IoT solution.

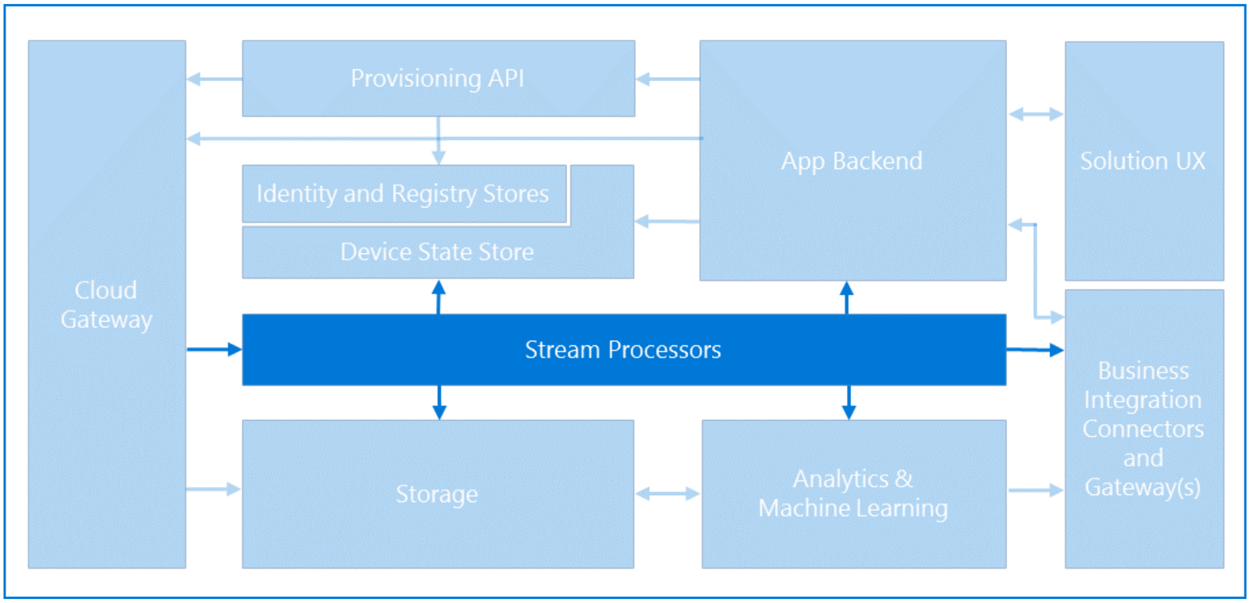

Provisioning API
App Backend
Solution UX
Identity and Registry Stores
Device State Store
Cloud Gateway
Stream Processors
Business Integration Connectors and Gateway(s)
Storage
Analytics & Machine Learning

Figure 5 Stream processors

The following paragraphs describe typical patterns and data flows of the reference architecture.

Raw telemetry. The simplest flow is facilitated by a data pump reading device telemetry from the cloud gateway and persisting it into storage. This data can be used as archive for raw device data as well as for batch analytics.

Bulk upload. Large amounts of raw telemetry data might also be collected on the edge and uploaded using bulk operations. This is a viable alternative for (cold) telemetry data, when there is no need to process individual records and to immediately act upon the data. It can be uploaded and used as archive or for batch analytics later.

Device state. In addition to processing the raw device data a stream processor can keep updating the "last known values" for devices. Specific aggregated or precalculated values can also be stored in the device state store for easy access by the app back end as required by the solution logic.

Device metadata. In some cases, devices may send messages indicating changes of their metadata attributes. Typically, those are separated from the general telemetry stream. An event processor can "listen" for those messages and update the device registry as appropriate. An example would be a configuration change performed on the device.

Special events. Special types of messages (that is, non-telemetry messages) coming from devices include solution-specific events such as alarms, notifications, and domain-specific updates. Those messages can be processed as a separate stream by a dedicated event processor, or in some cases combined with inquiries and command responses flowing to the back end.

