SCHEMA, CONTRACT & EVOLUTION PROBLEMS

Real Interview Scenarios & How to Handle Them

A practical guide for Data Engineers to answer

real-world schema, contract, and evolution questions with confidence

By Ankita Gulati

Senior Data Engineer | Interview Mentor

Interview Edition • Practical • Real Scenarios

Table Of Content

Scenario 1

Adding a Non-Optional Field to an Existing Schema

Problem Statement

An **upstream producer adds a new non-optional field** to an existing dataset. While new records include the field, **historical records do not,** causing **ETL pipelines to fail** during processing. The **SLA is 1 hour, historical data is business-critical**, and **multiple downstream pipelines depend on this dataset.**

Key Details

- New non-optional field added upstream
- Historical records missing the field
- ETL jobs failing on old data
- Multiple downstream dependencies
- SLA: 1 hour

Expected vs Actual Behavior

Expected	Actual
Schema changes are backward-compatible	Historical data breaks pipelines
ETL processes old and new data	ETL fails on missing field
Downstream pipelines remain stable	Cascading failures downstream
SLA met	SLA at risk

This is a **schema evolution and backward compatibility failure**, not a data quality issue.

Why This Problem Is High Risk

Because:

- Schema changes propagate instantly across systems
- Historical data cannot be retroactively fixed easily
- Downstream pipelines fail even without code changes
- Business impact multiplies with each dependency

Common but dangerous reactions:

- Rolling back producer changes under pressure
- Skipping historical records
- Ignoring failures to meet SLA

But **schema changes should never break existing consumers.**

Clarifying Questions

A senior data engineer asks:

- Is the new field truly required for business logic?
- How much historical data lacks the field?
- Are downstream consumers expecting this field?
- Can the ETL handle default or null values?
- Is schema versioning or compatibility enforced?

These questions focus on **stability over speed.**

Confirmed Facts & Assumptions

After investigation:

- Only new records contain the new field
- Historical records cannot be re-emitted quickly
- Rolling back producer delays upstream roadmap
- Skipping data violates data completeness
- ETL can be made backward-compatible

Interpretation:
This is a **consumer-resilience issue**, not a producer emergency.

What Teams Often Assume vs Reality

Assumption	**Reality**
Non-optional fields are safe	They break history
Producer rollback is fastest	It delays multiple teams
Skipping old data is acceptable	It causes silent data loss
Consumers should adapt later	Consumers must be resilient now

Senior engineers **design ETL to survive schema evolution.**

Root Cause Analysis

Step 1: Inspect Failure Pattern

Observed:

- ETL fails only on historical partitions
- Failures correspond to missing field

Conclusion:
The schema change is incompatible with historical data.

Step 2: Evaluate Compatibility Strategy

Observed:

- No default or null handling in ETL
- Field marked as non-optional

This confirms **lack of backward compatibility handling.**

Step 3: Conceptual Root Cause

The root cause is **introducing a non-optional field without consumer safety:**

- Producer change assumed uniform data
- Consumer pipelines not hardened

This is a **schema governance gap.**

Step 4 : Wrong Approach vs Right Approach

Wrong Approach

- Roll back producer schema
- Ignore ETL failures
- Skip historical records

Right Approach

- Update ETL to handle missing fields gracefully
- Treat new field as optional for historical data
- Apply defaults or null-safe logic

Senior engineers **absorb schema changes at consumption boundaries.**

Step 5 : Validation of the Fix

To validate:

- Update ETL to allow null/default for new field
- Re-run historical partitions
- Confirm downstream pipelines succeed
- Verify no data loss or duplication

Outcome:
Pipelines stabilize within SLA without upstream rollback.

Step 6 : Corrective Actions

- Enforce backward-compatible schema changes
- Add schema evolution tests in ETL
- Treat new fields as optional initially
- Document producer–consumer contracts
- Use schema registries with compatibility rules

These steps prevent **future schema-breaking incidents.**

Step 7 : Result After Fix

Before	After
ETL failures	Stable pipelines
Downstream breakage	Downstream continuity
SLA risk	SLA met
Reactive rollback	Forward-compatible design

Final Resolution

- **Root Issue:** Non-optional schema field broke historical data processing
- **Action Taken:** Updated ETL to handle missing values safely

Key Learnings

- Schema evolution is inevitable
- Backward compatibility is a consumer responsibility
- Historical data must always be considered
- Stability beats speed in distributed systems

Core Principle Reinforced

Schema changes should evolve systems—not break them.

