Azure Integration Services - Basic Summary

Audience: Primary: integration architects and senior Azure engineers; Secondary: security/governance leads and delivery teams; Azure intermediate+ **Job ID:** deee03f3-ed78-4099-ae28-544afe6b71b9 **Generated:** 2026-02-04

Table of Contents

1.1 Purpose, scope, and how to use this document

1.1.1 Purpose

You should use this document as a practical, architecture-first guide to **AIS (Azure Integration Services)**: how the core components fit together, how to choose an interaction style, and how to implement securely under an enterprise baseline (private networking preferred, multi-region DR required).

AIS (Azure Integration Services) is a set of Azure services used to build integration solutions across applications, data, and systems—typically including **APIM (Azure API Management)**, **Logic Apps**, **Service Bus**, **Event Grid**, and **Data Factory (ADF)**.

1.1.1.1 Key takeaways

- You should treat AIS as a small, standardized toolbox—not a “one-off per team” menu.
- You should optimize for repeatable patterns, security controls, and operability over feature breadth.
- You should use this document primarily for decision-making and architecture trade-offs, not tutorials.

1.1.2 Scope (what’s in / out)

1.1.2.1 In scope

- API-led integration patterns using **APIM**.
- Workflow orchestration using **Logic Apps**.
- Durable enterprise messaging using **Service Bus** (queues, topics/subscriptions, DLQ).
- Event routing/notification patterns using **Event Grid**.
- Batch-oriented integration and data movement using **Data Factory (ADF)**.
- Cross-cutting enterprise requirements: identity, private networking, governance, security baseline, reliability/DR, and observability.

  Note: Event Hubs is covered when streaming or large-scale telemetry-style ingestion is part of an integration solution. It is commonly paired with AIS patterns but is often treated as **adjacent** to the AIS “core” toolbox. As a rule of thumb: use **Event Hubs** for high-throughput streaming with retention/replay, and **Service Bus** for enterprise

messaging/work distribution patterns (commands, workflows, and delivery guarantees).

1.1.2.2 Out of scope (by design)

- Step-by-step tutorials, SDK walkthroughs, or “click-by-click” configuration.
- Product marketing and exhaustive feature matrices.
- Non-Azure integration platforms and detailed B2B/EDI implementations (unless explicitly called out elsewhere).

1.1.2.3 Key takeaways

- You should not expect implementation tutorials; you should expect architecture guidance and guardrails.
- You should treat Event Hubs as a specialized streaming choice, not the default integration backbone.
- You should use Azure documentation as the source of truth to validate current SKU/region/connector capabilities and constraints.

1.1.3 Terminology note (identity and control plane)

Microsoft Entra ID (Azure AD) provides identity and access management (authentication, authorization, and workload identities such as **Managed Identity**).

Azure Resource Manager (ARM) is the Azure control plane used to create, update, and manage Azure resources (for example via the Azure portal, Bicep/ARM templates, Terraform, or CI/CD pipelines).

> **Recommendation:** On first mention you should introduce **“Microsoft Entra ID (Azure AD)”**, then use **“Entra ID”** consistently thereafter; reserve “Azure AD” only when quoting legacy UI labels.

1.1.3.1 Key takeaways

- Entra ID governs authentication/authorization; it is not the Azure control plane.
- You should prefer **Managed Identity** for workload-to-workload access (secretless authentication).
- You should standardize terminology early to avoid policy and implementation drift.

1.1.4 Assumptions and constraints (design inputs)

Decisions in later sections depend on these baseline inputs:

- **Networking:** You should prefer **Private Endpoint (Private Link)** where supported; otherwise you should document an exception with compensating controls (firewall allowlists, strong TLS posture, monitoring) and obtain security sign-off. Private access support can vary by service, SKU, region, and (for Logic Apps) connector; you should validate feasibility during architecture using Azure service/connector documentation and your landing zone network/policy constraints. (See *Hybrid connectivity and network architecture*, s09; *Security baseline*, s11.)
- **Reliability/DR:** Multi-region DR is required by default; designs should state **RTO/RPO** and failover mechanics. (See *Reliability and resilience design*, s13.)
- **Observability:** You should propagate **W3C Trace Context** (`traceparent`) for distributed tracing plus a business **correlation ID** for business-level linkage across APIs, messages, workflows, and pipelines. As a minimum, you should carry `traceparent` in HTTP headers and map it into message/application properties where possible; you should carry the correlation ID similarly (HTTP header and message/application property) and treat it as stable across the end-to-end business transaction. (See *Observability and operations*, s14.)
- **Governance:** You should assume a **Landing Zone** provides guardrails (policy, networking, identity, logging) that enforce the baseline; if your design requires exceptions, you should document them and route them through the defined governance process. (See *Governance, policy, and landing zone considerations*, s12.)

1.1.4.1 Key takeaways

- You should validate private access feasibility early because support is SKU/region/connector dependent.
- You should treat multi-region DR as a first-class design constraint, not an afterthought.
- You should standardize tracing/correlation up front to avoid “untraceable” integrations in production.

1.1.5 How to navigate (reader pathways)

You should choose a path based on your role and the decision you need to make.

1.1.5.1 Architect (selection and end-to-end design)

- Start: **Integration styles and when to use them** (s03)
- Then: **Decision matrix and quick recommendations** (s17)
- Then: **Reference architectures (end-to-end examples)** (s16)
- Deep dives: APIM (s04), Logic Apps (s05), Service Bus (s06), Event Grid (s07), ADF (s08)

1.1.5.2 Security / governance

- Start: **Security baseline for AIS** (s11)
- Then: **Governance, policy, and landing zone considerations** (s12)
- Validate: **Hybrid connectivity and network architecture** (s09) and **Identity, authentication, and authorization** (s10)

1.1.5.3 Delivery / operations

- Start: **Observability and operations** (s14)
- Then: **Reliability and resilience design** (s13)
- Then: **DevOps, IaC, and release strategies** (s15)
- Use: **Appendices: terminology, constraints, and checklists** (s18) for go-live gates

1.1.5.4 Key takeaways

- You should use s03 + s17 as the primary “how do you choose?” shortcut.
- You should use s11–s14 as the enterprise baseline backbone (security, governance, reliability, ops).
- You should use s18 checklists to operationalize decisions and prevent last-minute gaps.

1.2 AIS component landscape (what fits where)

You should treat **AIS (Azure Integration Services)** as a toolbox: choose the smallest set of services that meets your **latency**, **durability**, **governance**, and **operational** needs, then standardize patterns (naming, identity, networking, observability) across teams.

1.2.1 What this section covers

- A high-level map of **core AIS services** and **common adjacent capabilities**

- Typical interaction paths (API-led, workflow orchestration, messaging, eventing, batch)
- Enterprise constraints that materially affect topology choices (private access, DR, audit)

1.2.1.1 Key takeaways

- You should use this section as a **role + connectivity** map; use the integration-style chapter for detailed “when to use” guidance.
- AIS services are mostly **PaaS**; you typically consume them via **Private Link (Private Endpoints)** rather than “deploying them into” a VNet.
- Private access and multi-region DR are **design-time constraints**; validate SKU/region support early to avoid redesign.

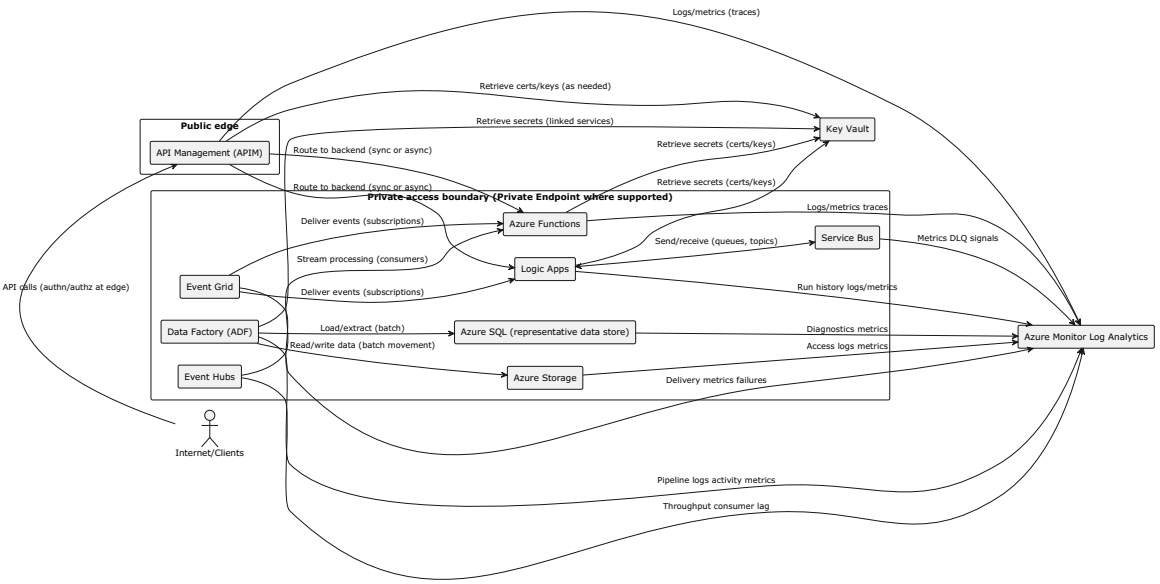
1.2.2 Design inputs (validate per SKU/region)

- **Identity**: **Microsoft Entra ID (Azure AD)** provides identity and access management; you should prefer **Managed Identity** for workload-to-workload access (secretless authentication).
- **Networking**: you should prefer **Private Endpoint (Private Link)** where supported; otherwise document exceptions and compensating controls (see Security baseline / Governance).
- **Resilience**: multi-region DR is required; you should state **RTO/RPO** and the routing/failover approach per workload.
- **Observability**: you should propagate W3C trace context (`traceparent`) plus a business **correlation ID** end-to-end.

1.2.2.1 Key takeaways

- You should validate **data-plane vs management-plane** private access for each service in your chosen SKU/region.
- You should plan DR for **network primitives** (private endpoints + private DNS) as well as application state.

1.2.3 AIS landscape overview and interactions



Note: Treat this diagram as a logical connectivity view. Many AIS components are PaaS services accessed *from* your VNets via **Private Link** (SKU/region-dependent), not deployed inside the VNet.

Recommendation: You should validate SKU/region support for **Private Link**, **customer-managed keys**, and **diagnostic settings** per service and per “hop” (publisher → broker/router → handler → datastore).

1.2.3.1 Key takeaways

- APIM is typically your **public edge**; most other AIS components are **private-accessed PaaS** behind it.
- Service Bus is the durable **messaging backbone**; Event Grid is **event notification routing**; Event Hubs is **stream ingestion/replay**.
- Monitoring and key management are cross-cutting; you should keep them centralized and consistent.

1.2.4 Mental model: what each component is “for”

Component	Primary role	Strengths	Not ideal when
APIM (Azure API Management)	API gateway + policy enforcement	Consistent authN/Z, throttling, routing, API lifecycle	You need durable async processing (use Service Bus)
Logic Apps	Workflow orchestration	Long-running workflows, connectors, human/system coordination	You need high-throughput compute or low-level streaming
Service Bus	Durable enterprise messaging	Queues, topics/subscriptions, sessions, DLQ, backpressure	You only need lightweight notifications/fan-out
Event Grid	Event routing	Push-based pub/sub, filtering, low ops overhead	You need durable work queues or replay log semantics
Data Factory (ADF)	Batch data movement + ETL/ELT orchestration	Pipelines, scheduling, integration runtimes	You need real-time stream processing
Event Hubs *(adjacent/common)*	High-throughput event ingestion + replay	Retention, consumer groups, streaming patterns	You need per-message workflows, DLQ, or strict command handling
Functions *(adjacent/common)*	Event-driven compute	Lightweight handlers, scaling, glue code	You need complex workflow state (use Logic Apps)

1.2.4.1 Key takeaways

- You should decide first whether you are moving **commands (durable)**, **events (notifications)**, **streams (replayable telemetry)**, **APIs (request/response)**, or **batch data**.
- You should use Service Bus for “do this” work items; Event Grid for “something happened” notifications; Event Hubs for “here is a stream” ingestion.

1.2.5 Adjacent platform capabilities you commonly pair with AIS

- **Key Vault** for secrets/keys/certificates, typically accessed via **Managed Identity**
- **Azure Monitor/Log Analytics** for centralized metrics/logs/traces and audit retention
- **Azure Storage** as durable state, payload storage, and occasional dead-letter/landing zones
- **Private Endpoint (Private Link)** + **private DNS** to keep data-plane access private where supported

1.2.5.1 Key takeaways

- You should treat “adjacent” services as **standard platform foundations**, not per-team choices.
- Private access is a **multi-hop concern** (publisher, broker/router, handler, datastore), not a single checkbox.

1.2.6 Section wrap-up: how to use this landscape

This section is the **map**: it helps you place components in an integration platform and understand typical interaction paths. You should use the next section (integration styles) for the **selection guide** (what to choose for a given contract and non-functional requirements).

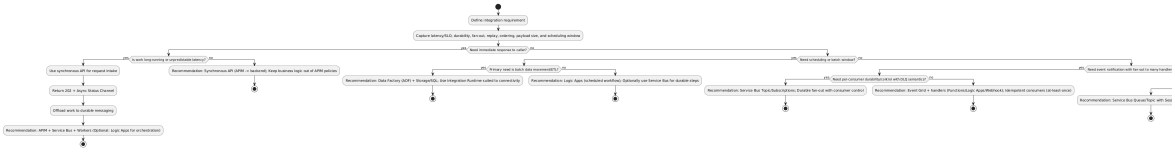
1.2.6.1 Key takeaways

- Use this landscape to standardize a **small set of repeatable patterns** and avoid bespoke point-to-point designs.
- You should treat private access + DR as **architecture constraints**, and document exceptions with compensating controls.
- You should carry a consistent **correlation ID** and W3C tracing context through every hop to keep operations viable at enterprise scale.

1.3 Integration styles and when to use them

This section is the *decision guide* for choosing an interaction style and the primary AIS (Azure Integration Services) component(s). It complements the component landscape in **s02** (what fits where) by focusing on *when* to use each style.

Recommendation: You should use the decision flow below as the default selection shortcut; the subsections that follow expand on constraints, security, and operational trade-offs.



1.3.1 Assumptions and constraints (decision-impacting)

- **Identity**: You should use **Microsoft Entra ID (Azure AD)** for authentication/authorization; prefer **Managed Identity** for service-to-service access.
- **Private networking**: You should prefer **Private Endpoint (Private Link)** *where supported*; otherwise document an exception (data classification, compensating controls, allowlists, TLS requirements) and get security sign-off.
- **Tracing**: You should propagate **W3C trace context** (`traceparent`) plus a business **correlation ID** (e.g., `Correlation-Id`) across APIs, messages, events, and batch runs.
- **DR**: You should define RTO/RPO per integration flow and validate service-specific multi-region patterns; failover often requires application behavior changes (not just “add a second region”).

1.3.1.1 Key takeaways

- Use the flow diagram first; use subsections for constraints and ops implications.
- Prefer private access where supported; exceptions must be explicit and reviewed.
- Standardize on `traceparent` + `Correlation-Id` end-to-end.

1.4 Synchronous integration (request/response APIs)

1.4.1 What it is

Synchronous integration is a request/response interaction where the caller waits for an immediate outcome (success/failure and optionally a response payload). In

AIS, this is typically **APIM (Azure API Management)** fronting a backend API or orchestrator.

1.4.2 When to use

- You should use it when the caller needs an immediate decision (authorize/validate/lookup).
- You should use it for low-latency CRUD-style operations with clear SLOs.
- You should use it at the edge to enforce consistent API governance (auth, throttling, versioning).

1.4.3 When not to use

- You should avoid it for long-running workflows (minutes+) or unpredictable latency (risking timeouts and poor user experience).
- You should avoid using APIM policies as a workflow engine (complex state, heavy transformations, multi-step business logic).

1.4.4 Key configurations

- **APIM gateway**: JWT validation, request size limits, required headers, throttling/quotas, response caching (where safe).
- **Async offload pattern**: Return `202 Accepted` with an **Async Status Channel** (status endpoint and/or event notification).
- **Correlation**: Ensure `traceparent` and `Correlation-Id` are forwarded to backends.

1.4.5 Security notes

> **Recommendation:** You should prefer **Private Endpoint (Private Link)** for backends **where supported**, and disable public network access **where feasible**. If a backend must remain public, you should document the exception and implement compensating controls.

- Enforce authentication/authorization at the boundary (APIM) and again at the backend for defense-in-depth.
- Treat payload logging as sensitive; avoid capturing bodies unless approved and redacted.

1.4.6 Ops notes

- Alert on: rising 4xx/5xx rates, latency, backend dependency failures, throttling events.
- Plan for multi-region: front door/routing + backend readiness; ensure clients tolerate retries and timeouts.

1.4.6.1 Key takeaways

- Synchronous APIs are for *immediate decisions*, not long-running work.
- APIM should stay “thin”; push orchestration to workflows/workers.
- Pair with async offload (APIM + Service Bus) to protect SLOs.

1.5 Asynchronous messaging (durable commands and work handoff)

1.5.1 What it is

Asynchronous messaging uses a durable **Queue** or **Topic/Subscription** to decouple producers from consumers. In AIS this is primarily **Service Bus** for enterprise messaging (durability, DLQ, sessions, transactions).

1.5.2 When to use

- You should use it when work must be processed reliably even if consumers are down.
- You should use it when you need backpressure control (competing consumers, queue depth).
- You should use it when you need ordering/session affinity or DLQ-driven operational control.

1.5.3 When not to use

- You should avoid it for high-throughput telemetry where replay/retention is a first-class requirement (use **Event Hubs**).
- You should avoid it for lightweight notifications where durability is not required and fan-out is the primary goal (use **Event Grid**).

1.5.4 Key configurations

- Queues vs topics: use **queues** for point-to-point; use **topics/subscriptions** for durable fan-out.
- Delivery controls: lock duration, max delivery count, retry strategy, DLQ handling.
- Ordering/affinity: sessions (where required); otherwise design consumers to be idempotent.

1.5.5 Security notes

- Use Managed Identity for producers/consumers and least-privilege RBAC.
- Prefer Private Endpoint where supported; align private DNS and firewall rules with your Landing Zone.

1.5.6 Ops notes

- Assume **at-least-once delivery**; enforce **idempotency** and define dedupe keys.
- Runbooks must cover: DLQ triage, poison message handling, replay/reprocess procedures, and backlog drain after incidents.
- DR cue: plan namespace/regional strategy and client reconnection behavior; test failover and replay.

1.5.6.1 Key takeaways

- Service Bus is the default for durable, controllable business messaging.
- DLQ and idempotency are not optional in enterprise operations.
- Use topics for durable fan-out; use Event Grid only when “notification” semantics are acceptable.

1.6 Event notification (reactive, fan-out integration)

1.6.1 What it is

Event notification publishes facts that “something happened” to interested subscribers. In AIS this is typically **Event Grid** routing events to handlers (Functions, Logic Apps, webhooks) with filtering and retries.

1.6.2 When to use

- You should use it for fan-out notifications and loosely coupled reactive integration.
- You should use it when handlers can tolerate at-least-once delivery and can be idempotent.
- You should use it to reduce point-to-point integrations with subscription-based routing.

1.6.3 When not to use

- You should avoid it as a durable command queue for required processing (use **Service Bus**).
- You should avoid it when you need stream replay/retention semantics (use **Event Hubs**).

1.6.4 Key configurations

- Event schema/versioning and filtering strategy (subject/type/attributes).
- Retry behavior and handler timeouts; design handlers to be fast and offload heavy work.
- Dead-lettering: configure dead-letter destinations **where supported by topic/subscription type and endpoint**; otherwise implement compensating capture (e.g., handler writes failed events to Storage).

  Warning: Readers often assume Event Grid dead-lettering is universal. You should validate dead-letter support for your specific Event Grid offering, subscription type, and delivery endpoint in your target region/SKU.

1.6.5 Security notes

- Private networking feasibility depends on **topic type, publish path, handler type, and region/SKU**; validate Private Endpoint support for each hop early.
- Enforce least privilege for publishers and subscribers; treat event payload classification explicitly.

1.6.6 Ops notes

- Assume **at-least-once delivery**; enforce idempotency in handlers.

- DR cue: failover is typically application-level (dual topics/regions + routing/registration strategy); duplication during failover must be handled by idempotent consumers.

1.6.6.1 Key takeaways

- Event Grid is for notifications and fan-out, not guaranteed processing.
- Private access varies materially; validate early to avoid redesign.
- Idempotent handlers are mandatory due to at-least-once delivery.

1.7 Batch and scheduled integration (data movement and ETL/ELT)

1.7.1 What it is

Batch integration runs on a schedule or in windows to move and transform data. In AIS this is typically **Data Factory (ADF)** orchestrating copy/transform activities and integration runtimes.

1.7.2 When to use

- You should use it for scheduled data movement, ETL/ELT orchestration, and bulk loads.
- You should use it when latency is minutes/hours and consistency can be eventual.
- You should use it for cross-system synchronization where throughput matters more than immediacy.

1.7.3 When not to use

- You should avoid it for low-latency request/response.
- You should avoid it as an event streaming engine (use **Event Hubs** for streams).

1.7.4 Key configurations

- Integration Runtime choice (managed vs self-hosted) based on network reachability and compliance constraints.
- Rerun strategy: make pipelines restartable and sinks idempotent (upserts, watermarking).

- Parameterize datasets/pipelines to propagate `traceparent`/`Correlation-Id` and business keys where useful for audit.

1.7.5 Security notes

- Prefer private connectivity to sources/sinks where supported; document exceptions and compensating controls.
- Secure secrets with Key Vault + Managed Identity; minimize exposure in run history.

1.7.6 Ops notes

- Treat reruns as normal operations; ensure idempotent loads and clear backfill procedures.
- DR cue: define how you re-run missed windows, where state/watermarks live, and how you validate correctness post-failover.

1.7.6.1 Key takeaways

- ADF is best for scheduled/batch movement and orchestration, not low-latency APIs.
- Idempotency matters for reruns and backfills just as much as messaging.
- DR is often “replay the batch,” so watermark/state design is critical.

1.8 Streaming integration (telemetry and replayable event streams)

1.8.1 What it is

Streaming integration ingests high-throughput event streams with retention and replay semantics. In AIS-adjacent patterns this is typically **Event Hubs** with consumer groups and checkpointed processors.