Diagnostics telemetry. In addition to device telemetry, there could be a data flow of diagnostics information emitted by devices. This stream related to health and operations conditions of a device most likely will be handled separately from the general telemetry.

Hot path analytics. A complex event processing engine can analyze ingested events in (near) real time, comparing multiple real-time streams or comparing real-time streams with historical values and models. This enables the detection of anomalies, recognition of patterns over rolling time windows, and the ability to trigger an alert when a specific error or condition appears in the stream. Generated alerts are forwarded to the app back end to be handled according to the business rules or can initiate an integration workflow with line-of-business systems directly.

Advanced analytics and machine learning. Incoming events can also be forwarded to specialized modules for advanced analytics and machine learning. Those can perform large-scale, in-motion analysis and visualizations.

The reference architecture assumes the use of multiple event processors, dedicated to the processing of one or several of the described core data flows. Devices will typically segregate the traffic in multiple data streams by using discriminators in the application protocol header (for example, message properties such as "stream-id" or "subject") that will allow for the routing and processing by the appropriate stream processor. Devices or gateways might also implement some data classification logic, and split the data into categories for cold or hot path processing, for example.

Technology options

In Microsoft Azure, the Stream Analytics service, Apache Storm implemented in Azure HDInsight, or custom event processors can facilitate the flow of data from the ingestion point in Azure IoT Hub or Event Hubs. They can perform timely processing of the data stream, including data aggregation, data enrichment through correlation with reference data, detection of threshold limits and producing alerts, and other analytics tasks. These event processors can also be used as a rule engine environment, where event rules for thresholds and limits can be defined, configured, and activated/deactivated.

The output of these processing tasks is commonly a permanent storage location in Azure Blobs, Azure Tables, Azure SQL Database, HDInsight HBase,[17] or some other store. The output can also be forwarded to Event Hubs or, when the data flow has been sufficiently reduced, also to Azure Service Bus Queues and Topics for distribution into subsystems where further downstream processing is performed.

> *Note*: Azure Stream Analytics currently allows forwarding data into Azure Blobs and Tables, SQL Database, DocumentDB, Event Hubs, Service Bus Queues and Topics, as well as Power BI. Custom event processors and HDInsight Storm bolts allow forwarding into any event sink either through custom-built code or available open-source components.

In addition to Azure Stream Analytics and Apache Storm, it's possible to build a Lambda architecture in Azure using frameworks such as Apache Kafka, Apache Cassandra, and Apache Spark for downstream processing. The incoming stream of data (for example, device telemetry) is going to be routed to the various processing stages via Kafka and stored in Cassandra so that batch and big data jobs (such as Spark jobs) can be executed on the data.

After data is at rest, it can be picked up, transformed, and stored in a different store using a continuously or periodically running data pump (or pipeline), available through Azure Data Factory. Data factories can perform nearly arbitrary transformations and transfers of at-rest data between Azure blobs, Azure tables, and SQL databases in Azure PaaS, IaaS, or running on on-premises servers.

Azure Stream Analytics uses SQL-based language for rapid development, while HDInsight Storm and custom event processors are programing code-based solutions that require deploying code in the runtime environment, but also provide full flexibility for processing. Because of the ease of use of Stream Analytics and transparency of SQL syntax used, an example of a minimal implementation using Stream Analytics will be described to better explain the core concepts of an event stream processor. In this example, Stream Analytics is connected directly to Azure IoT Hub as a stream input source and is used as a data pump and rule engine performing the following processing tasks:

- **Raw telemetry:** A Stream Analytics job based on "SELECT * FROM <iot-hub input>" query captures the incoming raw data unmodified. If special types of events (such as device metadata updates) should be excluded, a WHERE clause can be added to filter out those messages. The output of this job is configured to persist data into Azure Blob storage. Once in Azure storage additional transformations or data movement can be realized using Azure Data Factory.
  Note: If the order of events per device is relevant for the application and the way records are persisted, the query used in this example needs to be modified to "SELECT * FROM <iot-hub input> Partition By PartitionId" to ensure events within a partition are processed in the same order as received.
- **Telemetry transformations.** In addition, further Stream Analytics jobs can be defined to create projections through a range of operators from simple filters to complex correlations and lookups. This leads to new stream(s) originating at this stage of the data flow that can be captured as part of the application-specific model (used for backend analytics or application logic).
- **Rules and alerts.** As mentioned, Stream Analytics jobs can be used to execute rules for detection of thresholds and limits, such as using a syntax similar to "SELECT <alert> FROM <iot-hub input> WHERE <device-attribute> >= <value>". However, a single spike in measurement might not be enough to trigger an alert. An average value over a certain period of time violating the desired threshold might be a more appropriate way to express the rule.

[17] http://hbase.apache.org/

The generated alert will be output to a Service Bus Queue or Event Hubs for processing by the back end, which will typically trigger a preconfigured action.

- **Per-device telemetry.** A custom event processor[18] reading from Azure IoT Hub can be used to segregate the incoming data stream into a per-device state store. Using this technique, the data stream events for each device are written to a separate device state store based on Azure Blob storage.
  An alternative approach is to modify the Stream Analytics job for raw telemetry (based on the "SELECT *" query described previously in the "Raw telemetry" bullet) and configure its output to an internal-flow Event Hub. Then an event processor for Event Hub is used to segregate the incoming data stream into a per-device state store. The advantage of this technique is that in addition to the raw data from the device, projections from Stream Analytics jobs may produce the last known values, averages, or other pertinent device attributes, which can be stored as a separate record (for example when those are used to represent a device state in the application layer).
  In general, even for the raw telemetry, going through the Stream Analytics data path has the advantage of allowing for easy modification of the job query to create a data stream projection (for example, including filtering or correlation with reference data).

In addition to the described data flows, further Stream Analytics outputs or jobs can be used to output data to permanent storage locations (in Azure storage or databases), or to Service Bus Queues, Topics, or Event Hub, for distribution into subsystems and downstream analysis or processing.

In a very simplified form of implementation, when no rules engine and no additional data flow paths are required, an event processor can be used to process the incoming data stream directly from the ingestion IoT Hub into the device state store. Using Azure Stream Analytics or HDInsight Storm bolts as a data pump and rules engine provides flexibility to easily add new output projections and data streams as needed.

3.7. Solution UX

The solution user experience (UX) typically includes a website, but can also include web services and APIs with a graphical user interface in the form of a mobile or desktop app.

The solution UX is, as the name implies, part of the solution and implements access to and visualization of device data and analysis results, discovery of devices through the registry, command and control capabilities, and the workflows of the provisioning. In many cases, end users will be notified of alerts, alarm conditions, or necessary actions that need to be taken through push notifications.

The solution UX can also provide or integrate with live and interactive dashboards, which are a suitable form of visualizations for IoT scenarios with large population of devices.

IoT solutions often include geo-location and geo-aware services and the UI will need to provide appropriate controls and capabilities.

As stated in beginning of this document, security is key and the solution UX that provides control over the system and devices needs to be secured appropriately with access control differentiated by user roles and depending on authorization.

[18] http://azure.microsoft.com/en-us/documentation/articles/service-bus-event-hubs-csharp-ephcs-getstarted/

Technology options

Azure App Service is a managed platform with powerful capabilities for building web and mobile apps for many platforms and mobile devices. Web Apps and Mobile Apps allow developers to build web and mobile apps using languages like .NET, Java, NodeJS, PHP, or Python. In addition, Azure API Apps allows easy exposure and management of APIs, which can be accessed by mobile or web clients.

Azure Notification Hubs enables sending of push notifications to personal mobile devices (that is, smartphones and tablets). It supports iOS, Android, Windows, and Kindle platforms, while abstracting the details of the different platform notification systems (PNS). With a single API call, a notification can target an individual user or an audience segment with a large number of users.

