PROMYKON EDITIONS

FREE FIELD SAMPLE · VOLUME I

SYSTEM DESIGN · WORKED SCENARIO 01

Design an Enterprise Data Platform

under interview pressure.

A complete architecture conversation—from business trigger and requirements to trade-offs, evolutions and measurable outcome.

Business — Requirements — Trade-offs — Architecture — Technology — Outcome

INSIDE

- The complete 6-step reasoning framework
- Baseline architecture + interview-driven evolutions
- Follow-ups, measurable outcomes + Decision Card

FREE SAMPLE · VERSION 1.2 · JULY 2026

ARCHITECTURE
INTERVIEW SERIES ·
VOLUME I

Design an Enterprise Data Platform Under Interview Pressure

A fully worked architecture scenario — from business trigger to measurable outcome

Architecture Interview Series

Version 1.2 · July 2026

A PROMYKON EDITIONS PUBLICATION

PROMYKON EDITIONS

ARCHITECTURE
INTERVIEW SERIES ·
VOLUME I

Free Sample

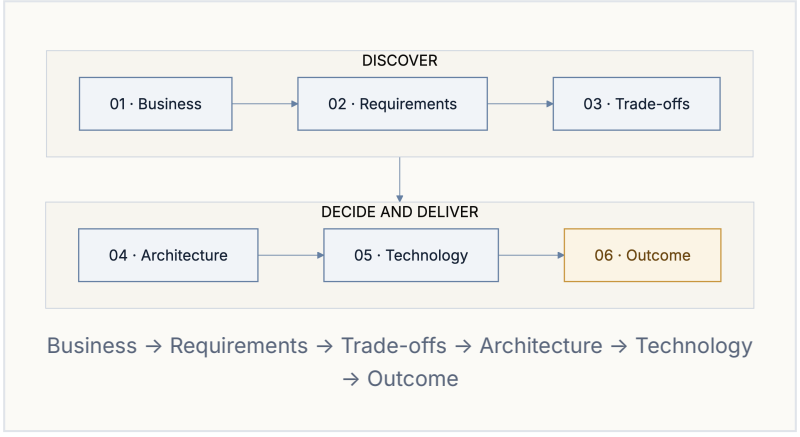
The Architecture Interview Bible
Scenario 1 · Version 1.2 · July 2026

Published by **Promykon Editions**

promykon.com/en/productos.html

ARCHITECTURE
INTERVIEW SERIES ·
VOLUME I

Scenario 1 — Internal Data Application → Enterprise Data Platform



Business → Requirements → Trade-offs → Architecture → Technology → Outcome

How to Use This Scenario in the Interview

Core rule. Start simple. Clarify requirements. State assumptions. Evolve the design only when the interviewer adds constraints. Always explain trade-offs.

1. Clarify the business goal and consumers before naming technologies.
2. Identify functional and non-functional requirements.
3. Propose the smallest viable architecture.
4. Explain why each component exists.
5. Evolve the design as scale, security, availability or integration needs change.
6. Close by summarizing trade-offs, risks and next steps.

The Opening Script to Memorize

> I'll begin with a simple architecture and evolve it as we clarify the requirements. First, I'd like to understand the consumers, data sources, access patterns, expected scale, freshness requirements, security classification and availability targets. Assuming this is initially an internal read-oriented application on Azure, I would use a web frontend, API Management as the controlled API entry point, a stateless backend service, and a data-access layer connected to the existing source. Authentication would be handled through Entra ID, secrets through Key Vault, and telemetry through Azure Monitor. I would avoid introducing microservices or event-driven components until the scale, domain boundaries or latency requirements justify the added complexity.

---

1. The System Design Conversation

The interviewer may intentionally start with an ambiguous prompt such as: *"Design an application that exposes data."* The objective is not to guess the final architecture immediately. The objective is to demonstrate

structured thinking, sound assumptions, architectural judgment and communication.