1.8.2 When to use

- You should use it for telemetry, logs, clickstreams, and high-volume event ingestion.
- You should use it when replay is a requirement (reprocess from a point in time).

1.8.3 When not to use

- You should avoid it for workflow state or per-message transactions (use **Service Bus** and/or **Logic Apps**).
- You should avoid it when you need simple notification fan-out without stream processing (use **Event Grid**).

1.8.4 Key configurations

- Partitioning strategy and consumer groups.
- Checkpointing and replay procedures; define how processors recover and reprocess safely.

1.8.5 Security notes

- Use Managed Identity where supported; constrain network paths (Private Endpoint where supported) and enforce encryption in transit.

1.8.6 Ops notes

- Assume duplicates and replays; enforce idempotency and define reprocessing runbooks.
- DR cue: plan regional pairing plus checkpoint/state replication and client reconnect behavior.

1.8.6.1 Key takeaways

- Event Hubs is the “replayable log” option; Service Bus is the “durable work” option.
- Checkpointing and replay runbooks are part of the design, not an afterthought.
- Idempotency is required due to replay and at-least-once processing patterns.

1.9 Trade-offs and selection cues (recap)

Need	Prefer	Why	Not ideal when
Immediate caller response	APIM + backend API	Lowest latency, clear contract	Work is long-running/ unpredictable
Durable processing	Service Bus	DLQ/control, sessions, durability	Pure notification/fan-out only

Need	Prefer	Why	Not ideal when
Fan-out notification	Event Grid	Simple pub/sub, filtering	You must guarantee processing
Replayable stream	Event Hubs	Retention + replay + scale	You need workflow state/transactions
Scheduled/batch movement	Data Factory (ADF)	Batch orchestration + data movement	You need low-latency interactions

1.9.1.1 Key takeaways

- Pick the style based on *latency, durability, fan-out, replay,* and *ops model*.
- Use combinations intentionally (e.g., **APIM + Service Bus**) and accept added component/operational overhead.
- Validate private networking and DR feasibility per service/SKU/region early.

1.10 API Management (APIM): product capabilities and reference usage

1.10.1 What it is

APIM (Azure API Management) is a managed API gateway plus management plane and developer portal used to publish, secure, transform, and observe APIs. In AIS (Azure Integration Services), APIM typically provides the **API edge** for API-led integration and a **governance choke point** for authentication/authorization, throttling, and consistent API behavior.

APIM includes:

- **Gateway (data plane)**: processes API requests (policies, routing, auth, throttling).
- **Management plane**: configuration, CI/CD/IaC integration, analytics, and operations.
- **Developer portal**: onboarding, documentation, self-service subscription keys (where used).

  Note: Identity terminology varies in Microsoft documentation. This guide assumes **Microsoft Entra ID (Azure AD)** as the identity provider; use **Entra ID** consistently unless quoting legacy labels.

1.10.1.1 Key takeaways

- You should treat APIM as the **API governance and security edge**, not as an orchestrator.
- You should separate **gateway/data plane concerns** from **management plane** operations and access.
- You should assume SKU/region features vary and validate early (private networking, multi-region, diagnostics).

1.10.2 When to use

Use APIM when you need one or more of the following:

- A **central API entry point** (external or internal) with consistent controls across teams.
- **Authentication/authorization** enforcement with Entra ID (OAuth2/JWT validation) and subscription constructs (products, subscriptions) where appropriate.
- **Traffic shaping**: rate limits, quotas, spikes protection, and backend protection.
- **Standardized API facade**: URL shape normalization, header enrichment (correlation ID), and lightweight transformations.
- **API lifecycle governance**: versioning, revisions, developer onboarding, and observability.

1.10.2.1 Key takeaways

- You should use APIM to **standardize API access** and protect backends.
- You should use APIM to **enforce cross-cutting policies consistently** (auth, throttling, headers, logging).
- You should pair APIM with **Service Bus** when backend work is long-running or bursty.

1.10.3 When not to use

Avoid APIM as a substitute for:

- **Long-running workflows** or complex stateful business orchestration (use Logic Apps / workers).
- **Durable messaging** (use Service Bus queues/topics with DLQ handling).

- **High-throughput streaming ingestion** and replay (use Event Hubs).
- **Batch ETL/ELT** (use Data Factory).

> **Warning:** You should not implement business workflows inside APIM policies. Policy execution is optimized for gateway concerns; complex logic increases latency, reduces debuggability, and can create brittle coupling at the edge.

1.10.3.1 Key takeaways

- You should keep APIM policies **thin and deterministic**.
- You should push business validation, orchestration, and side effects to **backends/workers**.

1.10.4 Key configurations

1.10.4.1 API surface and lifecycle

- **Products, APIs, operations**: use products to represent consumer-facing bundles and attach baseline policies at product/API scope.
- **Versioning and revisions**:
  - Versions = breaking contract changes.
  - Revisions = non-breaking iteration and safe rollout.
- **Backends**: define reusable backend entities (Logic Apps Standard/ Functions/other services) and control routing with policies.

1.10.4.2 Policy patterns (enterprise integration)

You should use policies for **edge concerns**:

- **Authentication/authorization**
  - Validate JWT tokens (issuer/audience/claims) for Entra ID-protected APIs.
  - Use managed identities from APIM to call Azure backends where supported.
- **Traffic shaping**
  - Rate limits and quotas per subscription, per IP, or per identity/claim (as applicable).
- **Request boundary checks**
  - Require standard headers (including **correlation ID**).
  - Enforce request size limits and content-type constraints.

    - Optional JSON schema validation where feasible; keep deep business validation in the backend.
- **Transformation and routing**
    - Header/body manipulation, rewrite-uri, set-backend-service.
- **Caching (selectively)**
    - Apply for read-heavy APIs with safe cache semantics and explicit TTL.
- **Observability enrichment**
    - Set/propagate `traceparent` (W3C) and a business **correlation ID** header (for example `Correlation-Id`), mapping to backend conventions.

Recommendation: Standardize on **W3C trace context** (`traceparent`) plus a business **correlation ID** (for example `Correlation-Id`). You should propagate both through APIM and into backend logs, Service Bus message properties, and workflow run metadata (see Observability and operations).

1.10.4.3 Key takeaways

- You should use policies for **auth, throttling, boundary checks, routing, and observability enrichment**.
- You should keep transformations and validation **minimal** to reduce coupling and latency.
- You should treat versioning/revisions as **release safety mechanisms** for enterprise change control.

1.10.5 Security notes

1.10.5.1 Identity and secrets

- Prefer **Managed Identity** for APIM-to-backend access where supported.
- Store certificates/keys/secrets in **Key Vault** and control access via least-privilege RBAC.
- Restrict the management plane with:
    - Entra ID role assignments and least privilege
    - Network restrictions where supported (management endpoint exposure differs by SKU and configuration)

1.10.5.2 Private networking (preferred)

Private access is feasible but depends on **APIM SKU/mode, backend type, and region**. You should validate **management-plane vs data-plane** private access requirements for your target SKU/region.

- For **private backends**, you typically:
  - Place APIM in a **shared integration/hub** network context (or equivalent).
  - Reach backends in spoke VNets via **VNet peering**.
  - Reach PaaS dependencies (Key Vault, Service Bus, Storage) via **Private Endpoint (Private Link)** where supported.
  - Control egress via **Azure Firewall/NVA** and explicit routing.

  Recommendation: You should prefer **Private Endpoint (Private Link)** for PaaS dependencies **where supported**, and disable public network access **where feasible**. If an endpoint must remain public, you should document the exception (data classification, compensating controls, firewall allowlists, TLS requirements) and get security sign-off per governance policy.

1.10.5.3 Key takeaways

- You should design APIM with a **clear separation of public edge vs private backend connectivity**.
- You should assume private networking features are **SKU/region dependent** and validate early.
- You should enforce least privilege with **Managed Identity + RBAC** and Key Vault controls.

1.10.6 Ops notes

- **Monitoring baseline**
  - Alert on gateway availability, 4xx/5xx rates, latency, throttling events, and backend failures.
- **Logging and data handling**
  - Centralize diagnostics to Log Analytics per Landing Zone standards.
  - Avoid logging sensitive payloads by default.

  Warning: Logging request/response bodies from APIM can violate data handling policies. If payload capture is required, you should implement

targeted sampling/redaction with explicit approval and retention controls.

- **Change management**
  - Use revisions for safe rollout, then promote to versions for breaking changes.
  - Automate configuration with IaC; avoid drift via policy-as-code and deployment pipelines.
- **DR and multi-region**
  - You should define RTO/RPO and explicitly design for failover of:
    - API entry (DNS/routing)
    - Gateway capacity and configuration replication model
    - Private networking dependencies (private endpoints are regional; DNS must be planned)
  - You should run DR drills that include **client reconnection behavior** and **private DNS resolution** checks.

1.10.6.1 Key takeaways

- You should operationalize APIM with **alerts on latency/errors/throttling**, not only availability.
- You should treat **private endpoints + DNS** as a DR dependency, not an implementation detail.
- You should use IaC + revisions to reduce change risk and configuration drift.

1.10.7 APIM with private backends (hub-spoke example)

1.11 Logic Apps: orchestration, connectors, and integration patterns

1.11.1 What it is

Logic Apps is a managed workflow service for orchestrating integrations using **triggers** and **actions** across Azure services and external systems via **connectors**. It is best treated as the **orchestrator**: it coordinates steps, manages workflow state, and handles long-running processes; it is not a general-purpose compute runtime.

Within **AIS (Azure Integration Services)**, Logic Apps typically sits behind **APIM (Azure API Management)** for API-led scenarios and alongside **Service Bus** / **Event Grid** for asynchronous and event-driven patterns.

> **Note:** Private networking and identity features can be **connector-, SKU-, and region-dependent**. You should validate **Private Endpoint (Private Link)** and managed identity support for every hop (trigger source, connector execution path, target system).

1.11.1.1 Key takeaways

- You should use Logic Apps as an **orchestrator** (coordination/state), not as a compute engine.
- Connectors accelerate integration but introduce **governance and networking** constraints you must validate early.
- Logic Apps pairs well with **Service Bus** for durable handoff and **APIM** for API-led front doors.

1.11.2 When to use

Use Logic Apps when you need workflow-centric integration:

- **Long-running business processes** (hours/days) with durable state and checkpoints.
- **Saga-style orchestration** with **compensation** for partial failure across systems.
- **Connector-rich integration** (SaaS/enterprise apps) where custom code would be expensive to maintain.
- **Asynchronous offload** patterns: accept a request, orchestrate, then publish commands to **Service Bus** for workers to execute side effects.
- **Human-in-the-loop or approvals** where appropriate to governance (subject to data handling policies).

1.11.2.1 Key takeaways

- You should prefer Logic Apps for **stateful orchestration** and **process coordination**.
- For durable downstream execution, you should pair Logic Apps with **Service Bus** (DLQ + replay runbooks).
- Logic Apps is a strong fit when **connectors** materially reduce delivery time.

1.11.3 When not to use

Avoid (or constrain) Logic Apps in these cases:

- **High-throughput streaming ingestion** or replay-oriented pipelines (use **Event Hubs** and stream processors).
- **Ultra-low-latency request/response** where every millisecond counts (push logic to a worker and keep orchestration minimal).
- **CPU-/memory-intensive work** or custom libraries that need tight runtime control.
- **Workflows requiring strict per-message transactions** across multiple systems (you should design for at-least-once + idempotency instead).
- **Scenarios requiring end-to-end private execution paths** when a required connector executes outside your private access boundary (connector-dependent).

  Warning: Do not put “business compute” into Logic Apps actions by default. When a step has significant side effects, you should isolate it behind a worker with explicit idempotency and operational controls.

1.11.3.1 Key takeaways

- You should not use Logic Apps as a streaming engine or heavy compute runtime.
- Connector execution location can break “private-only” assumptions; validate early.
- Favor explicit workers for side effects, and keep workflows focused on coordination.

1.11.4 Standard vs Consumption (high-level trade-offs)

Dimension	Consumption	Standard
Best fit	Spiky/low-volume workflows, simple orchestration	Enterprise workloads needing runtime control and predictable ops
Runtime/ deployment	Service-managed, per-workflow	App-managed, single workflow app hosting multiple workflows
Network control	More limited; many managed connectors execute outside your	More runtime control and enterprise alignment; still

Dimension	Consumption	Standard
	network boundary (connector-dependent)	connector/target-dependent for end-to-end private paths
Ops model	Less infrastructure to manage, more platform abstraction	More control over runtime settings; closer alignment to application ops patterns

Recommendation: You should choose **Standard** when private networking patterns, consistent deployment units, and operational control matter; choose **Consumption** when time-to-value and elastic scale are primary, and networking constraints are acceptable.

1.11.4.1 Key takeaways

- Standard generally aligns better with enterprise runtime control; Consumption optimizes for simplicity.
- Networking is not guaranteed in either model; **connectors are the critical dependency**.
- You should decide hosting model early because it shapes deployment and governance.

1.11.5 Key configurations

1.11.5.1 Connectors: built-in vs managed (governance-focused)

- **Built-in capabilities**: Prefer where they meet requirements, because they typically reduce external dependency surfaces and simplify governance.
- **Managed connectors**: Use when required for SaaS/enterprise systems, but treat them as governed dependencies:
  - Standardize allowed connectors and enforce via policy/blueprints where feasible.
  - Inventory connector usage and data classifications (inputs/outputs).

Recommendation: You should maintain an “approved connectors” list aligned to data classification and private networking feasibility.

1.11.5.2 Triggers/actions, retries, concurrency, and state

- **Retries**: Configure retries per action with bounded backoff; align budgets to downstream SLAs and avoid retry storms.

- **Concurrency**: Cap concurrency when calling fragile dependencies; use controlled fan-out where necessary.
- **Statefulness**: Use stateful workflows for long-running orchestration; ensure you understand how run history/state retention intersects with audit and data handling policies.
- **Idempotency**: Treat downstream calls and messages as at-least-once; implement idempotency at the workflow boundary and in workers (see diagram pattern below).

1.11.5.3 When to offload compute to Functions/Container Apps

You should offload when you need:

- Custom code, libraries, or tight runtime control.
- Better control over outbound networking, connection pooling, and dependency handling.
- Higher-performance execution than a workflow action provides.

  Note: The choice between Functions and Container Apps depends on runtime needs, scaling characteristics, and your private networking architecture. You should standardize one “worker” hosting model per platform team where possible to reduce operational variance.

1.11.5.4 Key takeaways

- You should govern connectors as dependencies with explicit approval and inventory.
- Configure retries/concurrency deliberately to protect downstream systems.
- Offload side-effecting compute to workers; keep Logic Apps focused on orchestration.

1.11.6 Security notes

- **Identity**: You should use **Managed Identity** for Azure resource access (secretless authentication). For non-Azure targets, prefer modern auth flows; store secrets in Key Vault when unavoidable.
- **Private networking**: You should prefer **Private Endpoint (Private Link)** and private DNS **where supported** for triggers and targets. If a connector path cannot be made private, you should document the exception and apply compensating controls (data classification, firewall allowlists, TLS requirements, monitoring).

- **Data exposure**: Workflow run history can contain payloads/headers. You should treat run history as a data store and align retention, access controls, and diagnostic settings to your enterprise baseline.
- **Least privilege**: Scope identities to the minimal set of actions (resource RBAC + connector permissions).

  Warning: Logging request/response bodies from Logic Apps can violate data handling policies. If payload capture is required, you should implement targeted sampling/redaction with explicit approval and retention controls.

1.11.6.1 Key takeaways

- You should default to Managed Identity and least privilege.
- Private networking is conditional; validate connector execution paths and document exceptions.
- Run history and diagnostics must be treated as sensitive data surfaces.

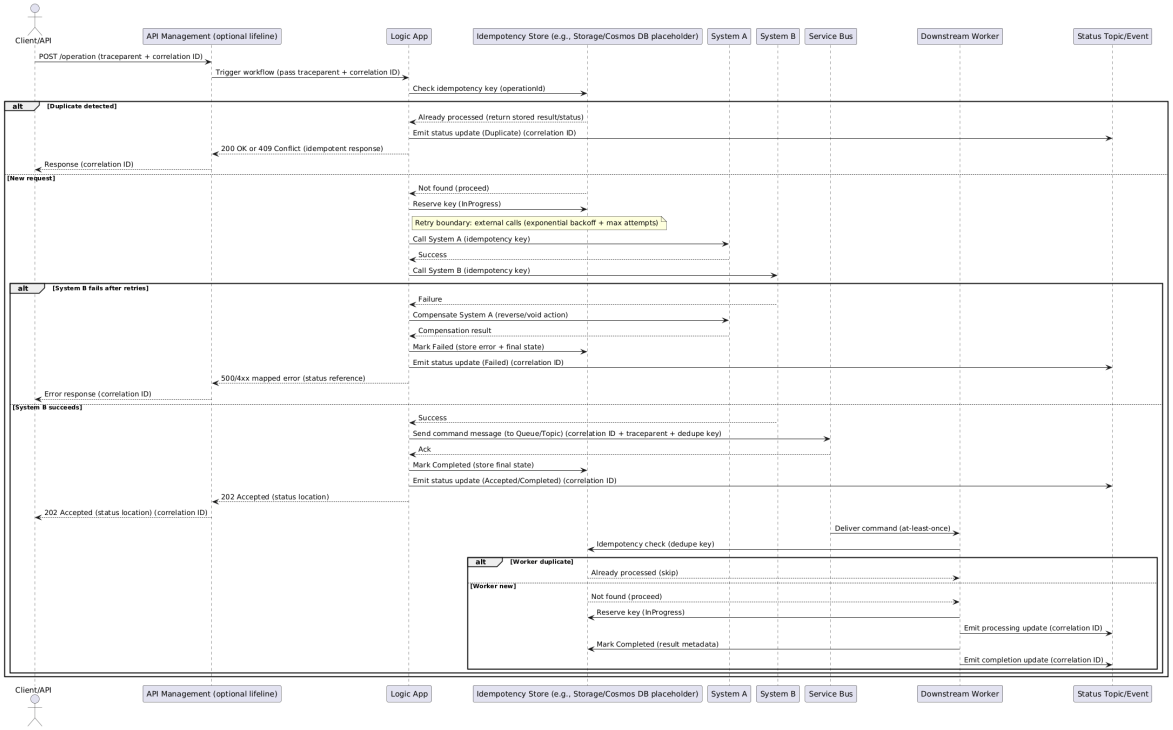
1.11.7 Ops notes

- **Observability**: You should propagate **W3C trace context** (`traceparent`) plus a business **correlation ID** through triggers/actions and into **Service Bus** message properties for end-to-end tracing.
- **Failure handling**:
  - Use explicit **retry boundaries** per dependency.
  - Use **compensation** for partial failure (saga-style) rather than assuming rollback.
  - Use **Service Bus DLQ** operationally for durable commands that fail downstream.
- **Runbooks**: Define runbooks for:
  - Manual replay from DLQ (with idempotency safeguards).
  - Workflow resubmission policies and poison-message handling.
  - DR failover behavior (including DNS/private endpoint regional dependencies).

1.11.7.1 Key takeaways

- You should standardize correlation and tracing across workflows and messages.
- Assume at-least-once delivery; enforce idempotency + replay procedures.
- Define runbooks for DLQ, compensation outcomes, and DR drills.

1.11.8 Diagram: Workflow orchestration with compensation (saga-style)



1.12 Service Bus: queues, topics, sessions, and enterprise messaging

1.12.1 What it is

Service Bus is an enterprise messaging broker for **durable business messaging**. It provides **queues** (point-to-point) and **topics/subscriptions** (pub/sub), with operational primitives such as **dead-letter queue (DLQ)**, message locks/settlement, and **at-least-once delivery**.

It is typically used inside AIS (Azure Integration Services) to decouple producers/consumers and to make “commands/work items” durable and controllable.

1.12.1.1 Key takeaways

- You should choose Service Bus when you need **durability + controlled consumption** (not just notification fan-out).
- Service Bus assumes **at-least-once delivery**; consumers must be **idempotent**.

- You should treat the **DLQ** as a first-class operational workflow (triage, replay, archive).
- Private networking is preferred via **Private Endpoint (Private Link) where supported**; plan DNS and DR accordingly.

1.12.2 When to use

Use Service Bus when one or more of the following are true:

- **Durable command/work queue**: work must not be lost even if consumers are down.
- **Backpressure + controlled concurrency**: you need competing consumers and explicit throttling via client/workers.
- **Multi-subscriber business events** where each subscriber needs its **own cursor/independent failure handling** (topics/subscriptions), often with **filters**.
- **Ordering and affinity requirements**: you need **sessions** (e.g., per customer/order key ordering).
- **Operational recoverability**: you need DLQ, max delivery count, TTL controls, and replay procedures.

1.12.2.1 Key takeaways

- Queues fit **competing consumers**; topics/subscriptions fit **independent downstreams**.
- Sessions are your primary tool for **ordering** and **per-key concurrency control**.
- Service Bus is a good fit for **async offload** (e.g., APIM → Service Bus → workers) to protect API SLOs.

1.12.3 When not to use

Avoid Service Bus when:

- You only need **event notifications** and can tolerate lightweight delivery semantics → prefer **Event Grid**.
- You need **high-throughput streaming ingestion + retention/replay semantics** → prefer **Event Hubs**.

- Your payloads are routinely large or binary blobs (Service Bus is not a file transfer mechanism); you should offload payloads to Storage and send **references**.
- You require exactly-once processing end-to-end without application idempotency (Service Bus does not remove the need for idempotent consumers).

  Warning: You should not treat Service Bus as a “database of record.” It is a messaging system; long-term retention and analytics belong elsewhere.

1.12.3.1 Key takeaways

- Service Bus is not a streaming log (Event Hubs) and not a notification router (Event Grid).
- Large payload anti-patterns are common; use **claim-check** (store payload, send pointer).
- Exactly-once outcomes require **idempotency + state management** in your application.

1.12.4 Key configurations

1.12.4.1 Entity model (queues vs topics/subscriptions)

- **Queue**: one message → one consumer (competing consumer pattern).
- **Topic/Subscription**: one message → copied to multiple subscriptions.
  - Use **subscription filters** to reduce fan-out and isolate consumer concerns.