Scenario 2

Column Renaming Without Deprecation

Problem Statement

An **upstream producer renames a column** in a dataset without providing any deprecation window or backward compatibility. The ETL pipeline does not fail outright, but **downstream aggregations silently break**, resulting in **incorrect metrics on business dashboards.** The **SLA is 2 hours**, metrics are **business-critical, and multiple consumers rely on this column.**

Key Details

- Column renamed upstream without deprecation
- ETL succeeds but aggregations break silently
- Dashboards show incorrect metrics
- Multiple downstream consumers affected
- SLA: 2 hours

Expected vs Actual Behavior

Expected	Actual
Schema changes communicated safely	Column renamed abruptly
Downstream metrics remain accurate	Metrics silently incorrect
Failures are visible	Errors go undetected
SLA protects correctness	SLA technically met, trust broken

This is a **silent data correctness failure,** which is often **more dangerous than hard failures.**

Why This Problem Is High Risk

Because:

- Silent failures are harder to detect than crashes
- Dashboards continue to update with wrong numbers
- Business decisions are made on incorrect data
- Trust erosion happens Before alerts fire

Common but risky reactions:

- Ignoring the issue because jobs didn't fail
- Rolling back upstream under pressure
- Skipping downstream validations

But **silent data corruption is worse than downtime.**

Clarifying Questions

A senior data engineer asks:

- Which column was renamed and where is it used?
- Did aggregations default to nulls or zero?
- Are there validation checks for metric sanity?
- How many consumers depend on this column?
- Is schema evolution governed or ad hoc?

These questions focus on **impact and detection,** not blame.

Confirmed Facts & Assumptions

After investigation:

- Old column name no longer exists upstream
- ETL does not error on missing column
- Aggregations compute on null/empty values
- Dashboards show incorrect but "valid" numbers
- Upstream rollback would delay other teams

Interpretation:
This is a **backward compatibility and observability gap**, not a compute issue.

What Teams Often Assume vs Reality

Assumption	Reality
Renaming is harmless	It breaks consumers
Pipelines will fail loudly	Many fail silently
Upstream rollback is easiest	It blocks progress
Dashboards will catch issues	They amplify them

Senior engineers **design for schema evolution, not ideal behavior.**

Root Cause Analysis

Step 1: Identify Silent Failure Pattern

Observed:

- Jobs succeed
- Aggregations output incorrect results

Conclusion:
Failures are logical, not technical.

Step 2: Inspect ETL Column Handling

Observed:

- ETL references old column name
- Missing column resolves to nulls

This confirms **lack of schema mapping.**

Step 3: Conceptual Root Cause

The root cause is **renaming without backward-compatible mapping**:

- Producer changed schema
- Consumers were not protected

This is a **schema governance failure**.

Step 4 : Wrong Approach vs Right Approach

Wrong Approach

- Ignore changes because jobs “run”
- Roll back upstream schema
- Skip failed aggregations

Right Approach

- Map old column name to new one in ETL
- Maintain backward compatibility
- Gradually deprecate old references

Senior engineers **absorb schema changes at consumption boundaries.**

Step 5 : Validation of the Fix

To validate:

- Add column mapping in ETL
- Recompute affected aggregations
- Compare metrics Before/After rename
- Confirm dashboards reflect correct values

Outcome:
Metrics corrected within SLA without upstream rollback.

Step 6 : Corrective Actions

- Enforce deprecation periods for renames
- Add column aliasing in ETL layers
- Implement metric sanity checks
- Document producer–consumer contracts
- Use schema registries with compatibility rules

These steps prevent **future silent data failures.**

Step 7 : Result After Fix

Before	After
Silent metric corruption	Accurate dashboards
Business confusion	Restored trust
Hidden failures	Visible, managed changes
Reactive firefighting	Controlled evolution

Final Resolution

- **Root Issue:** Column renamed without backward compatibility
- **Action Taken:** Mapped old column to new name in ETL

Key Learnings

- Silent failures are the most dangerous
- Schema evolution must protect consumers
- ETL should be resilient to upstream changes
- Correctness matters more than job success

Core Principle Reinforced

If a schema change doesn't break loudly, it must be handled carefully.