1.1 Clarifying Questions

- What data are we exposing, and where does it originate?
- Who are the consumers: internal users, mobile applications, external partners or public clients?
- Is the workload read-only, read-heavy, or transactional?
- How fresh must the data be: real time, near real time, minutes, or daily?
- What scale is expected: users, requests per second, data volume and growth?
- What availability target is required?
- What are the RTO and RPO expectations?
- Does the data contain sensitive, regulated or personally identifiable information?
- Is the target environment Azure, on-premises, or hybrid?
- Are there corporate standards or existing services that should be reused?

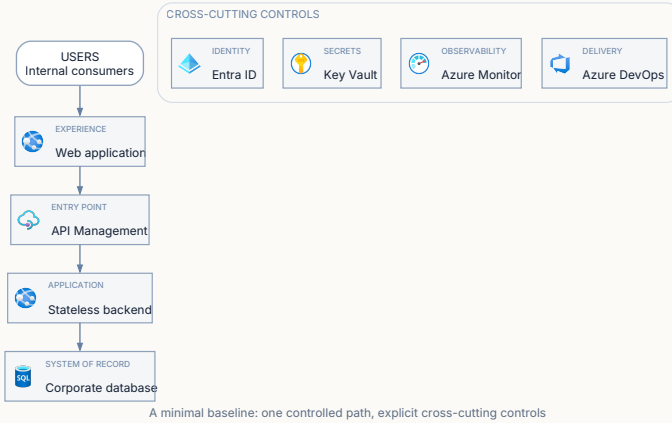
GOOD PRACTICE

> **Strong phrase:** *"Before selecting a technology, I want to understand the access pattern and operational requirements because those characteristics drive the architecture."*

---

2. Baseline Architecture: Internal Data Application

Assumption: internal web application, mostly read traffic, several thousand users, corporate database, Azure hosting, no strict real-time requirement.



Baseline architecture

ANSWER

Suggested Answer: For the initial version, I would keep the architecture intentionally simple. I'd use a web frontend consuming REST APIs through Azure API Management. The backend would contain the business logic and access the corporate data source through a controlled data-access layer. Authentication would be handled through Microsoft Entra ID. I'd add centralized monitoring, secrets management and automated deployment from the beginning.