1.12.4.2 Delivery semantics and lifecycle controls

- **At-least-once delivery**: duplicates can occur; consumers must be idempotent.
- **Lock duration**: tune to expected processing time; long processing should renew locks or hand off work.
- **Max delivery count**: controls when a message is moved to the **DLQ**.
- **TTL**: messages can expire; design for expiry handling (business meaning, alerting).
- **Duplicate detection** (where applicable): reduces accidental duplicates within a detection window; it is not a substitute for idempotency.

1.12.4.3 Ordering, correlation, and advanced features

- **Sessions**: enforce ordered processing per session key and enable session-aware scaling.
- **Transactions** (where supported by your client/SDK scenario): use to atomically send/complete within a scope; avoid overusing (adds coupling and complexity).
- **Deferral**: defer a message for later retrieval (typically for out-of-order handling or dependency waits).

  Recommendation: You should standardize a message envelope that includes business **correlation ID** and a stable **dedupe key** (and also propagate W3C `traceparent` where applicable). Treat this as part of your integration contract.

1.12.4.4 Key takeaways

- Lock/settlement + max delivery count + TTL are the operational “guardrails” that shape failure modes.
- Sessions are the cleanest way to get **per-key ordering** without building custom partitioning logic.
- Duplicate detection is helpful but not sufficient; you should still implement idempotency.

1.12.5 Security notes

1.12.5.1 Identity and access

- You should use **Microsoft Entra ID (Azure AD)** for authentication/authorization and prefer **Managed Identity** for workload access.
- Apply least privilege using RBAC scoped to:
  - Send-only identities for producers.
  - Listen/receive-only identities for consumers.
  - Separate identities per app/team where possible to reduce blast radius.

1.12.5.2 Network isolation

- You should prefer **Private Endpoint (Private Link)** for Service Bus **where supported**, and restrict/disable public access **where feasible**.
- Private endpoints are **regional**; multi-region designs must account for:
  - Duplicate private endpoints per region.

    - Private DNS zone links per VNet/region.
    - Tested DNS forwarding/failover behavior.

  Note: Private networking feasibility and controls can vary by SKU/region and by management-plane vs data-plane capabilities. You should validate early and document exceptions with compensating controls (firewall allowlists, TLS requirements, monitoring).

1.12.5.3 Key takeaways

- Managed Identity + least privilege RBAC should be your default.
- Private Endpoint is preferred but **must be validated per SKU/region**.
- DR can fail due to DNS/private endpoint coupling; you should test it explicitly.

1.12.6 Ops notes

1.12.6.1 Monitoring and alerting (minimum baseline)

You should alert on:

- **Active messages** and **queue/topic backlog growth** (capacity risk).
- **DLQ length** and **DLQ age** (operational failure signal).
- **Throttling/server busy** errors and client retry exhaustion (capacity/config risk).
- **Message processing failures** in consumers (application-level reliability).

1.12.6.2 DLQ operations (runbook-level expectations)

- Define ownership for **DLQ triage** (app team vs platform team).
- Standardize triage outcomes:
    - **Replay/reprocess** after fix (with safeguards to prevent repeat poison loops).
    - **Archive** with evidence for compliance/audit.
    - **Purge** only with explicit approval when appropriate.

  Recommendation: You should assume at-least-once delivery and enforce **idempotency + dedupe keys + replay procedures** as a production readiness requirement.

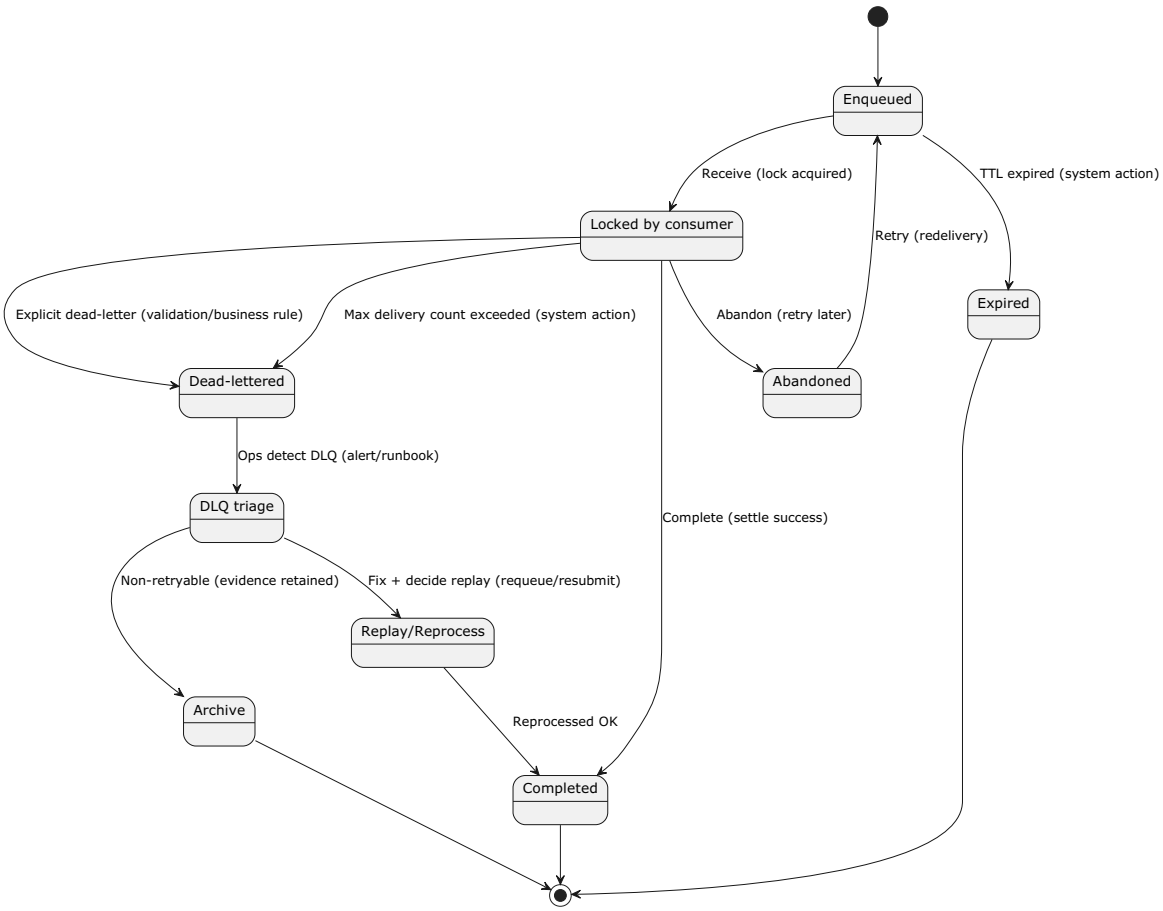
1.12.6.3 Multi-region DR considerations (service-specific cue)

- You should define how clients/workers reconnect and how producers fail over.
- You should practice failover drills that include **private networking + DNS** behavior, not just application redeploy.

1.12.6.4 Key takeaways

- DLQ handling is not optional; it is part of the operational model.
- Backlog, DLQ, and throttling are the fastest indicators of instability.
- DR readiness requires **tested runbooks** including network/DNS dependencies.

1.12.7 Service Bus message lifecycle and DLQ handling (diagram)



1.12.7.1 Key takeaways

- The lock/settlement model defines your retry behavior; configure it intentionally.
- Messages reach the DLQ via **explicit dead-letter**, **max delivery count**, or other non-success paths; you should operationalize each.
- DLQ replay must be controlled to avoid creating repeat poison-message loops.

1.13 Event Grid: eventing backbone and reactive integration

1.13.1 What it is

Event Grid is an event routing service for **publish/subscribe event notifications**. It connects **event sources** (Azure resources and custom publishers) to **event handlers** (for example, Logic Apps, Functions, and Webhooks) using **topics** and **event subscriptions** with filtering and retries.

Event Grid provides **at-least-once delivery** for notifications. It is not a durable command queue (see **Service Bus**) and not a replayable event log (see **Event Hubs**).

> **Note:** Private networking is feasible, but it **depends on topic type, publish path, handler type, and region/SKU**. You should validate **Private Endpoint/Private Link** support for each hop early to avoid redesign.

1.13.1.1 Key takeaways

- You should use Event Grid for **fan-out notifications** and reactive integration (not durable commands).
- You should design consumers for **at-least-once delivery** (idempotency required).
- You should validate **private access** per hop (source → topic, subscription → handler, dead-letter destination).

1.13.2 When to use

You should use Event Grid when you need:

- **Event-driven fan-out**: the same event should notify multiple independent consumers.
- **Lightweight integration** across services with **filtering** (by event type, subject, or advanced filters) and **routing**.
- **Near-real-time notifications** where the handler can process quickly or hand off to durable processing (often via Service Bus).

Common patterns:

- Resource lifecycle: “blob created”, “resource updated”, “policy changed”.
- Domain events: “OrderCreated”, “InvoicePosted” published to a custom topic and routed to different handlers.

1.13.2.1 Key takeaways

- You should prefer Event Grid for **notification + fan-out** with simple routing rules.
- You should pair Event Grid with **Service Bus** when downstream work must be **durable/controllable**.
- You should keep events **small**, immutable, and versioned; put large payloads in Storage and send references.

1.13.3 When not to use

You should not use Event Grid when you need:

- **Durable command processing**, backpressure control, sessions/ordering guarantees, or DLQ-centric operations → use **Service Bus**.
- **High-throughput telemetry ingestion** with retention/replay semantics → use **Event Hubs**.
- **Complex orchestration/state** within the broker → use **Logic Apps** (workflow state) plus messaging/eventing as needed.

  Warning: Event Grid delivery is at-least-once; if your handler is not idempotent, retries can cause duplicate side effects.

1.13.3.1 Key takeaways

- You should treat Event Grid as **notification routing**, not a workflow engine or queue.
- You should assume duplicates and implement **idempotency** in handlers.
- You should avoid pushing large payloads or sensitive data directly through events.

1.13.4 Key configurations

1.13.4.1 Topics and publishing model

- **System topics**: you should use these for Azure resource events (recommended default when supported).
- **Custom topics**: you should use these for your own domain events.
- **Domains** (where useful): you should consider domains when you need multi-tenant/event partitioning patterns across many event types and publishers.

1.13.4.2 Subscriptions, filters, and delivery

- You should use **one subscription per endpoint** (each subscription targets a single handler).
- You should configure **filters** to minimize noisy fan-out:
  - Event type filters (coarse)
  - Subject filters / advanced filters (fine-grained)
- You should plan for **event schema versioning**:
  - Prefer additive changes; version event types when breaking changes are unavoidable.
  - Include stable identifiers to support idempotency (for example, immutable event ID plus business keys).

1.13.4.3 Reliability behavior (high level)

- Event Grid uses **retries with backoff** over a bounded period; delivery remains **at-least-once**.
- You should configure **dead-lettering** (where supported for your subscription/endpoint type) to capture undeliverable events for later inspection/replay.

  Recommendation: You should standardize on **W3C trace context** (`traceparent`) plus a business **correlation ID** and propagate both in

event metadata to support end-to-end tracing (see Observability section).

1.13.4.4 Key takeaways

- You should model **subscription-per-handler** and filter aggressively to reduce noise.
- You should treat retries as a **normal control flow** and build idempotent handlers.
- You should implement **schema/versioning** and a dead-letter review process.

1.13.5 Security notes

Assumptions (enterprise baseline):

- **Identity**: Microsoft Entra ID (Azure AD) governs identity; you should prefer **Managed Identity** for workload-to-workload access.
- **Networking**: you should prefer **Private Endpoint (Private Link)** where supported; otherwise document exceptions and compensating controls.
- **Compliance**: you should minimize sensitive data in events; enforce encryption in transit/at rest and centralized audit logging.

Controls you should implement:

- **Publishing authorization**: restrict who can publish to custom topics; avoid shared keys when stronger identity-based controls are available.
- **Private access design dependencies**: topic type, publish path, handler type, region/SKU.
- **Dead-letter storage security**: encrypt, restrict access via RBAC, and ensure private access + private DNS where required.

  Warning: If dead-lettering is enabled, the dead-letter destination becomes part of your security boundary. You should classify and protect it like production data.

1.13.5.1 Key takeaways

- You should restrict publishers/subscribers using least privilege and minimize secrets.
- You should validate private access **per hop** and treat DNS as critical path for Private Endpoints.

- You should protect dead-letter data stores as production-grade assets.

1.13.6 Ops notes

You should operate Event Grid assuming at-least-once delivery:

- **Idempotency**: consumers should dedupe using event IDs/business keys; design replay-safe handlers.
- **Runbooks**:
  - Dead-letter inspection and replay procedure (including validation and throttling controls).
  - Handler failure triage (endpoint health, auth failures, networking/DNS, payload/schema mismatch).
- **Monitoring**:
  - Alert on delivery failures and dead-letter growth.
  - Track handler latency/error rates and correlate using `traceparent` and correlation ID.

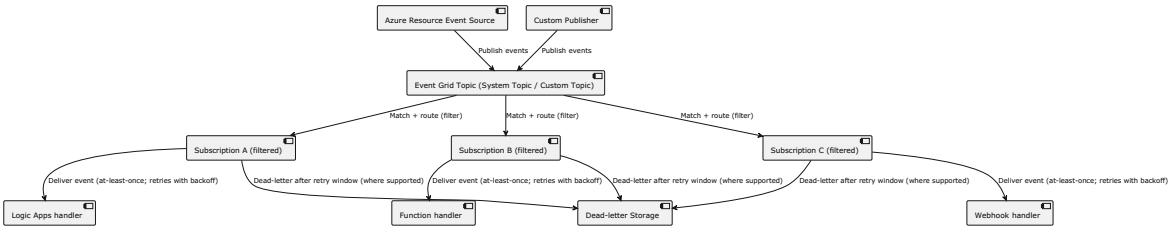
DR considerations (multi-region required):

- Event Grid itself is not a durable event log; your DR posture is typically **application-level**:
  - You should define how publishers fail over (dual publishing vs regional switch).
  - You should define how handlers fail over (active/active vs active/passive) and how duplicates are handled during failover.
  - You should ensure private endpoints and private DNS are duplicated per region (regional primitive).

1.13.6.1 Key takeaways

- You should assume at-least-once and standardize **idempotency + replay** procedures.
- You should monitor delivery health and dead-letter volume as first-class SLO signals.
- You should define DR at the **publisher/handler** level and test failover-induced duplication.

1.13.7 Diagram: Event Grid routing, filtering, and delivery



1.13.7.1 Key takeaways

- You should design for **subscription-per-endpoint** with filtering at the subscription.
- You should expect **retries** and implement **idempotent handlers**.
- You should treat **dead-lettering** as an operational and security control (and validate support per endpoint type).

1.14 Data Factory: batch integration and data movement in an integration platform

1.14.1 What it is

Data Factory (ADF) is a managed service for **batch-oriented integration**: scheduled or event-triggered pipelines that **move** and **transform** data using **pipelines**, **activities**, and **integration runtime (IR)**. In AIS (Azure Integration Services), ADF typically owns the **ETL/ELT orchestration plane** and "operational data movement" (extract → land → validate → load).

ADF is not an API gateway, not a low-latency request/response engine, and not a streaming ingestion log (that's Event Hubs). It is best treated as a **reliable batch orchestrator** with strong connectivity options and repeatable operational patterns.

> **Note:** ADF can orchestrate Databricks/Synapse jobs, but ADF itself is primarily an orchestration and data-movement service (not a streaming engine or an API gateway).

1.14.1.1 Key takeaways

- You should position ADF as your **batch orchestration + data movement** service within AIS.
- You should design for **reruns** (idempotency) and **operational control** (retries, alerts, lineage).

- You should expect **networking/IR choice** to be the primary architecture decision for enterprise deployments.

1.14.2 When to use

You should use ADF when you need one or more of the following:

- **Scheduled batch loads** (hourly/daily) with dependency management and retries.
- **Landing zone patterns**: extract from source → land immutable raw data → validate → load curated/serving stores.
- **Incremental loads** (watermarks) and **CDC-style** ingestion (conceptually: track changes and load deltas; the mechanics depend on the source).
- **Cross-system data movement** where connectivity and operationalization matter more than custom code.
- **Orchestration of external compute** (e.g., triggering a Spark job) while keeping the pipeline as the control plane.

1.14.2.1 Key takeaways

- You should use ADF for **repeatable, governed batch pipelines** (not ad-hoc scripts).
- You should use a **landing zone** to decouple extraction from transformation/loading and to support replay.
- You should treat **incremental/CDC** as an architecture pattern that drives metadata and idempotency requirements.

1.14.3 When not to use

You should avoid ADF when the primary requirement is:

- **Low-latency synchronous integration** (use APIM + backend compute; optionally async offload via Service Bus).
- **Durable commands and business messaging** with DLQ/runbooks and consumer control (use Service Bus).
- **Event notification fan-out** with lightweight handlers (use Event Grid).
- **High-throughput streaming ingestion with replay and retention** (use Event Hubs).

- **Heavy transformation at scale** where the dominant work is compute, not orchestration (use dedicated engines such as Databricks/Synapse; ADF can orchestrate them).

1.14.3.1 Key takeaways

- You should not use ADF as a **messaging backbone** or **API integration layer**.
- You should separate **orchestration** (ADF) from **heavy compute** (Spark/warehouse engines) when transformation dominates.
- You should align the service choice to the delivery semantics you need (batch vs messages vs events vs streams).

1.14.4 Key configurations

1.14.4.1 Core building blocks you should standardize

- **Pipelines**: parameterized and environment-agnostic (dev/test/prod).
- **Activities**: Copy activity for movement; Data Flow or external compute for transformations (choose based on scale/skill/ops model).
- **Triggers**:
  - Scheduled triggers for predictable batch windows.
  - Event-based triggers where supported (still batch-oriented in downstream behavior).
- **Integration runtime (IR)**:
  - **Azure IR (managed)**: simplest for cloud-to-cloud with supported private access patterns.
  - **Self-hosted IR (SHIR)**: required when you need private connectivity to on-prem/restricted networks or when the data plane must originate from your network boundary.

  Recommendation: You should choose IR placement early because it drives network design, DNS, throughput, and operational ownership (patching/availability for SHIR).

1.14.4.2 Common batch patterns you should implement

- **Landing zone** (raw/immutable):
  - Write extraction outputs to a controlled storage account/container with predictable partitioning and naming.

    - Store **metadata** (run ID, source watermark, schema version) alongside the data.
- **Incremental loads**:
    - Maintain a watermark per source entity (e.g., last modified timestamp, sequence, log position).
    - Treat watermark updates as a **transactional checkpoint** (update only after successful load).
- **Validation gates**:
    - Schema checks (where feasible), row counts, null thresholds, referential checks, and late-arriving data rules.
    - Quarantine invalid data into a controlled “rejects” area for triage.

1.14.4.3 Key takeaways

- You should standardize on **parameterized pipelines** + **IR strategy** as your main enterprise pattern.
- You should implement **landing + validation** to make reruns safe and auditable.
- You should treat **watermarks/checkpoints** as first-class state with clear failure semantics.

1.14.5 Security notes

1.14.5.1 Identity and secrets

- **Identity**: You should use **Managed Identity** for ADF to access Storage, Key Vault, and target stores where supported.
- **Secrets**: You should store connection secrets/keys in **Key Vault** and reference them from linked services; avoid embedding secrets in pipeline JSON.

  Recommendation: You should grant least-privilege data-plane roles (e.g., Storage Blob Data roles) scoped to the minimum containers/tables needed by each pipeline.

1.14.5.2 Private networking (preferred, but conditional)

- You should prefer **Private Endpoint (Private Link)** for supported data stores and key services **where supported**, and restrict public network access **where feasible**.

- When using **SHIR**, you should run it on hardened hosts, restrict outbound, and treat it as a production workload with patching and availability requirements.

  Warning: Private networking feasibility is **SKU/region/connector/target dependent**. You should validate **management-plane vs data-plane** private access for your chosen connectors early; if a hop must remain public, you should document the exception and apply compensating controls (firewall allowlists, TLS requirements, monitoring, and data classification controls).

1.14.5.3 Multi-region DR implications

- You should design the **data plane** (landing storage, target store) with replication appropriate to your RTO/RPO.
- You should expect DR to be **architecture-level**: redeploy pipelines, re-point IR/connectivity, and replay from landing/checkpoints as needed.

1.14.5.4 Key takeaways

- You should use **Managed Identity + Key Vault** as the default to avoid secret sprawl.
- You should prefer **Private Endpoint** paths where supported; otherwise you should document exceptions with compensating controls.
- You should design for DR primarily through **data replication + replayable landing zones**, not “ADF magic failover.”

1.14.6 Ops notes

1.14.6.1 Reruns, retries, and idempotency

- ADF pipelines and downstream stores often operate with **at-least-once** behavior (retries, reruns, partial failures). You should make loads **idempotent**:
  - Use deterministic file names/partitions (include pipeline run ID and source watermark).
  - Use upserts/merge semantics or staging + swap patterns for targets.
  - Ensure watermark updates occur only after successful validation and load.

Recommendation: You should treat “safe rerun” as a non-functional requirement and prove it with runbook-tested scenarios (partial extract, partial load, target outage).

1.14.6.2 Logging and correlation

- You should enable diagnostics to centralized logging and capture:
  - Pipeline run ID, activity runs, source watermark, target batch ID.
  - A business **correlation ID** propagated via pipeline parameters (and mapped to downstream message/event properties if you emit notifications).
- You should keep pipeline run history exposure in mind (avoid leaking sensitive values in parameters or logs).

1.14.6.3 Notifications and dependency signaling

- On completion/failure, you should emit a notification to downstream systems:
  - Use **Event Grid** for **non-durable fan-out notifications** (multiple subscribers; reactive processing).
  - Use **Service Bus** when downstream processing must be **durable/controllable** (competing consumers, ordering/sessions, DLQ/runbooks).