Scenario 3

Incompatible Kafka Schema Change

Problem Statement

An upstream **Kafka producer publishes Avro messages** with a **schema change that is incompatible** with existing consumers. As a result, **multiple consumer applications fail to deserialize messages,** causing **pipeline outages and data integrity risk**. The **SLA is 1 hour,** several consumers depend on the topic, and **data correctness is critical.**

Key Details

- Producer publishes incompatible Avro schema
- Existing consumers fail to deserialize
- Multiple downstream consumers affected
- Data integrity is critical
- SLA: 1 hour

Expected vs Actual Behavior

Expected	Actual
Schema evolution is safe	Incompatible schema pushed
Consumers continue processing	Consumers fail immediately
Data integrity preserved	Data loss / backlog risk
SLA protected	SLA breached

This is a **schema compatibility failure at the messaging layer**, not an ETL bug.

Why This Problem Is High Risk

Because:

- Kafka decouples producers and consumers
- Schema incompatibility breaks that contract
- Failures cascade across multiple services
- Stopped consumers can cause data loss or backlog

Common but dangerous reactions:

- Restarting consumers repeatedly
- Retrying ETL jobs
- Stopping producers under pressure

But **schema violations must be fixed at the source.**

Clarifying Questions

A senior data engineer asks:

- What compatibility mode is enforced in Schema Registry?
- Which fields were added, removed, or changed?
- Are consumers backward- or forward-compatible?
- Did the producer bypass schema validation?
- Can the schema be fixed without data loss?

These questions focus on **contract enforcement,** not firefighting.

Confirmed Facts & Assumptions

After investigation:

- New schema is incompatible with previous version
- Schema Registry compatibility checks were skipped or misconfigured
- Consumers expect the older schema
- Retrying consumers does not help
- Fixing schema compatibility is feasible quickly

Interpretation:
This is a **governance and validation gap**, not a scaling issue.

What Teams Often Assume vs Reality

Assumption	Reality
Consumers will adapt	They fail immediately
Restarting fixes the issue	Schema mismatch persists
Stopping producers is safest	It disrupts upstream systems
Compatibility checks are optional	They are mandatory

Senior engineers **treat schemas as contracts, not suggestions**.

Root Cause Analysis

Step 1: Inspect Consumer Failures

Observed:

- Deserialization exceptions across consumers
- Failures start exactly at schema deployment time

Conclusion:
Schema incompatibility is the trigger.

Step 2: Review Schema Registry Configuration

Observed:

- Compatibility mode not enforced
- Producer allowed to register incompatible schema

This confirms **missing schema governance.**

Step 3: Conceptual Root Cause

The root cause is **pushing incompatible schemas without validation:**

- Producer change not backward-compatible
- Consumers unprotected

This is a **contract enforcement failure.**

Step 4 : Wrong Approach vs Right Approach

Wrong Approach

- Ignore failures
- Retry consumers or ETL jobs
- Stop producers blindly

Right Approach

- Fix schema to maintain backward compatibility
- Enforce Schema Registry compatibility rules
- Redeploy producer safely

Senior engineers **fix contracts, not symptoms.**

Step 5 : Validation of the Fix

To validate:

- Register corrected, compatible schema
- Restart consumers
- Verify successful deserialization
- Confirm no message loss or duplication

Outcome:

Pipelines resume processing within SLA with data integrity intact.

Step 6 : Corrective Actions

- Enforce BACKWARD or FULL compatibility in Schema Registry
- Block incompatible schema registrations
- Add schema validation in CI/CD
- Educate producers on schema evolution rules
- Monitor schema changes proactively

These steps prevent **future Kafka-wide outages.**

Step 7 : Result After Fix

Before	**After**
Consumers failing	Consumers stable
Data backlog	Normal throughput
SLA breached	SLA met
Firefighting	Governed evolution

Final Resolution

- **Root Issue:** Incompatible Avro schema pushed to Kafka
- **Action Taken:** Fixed schema to maintain backward compatibility

Key Learnings

- Kafka schemas are system-wide contracts
- Schema Registry is not optional
- Compatibility failures impact many services
- Fix schema issues at the producer, not consumers

Core Principle Reinforced

If schemas evolve unsafely, distributed systems fail instantly.