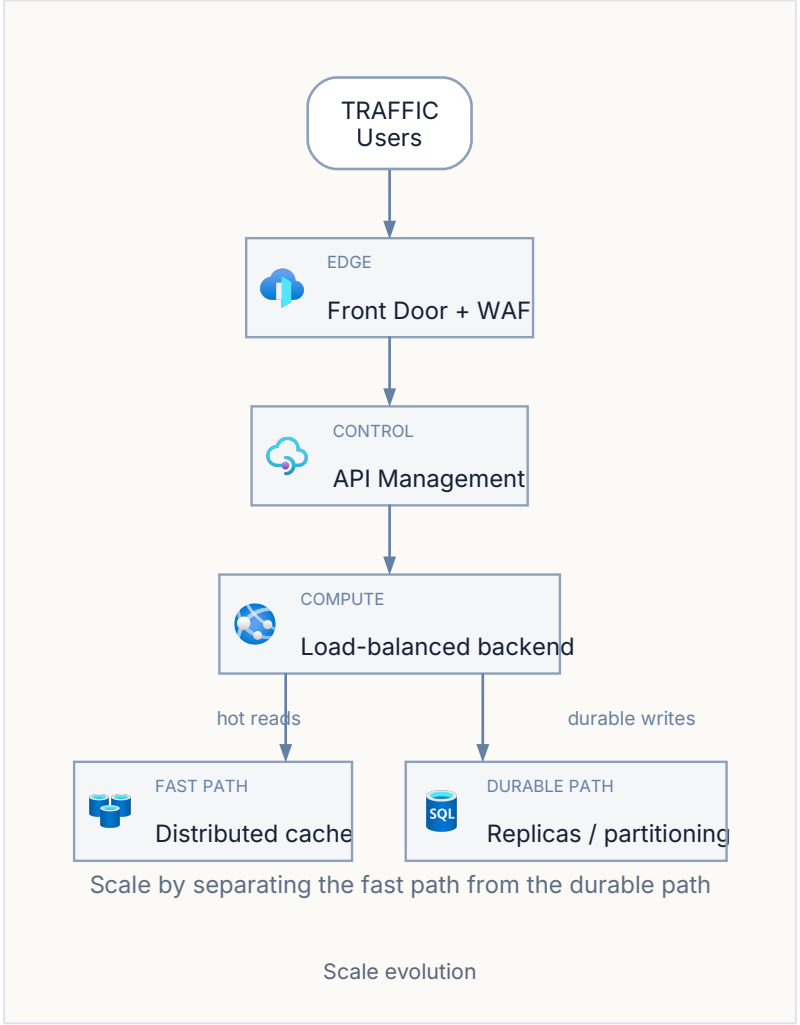
GOOD PRACTICE

> **Why This Is a Good Starting Point** - It meets the stated requirements without unnecessary complexity. - API Management provides a controlled contract and policy layer. - Stateless services can scale horizontally later. - Identity, secrets and observability are included from the beginning. - The design can evolve without prematurely adopting microservices.

---

3. Evolution — "Now It Must Scale"

Do not immediately answer "Kubernetes" or "microservices." Clarify what must scale: traffic, data volume, processing, geographic reach or team autonomy.



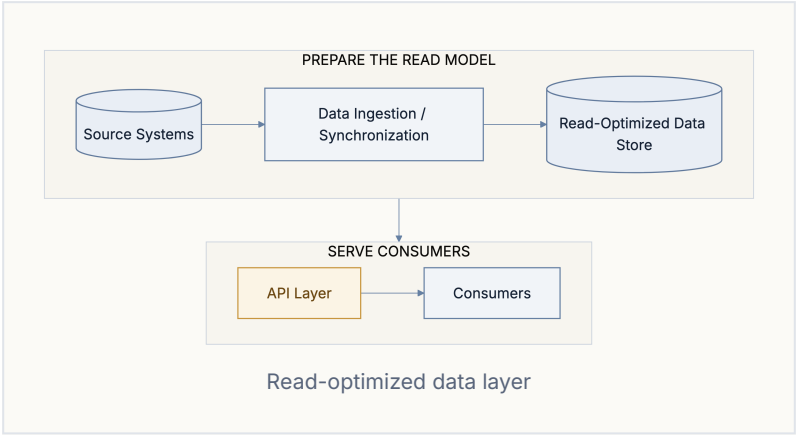
I would keep the application services stateless so they can scale horizontally. Frequently accessed and relatively stable data could be cached. At the database layer, I would review indexes, query patterns, read replicas, partitioning and connection management before introducing additional services.

INTERVIEWER SIGNAL

> **Architectural signal:** Scalability is not only about adding compute. It also includes database access patterns, caching, asynchronous processing, operational maturity and team ownership.