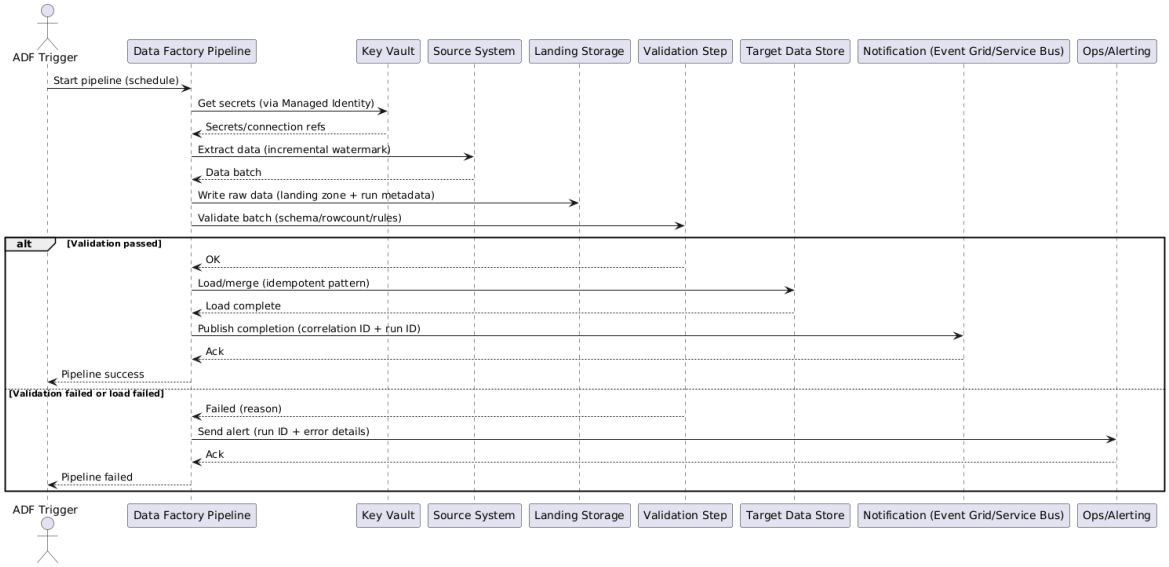
1.14.6.4 DR drills and operational readiness

- You should test:
  - Recreating IR connectivity in the secondary region (including SHIR host availability where used).
  - Replaying from landing zone with correct watermark semantics.
  - Alerting paths and on-call runbooks for common failure classes (auth failures, source throttling, schema drift, storage permission changes).

1.14.6.5 Key takeaways

- You should assume reruns and design pipelines/targets to be **idempotent**.
- You should standardize **metadata + correlation** so operators can trace end-to-end runs.
- You should choose **Event Grid vs Service Bus** for completion signals based on durability/control needs.

1.14.7 Diagram: ADF batch pipeline with landing zone and incremental load



1.14.7.1 Key takeaways

- You should model batch pipelines as **extract → land → validate → load → notify** with an explicit failure path.
- You should include **Key Vault + Managed Identity** and **ops alerting** as first-class steps.
- You should emit completion notifications with **run ID + correlation ID** to support downstream automation and troubleshooting.

1.15 Hybrid connectivity and network architecture

You should treat hybrid connectivity as a **platform capability** of your Landing Zone that AIS (Azure Integration Services) workloads consume, not re-implement per integration. The main design goal is to keep **integration traffic private by default** (Private Link via **Private Endpoint** where supported), while still supporting on-prem and partner connectivity with controlled egress, consistent DNS, and multi-region DR.

1.15.1 Assumptions and constraints (decision dependencies)

- **Landing Zone**: You deploy into an enterprise Landing Zone with hub-spoke networking, centralized firewalling, policy, and logging.
- **Identity**: Workloads use **Microsoft Entra ID (formerly Azure AD)** with **Managed Identity** wherever possible (secretless authentication).

- **Private networking preference**: You should use **Private Endpoint (Private Link)** where supported and disable public network access where feasible. If a hop cannot be private, you should document an exception (data classification, compensating controls, IP allowlists, TLS/mTLS, monitoring).
- **Hybrid**: On-prem connectivity is via **ExpressRoute** (preferred) or **site-to-site VPN**, with enterprise DNS resolution.
- **DR**: Multi-region DR is required; you should define **RTO/RPO** and validate that **network primitives** (Private Endpoints, DNS links, firewall rules, routes, NAT egress IPs) are duplicated and tested per region.

  Note: Private connectivity support varies by service, SKU, region, and sometimes by connector/endpoint type. You should validate **data-plane vs management-plane** private access early to avoid redesign.

1.15.1.1 Key takeaways

- You should design private connectivity as a **Landing Zone pattern**, not per-team bespoke networking.
- Private Endpoints are simple; **DNS is the critical path** that breaks most deployments.
- You should plan **regional duplication** of Private Endpoints and DNS links as part of DR, not as an afterthought.

1.15.2 Connectivity patterns (on-prem and partner)

1.15.2.1 What it is

Connectivity patterns define how on-premises networks and partner networks reach Azure VNets and how Azure workloads reach external systems.

1.15.2.2 When to use

- **ExpressRoute** when you need predictable latency, higher throughput, and enterprise routing control between on-prem and Azure.
- **Site-to-site VPN** when you need faster provisioning, lower cost, or as a backup path for ExpressRoute.
- **Hub-spoke** when you need shared security controls (firewall/NVA, DNS, private endpoints governance) across multiple workload spokes.

1.15.2.3 When not to use

- Don’t use ad-hoc point-to-point VPNs per workload unless you have a clear exception; it increases blast radius and operational burden.
- Don’t rely on “public endpoint + IP allowlist” as a default if Private Endpoint is available and aligns with policy.

1.15.2.4 Key configurations

- **Routing**: Define UDRs and propagation rules so spokes route through the hub firewall/NVA when required (egress inspection, partner routing).
- **Bandwidth and resiliency**: Use dual circuits/peering locations for ExpressRoute where your RTO requires it; ensure VPN failover is tested if used as backup.
- **Name resolution**: Align DNS (see DNS section) before enabling Private Endpoints broadly.

1.15.2.5 Security notes

- You should centralize ingress/egress inspection in the hub (firewall/NVA) for partner and Internet-bound flows.
- You should minimize exposed surfaces: prefer private endpoints and private routing over public exposure.
- For partner connectivity, you should distinguish:
    - **Outbound to partners** (your workloads calling partner endpoints): stable egress IPs via **NAT** + firewall egress rules.
    - **Inbound from partners** (partners calling your APIs): gateway/WAF placement + strong authentication (OAuth2/OIDC and/or mTLS) and IP allowlists where required.

1.15.2.6 Ops notes

- Monitor ExpressRoute/VPN tunnel health, route changes, and firewall denies as first-class SLO dependencies for integration.
- Run DR drills that include **network failover**, not just application failover.

1.15.2.7 Key takeaways

- ExpressRoute is typically the enterprise default; VPN is common for bootstrap/backup.

- Hub-spoke simplifies consistent controls but requires disciplined routing/DNS design.
- Hybrid availability is part of your end-to-end RTO/RPO.

1.15.3 Private access options across AIS components (and constraints)

1.15.3.1 What it is

Private access uses **Private Endpoint (Private Link)** and/or **VNet Integration** to keep traffic on private IP space and reduce reliance on public endpoints.

1.15.3.2 When to use

- Use **Private Endpoint** for PaaS services that support it (e.g., Service Bus, Key Vault, Storage) so workloads reach service data planes privately.
- Use **VNet Integration** when a compute/workflow runtime must reach private resources (e.g., private APIs, on-prem via ER/VPN) and the service supports it.

1.15.3.3 When not to use

- Don’t assume “private by default” is possible for every connector/hop. Some managed connectors execute outside your network boundary; you should validate connector execution and network path.
- Don’t mix private endpoints and public endpoints casually; inconsistent DNS often causes traffic to “leak” to public resolution.

1.15.3.4 Key configurations (enterprise pattern)

- **Private Endpoint**:
  - Create private endpoints per target service and per region as required.
  - Disable public network access **where supported** after validation/testing.
- **VNet Integration** (service-dependent):
  - Ensure outbound routes/DNS are consistent with hub policies (forced tunneling vs direct).
- **Egress control**:
  - Use NAT and firewall egress policies to provide stable outbound IPs and constrain destinations (partners commonly require allowlists).

1.15.3.5 Private access validation checklist (do this early)

- **From where**: test name resolution and connectivity from (1) each spoke subnet that will run workloads, and (2) on-prem networks that must reach the same endpoints.
- **DNS**: confirm the service FQDN resolves to **private IPs** when Private Endpoint is expected.
- **Data plane vs management plane**:
  - Verify the **runtime path** to the service data plane is private (e.g., messaging, secrets retrieval, storage IO).
  - Verify what still requires **public** access (often management/configuration endpoints, build agents, and some connector infrastructure).
- **Controls**: verify public access is disabled where intended, and that firewall/NSG/route policies do not force unintended egress paths.

1.15.3.6 Security notes

- Treat Private Endpoint as a **data-plane isolation control**, not a complete security model. You still need authentication/authorization (Entra ID, SAS where applicable, RBAC).
- You should enforce least privilege (RBAC) and avoid shared “integration super-identities.”

1.15.3.7 Ops notes

- Private endpoint deployments create dependencies on DNS and network teams; you should standardize automation (IaC) for:
  - Private Endpoint creation
  - Private DNS zone records and links
  - Firewall rules and route tables
- You should alert on “public resolution” indicators (see DNS section checks) because it often precedes outages or policy violations.

1.15.3.8 Key takeaways

- Private Endpoint + correct DNS is the baseline; VNet Integration is service/runtime-specific.
- Egress control is as important as ingress for partner integrations.
- You should treat private connectivity as a **multi-team operational dependency** (network + platform + app).

1.15.4 DNS and name resolution for Private Endpoints (enterprise pattern)

1.15.4.1 What it is

Private Endpoints rely on DNS so that a **public service FQDN** (e.g., `*.servicebus.windows.net`) resolves to a **private IP** in your VNet. In enterprises, this requires coordinated DNS across Azure and on-prem.

1.15.4.2 When to use

- Always, when you use Private Endpoints for any AIS-adjacent dependency (Service Bus, Key Vault, Storage, etc.).
- When on-prem clients must access Azure PaaS via Private Endpoint across ExpressRoute/VPN.

1.15.4.3 When not to use

- Don't rely on ad-hoc host file overrides or per-workload DNS exceptions; it breaks DR and is hard to audit.

1.15.4.4 Key configurations

- **Recommended enterprise model (typical)**:
  - Centralize **Private DNS zones** (e.g., `privatelink.*`) in a shared platform subscription/resource group.
  - Link zones to VNets deliberately (hub and/or spokes per design), using RBAC/policy to prevent ungoverned links.
  - Provide an **Azure DNS resolver/forwarder** in the hub, and forward relevant zones from on-prem.
- **DNS forwarding** from on-prem to an Azure DNS forwarder/resolver that can resolve private zones.
- **Split-horizon DNS**: ensure private resolution happens for internal clients while external clients (if any) resolve public endpoints intentionally.

  Warning: DNS misconfiguration is a common "silent failure mode" for Private Endpoints (workloads appear healthy but route to public endpoints or fail intermittently). You should validate DNS as part of every deployment and DR drill.

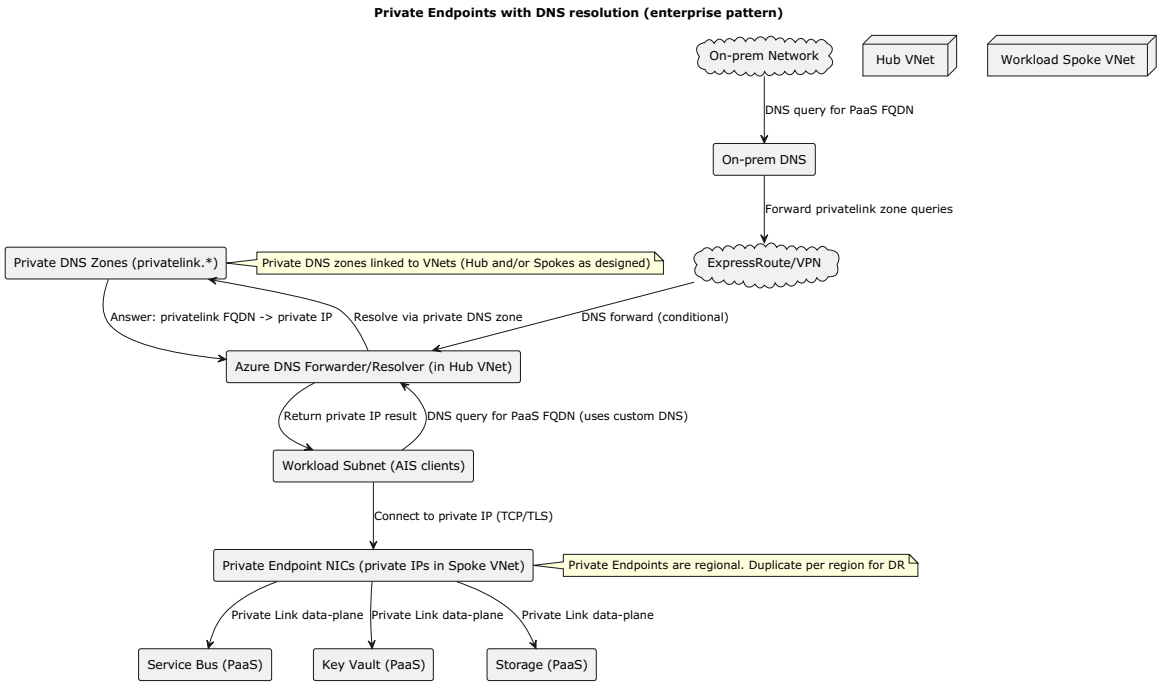
1.15.4.5 Security notes

- You should control who can create private DNS zone links and private endpoints (policy + RBAC) to prevent shadow connectivity paths.
- Treat DNS logs and resolver metrics as security telemetry (unexpected query sources can indicate misrouting or compromise).

1.15.4.6 Ops notes

- You should operationalize DNS validation:
  - Verify FQDN resolves to private IP from each spoke and from on-prem.
  - Verify traffic reaches private endpoint and not public IPs (packet capture/flow logs where appropriate).
  - Document conditional forwarders and replicate them for DR.
- For DR, you should document:
  - resolver/forwarder failover (active/active vs active/passive),
  - DNS TTL expectations (keep low where feasible),
  - the exact validation steps required to declare “private resolution is working.”

1.15.4.7 Diagram: Private Endpoints with DNS resolution (enterprise pattern)



1.15.4.8 Key takeaways

- Private Endpoint success depends on **DNS resolution** more than networking.
- On-prem must forward private zones to Azure; otherwise on-prem clients resolve public endpoints.
- You should design DNS and Private Endpoint duplication per region to make DR workable.

1.15.5 Segmentation, firewalling, and egress patterns

1.15.5.1 What it is

Segmentation defines where integration components live (spokes, shared integration subnets) and how traffic is inspected and constrained.

1.15.5.2 When to use

- Use **hub-spoke** with centralized firewalling when you have multiple teams and need consistent egress controls and partner governance.
- Use a **shared integration subnet** (in a dedicated spoke) for common integration runtimes/agents when you must centrally manage network paths (subject to team ownership and blast radius considerations).

1.15.5.3 When not to use

- Don’t put unrelated workloads into the same subnet just to “share private endpoints”; it creates lateral movement risk and operational coupling.

1.15.5.4 Key configurations

- **NSGs** for micro-segmentation (least privilege between subnets).
- **Firewall/NVA policies** for:
    - Partner allowlists
    - FQDN filtering (where supported)
    - TLS inspection policy decisions (only where approved/necessary)
- **NAT** for stable outbound IPs to partners.

1.15.5.5 Security notes

- You should assume partner integrations require:
  - IP allowlists (outbound NAT)
  - Certificate management (mTLS where required)
  - Tight protocol controls (no broad “any-any” rules)

1.15.5.6 Ops notes

- You should baseline and alert on:
  - Firewall denies for critical integration flows
  - SNAT port exhaustion (NAT)
  - Route changes affecting spokes (BGP/UDR drift)

1.15.5.7 Key takeaways

- Segmentation reduces blast radius; central egress simplifies partner governance.
- NAT is often mandatory for partners; plan SNAT capacity and DR.
- Network policy drift is a reliability risk—treat it as a governed artifact.

1.15.6 Multi-region DR implications for private networking

1.15.6.1 What it is

In DR, you must fail over **applications and their network dependencies**. For private networking, that includes private endpoints, private DNS zone links, and DNS forwarding paths.

1.15.6.2 When to use

- Always, because Private Endpoints are **regional** and must be recreated/managed per region.

1.15.6.3 When not to use

- Don’t assume “paired region” automatically handles private networking. You must design and test it.

1.15.6.4 Key configurations

- Duplicate per region:
  - Private Endpoints (for each PaaS dependency)
  - Private DNS zone links to each regional VNet
  - DNS forwarding/resolver infrastructure and conditional forwarders
  - Firewall rules/route tables and NAT public IPs (if partners require allowlists)

  Recommendation: You should include “DNS + Private Endpoint validation” as an explicit DR runbook step (resolve FQDNs, verify private IPs, verify connectivity) before declaring failover complete.

1.15.6.5 Security notes

- DR environments must enforce the same controls (no “temporary public enablement” without approval and audit trail).
- You should validate key/cert rotation and partner trust anchors in DR.

1.15.6.6 Ops notes

- You should test DR with realistic name resolution paths (from spoke workloads and from on-prem).
- Record DR evidence: resolved IPs, flow verification, and application-level smoke tests.

1.15.6.7 Key takeaways

- Private Endpoints are regional; DR requires **regional duplication + tested DNS behavior**.
- DNS forwarding and private DNS zone links are common DR breakpoints.
- You should treat network DR as part of your RTO/RPO commitment, not separate from it.

1.16 Identity, authentication, and authorization

1.16.1 Assumptions and constraints (decision drivers)

- **Identity provider**: **Microsoft Entra ID (formerly Azure AD)** provides identity and access management. Use **Entra ID** consistently for user and workload identity; Azure Resource Manager is the control plane.

- **Enterprise baseline**: you should meet an enterprise security baseline, prefer **private networking**, and plan **multi-region DR** (define **RTO/RPO**, routing, and runbooks).
- **Network reality**: many AIS components are PaaS; you should treat "private" as **private access** (e.g., **Private Endpoint (Private Link)**) rather than "deployed inside a VNet." Capability is **SKU/region/connector-dependent**.
- **Observability**: you should standardize on **W3C trace context** (`traceparent`) plus a business **correlation ID** (e.g., `Correlation-Id`) and propagate both through APIs, messages, and events.

1.16.1.1 Key takeaways

- You should default to **Entra ID + Managed Identity** for workload authentication.
- You should treat **Private Endpoint availability** as a first-order design dependency (validate early).
- You should standardize a **correlation ID** strategy alongside identity (security and ops are coupled).

1.16.2 What it is

Identity, authentication, and authorization in **AIS (Azure Integration Services)** spans:

- **Inbound authentication** for APIs/events (who is calling you).
- **Workload identity** for service-to-service access (how your workflows/compute access downstream resources).
- **Authorization controls** split between:
  - **Azure RBAC** (primarily management plane; and data plane where the target service supports Entra ID authorization),
  - **resource-specific access policies** (e.g., Service Bus authorization rules / SAS).

1.16.2.1 Key takeaways

- Authentication and authorization are separate concerns; you should design both explicitly.
- Use Entra ID where possible; treat shared keys/SAS as exceptions with compensating controls.
- Prefer "identity-based" access over "secret-based" access.

1.16.3 When to use

You should apply Entra ID-based patterns when you need:

- **Enterprise SSO / centralized access control** for API consumers (internal and partner where feasible).
- **Least privilege** and **auditable access** for workloads (Managed Identity + RBAC).
- **Secretless access** to Azure resources (Key Vault, Storage, SQL, Service Bus where supported).
- **Consistent rotation and governance** through policy (Landing Zone guardrails).

1.16.3.1 Key takeaways

- Entra ID is the default for humans and machine clients when you can enforce it.
- Managed Identity is the default for Azure-hosted workloads accessing Azure resources.
- Centralized governance is easier when you minimize shared keys and per-team secrets.

1.16.4 When not to use

You should avoid (or tightly constrain) identity patterns when they create unnecessary risk or operational burden:

- **Shared keys / shared access signatures (SAS)** as a primary strategy when Entra ID is available.
- **Embedding secrets in workflows/pipelines** instead of retrieving from Key Vault at runtime.
- **Over-scoping permissions** (e.g., subscription-wide roles) for convenience.
- **Relying on public endpoints** for identity-dependent flows when private access is required and supported.

  Warning: If you must use SAS/shared keys (partner constraints, legacy clients), you should treat it as an **exception** with compensating controls (short TTL, rotation, IP restrictions where possible, monitoring, and clear owner/runbook).

1.16.4.1 Key takeaways

- Use shared keys/SAS only when Entra ID is not feasible; document exceptions.
- Least privilege is a design requirement, not a post-deployment hardening task.
- Public access should be an explicit exception with controls and approval.

1.16.5 Key configurations (enterprise patterns)

1.16.5.1 Identity types and where they fit

Identity type	Typical use in AIS	Trade-off
Managed Identity	Workload → Azure resource access	Best security posture; Azure-hosted only
Service principal (client secret/cert)	Non-Azure workloads; some CI/CD	Secret lifecycle overhead; higher leakage risk
Federated workload identity (where applicable)	External CI/CD, Kubernetes to Entra ID	Requires setup discipline; reduces secret sprawl

> **Recommendation:** You should use **Managed Identity** for APIM-to-backend calls (where supported by your APIM/backends) and for Logic Apps/Functions/ADF access to Key Vault and downstream services, and reserve service principals for external workloads that cannot use Managed Identity.

1.16.5.2 Authentication patterns (inbound)

- **OAuth2/OIDC (Entra ID)**: preferred for client → APIM (user-delegated or client credentials).
- **Mutual TLS (mTLS)**: optional for high-assurance partner access (often in addition to OAuth2).
- **Signed requests / shared keys**: avoid where possible; if required, minimize scope and TTL.

1.16.5.3 Authorization patterns (inbound)

- **APIM policy enforcement**: validate tokens/claims, apply quotas/rate limits, and enforce request constraints.

- **Backend authorization**: re-check authorization in the backend (defense in depth), especially for high-value operations.

1.16.5.4 Authorization patterns (outbound)

- **Azure RBAC / Entra ID**: preferred when the service supports Entra ID-based data-plane authorization.
- **Resource access policies**: use when the service enforces them (e.g., Service Bus SAS rules); constrain scope to entity-level where possible.
- **Key Vault access**: use RBAC or access policies per org standard; ensure break-glass procedures exist.