Scenario 4

Data Type Change Breaks Downstream Jobs

Problem Statement

An upstream producer **changes a column's data type from INT to STRING.** While the change is valid for new data**, existing ETL logic and downstream aggregations expect an integer,** causing **job failures and incorrect dashboard metrics.** The **SLA is 2 hours, historical data must remain valid, and business dashboards are impacted.**

Key Details

- Column data type changed: INT → STRING
- Aggregations and calculations fail
- Historical data cannot be rewritten immediately
- Dashboards show errors or incorrect metrics
- SLA: 2 hours

Expected vs Actual Behavior

Expected	**Actual**
Type changes handled safely	Aggregations fail
Historical data remains usable	Mixed types break logic
Dashboards remain accurate	Dashboards incorrect or broken
SLA protected	SLA at risk

This is a **schema evolution and data type compatibility issue,** not a compute or performance problem.

Why This Problem Is High Risk

Because:

- Data type changes are common in evolving systems
- Aggregations depend heavily on type consistency
- Mixed historical and new data creates hidden failures
- Dashboards may fail loudly—or worse, compute wrong values

Common but risky reactions:

- Ignoring failures to meet SLA
- Retrying jobs repeatedly
- Skipping rows that don't parse

But **type mismatches must be handled deliberately, not ignored.**

Clarifying Questions

A senior data engineer asks:

- Is the STRING type numeric, alphanumeric, or mixed?
- Are nulls or invalid values introduced?
- Which downstream metrics depend on this column?
- Can historical data remain unchanged?
- Where is the safest place to handle conversion?

These questions focus on **data correctness and resilience.**

Confirmed Facts & Assumptions

After investigation:

- New data arrives as STRING
- Historical data remains INT
- Aggregations expect numeric values
- Retrying jobs does not fix logic errors
- ETL layer can safely cast values

Interpretation:
This is a **consumer-side compatibility problem**, not an upstream emergency.

What Teams Often Assume vs Reality

Assumption	**Reality**
Type change is trivial	It breaks aggregations
Dashboards will adapt	They rely on strict types
Skipping bad rows is safe	It causes silent data loss
Retrying fixes failures	Logic remains broken

Senior engineers **absorb type changes at the ETL boundary.**

Root Cause Analysis

Step 1: Inspect Aggregation Failures

Observed:

- Type casting errors during aggregation
- Failures start exactly After schema change

Conclusion:
Data type incompatibility is the root trigger.

Step 2: Review ETL Type Handling

Observed:

- No explicit casting logic
- Assumption of integer-only values

This confirms **lack of defensive type handling.**

Step 3: Conceptual Root Cause

The root cause is **unhandled data type evolution:**

- Producer changed representation
- Consumers assumed static types

This is a **schema evolution governance gap.**

Step 4 : Wrong Approach vs Right Approach

Wrong Approach

- Ignore failures
- Retry jobs repeatedly
- Skip affected rows

Right Approach

- Cast STRING → INT safely in ETL
- Handle invalid or null values explicitly
- Preserve historical data semantics

Senior engineers **make ETL resilient to representation changes.**

Step 5 : Validation of the Fix

To validate:

- Add safe casting logic in ETL
- Re-run affected partitions
- Verify aggregations compute correctly
- Confirm dashboards show accurate metrics

Outcome:
Metrics restored within SLA without altering historical data.

Step 6 : Corrective Actions

- Add explicit type casting in ETL
- Validate numeric strings Before conversion
- Log and monitor conversion failures
- Document data type expectations
- Enforce schema compatibility guidelines

These steps prevent **future type-related breakages.**

Step 7 : Result After Fix

Before	After
Aggregations failing	Aggregations stable
Dashboard errors	Accurate dashboards
SLA pressure	SLA met
Reactive fixes	Resilient design

Final Resolution

- **Root Issue:** Data type change (INT → STRING) broke downstream logic
- **Action Taken:** Safely cast data type in ETL

Key Learnings

- Data types are part of the schema contract
- Type changes impact more than storage
- ETL must handle mixed historical and new data
- Defensive casting prevents outages

Core Principle Reinforced

When data types evolve, consumers must adapt—silently and safely.