---

4. Evolution — "The Source Database Cannot Support Consumer Traffic"



Read-optimized data layer

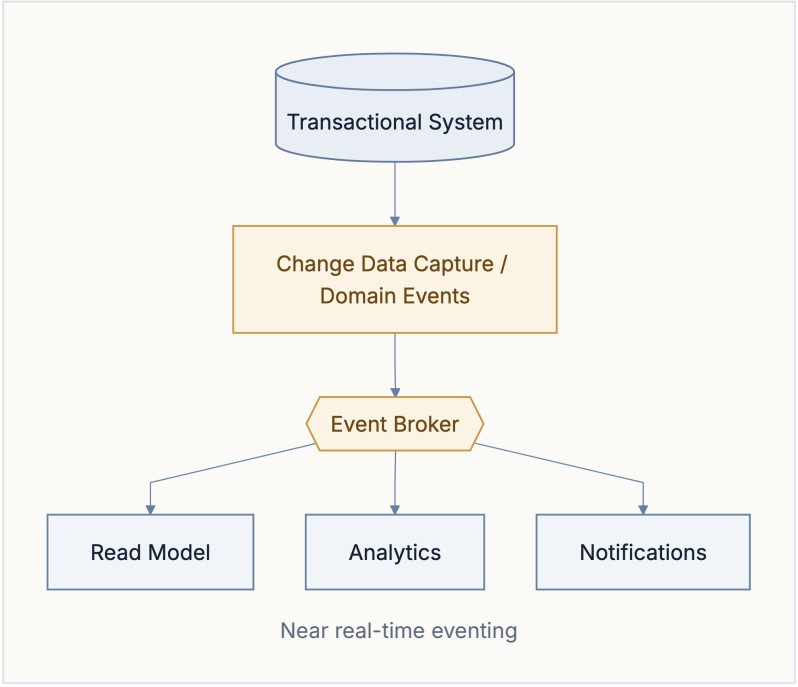
I would avoid exposing the transactional database directly to unpredictable consumer traffic. I'd introduce a read-optimized representation populated from the source system. Depending on freshness requirements, synchronization could be batch-based, change-data-capture or event-driven.

TRADE-OFF

> **Trade-off to state:** This protects the transactional system and improves scalability, but introduces data duplication and eventual consistency.

---

5. Evolution — "The Data Must Be Near Real Time"

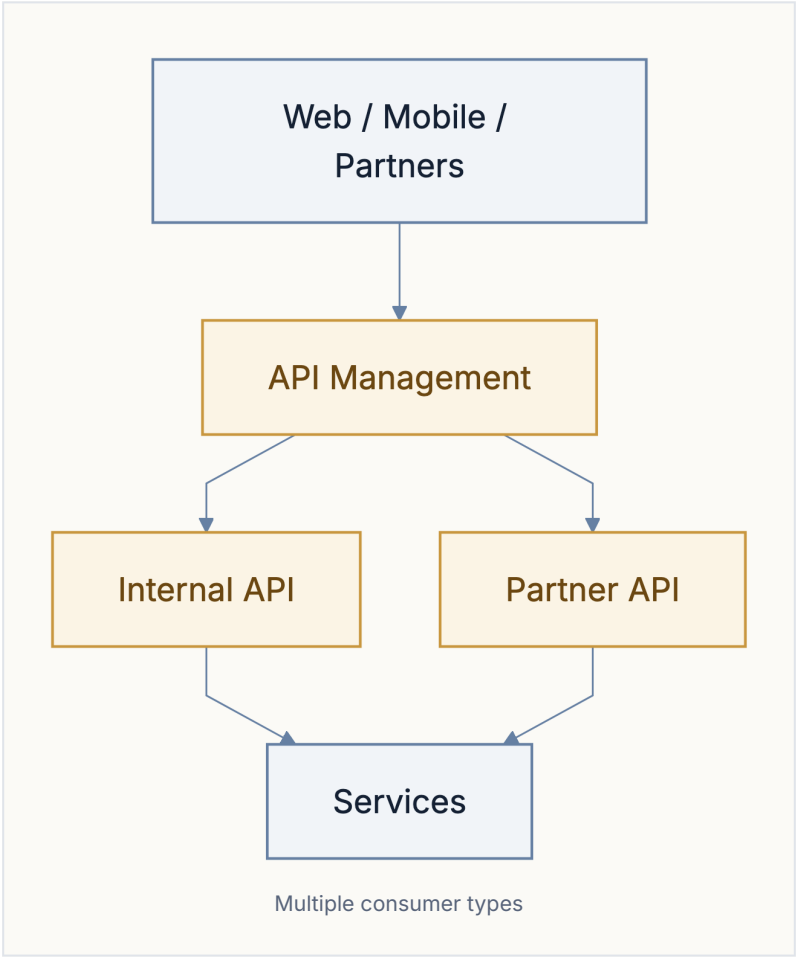


For near-real-time updates, I'd move from periodic synchronization to change-data-capture or domain events. Consumers would update their own read models asynchronously. I'd explicitly design for idempotency, retries, dead-letter handling, event ordering where required, and eventual consistency.