In addition to the traditional UI, dashboards are very important in IoT scenarios because they provide a natural way for aggregated views and help visualize a vast number of devices. Power BI is a cloud-based service that provides an easy way to create rich, interactive dashboards for visualizations and analysis. Power BI also offers live dashboards, which allow users to monitor changes in the data and indicators. Power BI includes native apps for desktop and mobile devices.

Another suitable technology for IoT visualizations is Bing Maps.[19] The Bing Maps APIs include map controls and services that you can use to incorporate Bing Maps in applications and websites. In addition to interactive and static maps, the APIs provide access to geospatial features such as geocoding, route and traffic data, and spatial data sources that you can use to store and query data that has a spatial component, such as device locations.

The web and mobile apps can be integrated with Azure Active Directory (AAD) for authentication and authorization control. The apps will rely on the management of user identities in AAD and can easily provide role-based access control for the application functionality. In many cases there will be logical associations between IoT devices and users (or between groups of devices and groups of users). For example, a device can be owned by someone, used by someone else, and installed or repaired by another user. Similar examples can be true for groups of devices and users. Permissions and role-based access control can be managed as part of an association matrix between device identities (maintained in the device identity store) and user identities managed by AAD. The specific design of this matrix, granularity of permissions, and level of control will depend on the specific solution requirements. This matrix can be implemented on top of the device registry or can use a separate store using different technology. For example, the device registry can be implemented using DocumentDB, while the association and permission matrix can be built using a relational SQL database. Please note that this topic is discussed in this section because user authentication and authorization is surfaced as part of the UX; however, the actual implementation will be spread across multiple underlying components, including the device registry and the app back end, discussed in the next section.

3.8. App back end

The application back end implements the required business logic of the solution. It implements the appropriate object models and abstractions for devices, groups of devices and relations between devices, business rules and actions, and also manages access and associations between devices and users. Important parts of the application backend are the “custom” control logic of the solution, device discovery and visualization, device state management and command execution, as

[19] http://msdn.microsoft.com/en-us/library/ff428643.aspx

well as the device management portion that controls the device life cycle, enables distribution of configuration and software updates, and remote control of devices.

Unlike traditional business systems, the business logic of an IoT solution might be spread across different components of the system. The solution's device management part will commonly use compute nodes, whereas the analytics portion of the solution will be largely implemented directly inside the respective analytics capabilities.

In extreme cases, simple solutions built along the lines of this guidance may indeed not have an independently deployed and managed "business logic" app back end, but the core logic may exist as rule expressions hosted inside the stream processors, some of the analytics capabilities, and/or as part of the business workflows and connector components.

Technology options

There are several implementation options for the backend logic. As mentioned above, some of the logic will be implemented in the event processors and analytics components of the system. Implementation choices for those components were covered in the respective sections. This section focuses specifically on the business logic back end.

Programming techniques that don't support hyper-scale. Many of the architectural patterns and programming techniques that have been popular for the past decades are applicable to IoT solutions, but might face scalability challenges at large number of devices. Hence, for large IoT deployments, these models should only be used with stateless app back end running on vastly scalable compute nodes. Scaling out a stateful application layer represents a difficult problem with traditional architectures. In those cases, scale appropriate compute models such as actor frameworks or batch processing can be used, as described in the next sections.

Actor frameworks. Actor frameworks represent an extremely well-suited compute model for IoT scenarios. The actor programming model is not new, but is currently gaining traction because it fits well in scenarios where there are a lot of "independent" units with a specific behavior and independent local data/state. The actor framework provides a good abstraction model for devices that need to communicate with backend services. A (physical) device can be modeled as an actor with defined behavior and local state that will run on the back end. The actor becomes a virtual representation of the physical device. An actor can represent a stateful computation unit that manages its own state. Unlike traditional programming techniques, where an instance of an object is created and the state needs to be loaded from outside, a stateful actor has immediately intrinsic state. With a 1:1 relationship between a device and backend "code," the actual implementation becomes easier and developers can focus on the specific behavior that is required to manage a single type of device.