Scenario 5

Nested Field Removal in JSON Breaks Consumers

Problem Statement

An upstream producer **removes one or more nested fields** from a JSON payload that **downstream consumers rely on.** While new records conform to the updated schema, **existing ETL logic and downstream pipelines fail** when attempting to access missing nested fields. The **SLA is 1 hour, multiple pipelines are impacted**, and **historical data must remain readable.**

Key Details

- Nested JSON field removed upstream
- Downstream consumers expect the field
- Multiple pipelines impacted
- Historical data cannot be rewritten
- SLA: 1 hour

Expected vs Actual Behavior

Expected	Actual
Optional fields handled safely	Pipelines fail on missing fields
JSON schema evolves without breaking consumers	Hard dependency causes failures
Historical data remains readable	Parsing errors on mixed data
SLA protected	SLA breached

This is a **schema evolution and consumer resilience issue,** not a performance or infrastructure problem.

Why This Problem Is High Risk

Because:

- Nested fields are often assumed to always exist
- JSON schemas evolve frequently and informally
- Missing nested fields cause runtime errors
- Failures cascade across dependent pipelines

Common but dangerous reactions:

- Skipping affected records
- Reprocessing entire datasets
- Ignoring failures to meet SLA

But **nested fields should always be treated as optional unless explicitly guaranteed.**

Clarifying Questions

A senior data engineer asks:

- Was the nested field documented as optional or required?
- How many consumers reference this field?
- Is the field removed permanently or conditionally?
- Can defaults or nulls be safely applied?
- Are schema contracts enforced or informal?

These questions focus on **consumer safety and backward compatibility.**

Confirmed Facts & Assumptions

After investigation:

- New JSON payloads no longer include the nested field
- Historical records still contain the field
- Consumers assume field existence
- Full dataset reprocessing violates SLA
- Consumers can be updated to handle missing fields

Interpretation:
This is a **consumer-side robustness gap**, not an upstream rollback issue.

What Teams Often Assume vs Reality

Assumption	Reality
Nested fields will always exist	They are frequently removed
Skipping records is acceptable	It causes silent data loss
Full reprocessing is safest	It's unnecessary and risky
Schema-less means flexible	It means consumers must be defensive

Senior engineers **treat nested fields as optional by default.**

Root Cause Analysis

Step 1: Inspect Failure Mode

Observed:

- Null pointer / key errors during parsing
- Failures start exactly After schema change

Conclusion:
Missing nested fields are the trigger.

Step 2: Review Consumer Parsing Logic

Observed:

- Direct access to nested fields
- No null or existence checks

This confirms **non-defensive parsing.**

Step 3: Conceptual Root Cause

The root cause is **assuming nested fields are permanent:**

- Upstream removed field
- Consumers not resilient

This is a **schema contract and design gap.**

Step 4 : Wrong Approach vs Right Approach

Wrong Approach

- Ignore failures
- Skip affected records
- Reprocess entire dataset

Right Approach

- Update consumers to handle missing nested fields
- Apply null-safe access or defaults
- Maintain backward compatibility

Senior engineers **make consumers resilient to schema change.**

Step 5 : Validation of the Fix

To validate:

- Update parsing logic to handle missing fields
- Re-run affected pipelines
- Validate historical and new data processing
- Confirm downstream outputs are correct

Outcome:
Pipelines stabilize within SLA without upstream rollback.

Step 6 : Corrective Actions

- Treat nested JSON fields as optional by default
- Add null-safe parsing and defaults
- Document required vs optional fields
- Add schema validation tests
- Enforce producer–consumer contracts

These steps prevent **future breakages due to nested schema changes.**

Step 7 : Result After Fix

Before	**After**
Pipeline failures	Stable pipelines
SLA breaches	SLA met
Reactive firefighting	Defensive design
Schema fragility	Schema resilience

Final Resolution

- **Root Issue:** Nested field removed without consumer resilience
- **Action Taken:** Updated consumers to handle missing nested fields safely

Key Learnings

- JSON schemas evolve frequently
- Nested fields are especially fragile
- Consumers must be defensive by design
- Backward compatibility is a consumer responsibility

Core Principle Reinforced

If a field can be removed, your code must survive its absence.