Azure Choices - **Azure Service Bus** — enterprise messaging, queues, topics, ordered delivery and dead-lettering. - **Azure Event Grid** — lightweight event distribution and event routing. - **Azure Event Hubs** — high-throughput streaming and telemetry ingestion.

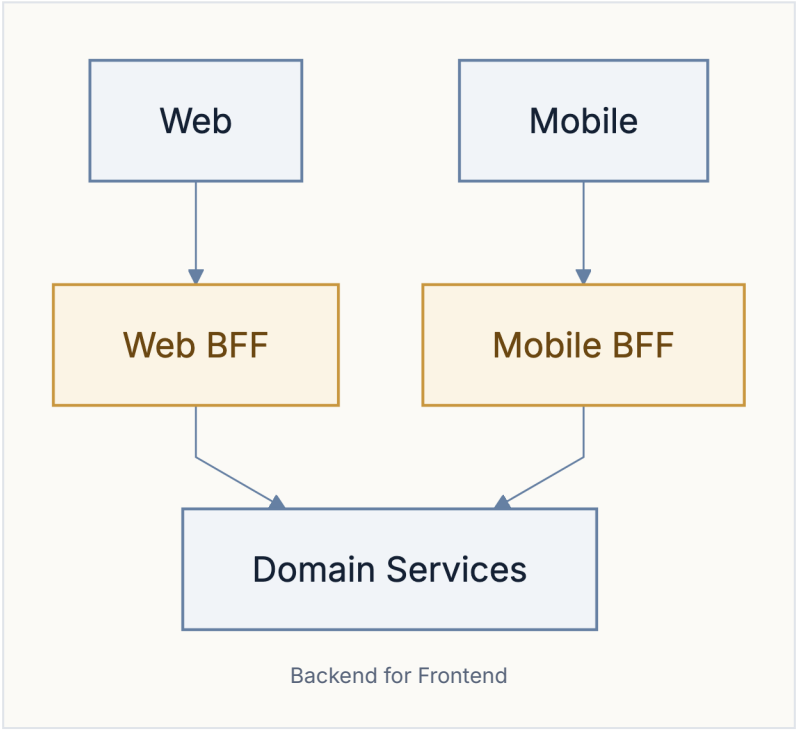
---

6. Evolution — "Multiple Consumer Types"



I'd use API Management as the external contract and policy enforcement layer. It can apply authentication, quotas, rate limits, transformations, versioning and analytics. I would avoid exposing internal service interfaces directly.

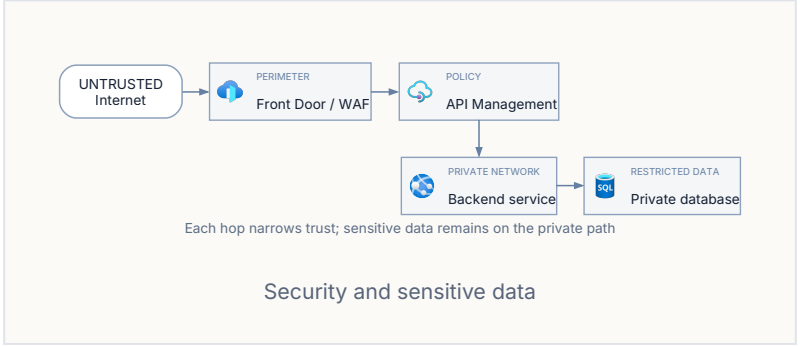
When to Introduce a BFF



Use separate Backends for Frontends only when web and mobile clients have materially different data composition, performance or orchestration needs.

---

7. Evolution — "Security and Sensitive Data"



I'd apply zero-trust principles: authenticate every request, authorize based on roles or claims, use managed identities between Azure services, store secrets in Key Vault, encrypt data in transit and at rest, and keep backend and database services on private networks.