In addition, actor models provide a way to create hierarchies of actors, in order to represent relationships among devices or group of devices. For instance, it is easy to model all the sensors in a building as a hierarchy of actors: a building can be an actor that is composed of a set of floor actors, a floor actor is defined as a set of room actors, and each room actor can control a set of sensors in that room. This way, it is easy to write complex rules and logic that iterate the actor hierarchy. Each element of the hierarchy provides the right behavior and state required to act or aggregate information at the higher level.

An actor can process messages from devices, perform computations, and send commands or notifications to devices when certain conditions are met on the back end. From an abstraction perspective, developers can focus on the code that is required to manage one device, which results in a simple programming model. Most actor frameworks use a message-based architecture, and communication with and among actors is managed by the framework. Actors are invoked only when one or more messages are available and need to be processed; that is, the actor is activated by the framework when there is work to be performed. There is no need to have any “worker role” type of component in the architecture that needs to stay alive to check if there is work to be done. The actor framework scheduler is responsible to schedule actors for execution with the goal of optimizing resource utilization. In the context of this architecture, an actor can be activated when a device event is received, or from the back end, based on events coming from business logic and rules, or a line-of-business system.

There are several actor frameworks available and developers can choose the one that best fits their programming, background, and scenario requirements. The following paragraphs introduce three popular actor frameworks: Azure Service Fabric Reliable Actors, Akka, and Akka.NET.

Azure Service Fabric Reliable Actors

Azure Service Fabric[20] enables developers to build and manage scalable and reliable applications composed of microservices running at very high density on a shared pool of machines, commonly referred to as a Service Fabric cluster. It provides a sophisticated runtime for building distributed, scalable, stateless and stateful microservices and comprehensive application management capabilities for provisioning, deploying, monitoring, upgrading/patching, and deleting deployed applications. Stateful services in Service Fabric offer the benefits of having fully replicated local data that can be used directly by the service without the need for relying on external tools such as cache systems or storage.

Service Fabric provides the Reliable Actors programming model. It is an actor-based programming model that uses the strength of the Service Fabric runtime infrastructure to provide a scalable and reliable model that developers with an object-oriented programming background will find very familiar. The Reliable Actors programming model is very similar to Orleans, and developers that are familiar with Orleans can easily migrate to Reliable Actors or can keep using the Orleans runtime.

In addition to the Reliable Actors, Service Fabric also provides a lower level programming model Reliable Services[21] that has different tradeoffs between simplicity and flexibility in terms of concurrency, partitioning, and communication[22]. With this model Reliable Collections[23] can be used to store and manage device state.

Akka

Akka[24] is a well-known Actor programming model that runs on a Java virtual machine (JVM). It is developed using the Scala programming language, but provides Java APIs as well. Akka-based backend applications can be hosted in Azure and can use Azure IoT services, while enabling a familiar programming model for developers that are already using Java or Scala as their language of choice.

Akka.NET

Akka.NET[25] is a port of the Akka programming model to the .NET runtime and supports both C# and F# . Along with Akka, it provides a way for developers to use the Akka programming model, but run the code on top of the .NET runtime.

[20] http://azure.microsoft.com/en-us/campaigns/service-fabric/
[21] https://azure.microsoft.com/documentation/articles/service-fabric-reliable-services-introduction/
[22] https://azure.microsoft.com/documentation/articles/service-fabric-choose-framework/
[23] https://azure.microsoft.com/documentation/articles/service-fabric-reliable-services-reliable-collections/
[24] http://akka.io/
[25] http://getakka.net/

Azure Batch. Batch processing is well suited for workloads that require running lots of automated tasks, such as performing regular (such as monthly or quarterly) processing, risk calculations, or different types of simulations. Azure Batch[26] is a cloud-scale job scheduling and compute management service that enables users to run highly parallelizable compute workloads. The Azure Batch scheduler can be used to dispatch and monitor the execution of "work" across large-scale compute clusters. It takes care of starting a pool of compute virtual machines, installing processing jobs and staging data, running the jobs, identifying failures, and re-queuing work as needed. It also automatically scales down the pool of resources as the work completes.