1.16.5.5 Correlation standard (make it enforceable)

- At ingress (typically APIM), you should:
  - **Preserve** incoming `traceparent` and `Correlation-Id` if present and valid.
  - **Generate** them if absent (at minimum generate a `Correlation-Id`; prefer generating both where supported).
- You should map correlation to messaging consistently (e.g., message/application properties) and include it in logs.

1.16.5.6 Key takeaways

- You should standardize on a small set of patterns: OAuth2/OIDC, mTLS (optional), Managed Identity.
- Prefer RBAC over access keys; treat policy-based access (SAS) as a bounded exception.
- Configure identity with DR in mind (region pairing, failover identities/permissions).

1.16.6 Security notes

- **Least privilege**: you should grant identities the minimum roles required (resource group/resource scope; entity scope for messaging when available).
- **Key management**:
  - you should store secrets/keys/certs in **Key Vault**,
  - enforce rotation and monitor retrieval anomalies.

- **Private access**:
  - you should use **Private Endpoint (Private Link)** where supported and disable public access where feasible,
  - validate **management-plane vs data-plane** private access for your target SKU/region.
- **Token handling**:
  - you should validate JWTs at APIM (issuer, audience, signature, expiry),
  - avoid forwarding bearer tokens to backends unless required (consider token exchange/on-behalf-of patterns where applicable).

Warning: Logging tokens, secrets, or sensitive headers can violate data handling policies. You should redact sensitive fields and restrict diagnostic verbosity by environment.

1.16.6.1 Key takeaways

- You should enforce least privilege and secretless patterns by default.
- Private access and identity must be designed together (DNS and endpoint strategy affects auth paths).
- You should treat diagnostics as sensitive data and apply redaction/retention controls.

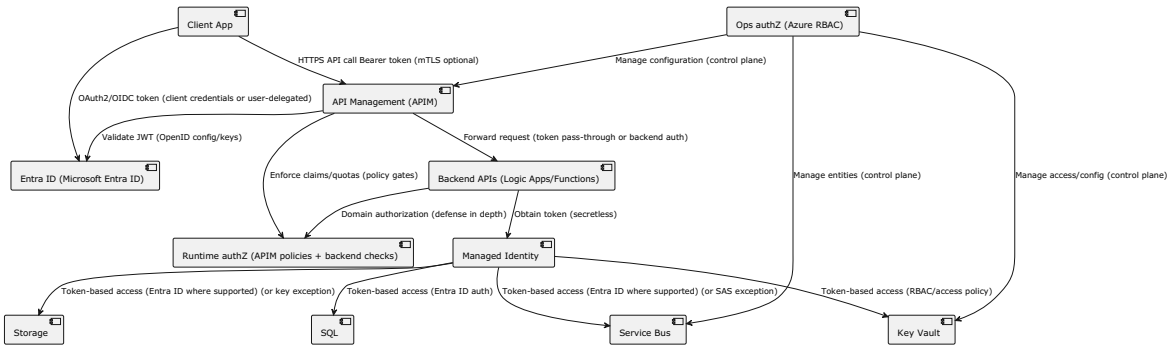
1.16.7 Ops notes

- **Rotation and expiry**:
  - Managed Identity reduces secret rotation, but you still operate **cert rotation** (mTLS) and any unavoidable SAS/key material.
- **Access review**:
  - you should implement periodic role/access reviews for service principals and managed identities.
- **Incident response**:
  - you should have runbooks for credential compromise (revoke tokens, rotate keys, disable identities, audit access).
- **Correlation and tracing**:
  - you should propagate `traceparent` and a business `Correlation-Id` through APIM, workflows, and messaging metadata to support forensic analysis.

1.16.7.1 Key takeaways

- You should run identity as an operational discipline: review, rotate, and rehearse response.
- Correlation standards materially improve security investigations and DR replay safety.
- You should monitor for abnormal auth patterns (failed logins, token validation failures, unusual Key Vault access).

1.16.8 Diagram: Workload identity and access paths in AIS



1.17 Security baseline for AIS

1.17.1 Assumptions and constraints (decision drivers)

- **Identity**: **Microsoft Entra ID (formerly Azure AD)** provides identity and access management; **Managed Identity** is preferred for workload-to-workload access (secretless authentication).
- **Networking**: Private networking is preferred. You should use **Private Endpoint (Private Link) where supported**, and disable public network access **where feasible**. If an endpoint must remain public, you should document the exception (data classification, compensating controls, firewall allowlists, TLS requirements).
- **Operations**: Centralized diagnostics to Azure Monitor/Log Analytics and end-to-end tracing are required.
- **Resilience**: Multi-region DR is required; designs must state **RTO/RPO** and a tested failover/recovery procedure (service-by-service mechanics differ).

  Note: Capabilities such as Private Link, customer-managed keys (CMK), and “disable public access” vary by service, SKU, and region

(and sometimes by connector). You should validate **management-plane vs data-plane** private access early.

1.17.1.1 Key takeaways

- You should treat **Private Endpoint support as a per-hop dependency** (service/SKU/region/connector), not a blanket capability.
- You should default to **Managed Identity + least privilege** and treat secrets as exceptions.
- You should treat **multi-region DR** as an explicit design artifact (not an aspiration), including DNS and private endpoint implications.

1.17.2 Baseline checklist (cross-cutting controls)

1.17.2.1 Network isolation and traffic boundaries

- **Ingress placement**
  - Put **APIM** at the gateway boundary appropriate to exposure (internal vs external); keep business logic behind it.
  - Use WAF/DDoS controls at the edge **where applicable** (selection depends on exposure model and org standard).
- **Private connectivity**
  - Use **Private Endpoint (Private Link)** for AIS-adjacent data planes where supported (Service Bus, Key Vault, Storage, etc.).
  - For **APIM**, private networking approach is **SKU/mode-dependent** (e.g., VNet injection for gateway placement; Private Link availability differs for gateway/management endpoints). You should validate your target SKU/region and exposure model.
  - Ensure **private DNS zones** and VNet links are deployed and included in DR planning.
- **Egress control**
  - Route outbound traffic through centralized egress controls (firewall/NVA/NAT) per Landing Zone patterns.
  - Restrict outbound destinations for workflows/workers to required FQDNs/endpoints only.

Warning: Private endpoints are **regional**. DR requires duplicate private endpoints per region, private DNS zone links per VNet/region, and tested DNS forwarding/failover behavior.

1.17.2.2 Key takeaways

- You should design for a **private access boundary** (via Private Endpoints) rather than assuming services are “in the VNet.”
- You should treat **DNS as a critical path** for Private Link and DR.
- You should enforce **egress restrictions**; “private ingress” alone is not a baseline.

1.17.2.3 Identity, authentication, and authorization

- **Authentication**
  - Use Entra ID for user/app authentication; prefer OAuth2/OIDC flows aligned to client type.
  - For service-to-service, prefer **Managed Identity** over client secrets/certificates where supported.
- **Authorization**
  - Apply least privilege using RBAC/resource policies; scope narrowly (resource group/resource) and separate duties (operators vs developers).
  - For APIs, enforce authorization at the gateway **and** in the backend (defense in depth).
- **Token and header propagation**
  - Standardize on **W3C trace context** (`traceparent`) plus a business **correlation ID** header/property (e.g., `Correlation-Id`).
  - You should **generate at ingress if missing**, preserve if present, and propagate both through APIM headers, workflow metadata, and message/event properties.

Recommendation: You should centralize identity policy (Conditional Access, app registrations, managed identity lifecycle) as Landing Zone governance to prevent per-team drift.

1.17.2.4 Key takeaways

- You should default to **Managed Identity-first** and use secrets only with documented justification.
- You should implement **authorization twice** (edge + backend) for high-value APIs.
- You should standardize **traceparent + correlation ID** end-to-end to make audit/forensics viable.

1.17.2.5 Secrets, keys, and certificates

- **Key Vault as the system of record**
  - Store secrets/keys/certs in **Key Vault**; restrict access via private networking where supported.
  - Prefer **certificate-based** auth over shared secrets when MI is not available (trade-off: lifecycle complexity).
- **Lifecycle controls**
  - Automate rotation and expiration alerting; treat “manual rotation” as a production risk.
  - Separate encryption keys from data stores where required by compliance (CMK), validating per-service support.
- **Policy**
  - Deny secret retrieval by human principals by default; use break-glass only with approval and audit.

1.17.2.6 Key takeaways

- You should treat **certificate lifecycle** (rotation, expiry, revocation) as an operational dependency.
- You should isolate Key Vault access with **least privilege + private connectivity** where supported.
- You should validate **CMK support per service/SKU/region** before committing to it as a requirement.

1.17.2.7 Data protection and privacy

- **Encryption**
  - Encrypt in transit (TLS) for all hops; terminate TLS only at approved boundaries.
  - Encryption at rest is default; use CMK where mandated and supported.
- **Payload minimization**
  - Avoid sending sensitive data through events/queues unless required; prefer references (IDs) and fetch securely.
- **PII handling**
  - Classify data and apply handling rules (masking/redaction, retention, access controls).
  - Avoid storing PII in workflow run history, message properties, or diagnostic logs.

Warning: Logging request/response bodies from APIM/Logic Apps can violate data handling policies. If payload capture is required, you should use targeted sampling and redaction with explicit approval and retention controls.

1.17.2.8 Key takeaways

- You should design integrations to be **data-minimizing by default** (IDs over full payloads).
- You should prevent PII leakage via **run history, message metadata, and logs**.
- You should treat “turn on verbose logging” as a **security event**, not a default.

1.17.2.9 Secure defaults and configuration hygiene

- Disable public network access **where feasible**; otherwise restrict by firewall allowlists and strong auth.
- Enforce TLS versions/ciphers per baseline; prefer mTLS only when justified (adds operational burden).
- Apply resource locks, tagging, and policy enforcement per Landing Zone standards.
- Use separate environments/subscriptions for dev/test/prod with controlled promotion.

1.17.2.10 Key takeaways

- You should treat “public endpoint + no allowlist” as an exception requiring compensating controls.
- You should enforce baseline via **policy** (not wiki guidance).
- You should keep environments isolated to reduce blast radius and privilege creep.

1.17.2.11 Logging, alerting, and auditability (security-relevant)

- **Diagnostics**
  - Enable diagnostics for AIS services to centralized Log Analytics (and SIEM if required).
  - Capture authentication/authorization decisions, admin actions, and dependency failures.

- **Correlation**
  - Require `traceparent` + correlation ID in ingress requests; propagate to messages/events and logs.
- **Alerting**
  - Alert on suspicious auth patterns, sudden 4xx/5xx spikes, DLQ growth, and unexpected egress destinations.

1.17.2.12 Key takeaways

- You should treat **audit logging as a control**, not merely an ops convenience.
- You should make **correlation mandatory** to support incident response.
- You should alert on **security signals** (auth anomalies, DLQ surges, egress drift), not only availability.

1.17.3 Service-specific security guidance (minimum deltas)

Recommendation: Use this section as the **baseline delta** per service; deeper product guidance belongs in the service chapters.

Note: Many messaging/eventing systems are **at-least-once**. You should enforce **idempotency** (processing the same message/event more than once does not change the outcome beyond the first success) and maintain DLQ/replay runbooks.

1.17.3.1 APIM (Azure API Management)

- **Key configurations**
  - JWT validation, required headers, size limits, quotas/rate limits; optional JSON schema validation **where feasible**.
  - Offload long-running work via Service Bus; return **202 + async status channel**.
- **Security notes**
  - Prefer private backends; restrict inbound exposure model; validate private networking approach (VNet injection and/or Private Link capabilities) by SKU/region.
  - Avoid embedding secrets in policies; use managed identity/Key Vault references where supported.
- **Ops notes**
  - Monitor auth failures, policy errors, backend latency, and throttling events.

1.17.3.2 Logic Apps

- **Key configurations**
  - Connector governance (approved connectors), retry policies, run history retention controls.
- **Security notes**
  - Validate connector network path: some managed connectors may execute outside your private boundary (connector/plan-dependent).
  - Mitigations (capability-dependent): prefer built-in connectors where possible; use VNet integration options where supported; for on-prem access use approved gateway patterns; store unavoidable secrets in Key Vault with rotation.
  - Use managed identity for supported connectors; otherwise store secrets in Key Vault with rotation.
- **Ops notes**
  - Treat run history as potentially sensitive; restrict access and monitor for failures/timeouts.

1.17.3.3 Service Bus

- **Key configurations**
  - Use queues/topics appropriately; configure DLQ, max delivery count, lock duration, TTL; use sessions when required for ordering.
- **Security notes**
  - Prefer Private Endpoint; disable public access where feasible; restrict SAS usage (prefer Entra ID auth where supported).
- **Ops notes**
  - Assume at-least-once delivery; enforce idempotency and DLQ replay runbooks.

1.17.3.4 Event Grid

- **Key configurations**
  - Filters, retry settings, dead-lettering **where supported** (depends on subscription type/endpoint).
- **Security notes**
  - Private networking feasibility depends on topic type, publish path, handler type, and region/SKU; validate each hop.
- **Ops notes**
  - Assume at-least-once delivery; handlers must be idempotent; events are notifications, not durable command queues.

1.17.3.5 Data Factory (ADF)

- **Key configurations**
  - Use Managed Identity for sources/sinks where supported; control integration runtime placement for private connectivity.
- **Security notes**
  - Treat linked services/datasets as credential-bearing; enforce Key Vault references and restricted authoring rights.
  - Validate runtime network boundary (managed vs self-hosted integration runtime) and ensure the chosen model satisfies private connectivity requirements.
- **Ops notes**
  - Design for reruns (at-least-once execution); ensure idempotent loads and auditable pipeline runs.

1.17.3.6 Event Hubs (adjacent; when streaming is part of integration)

- **Key configurations**
  - Partitioning strategy, retention, consumer groups; checkpointing in consumer store.
- **Security notes**
  - Validate Private Endpoint and network path per SKU/region; restrict producer/consumer authorization.
- **Ops notes**
  - Plan replay and retention as part of DR and incident response (checkpoint recovery).

1.17.3.7 Key takeaways

- You should keep the **global baseline** consistent, then apply **service-specific deltas** (connectors, DLQ, replay, retention).
- You should assume **at-least-once** delivery across eventing/messaging and enforce idempotency consistently.
- You should validate **private networking feasibility per hop**, especially for connectors and Event Grid paths.

1.18 Governance, policy, and landing zone considerations

You should treat **AIS (Azure Integration Services)** as a governed toolbox deployed into an enterprise **Landing Zone**, with standardized guardrails (policy,

naming, logging, networking) that make delivery teams fast *without* weakening the enterprise security baseline.

1.18.1 Assumptions and constraints

> **Note:** These assumptions align with the document-wide baseline. Where service/SKU capabilities differ (especially around Private Link, CMK, and "disable public network access"), you should validate early and record exceptions.

- **Identity**: **Microsoft Entra ID (Azure AD)** provides identity and access management; you should prefer **Managed Identity** for workload-to-workload access (secretless authentication).
- **Networking**: You should prefer **Private Endpoint (Private Link) where supported**, and disable public network access **where feasible**.
  - If an endpoint must remain public, you should document the exception (data classification, compensating controls, firewall allowlists, TLS requirements) and obtain security approval.
  - You should explicitly validate **management-plane vs data-plane** access requirements for each service.
- **Governance**: Workloads deploy via IaC into standardized subscriptions, with enforced tagging, diagnostics, and policy compliance.
- **Resilience**: Multi-region DR is required; you should define **RTO/RPO** and test failover—including **Private Endpoint** and **private DNS** dependencies.
- **Observability**: Centralized audit logging is mandatory; correlation should be standardized end-to-end (see observability standards elsewhere in the document).

1.18.1.1 Key takeaways

- You should centralize guardrails in the Landing Zone and push "safe defaults" through policy.
- You should treat private networking controls as policy-enforced, with an explicit exception process.
- You should operationalize governance as a workflow (request → review → provision → monitor), not a document.

1.18.2 Resource organization model (management groups, subscriptions, resource groups)

1.18.2.1 Management groups

You should organize by *governance intent* and policy scope:

- **Platform management group**: landing zone subscriptions, shared services (hub networking, central logging).
- **Workload management groups**: business unit / domain groupings, with consistent baseline policies applied.

1.18.2.2 Subscriptions

You should separate subscriptions by **environment** and **blast radius**:

- **Dev/Test subscription(s)**: higher change velocity; still policy-enforced (especially diagnostics/tagging).
- **Prod subscription(s)**: strict change control, limited role assignments, stronger deny policies.
- **Shared integration subscription (optional)**: only if you operate a centralized integration platform team; otherwise prefer "you build it, you run it" within workload subscriptions.

  Recommendation: You should avoid mixing unrelated domains in a single subscription when they have different compliance profiles, cost owners, or DR obligations.

1.18.2.3 Resource groups

You should group by **lifecycle** and **ownership**, not by Azure service type:

- `rg-<workload>-ais-<env>-<region>` for AIS runtime components (APIM, Logic Apps, Service Bus, Event Grid, ADF, Event Hubs as applicable).
- `rg-<workload>-shared-<env>-<region>` for shared workload resources (Key Vault, private DNS links if workload-managed, monitoring artifacts where not centralized).

1.18.2.4 Key takeaways

- You should scope policy at management group/subscription and keep resource groups focused on lifecycle boundaries.
- You should separate prod by subscription where possible to enforce least privilege and change control.
- You should only centralize AIS into a shared subscription when you can also centralize operations and SLO ownership.

1.18.3 Standardization: naming, tagging, and cost allocation

1.18.3.1 Naming conventions (minimum)

You should standardize names to support operations, policy targeting, and DR mapping:

- Include: `workload`, `environment`, `region`, `instance` (if needed), and `service`
- Examples (illustrative):
  - `sb-<workload>-<env>-<region>-01`
  - `apim-<workload>-<env>-<region>`
  - `la-<workload>-<env>-<region>-<workflowgroup>`

  Note: Some Azure resource types have naming constraints (length/characters). You should codify the exact pattern per resource type in your engineering standards.

1.18.3.2 Tagging strategy (minimum)

You should enforce tags via Azure Policy at create/update:

Tag	Purpose
`CostCenter`	Chargeback/showback
`Owner`	Human/accountable team
`Service`	Workload/service identifier
`Environment`	dev/test/prod
`DataClassification`	Drives policy and logging rules
`RTO` / `RPO`	Operational intent for DR reviews

> **Recommendation:** You should standardize `RTO`/`RPO` tag formats (e.g., ISO 8601 durations like `PT4H`, `PT15M`) and treat them as *workload intent metadata* (not a per-resource guarantee).

1.18.3.3 Cost allocation

You should align tags and subscription design so costs can be attributed to:

- **Workload/product team** (primary)
- **Shared platform** (APIM shared gateways, central log ingestion, shared networking) where applicable

1.18.3.4 Key takeaways

- You should enforce tagging with policy; do not rely on “team discipline.”
- You should encode environment/region in naming to reduce DR ambiguity.
- You should treat RTO/RPO as governance metadata, not only a design document detail.

1.18.4 Policy guardrails (conceptual Azure Policy examples)

You should implement “deny by default” for enterprise baseline controls, with scoped exemptions.

> **Note:** Policy feasibility varies by **resource type, API version, SKU, region, and feature flags**. You should treat these examples as patterns and implement them with explicit scoping (and automated tests) rather than blanket denies.

1.18.4.1 Private networking and public access controls

- Deny creation where **public network access** is enabled (**only** for resource types that expose and support a `publicNetworkAccess`-style property).
- Require **Private Endpoint** for supported services (or require an approved exception tag/workflow).
- Restrict inbound exposure for APIM depending on chosen topology (internal vs external gateway patterns).

> **Warning:** Private networking feasibility is **SKU/region/feature dependent** and can differ between management-plane and data-plane paths. You should validate Private Endpoint support for each hop early and capture exceptions explicitly.

1.18.4.2 Diagnostic settings and log forwarding

- Require diagnostic settings to a central Log Analytics workspace / eventing sink (per Landing Zone standard).
- Prefer **DeployIfNotExists** (with a remediation identity) to auto-deploy diagnostic settings.
- Use **Deny** sparingly for diagnostics (typically only when a resource type reliably supports it), and monitor for drift.

1.18.4.3 Tag enforcement

- Deny create/update when required tags are missing.
- Append/update certain tags from subscription context (e.g., `Environment`) where appropriate.

1.18.4.4 Key takeaways

- You should use policy to enforce baseline controls, not wiki pages.
- You should design policy with an explicit exception mechanism and audit trail.
- You should standardize diagnostics and tags as “day-0 requirements” for AIS components.

1.18.5 Approval workflows and lifecycle governance (APIs, events, and schemas)

You should implement lightweight governance gates that scale:

- **API onboarding (APIM)**:
  - Contract review (authn/authz, versioning, rate limits, PII handling)
  - Threat modeling for externally exposed APIs
  - APIM policy governance (policy-as-code, linting, forbidden patterns such as secrets in policies and broad body logging)
  - Approval for policy exceptions (public endpoints, body logging)
- **Event onboarding (Event Grid / Event Hubs)**:
  - Event schema and versioning review
  - Producer ownership and consumer compatibility expectations
  - DR and duplication risk review (at-least-once delivery + failover behavior)
- **Messaging onboarding (Service Bus)**:
  - Queue/topic purpose, DLQ/runbook ownership, retention/TTL policy

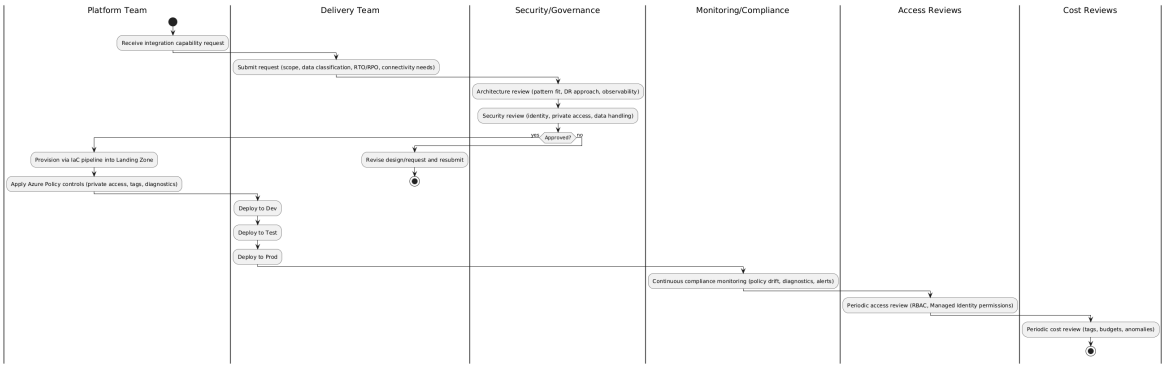
- Idempotency strategy for consumers (explicit dedupe/idempotency keys)

Recommendation: You should treat new APIs/events as “platform assets” with a registration step (catalog entry, owners, support tier, DR intent) before provisioning.