- Entra ID with OAuth 2.0 / OpenID Connect.
- Role- and claim-based authorization.
- Managed Identity for service-to-service authentication.
- Private Endpoints and Network Security Groups.
- Key Vault for secrets, certificates and keys.
- Encryption in transit and at rest.
- Data masking, tokenization or field-level protection when required.
- Auditability and retention aligned with compliance requirements.

8. Microservices: When to Use Them and When Not To

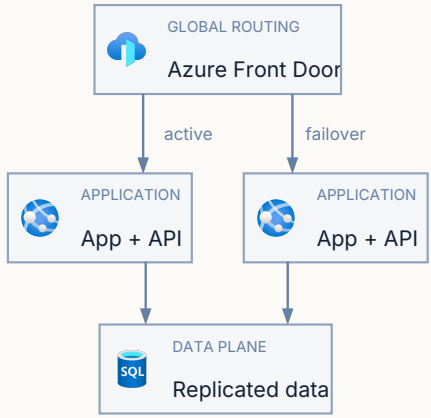
I would start with a modular service or modular monolith unless independent scaling, independent deployment or clear domain ownership justifies microservices. Microservices add network latency, distributed transactions, operational overhead and more complex observability.

Good Reasons to Split - Clear bounded contexts and ownership boundaries. - Independent deployment produces material business value. - Components have significantly different scaling characteristics. - Teams require autonomy and can own services end to end. - The organization has sufficient platform and operational maturity.

Poor Reasons to Split - Because microservices are considered modern. - Because Kubernetes is available. - Before the domain boundaries are understood. - When the team is too small to operate distributed systems effectively.

---

9. Evolution — "Global Availability and Disaster Recovery"



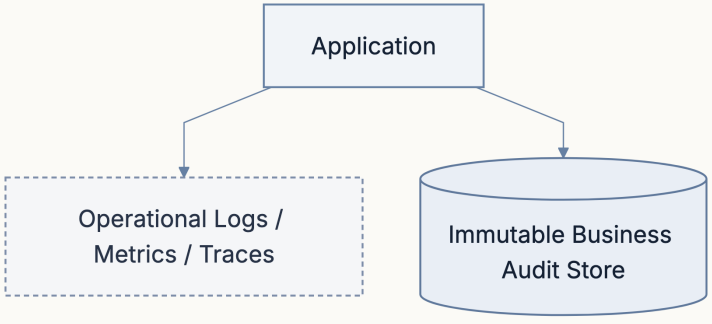
Routing, compute and data recovery are separate design decisions

Global availability and disaster recovery

First I would clarify whether the requirement is disaster recovery or active-active global operation. Those are very different cost, consistency and operational models.

- **Active-passive:** lower cost and complexity; suitable for disaster recovery.
- **Active-active:** lower latency and stronger availability; higher data and operational complexity.
- Define RTO and RPO explicitly.
- Consider data residency and regulatory constraints.
- Automate and regularly test failover procedures.

10. Business Audit vs. Technical Logging

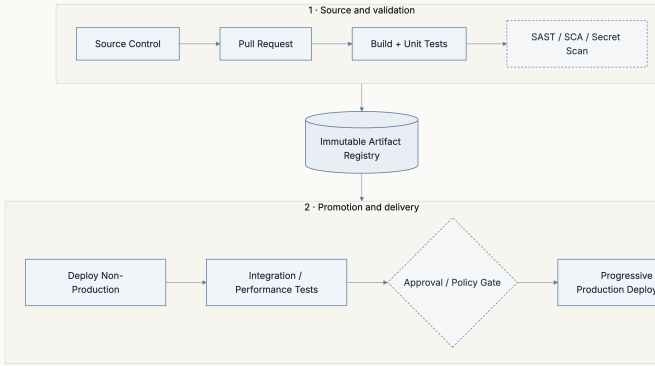


Business audit vs. technical logging

I would distinguish technical telemetry from business audit events. Application logs tell us whether the system is healthy; audit records capture who accessed which data, when, from where and under which authorization context.

- User and service identity.
- Timestamp and correlation ID.
- Resource or data accessed.
- Authorization result.
- Client/channel information.
- Retention and immutability controls.