3.9. Business systems integration

The business integration layer is responsible for the integration of the IoT environment into downstream business systems such as CRM, ERP, and line-of-business (LOB) applications. Typical examples include service billing, customer support, dealers and service stations, replacement parts supply, third-party data sources, operator profiles and shift plans, time and job tracking, and more.

The IoT solution ties into existing line-of-business applications and standard software solutions through business connectors or EAI/B2B gateway capabilities. End users in B2B or B2C scenarios will interact with the device data and special-purpose IoT devices through this layer. In many cases the end users will use personal mobile devices to access the functionality. Those personal mobile devices are conceptually different than the IoT devices, although in some cases there will be association or mapping between the end user's mobile device and IoT devices. For example, in a home automation scenario, a mobile phone might act as field gateway, connecting to IoT devices and facilitating the communication for those. From an authorization perspective the associations between end users, personal mobile devices, and IoT devices will be managed by the IoT solution back end.

Technology options

Azure Logic Apps provide a reliable way to automate business processes. The service supports long-running process orchestrations across different systems hosted in Azure, on-premises, or in third-party clouds. Logic Apps allow users to automate business process execution and workflow via an easy-to-use visual designer. The workflows start from a trigger and execute a series of steps, each invoking connectors or APIs, while taking care of authentication, check-pointing, and durable execution. There is a very rich set of available connectors to a number of first-party and third-party systems, such as database, messaging, storage, ERP, and CRM systems, as well as support for EAI and EDI services and advanced integration capabilities through BizTalk API Apps.

For API integration, Azure API Management provides a comprehensive platform for exposing and managing APIs that includes end-to-end management capabilities such as: security and protection, usage plans and quotas, policies for transforming payloads, as well as analytics, monitoring, and alerts.

Integration at the data layer can be enabled by Azure Data Factory, which provides an orchestration layer for building data pipelines for transformation and movement of data. Data Factory works across on-premises and cloud environments to read, transform, and publish data. It allows users to visualize the lineage and dependencies between data pipelines and monitor data pipeline health.

[26] https://azure.microsoft.com/en-us/services/batch/

3.10. At-rest data analytics

At-rest data analytics is performed over the collected device telemetry data, and often this data is blended with other enterprise data or secondary sources of telemetry from other systems or organizations. Analyzing and predicting device operational data and behavior, based on device telemetry correlated with ambient parameters and telemetry, is a powerful pattern.

There are a significant number of scenarios for when, why, and how to analyze data after it is at rest, and this reference architecture document does not aim to provide an in-depth explanation of these options or of at-rest data analytics. IoT scenarios and the general-purpose guidance for these capabilities directly applies to IoT solutions, but is not limited to that. Advanced analytics and big data solutions can be used in these cases.

Technology options

With HDInsight, the Azure platform provides a hosted implementation of the Apache Hadoop[27] platform, providing Apache Hive,[28] Apache Mahout,[29] MapReduce,[30] Pig,[31] and Apache Storm[32] as analysis capabilities.

Power BI enables the creation of models, KPIs, and their visualization through interactive dashboards. It provides a powerful analytics solution for monitoring the performance of processes or operations and can help to identify trends and discover valuable insights.

For data scientists acquainted with the algorithmic foundation, Azure Machine Learning provides a hosted machine learning capability. It offers ease of use with straightforward integration into solutions using a generated web service interface.

Other options include Apache Spark, which can be used to run big data jobs, but also provides modules for graph analysis and machine learning.