1.18.5.1 Key takeaways

- You should govern the contract surface (API/event/message) more than the implementation detail.
- You should require explicit ownership for DLQ triage, schema evolution, and DR behavior.
- You should keep gates automated where possible (template-driven approvals, policy compliance checks).

1.18.6 Governance model for AIS across environments (diagram)



1.18.6.1 Key takeaways

- You should make provisioning an IaC-backed workflow gated by architecture/security review.
- You should apply Azure Policy immediately after provisioning to prevent drift.
- You should treat access reviews and cost reviews as recurring governance controls, not ad hoc tasks.

1.19 Reliability and resilience design

You should treat reliability as an end-to-end property of the integration, not as a single AIS (Azure Integration Services) feature. In practice, you combine *platform knobs* (retries, throttles, DLQs) with *application patterns* (idempotency, outbox,

circuit breakers) and *operational readiness* (runbooks, DR drills) to meet explicit SLOs and RTO/RPO.

> **Note:** Many capabilities are SKU/region-dependent. You should validate **Private Endpoint (Private Link)** support, diagnostics coverage, and any geo/DR features for each AIS component in your target regions.

1.19.1 Assumptions and constraints (design inputs)

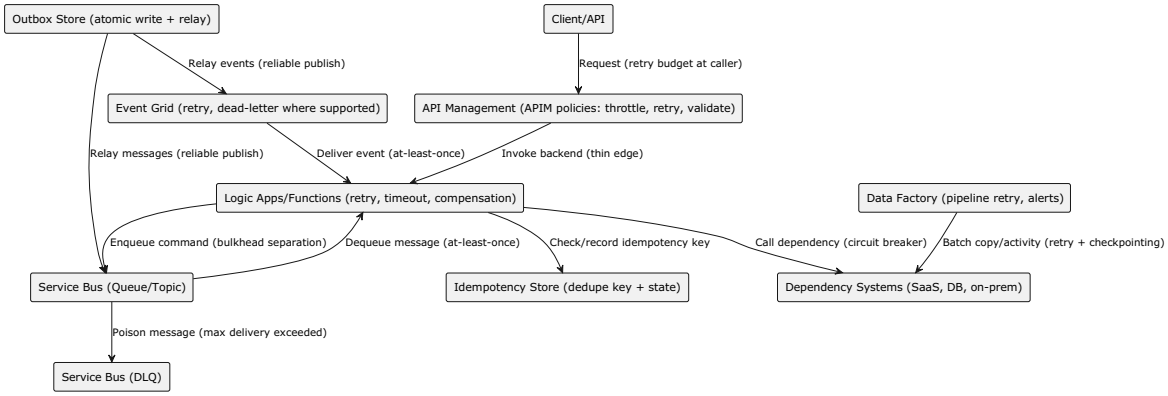
- **Identity**: You should use **Microsoft Entra ID (Azure AD)** for authentication/authorization, preferring **Managed Identity** for service-to-service access.
- **Networking**: You should prefer **Private Endpoint (Private Link)** and restrict public access **where supported**; otherwise, you should document exceptions with compensating controls (allowlists, WAF, TLS posture, additional monitoring).
- **DR**: You should design for **multi-region DR**, with explicit **RTO/RPO** and tested failover/recovery runbooks.
- **Observability**: You should propagate W3C trace context (`traceparent`) plus a business **correlation ID** across APIs, messages, events, and workflows.

1.19.1.1 Key takeaways

- You should assume *partial failure* is normal and design for it (timeouts, retries, DLQs, idempotency).
- You should treat “at-least-once” delivery as the default and design handlers to be idempotent.
- You should make RTO/RPO explicit early; the topology follows from it.
- You should validate SKU/region/networking features early to avoid late redesign.

1.19.2 Reliability patterns for AIS solutions

This section maps common reliability patterns to where you typically implement them in AIS solutions.



1.19.2.1 Pattern guide (what / when / when not)

1.19.2.1.1 What it is

- **Retries + exponential backoff**: Controlled re-attempts for transient failures.
- **Circuit breaker**: Stops calls to an unhealthy dependency to protect upstream capacity.
- **Bulkhead**: Isolates workloads so one failure/backlog does not collapse everything (e.g., separate queues/topics/subscriptions).
- **Poison message handling + DLQ**: Moves repeatedly failing messages aside for triage and replay.
- **Idempotency**: Ensures repeated deliveries do not cause repeated side effects.
- **Outbox pattern**: Writes business state and “to-be-published” messages/events atomically, then relays asynchronously.

1.19.2.1.2 When to use

- You should use **retries** for *transient* network/service throttling failures.
- You should use **circuit breakers** around unstable third-party/on-prem dependencies.
- You should use **bulkheads** when a single producer or tenant can overwhelm shared processing.
- You should use **DLQs** whenever you use **Service Bus** for durable commands/tasks.
- You should use **idempotency** for any handler of **Service Bus** or **Event Grid**.
- You should use **outbox** when you must guarantee “state change + publish” consistency.

1.19.2.1.3 When not to use

- You should not use aggressive retries for **non-transient** errors (e.g., validation failures) or you will amplify outages.
- You should not rely on Event Grid as a durable command queue; it is a notification router, not a work backlog.
- You should not skip idempotency if you cannot guarantee exactly-once effects (most integrations cannot).

1.19.2.1.4 Key configurations

Component	Where to configure	Guardrails you should apply
APIM (APIM policies)	Retry/timeout policies; rate limits/quotas	Cap total retry time; fail fast on 4xx; protect backends with throttles
Logic Apps	Action retry policy, timeout, concurrency	Set per-action retry; use deterministic dedupe keys; avoid infinite loops
Service Bus	Max delivery count, lock duration, DLQ, sessions	Tune lock vs processing time; enable DLQ; define replay runbook
Event Grid	Subscription filters, retry, dead-letter (where supported)	Treat as at-least-once; design for duplicates and reordering
Data Factory (ADF)	Activity retry, timeouts, trigger reruns, alerts	Make pipelines restartable; avoid non-idempotent loads without checkpoints

1.19.2.1.5 Security notes

- You should ensure **retry + DLQ** does not leak sensitive payloads into logs or run histories; apply data classification and masking/redaction where possible.
- You should prefer **Managed Identity** for accessing the **Idempotency Store** / **Outbox Store** and protect them with least privilege.
- You should enforce private access paths (Private Endpoint where supported) for **Service Bus**, storage-based DLQ/archives, and state stores.

1.19.2.1.6 Ops notes

- You should publish a **DLQ runbook**: triage → fix → replay → verify → archive.

- You should alert on **DLQ depth**, **age of oldest message**, and **repeated transient failures** (which often indicate a dependency outage).
- You should maintain a “retry budget” per dependency tier (internal vs partner vs SaaS) to avoid cascading failure.

1.19.2.2 Key takeaways

- You should implement reliability patterns *at multiple layers*: caller, gateway, orchestrator, and messaging.
- You should treat idempotency + DLQ handling as mandatory for at-least-once systems.
- You should use bulkheads (separate queues/topics) to prevent noisy-neighbor collapse.
- You should use an outbox when publish consistency matters more than simplicity.

1.19.3 Multi-region and failure handling

1.19.3.1 What it is

Multi-region resilience is the combination of (1) how you deploy across regions and (2) how you fail over traffic and state safely, without creating duplication or data loss outside the agreed RTO/RPO.

1.19.3.2 When to use

- You should use **active-active** when you need lower RTO and can tolerate/mitigate duplication (idempotency, conflict handling).
- You should use **active-passive** when simplicity and deterministic failover are more important than RTO.

1.19.3.3 When not to use

- You should not use active-active when you cannot define conflict resolution or idempotency semantics for side effects (payments, entitlement changes) without strong business controls.

1.19.3.4 Key configurations

Minimum DR decision template (fill this out per integration):

Item	You should define
RTO target	Time to restore service in a region-loss scenario
RPO target	Acceptable data loss window (if any)
Failover trigger	Manual vs automated; health signals used
Routing mechanism	DNS/global routing to APIM/ingress; backend selection logic
State replication	Datastores replicated? outbox/idempotency store replicated?
Replay source	Service Bus reprocessing, outbox relay replay, event re-drive strategy
Validation steps	Post-failover smoke tests; backlog/DLQ checks; data consistency checks

> **Warning:** Multi-region designs frequently fail during DR because **Private Endpoints are regional** and **private DNS** does not fail over automatically unless you design and test it. You should include private endpoint + DNS validation in DR drills.

1.19.3.5 Security notes

- You should ensure region failover does not bypass enterprise controls (firewall/NVA routing, private DNS, inspection policies).
- You should ensure DR access paths remain least-privilege (RBAC, key access) in both regions.

1.19.3.6 Ops notes

- You should run DR drills that include **duplication scenarios** (replayed messages/events) and verify idempotency controls.
- You should maintain dependency mapping: APIM → workflows → messaging/eventing → state stores → downstream systems, and test failover in that order.

1.19.3.7 Key takeaways

- You should decide active-active vs active-passive based on RTO/RPO *and* conflict/idempotency complexity.
- You should treat private endpoint + private DNS duplication as first-class DR work.
- You should drill failover including replays/duplicates, not just component availability.

1.19.4 Capacity, throttling, and backpressure

1.19.4.1 What it is

Backpressure is how the system remains stable when downstream dependencies are slow or unavailable (queueing, throttling, shedding load) rather than timing out everywhere.

1.19.4.2 When to use

- You should apply backpressure controls whenever a caller can outpace a dependency (common in APIM → backend and event fan-out cases).

1.19.4.3 When not to use

- You should not “buffer indefinitely” without business agreement; unbounded queues become hidden outages.

1.19.4.4 Key configurations

- **APIM**: you should configure quotas/rate limits and timeouts to protect backends; keep policies “thin.”
- **Logic Apps**: you should control concurrency and action timeouts; avoid unconstrained parallel loops against fragile systems.
- **Service Bus**: you should size throughput (e.g., Premium capacity) and monitor queue depth; use separate entities for bulkheads.
- **Event Grid**: you should filter aggressively and avoid using it as a backlog; scale handlers independently.
- **ADF**: you should control parallelism and partitioning; design rerunnable pipelines.

1.19.4.5 Security notes

- You should ensure throttling responses do not disclose sensitive internals (consistent error handling at APIM).
- You should ensure backlog storage/archives are encrypted and access-controlled (especially if DLQ payloads contain regulated data).

1.19.4.6 Ops notes

- You should alert on “approaching capacity,” not just “failure”:
  - APIM: sustained 429/5xx, latency spikes
  - Service Bus: active messages, DLQ length, oldest message age
  - Logic Apps/Functions: failure rate, retry saturation, throttled actions
  - ADF: activity failures, long runtimes vs baseline

1.19.4.7 Key takeaways

- You should protect downstream systems with throttling and bounded queueing.
- You should design for controlled degradation (429/202 patterns) rather than global timeouts.
- You should monitor queue depth/age as primary early-warning signals.

1.19.5 Graceful degradation and partial failure handling

1.19.5.1 What it is

Graceful degradation means preserving core business functions while shedding non-critical work (e.g., async processing, deferred enrichment) during dependency impairment.

1.19.5.2 When to use

- You should degrade when dependencies are optional or can be reconciled later (e.g., analytics events, non-critical notifications).

1.19.5.3 When not to use

- You should not degrade when it creates unsafe business outcomes (e.g., “accepted” commands that cannot be recovered/replayed).

1.19.5.4 Key configurations

- You should prefer **async offload via Service Bus** with `202 Accepted` + an **async status channel** for long-running work.
- You should implement compensating actions in **Logic Apps** when true atomicity is not feasible (saga pattern).

1.19.5.5 Security notes

- You should ensure degraded modes still enforce authentication/authorization and do not broaden access.
- You should ensure deferred processing does not violate retention policies (e.g., queue TTL vs business requirements).

1.19.5.6 Ops notes

- You should document “degraded mode” behaviors in runbooks (what is delayed, where it queues, how it drains).
- You should include replay/compensation steps in incident response and DR drills.

1.19.5.7 Key takeaways

- You should design explicit degraded modes for non-critical capabilities.
- You should pair async patterns with status visibility and replay procedures.
- You should validate degraded-mode security posture (no “temporary” bypasses).

1.20 Observability and operations

1.20.1 Assumptions and constraints (design inputs)

> **Note:** This section assumes the Landing Zone baseline in the governance section; the focus here is on telemetry, runbooks, and operational readiness.

- **Identity**: You use **Microsoft Entra ID (Azure AD)** for authentication/authorization; you should prefer **Managed Identity** for workload-to-workload access.
- **Networking**: You should use **Private Endpoint (Private Link)** where supported and disable public network access where feasible. If a hop must remain public, you should document the exception and compensating controls (IP allowlists, WAF, TLS requirements, logging/alerting).
- **DR**: Multi-region DR is required; you should define **RTO/RPO** per integration and test failover, including private DNS behavior for Private Endpoints (regional dependency).

- **Telemetry platform**: You should centralize logs/metrics/traces in **Azure Monitor + Log Analytics**, and use **Application Insights** where applicable for distributed tracing.

1.20.1.1 Key takeaways

- You should treat observability as an end-to-end architecture concern, not a per-component add-on.
- You should assume at-least-once delivery and build for duplicates; observability must make retries/replays explicit.
- You should make private DNS and Private Endpoint regional dependencies first-class in ops/DR runbooks.

1.20.2 End-to-end observability approach for integrations

1.20.2.1 What it is

An end-to-end observability approach defines how integrations emit and correlate **logs**, **metrics**, and **traces** across APIM, workflows/compute, messaging, and downstream processing so operators can:

- Identify where failures occur (edge, orchestration, broker, consumer, dependency).
- Quantify health (SLO signals, backlog, error rates).
- Execute safe recovery actions (replay, reprocess, DLQ handling) with traceability.

1.20.2.2 When to use

- Always, for production integrations under enterprise governance and audit requirements.
- Especially for **asynchronous** patterns (APIM → Service Bus → worker) where failures are delayed and distributed.

1.20.2.3 When not to use

- You should not rely on ad-hoc per-team logging formats or component-local dashboards; they prevent cross-service correlation and slow incident response.

1.20.2.4 Key configurations

- **Diagnostics enablement (per component)**:
  - APIM diagnostic settings → Log Analytics / App Insights (gateway logs, backend latency, failures).
  - Logic Apps/Functions → runtime logs + App Insights tracing (where applicable).
  - Service Bus → metrics and resource logs (when available), plus consumer-side telemetry.
- **Standard fields** (apply consistently):
  - `traceparent` (W3C trace context)
  - `Correlation-Id` (business correlation ID; stable across retries/replays)
  - `Idempotency-Key` (or equivalent) for dedupe/idempotent processing (may differ from `Correlation-Id`)
  - `OperationName`, `ComponentName`, `Environment`, `Region`
  - Message metadata: `MessageId`, `SessionId` (if used), `DeliveryCount`, `EnqueuedTime`, `DeadLetterReason` (DLQ)

Recommendation: You should standardize on **W3C trace context** (`traceparent`) plus a business `Correlation-Id` header/property, and use a separate **idempotency/deduplication key** when consumer semantics require it.

1.20.2.5 Security notes

- You should treat telemetry as **sensitive data** (often contains identifiers, payload fragments, or operational secrets).
- You should enforce:
  - Least-privilege access to Log Analytics workspaces (RBAC + role separation).
  - Data retention aligned to compliance requirements.
  - Encryption requirements (including CMK where mandated and supported).

Warning: Logging request/response bodies from APIM/Logic Apps can violate data handling policies. If payload capture is required, you should implement targeted sampling/redaction with explicit approval and retention controls.

1.20.2.6 Ops notes

- You should define a “golden path” dashboard that spans:
  - **Ingress** (APIM): 4xx/5xx rate, latency, throttles.
  - **Workflow/compute**: failure count, retry count, duration anomalies.
  - **Messaging** (Service Bus): active message count, scheduled messages, **DLQ length**, oldest message age.
  - **Dependencies**: downstream error rate/latency, private endpoint/DNS resolution failures (where detectable).
- You should ensure operational access supports incident response without broad contributor permissions (break-glass procedures + audit logs).

1.20.2.7 Key takeaways

- You should standardize telemetry schemas early so cross-team operations scale.
- You should monitor both “failure” and “stuck” signals (e.g., backlog age, DLQ growth).
- You should align telemetry retention and access control to compliance and least privilege.

1.20.3 Correlation IDs and distributed tracing strategy

1.20.3.1 What it is

A correlation strategy defines how you link a single business transaction across:

- **HTTP** (Client → APIM → backend)
- **Workflows/compute** (Logic Apps/Functions)
- **Messaging** (Service Bus messages)
- **Monitoring** (Azure Monitor / Application Insights)

This typically combines:

- **W3C trace context** for distributed tracing (`traceparent` / `tracestate`).
- A **business correlation ID** (`Correlation-Id`) for grouping related work and human-friendly incident handling.
- An **idempotency/deduplication key** (e.g., `Idempotency-Key`, `DeduplicationId`) when consumers must safely process duplicates.

1.20.3.2 When to use

- Always for multi-hop integrations.
- Mandatory for asynchronous patterns where the request/response boundary is separated from completion.

1.20.3.3 When not to use

- You should not invent per-service correlation keys without mapping them to W3C tracing and a single business correlation ID; it fragments traces.

1.20.3.4 Key configurations

- **HTTP (APIM)**:
  - Accept inbound `traceparent` and `Correlation-Id`; if missing, generate both.
  - Validate/sanitize correlation headers (size limits; allowed character set).
  - Forward both to the backend.
- **Service Bus**:
  - Set application properties: `Correlation-Id` (business), optionally `traceparent`.
  - If using sessions, ensure `SessionId` aligns to ordering requirements (not correlation by default).
- **Workers/consumers**:
  - Read `Correlation-Id` and `traceparent`, start/continue the trace, emit logs/metrics with both.
  - Use an explicit idempotency key (from message metadata or payload) to implement dedupe/idempotency where required.

1.20.3.5 Security notes

- You should treat correlation IDs as **metadata that can be abused for data inference** (especially if derived from PII). Prefer random/opaque identifiers.
- You should validate and size-limit correlation headers at the edge to reduce abuse.

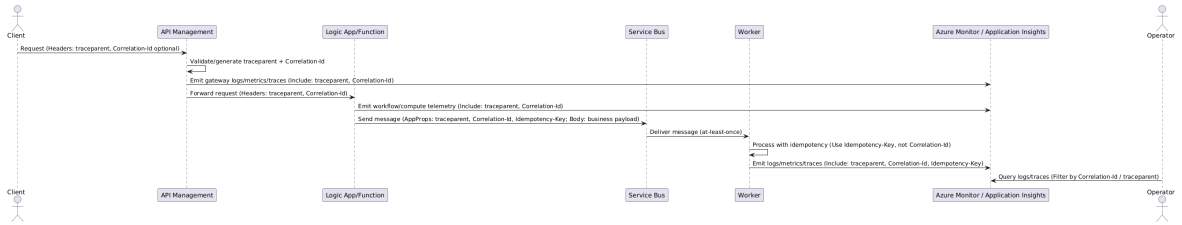
1.20.3.6 Ops notes

- You should make `Correlation-Id` a first-class filter in dashboards and incident triage queries.

- You should include `Correlation-Id` in DLQ inspection tooling and replay pipelines.

1.20.3.7 Key takeaways

- You should propagate **both** `traceparent` and `Correlation-Id`.
- You should keep correlation IDs opaque and non-PII.
- You should design replay/DLQ handling around correlation IDs *and* explicit idempotency keys to keep recovery auditable.



1.20.4 Dashboards and alerts (minimum operational baseline)

1.20.4.1 What it is

A baseline set of dashboards and alerts ensures operators detect and respond to integration failures quickly, with clear ownership per component.

1.20.4.2 When to use

- For all production workloads; it should be part of the “go-live gate”.

1.20.4.3 When not to use

- You should not rely solely on “availability” or single synthetic checks; integrations often fail through partial degradation (timeouts, backlog growth, poison messages).

1.20.4.4 Key configurations

Component	Minimum signals	Typical alerts
APIM	4xx/5xx, latency, throttles	5xx spike, backend timeout spike, auth failures
Logic Apps/ Functions	failures, retries, duration	workflow run failures, retry storms, timeout rate

Component	Minimum signals	Typical alerts
Service Bus	active messages, oldest age, DLQ length	DLQ growth, backlog age breach, sudden drop in throughput
Event Grid (if used)	delivery failures, dead-letter (where supported)	delivery failure rate, subscription disabled
Monitoring plane	ingestion health, query latency	workspace ingestion delay, alert rule failures

> **Note:** Some diagnostics and dead-letter capabilities are **SKU/region/endpoint dependent**. You should validate availability for your target services early and document gaps.

1.20.4.5 Security notes

- You should restrict alert destinations (paging, email, chat) based on data classification; avoid sending sensitive identifiers to non-secure channels.
- You should audit changes to alert rules and dashboards (infrastructure-as-code preferred).

1.20.4.6 Ops notes

- You should define on-call ownership by component boundary (APIM, workflow/compute, messaging, downstream dependency).
- You should create “fast triage” queries that pivot on `Correlation-Id` and message identifiers.

1.20.4.7 Key takeaways

- You should alert on backlog age and DLQ growth, not only error rates.
- You should define ownership and triage queries as part of the monitoring baseline.
- You should validate diagnostics support per SKU/region and capture exceptions explicitly.

1.20.5 Operational runbook pointers (replay, reprocess, DLQ, incident response)

1.20.5.1 What it is

Runbooks define safe, repeatable operational actions during incidents and planned recovery, including evidence capture for audit/compliance.

1.20.5.2 When to use

- Always for asynchronous and message-driven integrations.
- Required when you claim an RTO/RPO and multi-region DR posture.