Scenario 6

Multiple Teams Modifying the Same Schema

Problem Statement

Multiple teams independently modify the **same shared schema** without coordination. Conflicting changes are deployed close together, causing **schema incompatibilities** that lead to **pipeline failures across multiple downstream systems.** The **SLA is 2 hours,** several pipelines are impacted, and **collaboration between teams is limited.**

Key Details

- Multiple teams modifying the same schema
- No coordination or versioning discipline
- Conflicting schema changes deployed
- Multiple pipelines impacted
- SLA: 2 hours

Expected vs Actual Behavior

Expected	Actual
Schema changes are coordinated	Conflicting changes deployed
Pipelines remain stable	Pipelines fail unExpectedly
Changes are versioned and reviewed	No single source of truth
SLA protected	SLA at risk

This is a **schema governance failure,** not a tooling or performance issue.

Why This Problem Is High Risk

Because:

- Shared schemas act as contracts across teams
- Uncoordinated changes multiply blast radius
- Failures are hard to attribute to one change
- Trust between teams erodes quickly

Common but dangerous reactions:

- Retrying jobs repeatedly
- Asking teams to “just coordinate better”
- Ignoring failures until escalation

But **process problems cannot be solved with retries.**

Clarifying Questions

A senior data engineer asks:

- Who owns the schema officially?
- Are changes versioned and reviewed?
- How are compatibility checks enforced?
- Do producers validate against a shared registry?
- Can consumers rely on backward compatibility guarantees?

These questions focus on **system ownership and governance,** not blame.

Confirmed Facts & Assumptions

After investigation:

- No centralized schema authority exists
- Teams deploy schema changes independently
- Compatibility checks are inconsistent or missing
- Multiple pipelines broke simultaneously
- Centralized governance is feasible

Interpretation:
This is a **coordination and ownership gap**, not a technical limitation.

What Teams Often Assume vs Reality

Assumption	Reality
Teams will self-coordinate	Conflicts are inevitable
Pipelines will fail loudly	Some break silently
Notifications are enough	Enforcement is required
Retry fixes failures	Conflicts persist

Senior engineers **build guardrails instead of relying on discipline.**

Root Cause Analysis

Step 1: Correlate Failures

Observed:

- Pipeline failures align with multiple schema deployments
- Different fields modified by different teams

Conclusion:
Conflicting schema changes caused incompatibility.

Step 2: Review Change Process

Observed:

- No single schema registry or approval flow
- No enforced compatibility checks

This confirms **lack of centralized schema governance.**

Step 3: Conceptual Root Cause

The root cause is **distributed ownership without control:**

- Shared schema
- Independent changes
- No enforcement mechanism

This is a **design and process failure.**

Step 4 : Wrong Approach vs Right Approach

Wrong Approach

- Ignore schema conflicts
- Retry jobs
- Ask teams to coordinate manually

Right Approach

- Implement a centralized schema registry
- Enforce compatibility rules
- Require versioned, reviewed changes

Senior engineers **solve coordination problems with systems, not meetings.**

Step 5 : Validation of the Fix

To validate:

- Register schemas centrally
- Block incompatible schema updates
- Re-deploy corrected schemas
- Restart pipelines and verify stability

Outcome:

Pipelines stabilize within SLA, and future conflicts are prevented.

Step 6 : Corrective Actions

- Introduce centralized schema registry
- Enforce backward/forward compatibility modes
- Define clear schema ownership
- Require schema reviews in CI/CD
- Educate teams on schema evolution rules

These steps prevent **multi-team schema conflicts**

Step 7 : Result After Fix

Before	**After**
Conflicting schema changes	Controlled evolution
Pipeline failures	Stable pipelines
SLA risk	SLA met
Reactive firefighting	Proactive governance

Final Resolution

- **Root Issue:** Uncoordinated schema changes across teams
- **Action Taken:** Implemented centralized schema registry and governance

Key Learnings

- Shared schemas require ownership
- Coordination doesn't scale without enforcement
- Schema registries are essential in multi-team systems
- Governance prevents outages

Core Principle Reinforced

In multi-team systems, schema governance is not optional—it's infrastructure.