[27] http://hadoop.apache.org/
[28] http://hive.apache.org/
[29] http://mahout.apache.org/
[30] http://en.wikipedia.org/wiki/MapReduce
[31] http://en.wikipedia.org/wiki/Pig_(programming_tool)
[32] http://storm.incubator.apache.org/

4. Appendix

4.1. Terminology

This section provides scoped definitions for several terms that are used throughout this document.

Devices. There are several categories of devices: personal devices, special-purpose devices, or industrial equipment to name a few. Personal computers, phones, and tablets are primarily interactive information devices. From a systems perspective, these information technology devices are largely acting as proxies toward people. They are "people actuators" suggesting actions and "people sensors" collecting direct input or input related to the device use. These devices are referred to as "personal mobile devices" in the document.

Special-purpose devices, from simple temperature sensors to complex factory production lines with thousands of components inside them, are different. These devices are much more scoped in purpose, and even if they provide some level of a user interface (for interactions with people), they're largely scoped to interface with or be integrated into assets in the physical world. They measure and report environmental circumstances, turn valves, control servos, sound alarms, switch lights, and do many other tasks. They help doing work for which an information device is either too generic, too expensive, too big, or too brittle. The actual purpose for these devices will dictate their technical design as well as the amount of resources needed for their production and scheduled lifetime operation. The combination of these two key factors will define the available operational energy, physical footprint, and thus available storage, compute, and security capabilities. Special-purpose devices, especially industrial equipment devices, may also be complex systems, with multiple subcomponents or subsystems in them.

These special-purpose devices, referred to as "devices," are the primary focus for this discussion, whereas information devices (that is, personal mobile devices) are merely playing a proxy role toward human actors in the scenarios discussed in this document.

Device environment. The device environment is the immediate physical space around the device where physical access and/or "local network" peer-to-peer, digital access to the device is feasible.

Local network. A "local network" is assumed to be a network that is distinct and insulated from—but potentially bridged to—the public Internet, and includes any short-range wireless radio technology that permits peer-to-peer communication of devices. This notion of "local network" does *not* include network virtualization technology creating the illusion of such a local network and it does also *not* include public operator networks that require any two devices to communicate across public network space if they were to enter a peer-to-peer communication relationship.

Field gateway. A field gateway is a specialized appliance, or some general-purpose server computer software that acts as communication enabler and, potentially, as a device control system and device data processing hub.

The field gateway's scope includes the field gateway itself and all devices that are attached to it. As the name implies, field gateways act outside dedicated data processing facilities and are usually location bound.

They are potentially subject to physical intrusion, and might have limited operational redundancy.

A field gateway is different from a mere traffic router in that it plays an active role in managing access and information flow, meaning it is an application-addressed entity and network connection or session terminal. NAT devices or firewalls, in contrast, do not qualify as field gateways because they are not explicit connection or session terminals, but rather route (or block) connections or sessions made through them.

A field gateway has two distinct surface areas. One faces the devices that are attached to it and represents an inside of a zone, and the other faces external parties (such as a cloud gateway) and is the edge of the zone.

Cloud gateway. A cloud gateway is a system that enables remote communication from and to devices or field gateways, potentially residing at several different sites, connecting across public network space.

The cloud gateway handles both inbound and outbound communication between devices and a cloud-based backend system, or a federation of such systems.

In the context discussed here, "cloud" is meant to refer to a dedicated data processing system that is not bound to the same site as the attached devices or field gateways, and where operational measures prevent targeted physical access, but is not necessarily a "public cloud" infrastructure.

A cloud gateway may potentially be mapped into a network virtualization overlay to insulate the cloud gateway and all of its attached devices or field gateways from any other network traffic.

The cloud gateway itself is neither a device control system nor a processing or storage facility for device data; those facilities interface with the cloud gateway. The cloud gateway's scope includes the cloud gateway itself along with all field gateways and devices directly or indirectly attached to it.

A cloud gateway has two distinct surface areas. One faces the devices and field gateways that are attached to it, and the other faces backend services and potentially external parties.