11. CI/CD Architecture



CI/CD architecture

I would use reusable pipeline templates rather than allowing every team to reinvent its pipeline. The pipeline should produce an immutable artifact once, validate it through security and quality gates, and promote that same artifact across environments.

GOOD PRACTICE

> **Key phrase:** *"Build once, promote many."*

- Infrastructure as Code.
- Workload identity instead of long-lived credentials.
- Reusable templates and policy enforcement.
- Quality and security gates.

- Artifact signing and traceability.
- Progressive deployment and automated rollback.
- Full deployment evidence for audit purposes.

---

12. Azure Cheat Sheet for an AWS Architect

Architecture Need	AWS	Azure
Identity	IAM / Cognito	Entra ID / External ID
Compute	EC2	Azure Virtual Machines
Containers	ECS / EKS	Container Apps / AKS
Serverless	Lambda	Azure Functions
API gateway	API Gateway	API Management
Object storage	S3	Blob Storage
Container registry	ECR	Azure Container Registry
Secrets	Secrets Manager / KMS	Key Vault

Architecture Need	AWS	Azure
Monitoring	CloudWatch	Azure Monitor / Log Analytics / App Insights
Audit	CloudTrail	Azure Activity Log
Network	VPC	Virtual Network
Firewall groups	Security Groups	Network Security Groups
Global routing	CloudFront / Route 53	Front Door / Azure DNS
IaC	CloudFormation / CDK	Bicep / ARM / Terraform
Messaging	SQS / SNS	Service Bus / Event Grid
Streaming	Kinesis	Event Hubs
Relational DB	RDS / Aurora	Azure SQL / PostgreSQL Flexible Server

Azure Answers That Sound Architectural

> My strongest hands-on cloud experience is AWS, but the architectural principles are cloud-agnostic. Identity, networking, containers, observability, automation, resilience and security exist in every cloud. The main learning curve is understanding the managed service ecosystem and organizational conventions.

> If Kubernetes introduces unnecessary operational complexity, I would evaluate Azure Container Apps. If the workload requires fine-grained orchestration, custom scheduling, service mesh capabilities or greater platform control, AKS becomes a stronger option.

---

13. Architecture Framework for Any Whiteboard Question

Step	What to Cover
Business Goal	What outcome does the organization need?
Consumers	Who uses the system and through which channels?

Step	What to Cover
Data	Where does it originate, who owns it, and how fresh must it be?
Scale	Users, requests, data growth, processing and regions.
Security	Identity, authorization, classification, compliance and audit.
Availability	SLA, RTO, RPO, failure domains and recovery model.
Architecture	Start with logical components and flows.
Technology	Map components to services only after requirements are known.
Operations	Observability, CI/CD, support model and cost.
Trade-offs	State benefits, costs, risks and assumptions.

14. Interviewer Follow-Ups and Compact Answers

Question	Compact Answer
Why API Management?	To create a stable external contract and centralize authentication, authorization, rate limiting, quotas, transformations, versioning and analytics.
Why not connect directly to the database?	It couples consumers to the source schema, creates security exposure and allows unpredictable traffic to impact the transactional system.
Why use a cache?	To reduce latency and database load for frequently accessed data, provided the freshness and invalidation strategy are acceptable.
What happens when the cache is stale?	Define TTL and invalidation based on business tolerance, monitor hit ratio, and treat the source of truth as authoritative.
How do you prevent duplicate event processing?	Use idempotent consumers, stable event identifiers and deduplication or transactional outbox patterns when needed.

Question	Compact Answer
How do you handle failures?	Timeouts, bounded retries with backoff, circuit breakers where appropriate, dead-letter handling, health checks and clear operational ownership.
How do you version APIs?	Prefer backward-compatible changes; for breaking changes use explicit versioning, a deprecation period and consumer communication.
REST or GraphQL?	REST is simple and cache-friendly for stable resources. GraphQL is valuable for consumer-driven queries but adds governance, caching and complexity considerations.
Containers or serverless?	Use serverless for event-driven or bursty workloads with minimal operational needs; use containers for long-running services, custom runtimes and predictable control requirements.
How do you control cost?	Right-size services, autoscale, use budgets and tagging, monitor unit economics, and avoid architectural complexity without business value.