1.20.5.3 When not to use

- You should not perform manual “fix and resubmit” actions without:
  - An idempotency strategy
  - A documented approval/audit trail
  - A known-safe replay mechanism

1.20.5.4 Key configurations

- **Replay/reprocess**:
  - Store enough metadata to reconstruct/retry safely (`Correlation-Id`, `MessageId`, explicit `Idempotency-Key`/dedupe key).
  - Define bounded retry policies and a terminal failure path into DLQ.
- **DLQ handling**:
  - Define classification (poison vs transient vs validation failure).
  - Define remediation steps and tooling (re-drive with guardrails; purge with approval).
- **Incident response**:
  - Capture: impacted correlation IDs, time window, regions, dependency health, mitigation steps.

Recommendation: You should assume **at-least-once delivery** and enforce **idempotency + explicit dedupe keys + replay procedures** for all message/event consumers.

1.20.5.5 Security notes

- You should gate replay and DLQ re-drive operations via least privilege and approvals (separation of duties).
- You should treat DLQ payloads as production data (same classification, encryption, retention).

1.20.5.6 Ops notes

- You should practice DLQ and replay runbooks in non-production with production-like data shapes.
- You should include DR-specific runbook steps for:
  - Regional private endpoint dependencies (recreate/activate in secondary region)
  - DNS forwarding/conditional resolution validation
  - Client reconnection behavior (APIM endpoints, Service Bus connections)

Minimum runbook template (recommended)

- Owner/on-call rotation
- Preconditions (required roles, break-glass path)
- Step-by-step procedure (including “stop the bleed” actions)
- Validation steps (queries/metrics to confirm recovery)
- Evidence capture (ticket ID, queries used, approvals)
- DR-specific deltas (secondary region endpoints, DNS checks)

1.20.5.7 Key takeaways

- You should operationalize recovery: replay, reprocess, and DLQ handling are first-class features, not afterthoughts.
- You should align runbooks to idempotency assumptions and audit requirements.
- You should test DR runbooks including private DNS and Private Endpoint regional behavior.

1.21 DevOps, IaC, and release strategies

1.21.1 Assumptions and constraints (delivery context)

> **Note:** This section builds on the governance and observability baselines. The focus here is how you encode those controls into delivery automation.

- **Identity**: **Microsoft Entra ID (Azure AD)** provides identity and access management; you should prefer **Managed Identity** for workload-to-workload access (secretless authentication).
- **Networking**: you should prefer **Private Endpoint (Private Link) where supported**, and disable public network access **where feasible**. If an endpoint must remain public, you should document the exception (data classification, compensating controls, firewall allowlists, TLS requirements).
- **Governance**: deployments occur within a governed **Landing Zone** (policy, tagging, diagnostics, approved regions/SKUs).
- **Reliability**: **multi-region DR** is required; you should define and test **RTO/RPO** and failover procedures per environment and per dependency (including **private DNS** parity when using Private Endpoints).
- **Observability**: you should propagate W3C trace context (`traceparent`) plus a business **correlation ID** end-to-end; use explicit idempotency/dedupe keys for consumers as required; enforce diagnostic settings via policy.

> **Note:** Private Endpoint/Private Link, CMK, and "disable public network access" capabilities vary by **service, SKU, connector, and region**. You should validate **management-plane vs data-plane** private access early to avoid redesign.

1.21.1.1 Key takeaways

- You should treat delivery constraints (identity, private networking, DR) as **design inputs** to the pipeline, not afterthoughts.
- You should standardize on **Managed Identity + Key Vault references** to minimize secret handling in CI/CD.
- You should validate **SKU/region feature availability** during build/plan stages, not at deploy time.

1.21.2 Infrastructure as Code (IaC)

1.21.2.1 What it is

IaC defines your integration platform and AIS components (APIM, Logic Apps, Service Bus, Event Grid, ADF, Event Hubs) as versioned, reviewable code using **Bicep/ARM** or **Terraform**, including networking (Private Endpoints, private DNS links), identity assignments, and diagnostics.

1.21.2.2 When to use

- You need **repeatable deployments** across dev/test/prod with consistent controls and tagging.
- You must enforce **enterprise security baseline** (private networking, diagnostics, approved SKUs/regions) through policy and templates.
- You need multi-region footprints and DR parity (including **private DNS** and **Private Endpoints**).

1.21.2.3 When not to use

- You are doing one-off experimentation that will be discarded (still keep a minimal template if it might persist).
- You cannot control target subscriptions/resource groups (e.g., unmanaged partner environments); in that case, you should codify **contract tests** and drift detection instead.

1.21.2.4 Key configurations

- **Parameterization**: separate *platform parameters* (regions, VNets, DNS zones, SKUs) from *workload parameters* (API names, workflow settings, pipeline names).
- **Environment configuration separation**:
  - Use per-environment parameter files (Bicep) or workspaces/variable sets (Terraform).
  - Keep non-secret config in repo; keep secrets in **Key Vault**.
- **Policy alignment**: ensure templates comply with Landing Zone policy (diagnostics, tags, private endpoints, encryption settings).
- **Deterministic naming**: apply a consistent naming scheme across environments to simplify RBAC, DNS, monitoring, and DR runbooks.

1.21.2.5 Security notes

- You should avoid embedding secrets in templates; use **Key Vault references** and **Managed Identity**.
- You should deploy **Private Endpoints** and **private DNS zone links** via IaC; treat DNS as a first-class dependency for DR.
- You should scope RBAC to least privilege for deployment identities (separate “plan” vs “apply” rights if required).

1.21.2.6 Ops notes

- You should run **drift detection** (what changed outside IaC) on a schedule for prod.
- You should treat DNS and Private Endpoint state as part of operational readiness checks (pre-flight in pipelines).
- You should pre-provision role assignments and diagnostic settings consistently to reduce “works in dev” issues.

1.21.2.7 Key takeaways

- IaC should capture **networking + DNS + diagnostics** alongside the AIS resources (not just the resources).
- You should separate **platform** and **workload** parameters to reduce coupling and blast radius.
- You should validate policy compliance as part of build/plan to prevent late deployment failures.

1.21.3 CI/CD for AIS artifacts

1.21.3.1 What it is

CI/CD automates build, validation, and promotion of both infrastructure and service-level artifacts, including:

- **APIM** APIs, products, and policies
- **Logic Apps (Standard)** application package and workflow artifacts
- **ADF** pipelines, linked services, datasets, triggers (and their deployment representation via ARM/Bicep or publish artifacts)
- Configuration via **Key Vault references** and environment settings

1.21.3.2 When to use

- You release integrations frequently and need safe, traceable promotion across environments.
- You need controlled rollout patterns (blue/green, canary) and fast rollback for APIs and handlers.
- You must support audited change control and approval gates.

1.21.3.3 When not to use

- You cannot establish stable lower environments representative of production; in that case, prioritize contract tests and production-safe progressive delivery with strict blast-radius controls.

1.21.3.4 Key configurations

- **Pipeline stages**:
  - Build and validate (lint, template validation, policy checks, unit tests)
  - IaC plan/what-if (detect drift and policy violations)
  - Deploy to dev → integration tests → promote to test → approval → deploy to prod
- **Artifact packaging**:
  - APIM: versioned API/policy artifacts (treat policies as code; enforce linting)
  - Logic Apps Standard: package as deployable unit with environment settings externalized
  - ADF: deploy pipelines/triggers with environment-specific linked service parameters
- **Configuration injection**:
  - Use environment-specific variables for non-secrets
  - Use **Key Vault references** for secrets and certificates

1.21.3.5 Security notes

- You should use **workload identity/Managed Identity** for deployment agents where possible; otherwise tightly control service principals and rotate credentials.
- You should gate production with approvals and require evidence (tests, what-if/plan output, policy compliance).
- You should restrict who can modify pipelines and who can approve production releases (segregation of duties).

1.21.3.6 Ops notes

- You should include post-deploy health checks (APIM synthetic calls, workflow trigger validation, ADF pipeline test run, Service Bus connectivity checks).
- You should record deployment metadata (build number, git SHA, correlation ID) into centralized logs for incident traceability.
- You should automate rollback paths (see next section) rather than relying on manual remediation.

1.21.3.7 Key takeaways

- You should treat **APIM policies, Logic Apps artifacts, and ADF pipelines** as versioned release artifacts.
- You should inject configuration via **environment variables + Key Vault references**, not per-environment code forks.
- You should require **integration tests + approvals** before production promotion.

1.21.4 Secrets and configuration management

1.21.4.1 What it is

A consistent approach to managing environment configuration and secrets across AIS services using:

- **Key Vault** for secrets/keys/certificates
- **Key Vault references** where supported
- Centralized non-secret configuration (variables/parameter files)

1.21.4.2 When to use

- Any integration touching regulated or sensitive data.
- Any production workload where secret sprawl or inconsistent configuration is a risk.

1.21.4.3 When not to use

- Not applicable in enterprise baselines; even low-risk workloads should avoid storing secrets in code.

1.21.4.4 Key configurations

- **Key Vault references**: prefer platform-native references over runtime secret retrieval logic.
- **Rotation strategy**: define ownership, rotation intervals, and blast-radius-aware rollout (especially for certificates and partner credentials).
- **Config contract**: define required settings per component (e.g., endpoints, topic names, correlation ID header name, retry budgets).

1.21.4.5 Security notes

- You should restrict Key Vault access to the smallest set of identities (Managed Identities for runtime, limited deploy identity for reference setup).
- You should enable diagnostics/audit logs for Key Vault and alert on suspicious access patterns.

1.21.4.6 Ops notes

- You should test secret rotation in non-prod and validate rollback (old+new coexistence window).
- You should treat Key Vault availability and private DNS resolution as dependencies in DR drills.

1.21.4.7 Key takeaways

- You should prefer **Managed Identity + Key Vault references** to eliminate secret handling in pipelines.
- You should standardize a **configuration contract** per service to reduce release risk.
- You should operationalize **rotation + rollback** as a practiced procedure.

1.21.5 Release patterns and rollback strategies

1.21.5.1 What it is

Release strategies that reduce customer impact and protect SLOs during changes to APIs, workflows, and message/event schemas.

1.21.5.2 When to use

- Any change that can break consumers (API contract, message schema, routing/policies).
- Any change that affects throughput limits or retry behavior (risk of cascading failure).

1.21.5.3 When not to use

- Trivial cosmetic changes with no runtime impact (still deploy via pipeline, but progressive delivery may be unnecessary).

1.21.5.4 Key configurations

- **Blue/green for APIs (APIM)**:
  - Deploy a new API revision/version behind APIM and shift traffic via routing rules.
  - Keep policies backward compatible; avoid long-running logic in policies.
- **Canary releases**:
  - Route a small percentage of traffic to new backend/workflow/handler.
  - Use error budget guardrails (rollback on elevated 5xx/latency).
- **Backward-compatible event/message evolution**:
  - Additive schema changes first; deprecate fields later.
  - Version topics/events where needed; coordinate consumer upgrades.
- **Rollback nodes (standard)**:
  - Revert APIM revision/route
  - Redeploy last-known-good Logic Apps package
  - Disable ADF triggers and revert pipeline artifacts
  - Prefer **consumer-side pause** (scale down/disable processors) over broker-side manipulation; if you change subscriptions/rules, you should document ordering/session impacts and recovery steps.

  Warning: Because most AIS eventing/messaging paths are **at-least-once delivery**, rollbacks can produce duplicates and reprocessing.
  You should enforce **idempotency** and have replay/reprocess runbooks before using aggressive canary/blue-green patterns.

1.21.5.5 Security notes

- You should treat release routing rules as security-impacting configuration (could bypass authZ/throttling if misconfigured); require review.

- You should ensure production approvals include verification that private endpoints/DNS are unchanged or updated safely for the release.

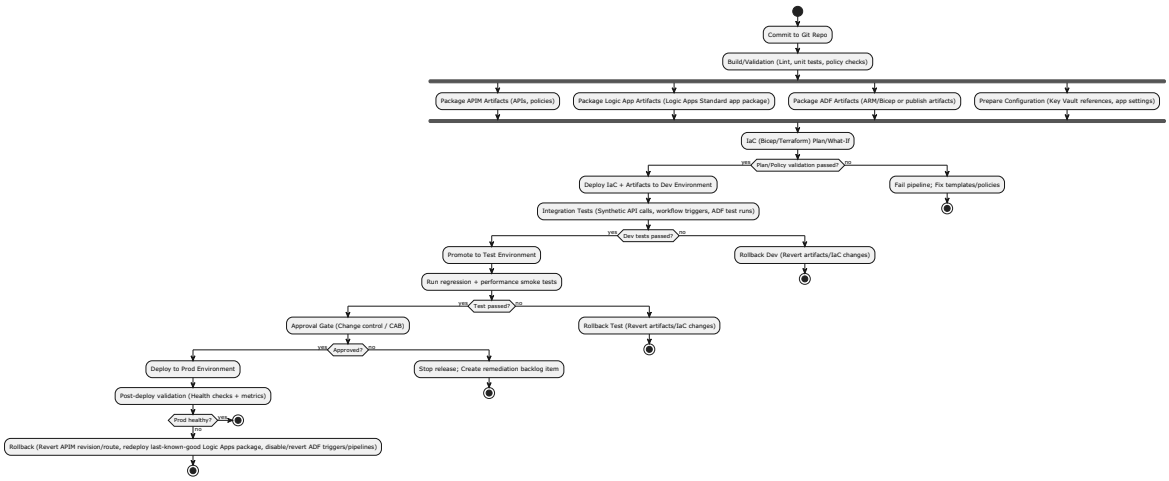
1.21.5.6 Ops notes

- You should define “rollback triggers” as objective signals (error rate, latency, DLQ growth, workflow failure rate).
- You should validate rollback regularly (game days), including cross-region DR scenarios where DNS and routing differ.

1.21.5.7 Key takeaways

- You should prefer **blue/green and canary** for APIM/backends to reduce blast radius.
- You should evolve schemas **backward-compatibly** and assume duplicates during both forward and rollback moves.
- You should automate rollback and tie it to **measurable guardrails**.

1.21.6 CI/CD and IaC flow for AIS components



1.21.6.1 Key takeaways

- You should validate **policy + IaC plans** before deploying any AIS artifacts.
- You should promote the same versioned artifacts through dev/test/prod with **approval gates** for prod.
- You should implement rollback as a **first-class pipeline path**, not an ad-hoc manual procedure.

1.22 Reference architectures (end-to-end examples)

This section provides cohesive, end-to-end examples that combine **APIM (Azure API Management)**, **Logic Apps**, **Service Bus**, and foundational controls (identity, private access, observability). It is pattern-centric; service deep dives follow the standard template elsewhere.

1.22.1 Assumptions and constraints (decision-impacting)

- **Identity**: **Microsoft Entra ID (Azure AD)** provides identity and access management; you should prefer **Managed Identity** for workload-to-workload access (secretless authentication).
- **Networking**: you should use **Private Endpoint (Private Link)** where supported and disable public network access where feasible; if an endpoint must remain public, you should document the exception and compensating controls (data classification, firewall allowlists, TLS requirements).
- **DR**: **multi-region DR is required**; you should define **RTO/RPO**, failover triggers, and runbooks, and you should drill failover including replay/duplication behavior.
- **Observability**: you should standardize on **W3C trace context** (`traceparent`) plus a business **correlation ID**, and propagate both end-to-end (see Observability/operations conventions in your platform standard).

  Note: Validate **SKU/region** support for Private Link, CMK, and diagnostics early. Capabilities differ by service, SKU, region, and sometimes connector/endpoint type, and they change over time.

1.22.1.1 Key takeaways

- You should treat this as a *pattern* view; confirm SKU/region feature support during design.
- You should design for at-least-once delivery and implement **idempotency** in consumers.
- You should make DNS and private endpoint replication part of DR, not an afterthought.

1.23 Reference architecture: API-led integration with asynchronous backend

1.23.1 What it is

An **API-led** pattern where external clients call a governed API surface (**APIM**). A backend **Orchestrator (Logic Apps/Functions)** performs lightweight coordination and publishes a **durable command** to **Service Bus (Command Queue/Topic)**. **Worker(s)** execute side effects against **Downstream Systems**. Completion is communicated via an **Async Status Channel** (status endpoint and/or event).

> **Recommendation:** You should define responsibilities explicitly: **Orchestrator = coordination/state**, **Worker(s) = execution/side effects**. This prevents business logic drift into APIM policies or workflows.

1.23.1.1 Key takeaways

- You should use APIM for edge controls, not orchestration.
- You should use Service Bus to decouple client SLOs from downstream latency/failures.
- You should provide an explicit Async Status Channel for long-running work.

1.23.2 When to use

- You must protect synchronous APIs from variable downstream latency and outages (return `202 Accepted` + status).
- You need **durable command handling** with operational controls (DLQ, controlled retries, backpressure).
- You require strong governance at the API boundary (authentication/authorization, quotas, versioning).

1.23.2.1 Key takeaways

- Choose this when you need *durable work queues* behind a stable API contract.
- This pattern fits enterprise controls (least privilege, private access, centralized monitoring).

1.23.3 When not to use

- You need near-real-time streaming ingestion with replay/retention semantics (prefer **Event Hubs**).
- You only need best-effort notifications/fan-out without durable command processing (prefer **Event Grid**).
- The workflow is truly synchronous and short-lived end-to-end; async adds components and operational overhead.

1.23.3.1 Key takeaways

- Avoid using Service Bus “just in case” if synchronous processing meets SLOs and failure modes.
- Avoid using APIM policies for complex validation/orchestration.

1.23.4 Key configurations

APIM (edge controls; keep policies thin)

- Authentication/authorization: JWT validation (Entra ID), per-API/product access control.
- Request boundary checks: required headers, payload size limits, optional JSON schema validation where feasible.
- Protection: rate limits/quotas, spike arrest, IP filtering (where required).
- Correlation: inject/propagate `traceparent` and your business **correlation ID** header.
- Async behavior: return `202 Accepted` with an operation/status resource link.

Orchestrator (Logic Apps/Functions)

- Validate *business* rules, enrich messages, and choose routing keys/metadata for workers.
- Publish command to Service Bus with idempotency keys and correlation metadata.
- Implement compensation/timeout logic where applicable (especially for long-running workflows).

Service Bus (Command Queue/Topic)

- Choose **Queue** for single logical consumer group; **Topic/Subscription** for multi-consumer fan-out with independent processing.

- Configure DLQ behavior (max delivery count, TTL, dead-letter reason usage).
- Use sessions/ordering only when required (trade-off: throughput/complexity).
- Consider duplicate detection where it fits the message model (trade-off: memory/time window).

Async Status Channel

- Status endpoint: store operation state (e.g., in a transactional store) and expose via APIM.
- Optional event notification: emit completion events for subscribers (still assume at-least-once).

1.23.4.1 Key takeaways

- You should do *edge checks* in APIM and *business validation* in the orchestrator/workers.
- You should design DLQ and replay as first-class operational flows.
- You should treat status reporting as part of the API contract, not an afterthought.

1.23.5 Security notes

- **Private access**: you should prefer **Private Endpoint (Private Link)** for **Service Bus** and **Key Vault** where supported; disable public network access where feasible.
- **Private DNS**: because private endpoints are **regional**, DR requires:
  - duplicate private endpoints per region,
  - private DNS zone links per VNet/region,
  - tested conditional forwarding from on-prem (if hybrid).
- **Secrets**: you should use **Key Vault** for secrets/keys/certs; workloads should authenticate using **Managed Identity**.
- **Least privilege**: scope RBAC to the smallest set (e.g., “send” vs “listen” roles for Service Bus, read-only secrets where required).
- **Data handling**: you should avoid logging sensitive payloads in APIM/workflows; apply sampling/redaction with approved retention if capture is required.

1.23.5.1 Key takeaways

- You should plan private endpoints and DNS as *regional resources* in DR.

- You should use Managed Identity + Key Vault to avoid embedded secrets.
- You should enforce least privilege separately for producers and consumers.

1.23.6 Ops notes

Delivery semantics and correctness

- Assume **at-least-once** delivery for messaging and event notifications; you should enforce **idempotency** in Worker(s) and orchestrations.
- You should define DLQ triage/replay runbooks (owner, SLA, quarantine criteria, replay tooling).

Monitoring baseline (minimum signals)

Component	Key signals	Typical alerts
APIM	4xx/5xx rate, latency, backend failures	Elevated 5xx, p95 latency, auth failures spike
Service Bus	active messages, DLQ length, throttling, failures	DLQ growth, backlog growth, send/listen errors
Worker(s)	failures, retries, dependency latency	error rate, retry storm, dependency timeouts
Orchestrator	run failures, action retries, long-running instances	failure rate, stuck/timeout instances

DR options (trade-offs)

- **Active/active**: higher cost/complexity; you must manage multi-region routing and duplication risk; clients and workers must handle replay and duplicate processing cleanly.
- **Active/passive**: simpler steady-state; you must automate/promote failover and validate client reconnection behavior.

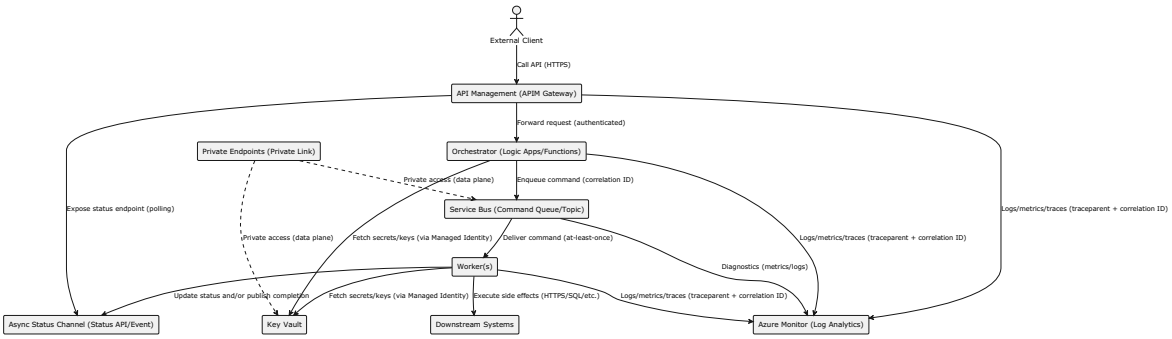
  Warning: DR for private access commonly fails due to DNS/private endpoint coupling. You should include DNS resolution tests and private endpoint reachability checks in every DR drill.