Scenario 7

Optional Field Becomes Mandatory Mid-Month

Problem Statement

An upstream producer **changes an optional field to mandatory** in the middle of the month. While new records contain the field, **historical records do not,** causing **ETL jobs to fail** when processing older data. The **SLA is 1 hour, historical metrics are business-critical**, and **multiple downstream jobs are impacted.**

Key Details

- Field changed from optional → mandatory
- Historical records missing the field
- ETL fails on older partitions
- Multiple pipelines impacted
- SLA: 1 hour

Expected vs Actual Behavior

Expected	Actual
Schema evolution is backward-compatible	Historical data breaks ETL
Old and new data processed together	Jobs fail on missing values
Metrics remain continuous	Gaps in historical metrics
SLA protected	SLA at risk

This is a **schema evolution and backward-compatibility failure,** not a performance issue.

Why This Problem Is High Risk

Because:

- Historical data cannot be regenerated easily
- Metrics depend on continuity across time
- Mid-period schema changes create partial failures
- Multiple jobs amplify the blast radius

Common but risky reactions:

- Reprocessing entire datasets
- Skipping records with missing values
- Ignoring failures to meet SLA

But **making a field mandatory does not make historical data magically compliant.**

Clarifying Questions

A senior data engineer asks:

- Is the field truly required for historical metrics?
- How much historical data is missing the field?
- Can a default or null safely represent missing values?
- Which downstream jobs depend on this field?
- Was a deprecation window provided?

These questions focus on **data continuity and correctness.**

Confirmed Facts & Assumptions

After investigation:

- Only recent records contain the field
- Historical data cannot be backfilled quickly
- Skipping rows causes metric gaps
- Full reprocessing violates SLA
- ETL can handle null/default values safely

Interpretation:

This is a **consumer-resilience issue**, not an upstream emergency.

What Teams Often Assume vs Reality

Assumption	Reality
Mandatory means always present	History disproves that
Reprocessing will fix it	Time and cost prohibitive
Skipping rows is harmless	It corrupts metrics
Upstream should roll back	Downstream can adapt faster

Senior engineers **design ETL to survive schema evolution.**

Root Cause Analysis

Step 1: Analyze Failure Pattern

Observed:

- Failures occur only on historical partitions
- Missing mandatory field triggers errors

Conclusion:
Historical data is incompatible with new requirement.

Step 2: Review ETL Assumptions

Observed:

- ETL assumes mandatory field always exists
- No null/default handling implemented

This confirms **lack of backward compatibility handling.**

Step 3: Conceptual Root Cause

The root cause is **evolving field requirements without consumer safeguards:**

- Producer tightened constraints
- Consumers not updated defensively

This is a **schema governance gap.**

Step 4 : Wrong Approach vs Right Approach

Wrong Approach

- Ignore failures
- Skip affected rows
- Reprocess entire dataset

Right Approach

- Handle nulls/defaults in ETL
- Preserve historical data continuity
- Maintain backward compatibility

Senior engineers **adapt consumers instead of breaking history.**

Step 5 : Validation of the Fix

To validate:

- Update ETL to handle missing values
- Re-run historical partitions
- Verify metric continuity
- Confirm downstream jobs succeed

Outcome:
Pipelines stabilize within SLA with no data loss.

Step 6 : Corrective Actions

- Treat newly mandatory fields as optional in ETL
- Apply defaults or null-safe logic
- Enforce deprecation windows for schema changes
- Add schema evolution tests
- Document producer–consumer contracts

These steps prevent **future mid-period schema breakages.**

Step 7 : Result After Fix

Before	After
ETL failures	Stable pipelines
Metric gaps	Continuous metrics
SLA pressure	SLA met
Reactive fixes	Forward-compatible design

Final Resolution

- **Root Issue:** Optional field made mandatory without historical compatibility
- **Action Taken:** Updated ETL to handle missing values safely

Key Learnings

- Historical data always matters
- Mandatory fields must respect legacy data
- ETL should be backward-compatible by design
- Schema evolution needs governance

Core Principle Reinforced

A field can become mandatory in code—but history doesn't update itself.