15. High-Value Phrases to Reuse

- Architecture is about making informed trade-offs.
- Technology should support business objectives, not drive them.
- I would begin with the simplest architecture that meets the current requirements.
- That decision depends on the workload characteristics and operational constraints.
- I would separate the external API contract from the internal implementation.
- I would avoid exposing the transactional database to unpredictable consumer traffic.
- I would design explicitly for idempotency, retries and observability.
- I would distinguish disaster recovery from active-active availability.
- I would prefer backward compatibility and controlled deprecation.
- I would build the artifact once and promote it across environments.
- I would document the decision, alternatives considered and consequences.
- I would validate the assumptions through a prototype or proof of concept where uncertainty is high.

16. Outcome — How I Would Measure Success

- Consumer traffic no longer threatens the transactional source's availability or latency target.
- API latency, availability and freshness meet explicit SLOs agreed with consumers.
- Recovery tests demonstrate the stated RTO and RPO instead of treating them as documentation only.
- Cost is visible per consumer, workload or environment, with unit economics reviewed as usage grows.
- Audit events answer business-access questions independently from technical logs.

17. Architecture Decision Card

Decision	Introduce a controlled API and read-optimized data layer, then add eventing, regional resilience and specialized consumer paths only when requirements justify them.
Why	It protects the source system while keeping the first release simple and evolvable.

Alternative rejected	Direct consumer access to the transactional database or a fully distributed platform from day one.
Trade-off accepted	Data duplication, synchronization logic and eventual consistency in exchange for isolation and scale.
Business impact	Predictable consumer access without placing the system of record at operational risk.
Interview takeaway	Start with the smallest architecture that meets current requirements and make every evolution traceable to a new constraint.

18. Final 5-Minute Review Before Joining

- Do not start with tools. Start with questions.
- State assumptions clearly.
- Keep the first architecture simple.
- Add complexity only when the interviewer introduces a requirement.
- Always state at least one trade-off.
- Include security, observability and delivery.

- Explain why each component exists.
- Anchor claims in real evidence — production migrations, platform rollouts, vendor assessments.
- Be honest about depth gaps while confidently mapping cloud principles across providers.
- End by summarizing the architecture, risks and next validation steps.

NOTE

> **Last reminder:** They are evaluating how you think and communicate, not whether you remember every Azure product name. Stay calm, clarify the problem and guide the conversation.

Version History

Version	Date	Scope	Status
1.0	2026-07-17	System-design interview flow, evolving Azure architecture, CI/CD and Azure cheat sheet	Interview-ready
1.1	2026-07-29	Adds repeated framework, measurable Outcomes and Architecture Decision Card	Interview-ready

ARCHITECTURE
INTERVIEW SERIES ·
VOLUME I

Get the Full Early Access Edition — Version 1.2

This was Scenario 1 of 7 in the current Early Access edition — the same depth, applied to CI/CD platform migrations, AI coding assistant governance, developer portals, technology evaluation, multi-region rollouts, and container supply chain security.

Also included: the full 6-step framework (Chapter 3), a live interviewer-curveball exchange, and a calibrated-confidence playbook for when you don't have the story.

Get the Early Access Edition → promykon.com/book

Your purchase includes all future PDF and EPUB updates released for *The Architecture Interview Bible, Volume I*, delivered through your original Lemon Squeezy order at no additional charge. It does not include future volumes, print editions, translations, courses, coaching, templates,

communities or other standalone products. No specific update schedule or final page count is guaranteed.

A Promykon Editions Publication

Promykon Editions publishes field guides for technical leadership: practical resources for engineers and architects navigating interviews, platform decisions, cloud systems and AI adoption.

Explore the catalog: promykon.com/en/productos.html