Service. In the context of this document a service is defined as any software component or module that is interfacing with devices through a field gateway or cloud gateway for data collection and analysis, as well as for command and control interactions. Services are mediators. They act under their own identity toward gateways and other subsystems, store and analyze data, autonomously issue commands to devices based on data insights or schedules, and expose information and control capabilities to authorized end users.

Solution. A solution for a particular IoT scenario is a composition of system building blocks, including all user-contributed rules, extensions, and code. It includes all data storage and analysis capabilities specific to the known scope of the solution.

The solution interacts and integrates with other systems that exist as shared enterprise resources such as CRM or ERP systems or other line-of-business solutions. A CRM system used as a job ticketing system for support technicians that is specifically introduced for a predictive maintenance solution would be in the solution scope, but very often CRM systems are already in place for customer support. In these cases, the new solution will integrate with the existing support job ticketing system rather than introducing a new one.

4.2. References

To learn more about Azure IoT, visit our website.

The following Microsoft products support Azure IoT scenarios:

Azure IoT Suite

Azure IoT Hub

Azure Storage

Azure Data Lake

Azure DocumentDB

Azure SQL Database

Azure HDInsight

Azure Stream Analytics

Azure Event Hubs

Azure Web Apps

Azure Mobile Apps

Azure Logic Apps

Azure Notification Hubs

Azure Machine Learning

Azure Machine Learning Studio

Power BI

Azure Active Directory

Azure Key Vault

For more references and information supporting this document, take a look at the following:

Service assisted communication	http://blogs.msdn.com/b/clemensv/archive/2014/02/10/service-assisted-communication-for-connected-devices.aspx
TCP	http://tools.ietf.org/html/rfc793
UDP	http://tools.ietf.org/html/rfc768
AMQP	http://www.amqp.org/
AMQP Core	http://docs.oasis-open.org/amqp/core/v1.0/os/amqp-core-overview-v1.0-os.html
MQTT	http://mqtt.org/ http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html
CoAP	http://en.wikipedia.org/wiki/Constrained_Application_Protocol
OPC Foundation	https://opcfoundation.org/ http://en.wikipedia.org/wiki/OPC_Foundation
WebSockets	http://en.wikipedia.org/wiki/WebSockets

TLS	http://tools.ietf.org/html/rfc5246 http://tools.ietf.org/html/rfc4279
Azure VPN	http://azure.microsoft.com/en-us/services/virtual-network/
ExpressRoute	http://azure.microsoft.com/en-us/services/expressroute/
Azure API applications	https://azure.microsoft.com/en-us/documentation/articles/app-service-api-apps-why-best-platform/
Azure Search	https://azure.microsoft.com/en-us/documentation/articles/search-what-is-azure-search/
Bing Maps	http://msdn.microsoft.com/en-us/library/ff428643.aspx
Service Fabric	http://azure.microsoft.com/en-us/campaigns/service-fabric/
Akka	http://akka.io/
Akka.Net	http://getakka.net/
Azure Batch	https://azure.microsoft.com/en-us/services/batch/
MapReduce	http://en.wikipedia.org/wiki/MapReduce
Pig	http://en.wikipedia.org/wiki/Pig_(programming_tool)
Apache Storm	http://storm.incubator.apache.org/
Apache Spark	http://spark.apache.org/
Apache HBase	http://hbase.apache.org/
Apache Hadoop	http://hadoop.apache.org/
Apache Hive	http://hive.apache.org/
Apache Mahout	http://mahout.apache.org/
CAP Theorem	https://en.wikipedia.org/wiki/CAP_theorem
Azure Business Continuity Technical Guidance	https://msdn.microsoft.com/library/azure/hh873027.aspx
HADR for Azure applications	https://msdn.microsoft.com/library/azure/dn251004.aspx
Securing your Internet of Things from the ground up	http://download.microsoft.com/download/8/C/4/8C4DEF9B-041B-47F3-AD7F-52F391B1D0AB/Securing_your_Internet_of_Things_from_the_ground_up_white_paper_EN_US.pdf