1.23.6.1 Key takeaways

- You should operationalize DLQ handling and replay (runbooks + tooling).
- You should alert on backlogs and DLQ growth as early indicators of downstream failure.

- You should drill DR with duplication/replay scenarios, not just “service up” checks.

1.23.7 Diagram: API-led integration with asynchronous backend



1.24 Decision matrix and quick recommendations

This section is your fast path to selecting the right **AIS (Azure Integration Services)** building blocks and avoiding common enterprise integration failure modes. Use it after you understand the reference architecture patterns (see **s16**) and before you start detailed implementation.

> **Note:** This matrix assumes an enterprise baseline: **Entra ID** + **Managed Identity**, **Private Endpoint (Private Link)** preferred where supported, centralized logging/metrics, and **multi-region DR** with explicit **RTO/RPO** and tested runbooks.

1.24.1 Quick selection matrix (default choices)

Need	Prefer	Why	Not ideal when
Publish and govern APIs (internal or external)	**APIM (Azure API Management)**	Central authZ/authN, throttling, transformation, consistent API lifecycle	You need only raw L4/L7 pass-through without policy, analytics, or developer onboarding
Durable async commands / work queue	**Service Bus**	Durable **queues**/**topics/subscriptions**, DLQ, backpressure controls	You only need best-effort notifications with fan-out and minimal ops
Event notifications /	**Event Grid**		You need durable commands, strict

Need	Prefer	Why	Not ideal when
fan-out to many handlers		Pub/sub event routing, filtering, retry, push-based delivery	ordering, or replay as a log
Long-running orchestration across systems	**Logic Apps**	Stateful workflows, connectors, built-in retry/compensation patterns	You need high-throughput compute-heavy processing as the primary workload
Batch ingestion / ETL/ELT / scheduled movement	**Data Factory (ADF)**	Pipeline scheduling, data movement, batch transforms and integration runtimes	You need low-latency transactional integration or streaming semantics
High-throughput streaming ingestion with replay	**Event Hubs** (adjacent)	Retention/replay, partitions, consumer groups	You need per-message business DLQ semantics and command processing controls

1.24.1.1 Key takeaways

- You should treat **Service Bus** as the default for **durable business messaging** (commands/work items).
- You should treat **Event Grid** as the default for **event notifications** (facts that something happened).
- You should treat **ADF** as the default for **batch** (not low-latency integration).
- You should treat **APIM** as the default **API governance boundary**, even for internal APIs when standardization matters.

1.24.2 APIM vs direct ingress (when to put a gateway in front)

1.24.2.1 Recommendation

- Use **APIM** when you need any of: consistent authentication/authorization, throttling/quotas, transformations, versioning, centralized observability, or a managed API lifecycle.
- Use **direct ingress** only when the client set is tightly controlled and gateway features add little value (and you still meet security baseline).

Decision factor	Prefer APIM	Prefer direct ingress
AuthN/AuthZ standardization	Yes	Only if backend already enforces consistently
Throttling/quotas	Yes	Limited / backend-specific
API catalog + onboarding	Yes	No
Zero-trust segmentation	Often	Sometimes (if no shared gateway is allowed)

> **Recommendation:** You should use APIM policies for edge concerns (JWT validation, required headers, request size limits, optional schema validation where feasible, throttling, correlation header propagation). You should keep business validation and orchestration in the backend.

1.24.2.2 Key takeaways

- You should place policy and governance at **APIM**, not in every backend.
- You should avoid pushing complex business validation into APIM policies.
- You should validate **Private Link / network mode** capabilities per APIM SKU/region early.

1.24.3 Service Bus vs Event Grid (commands vs notifications)

1.24.3.1 Use Service Bus when

- You need **durability** and operator control (DLQ triage/replay).
- You need competing consumers on a **queue** or independent consumers via **topic/subscription**.
- You need enterprise messaging controls (sessions, TTL, lock management).

1.24.3.2 Use Event Grid when

- You need **reactive fan-out** notifications and lightweight routing/filtering.
- You want push delivery to handlers with retry and dead-lettering **where supported** by topic type/subscription configuration.

> **Warning:** You should not use **Event Grid** to deliver **commands** that must be processed exactly once. Delivery is typically **at-least-once**; handlers must be **idempotent**.

1.24.3.3 Key takeaways

- You should model **commands/work** as **Service Bus** messages; model **events** as **Event Grid** notifications.
- You should assume **at-least-once delivery** for both; design handlers to be **idempotent**.
- You should plan DLQ/dead-letter operational runbooks (triage, replay, archive) upfront.

1.24.4 Logic Apps vs Functions (orchestrator vs compute)

1.24.4.1 Definitions (consistent roles)

- **Orchestrator**: coordinates steps and state, handles retries/compensation, and manages long-running work.
- **Worker**: executes side effects and CPU-heavy work, scales for throughput, and stays stateless where possible.

Decision factor	Prefer Logic Apps	Prefer Functions
Long-running orchestration	Yes	Only with additional state patterns
Connector-driven integration	Yes	No (unless you wrap SDKs)
High-throughput custom compute	Not ideal	Yes
Fine-grained runtime control	Standard is better; connector-dependent	Yes (and/or Container Apps)

> **Recommendation:** You should use **Logic Apps** as the **orchestrator** and offload compute-heavy work to **Functions** (or another worker host), using **Service Bus** for durable handoff where needed.

1.24.4.2 Key takeaways

- You should keep orchestration and state in **Logic Apps**; keep compute in **workers**.
- You should validate connector networking paths early (some connectors are not fully private end-to-end).
- You should standardize correlation propagation across workflow actions and worker logs.

1.24.5 ADF vs workflow-based movement (batch vs transactional orchestration)

1.24.5.1 Use ADF when

- You need scheduled/batch pipelines, structured ETL/ELT orchestration, or integration runtimes for controlled data movement.

1.24.5.2 Not ideal when

- You need request/response APIs, near-real-time transactional workflows, or event streaming semantics.

  Note: You should treat ADF reruns as normal operations and design sinks to be idempotent (e.g., upserts, partition overwrite rules, dedupe keys).

1.24.5.3 Key takeaways

- You should use **ADF** for batch; pair it with events/messaging only for notifications and downstream coordination.
- You should design data loads to tolerate retries and reruns without data corruption.
- You should align SHIR/network placement with private connectivity requirements.

1.24.6 Anti-patterns (what to stop in design reviews)

- **Synchronous chaining across multiple systems** (fragile latency coupling and cascading failures).
- **Using events for commands** (notifications are not durable work items).
- **No idempotency** in consumers/handlers (breaks under retries, DR, and at-least-once delivery).
- **Public endpoints by default** when private access is feasible (creates avoidable exposure and compliance friction).
- **No DLQ/dead-letter runbook** (messages pile up; no operational recovery path).

1.24.6.1 Key takeaways

- You should treat retries as guaranteed and duplicates as normal.
- You should use the right primitive: **API** for queries/commands, **Service Bus** for durable work, **Event Grid** for notifications.
- You should require an explicit operations story (DLQ, replay, dashboards, alerts) before go-live.

1.24.7 Solution review checklist (concise)

1.24.7.1 Security

- Entra ID authentication is enforced; workloads use **Managed Identity** (no embedded secrets).
- **Private Endpoint (Private Link)** is used **where supported**; exceptions are documented with compensating controls.
- Key material/secrets are in **Key Vault**; diagnostics do not leak sensitive payloads.

1.24.7.2 Resilience + DR

- **RTO/RPO** are stated; failover triggers and runbooks are documented and tested.
- Messaging/event handling is **idempotent**; replay/reprocess procedures exist (including DLQ and dead-letter handling).
- Private connectivity DR is planned (regional private endpoints + private DNS behavior).

1.24.7.3 Operations

- Correlation is standardized: **W3C** `traceparent` plus a business **correlation ID** propagated across APIs/messages/events/pipelines.
- Alerts exist for minimum signals per component (gateway errors/latency, queue depth/DLQ, workflow failures, pipeline failures).
- Ownership is clear for DLQ triage, replay, and incident response.

1.24.7.4 Cost + governance

- SKU choices are justified (networking features, throughput, DR needs); scale limits are known.

- Landing Zone policies (tags, diagnostics, network controls) are applied consistently.

  Recommendation: You should treat this checklist as a gate for architecture approval; service-specific deep dives remain the source of truth for knobs and limits (APIM/Logic Apps/Service Bus/Event Grid/ADF sections).

1.24.7.5 Key takeaways

- You should not approve integrations without: RTO/RPO, private access stance, idempotency, and DLQ/dead-letter runbooks.
- You should standardize correlation and diagnostics before production to avoid “blind” incidents.
- You should document SKU/region constraints early because they can force architecture changes late.

1.25 Appendices: terminology, constraints, and checklists

1.25.1 What this section is for

This appendix centralizes reusable definitions and operational checklists so you can apply consistent patterns across AIS (Azure Integration Services) solutions and reduce drift between teams.

1.25.1.1 Key takeaways

- You should treat this appendix as the **single source of truth** for **baseline constraints** and **checklists**.
- You should link to these checklists from delivery plans, architecture decisions, and go-live gates.
- You should standardize terminology here to keep diagrams, policies, and runbooks consistent.

1.25.2 How to navigate this appendix

- **A. Glossary and acronyms**: shared language for architects, security, and operations.

- **B. Baseline constraints and assumptions**: enterprise defaults you should design to unless an approved exception exists.
- **C. Checklists**: practical gates for onboarding and go-live readiness.

  Recommendation: You should reference these checklists from the decision matrix section (s17) and from service deep dives (s04–s08) to keep “decision” and “execution” aligned.

1.25.2.1 Key takeaways

- You should read **B** first if you are starting a new design.
- You should use **C** as delivery gates (DoD) and for DR drills.
- You should treat exceptions as explicit, documented decisions with compensating controls.

1.26 A. Glossary and acronyms

1.26.1 Terminology normalization (use these terms consistently)

- **Microsoft Entra ID (Azure AD)**: Use this full term on first mention; use **Entra ID** thereafter. Reserve “Azure AD” only when quoting legacy UI labels.
- **Private Link / Private Endpoint**: **Private Link** is the Azure capability; **Private Endpoint** is the NIC you deploy into a VNet to consume that capability. Prefer “**Private Endpoint (Private Link)**” on first mention per section; then “Private Endpoint”.
- **correlation ID**: Use **“correlation ID”** casing consistently.
- **Authentication/authorization**: Prefer full terms over “authn/authz” in architecture guidance.

1.26.1.1 Key takeaways

- You should standardize on one identity term (**Entra ID**) to avoid policy and documentation ambiguity.
- You should distinguish **Private Link** (service offering) vs **Private Endpoint** (your deployed resource).
- You should standardize “correlation ID” casing to prevent telemetry fragmentation.

1.26.2 Glossary (AIS-focused)

Term	Definition
AIS (Azure Integration Services)	A set of Azure services used to build integration solutions across applications, data, and systems, typically including API Management, Logic Apps, Service Bus, Event Grid, and Data Factory.
APIM (Azure API Management)	Azure service for publishing, securing, transforming, and monitoring APIs via a gateway and management plane.
Logic Apps	Azure workflow service for orchestrating processes using triggers/actions and connectors; supports long-running and event-driven workflows.
Service Bus	Enterprise messaging service providing durable queues and topics/subscriptions with advanced messaging features such as sessions and dead-letter queues.
Event Grid	Event routing service for pub/sub event notifications from Azure services and custom publishers to subscribers with filtering and retry.
Data Factory (ADF)	Cloud ETL/ELT and data movement service using pipelines, activities, and integration runtimes for batch-oriented integration.
Event Hubs	High-throughput event ingestion and streaming service designed for telemetry and large-scale event streams with replay/retention semantics.
Queue	Point-to-point messaging construct where a single consumer (or competing consumers) processes each message once (with at-least-once delivery).
Topic/Subscription	Publish/subscribe messaging construct where messages sent to a topic are copied to one or more subscriptions for independent consumption.
Dead-letter queue (DLQ)	A sub-queue that stores messages that cannot be delivered or processed successfully after retries or validation.
Idempotency	A property of an operation where processing the same request/message multiple times results in the same outcome, important for at-least-once delivery systems.
Managed Identity	An Entra ID identity automatically managed by Azure for a workload to access Azure resources without storing secrets in code.

Term	Definition
Private Endpoint (Private Link)	A network interface that connects privately to a service powered by Private Link, enabling access over a private IP in a VNet.
VNet Integration	A mechanism for certain services to access resources within a VNet, influencing outbound connectivity and routing.
Correlation ID	An identifier propagated across services and messages to link logs/traces for end-to-end observability.
RTO/RPO	Recovery Time Objective and Recovery Point Objective; target time to restore service and acceptable data loss window after failure.
Landing Zone	A standardized Azure environment (subscriptions, policies, networking, identity) that provides governance and guardrails for workloads.

1.26.2.1 Key takeaways

- You should align component naming in diagrams and code to these glossary terms.
- You should treat Event Hubs as **adjacent/commonly paired** with AIS when streaming is required.
- You should assume **at-least-once** delivery semantics unless you have explicitly proven otherwise.

1.27 B. Baseline constraints and assumptions (authoritative)

1.27.1 Identity and access (Entra ID)

- You should use **Microsoft Entra ID (Azure AD)** for workforce and workload identity.
- You should prefer **Managed Identity** for workload-to-workload access (secretless authentication).
- You should enforce **least privilege** using RBAC and resource-scoped permissions.

  Note: Entra ID governs identity and access; **Azure Resource Manager** is the control plane for resource deployment and policy enforcement.

1.27.1.1 Key takeaways

- You should design for secretless access by default (Managed Identity).
- You should treat RBAC scope and separation of duties as baseline, not an enhancement.
- You should align identity choices to audit requirements (who/what accessed what, when).

1.27.2 Networking (private access first)

- You should prefer **Private Endpoint (Private Link)** for AIS data-plane connectivity **where supported**.
- You should disable public network access **where feasible**; when not feasible, you should document an exception with compensating controls (data classification, firewall allowlists, TLS requirements, monitoring).
- You should treat **DNS** as critical path for Private Endpoints:
    - private DNS zones, VNet links, and on-prem conditional forwarding must be designed and tested.
- You should validate private access feasibility early because it **depends on topic type, publish path, handler type, and region/SKU**.

  Warning: Private Endpoint support and “public network disable” capabilities can vary by **service, SKU, region, and connector/endpoint type**. You should validate both **management-plane vs data-plane** private access per service in your target regions.

1.27.2.1 Key takeaways

- You should plan Private Endpoint + DNS as first-class architecture, including DR.
- You should treat “public access required” as an exception that must be justified and controlled.
- You should validate SKU/region support early to avoid redesign.

1.27.3 Reliability and DR (multi-region by default)

- You should design for **multi-region DR** and explicitly state **RTO/RPO** for each integration.
- You should assume **at-least-once delivery** and require **idempotent** consumers/handlers.
- You should create runbooks for:
    - DLQ triage and replay (Service Bus)

    - event reprocessing strategies (Event Grid handlers, Event Hubs consumers)
    - pipeline reruns and data reconciliation (ADF)

  Recommendation: You should perform periodic DR drills that include **Private Endpoint + DNS failover behavior**, not only application failover.

1.27.3.1 Key takeaways

- You should make RTO/RPO explicit and testable.
- You should design idempotency as a requirement, not a best practice.
- You should treat DNS/private connectivity as part of DR readiness.

1.27.4 Observability and audit (centralized)

- You should centralize logs/metrics to **Azure Monitor / Log Analytics** to meet enterprise audit requirements.
- You should standardize trace propagation:
    - **W3C trace context**: `traceparent`
    - **Business correlation ID**: `Correlation-Id` (canonical name)

1.27.4.1 Key takeaways

- You should standardize both technical tracing (`traceparent`) and business correlation (`Correlation-Id`).
- You should centralize telemetry to enable audit and incident response.
- You should treat missing correlation propagation as a production defect.

1.28 C. Conventions: correlation & idempotency (reusable)

1.28.1 Correlation conventions (minimum standard)

You should propagate both:

- `traceparent` (W3C trace context)
- `Correlation-Id` (business correlation ID)

Mapping guidance (typical):

- **APIM**: require/forward `traceparent` and `Correlation-Id` headers.
- **Service Bus**: store correlation in message application properties (and/or `CorrelationId` field if used); keep `traceparent` in properties.
- **Event Grid**: include `Correlation-Id` in event `data` (and propagate `traceparent` if you control the publisher); handlers must forward.
- **ADF**: pass `Correlation-Id` as pipeline parameters and into downstream activities/logging.

  Recommendation: You should treat `traceparent` as the distributed tracing standard and `Correlation-Id` as the business-level join key; use both because they solve different problems.

1.28.1.1 Key takeaways

- You should adopt one canonical header/property name: `Correlation-Id`.
- You should ensure every hop preserves correlation for audit and debugging.
- You should document service-specific metadata mapping at design time.

1.28.2 Idempotency conventions (minimum standard)

- You should assume retries and duplicates for: **Service Bus**, **Event Grid**, **Event Hubs**, and workflow/pipeline reruns.
- You should implement idempotency using a durable dedupe key strategy (examples):
  - message ID + producer ID
  - business key + event type + version
  - command ID stored in a state store (e.g., transactional DB table)

1.28.2.1 Key takeaways

- You should design idempotency into handlers and workers, not into transport assumptions.
- You should define dedupe keys as part of the API/event contract.
- You should include replay procedures in runbooks.

1.29 D. Checklists

1.29.1 C-API: API onboarding checklist (APIM)

- **Contract and governance**
  - API has an owner, SLO/SLA target, and versioning policy.
  - Backend dependencies and data classifications are documented.
- **Edge controls (APIM policy)**
  - Authentication/authorization enforced (JWT validation where applicable).
  - Required headers enforced (including `Correlation-Id`; `traceparent` forwarded).
  - Request limits configured (size, rate limits/quotas); error mapping standardized.
- **Networking**
  - Private Endpoint to backends **where supported**; exceptions documented and approved.
  - DNS/private zones validated for all consuming networks (Azure + on-prem if applicable).
- **Operations**
  - Diagnostics enabled; alerts for 4xx/5xx and latency; runbooks exist.

1.29.1.1 Key takeaways

- You should keep APIM focused on boundary controls and governance, not business logic.
- You should treat private backend connectivity as the default.
- You should require correlation headers at the edge to make tracing reliable.

1.29.2 C-EVENT: Event publishing checklist (Event Grid)

- **Event contract**
  - Event schema documented (including versioning); `Correlation-Id` included in event data.
  - Producers define idempotency/dedupe expectations for consumers.
- **Routing**
  - One subscription per endpoint; filters defined and tested.
  - Dead-lettering configured **where supported by the chosen subscription/endpoint type**.

- **Security**
  - Private access validated per publish path and handler type; exceptions documented.
  - Subscriber endpoints authenticated (avoid anonymous webhooks).
- **Operations**
  - Alerts on delivery failures/dead-letter (where supported) and handler failure rates.
  - Replay strategy documented (Event Grid is not a log; replay is typically application-driven).

1.29.2.1 Key takeaways

- You should treat Event Grid as **notification routing**, not durable business messaging.
- You should plan for application-level replay and idempotency.
- You should validate private access per hop early (topic/source/handler/SKU/region).

1.29.3 C-MSG: Messaging checklist (Service Bus)

- **Topology**
  - Queue vs topic/subscription choice is justified; DLQ strategy defined.
  - Ordering/session needs documented (use sessions only when required).
- **Delivery and retries**
  - Max delivery count, lock duration, TTL configured per workload.
  - Consumer implements idempotency; dedupe keys defined.
- **Security and networking**
  - Private Endpoint configured where supported; public access disabled where feasible.
  - RBAC/least privilege and separation of duties enforced.
- **Operations**
  - DLQ triage + replay runbooks exist; alerts on active messages, DLQ length, throttling.

1.29.3.1 Key takeaways

- You should use Service Bus for durable commands/work coordination with operational control (DLQ).
- You should implement idempotent consumers because delivery is typically at-least-once.

- You should operationalize DLQ handling as a first-class process.

1.29.4 C-WF: Workflow checklist (Logic Apps)

- **Design**
  - Workflow scope and state model defined; compensation strategy documented if needed.
  - Connectors used are approved and reviewed for data handling.
- **Networking**
  - Private networking is validated for each connector/target; exceptions documented.
- **Security**
  - Managed Identity used where possible; secrets stored in Key Vault if required.
  - Run history data exposure reviewed (PII/PHI considerations).
- **Operations**
  - Alerts for failed runs and excessive retries; correlation propagated to downstream calls/messages.

1.29.4.1 Key takeaways

- You should use Logic Apps for orchestration and long-running workflows, not compute-heavy work.
- You should treat connector networking as a design dependency (often the limiting factor).
- You should review run history exposure as a security control.

1.29.5 C-BATCH: Batch pipeline checklist (ADF)

- **Data movement**
  - Sources/sinks documented; schema drift and rerun behavior addressed.
  - Integration Runtime placement chosen (managed vs self-hosted) based on connectivity.
- **Security**
  - Private Endpoint used where supported for data stores; keys/secrets handled via Key Vault.
  - Data exfiltration controls and approvals documented for any public endpoints.

- **Operations**
  - Rerun strategy and reconciliation checks defined; alerts on pipeline failures and duration anomalies.

1.29.5.1 Key takeaways

- You should design for safe reruns and reconciliation (batch is retried operationally).
- You should choose IR placement based on real network paths, not assumptions.
- You should treat ADF as orchestration/movement, not streaming or API management.

1.29.6 C-GO: Go-live readiness checklist (cross-cutting)

- **Security and compliance**
  - Threat model completed; least privilege verified; audit logging enabled.
  - Public exposure exceptions documented with compensating controls and approvals.
- **Resilience**
  - RTO/RPO documented; DR runbooks written; DR drill performed (including Private Endpoint + DNS).
- **Operations**
  - Monitoring/alerts configured; runbooks exist; on-call ownership defined.
  - Correlation (`traceparent`, `Correlation-Id`) validated end-to-end.
- **Delivery**
  - IaC pipelines in place; rollback plan and change windows defined.

1.29.6.1 Key takeaways

- You should not treat DR as “optional later”; it is part of the baseline.
- You should prove operability (alerts + runbooks + ownership) before production.
- You should validate correlation end-to-end to avoid “unobservable” integrations.