Scenario 8

Upstream API Contract Change Breaks ETL Parsing

Problem Statement

An **upstream API changes its response structure** (field names, nesting, or payload format) without proper notification. As a result, **ETL parsing logic fails**, causing **pipeline failures and incorrect downstream metrics**. The **SLA is 1 hour, multiple pipelines depend on this API, and data correctness is business-critical.**

Key Details

- API response structure changed upstream
- ETL parsing logic breaks
- Multiple dependent pipelines affected
- Data correctness is critical
- SLA: 1 hour

Expected vs Actual Behavior

Expected	Actual
API changes are contract-aware	Response structure changed unexpectedly
ETL adapts safely to changes	Parsing fails
Downstream pipelines remain stable	Cascading failures
SLA protected	SLA at risk

This is a **contract evolution failure,** not a compute or retry issue.

Why This Problem Is High Risk

Because:

- APIs are upstream contracts for many systems
- Structural changes break parsers immediately
- Failures cascade across dependent pipelines
- Silent changes undermine trust between teams

Common but risky reactions:

- Retrying ETL jobs repeatedly
- Rolling back upstream blindly
- Ignoring failures to meet SLA

But **API contracts must be validated and enforced.**

Clarifying Questions

A senior data engineer asks:

- What exactly changed in the API response?
- Was versioning or deprecation provided?
- Are schema validations in place?
- How many pipelines depend on this API?
- Can ETL be updated quickly without data loss?

These questions focus on **contract stability and impact scope.**

Confirmed Facts & Assumptions

After investigation:

- API response fields/nesting changed
- ETL parsing assumes old structure
- Retrying does not fix parsing logic
- Rolling back upstream impacts other consumers
- ETL can be updated to handle new contract

Interpretation:
This is a **consumer-side contract validation gap**, not an upstream outage.

What Teams Often Assume vs Reality

Assumption	Reality
API changes are backward-compatible	Many are breaking
Retries will fix failures	Logic remains broken
Rollback is fastest	It disrupts multiple teams
Parsers can be brittle	They must be defensive

Senior engineers **treat APIs as versioned contracts, not flexible inputs.**

Root Cause Analysis

Step 1: Inspect ETL Failure Logs

Observed:

- Parsing errors due to missing/renamed fields
- Failures align with API deployment time

Conclusion:
API contract change is the trigger.

Step 2: Review Contract Validation

Observed:

- No schema validation at ingestion
- ETL assumes fixed response structure

This confirms **missing contract enforcement.**

Step 3: Conceptual Root Cause

The root cause is **API contract change without consumer safeguards:**

- Producer changed structure
- Consumers not protected

This is a **contract governance failure.**

Step 4 : Wrong Approach vs Right Approach

Wrong Approach

- Ignore failures
- Retry jobs repeatedly
- Roll back upstream blindly

Right Approach

- Validate API contract changes
- Update ETL parsing logic
- Communicate changes to dependent teams

Senior engineers **fix contracts, not symptoms.**

Step 5 : Validation of the Fix

To validate:

- Update ETL to parse new API structure
- Add schema/contract validation
- Re-run failed pipelines
- Verify downstream metrics correctness

Outcome:
Pipelines resume successfully within SLA with correct data.

Step 6 : Corrective Actions

- Enforce API contract versioning
- Add schema validation at ingestion
- Implement backward-compatible parsing where possible
- Require change notifications from API providers
- Monitor contract changes proactively

These steps prevent **future API-driven outages.**

Step 7 : Result After Fix

Before	After
ETL failures	Stable ingestion
Downstream breakage	Correct metrics
SLA pressure	SLA met
Reactive firefighting	Governed contracts

Final Resolution

- **Root Issue:** Upstream API contract change broke ETL parsing
- **Action Taken:** Validated contract and updated ETL logic

Key Learnings

- APIs are data contracts
- Contract changes must be versioned and communicated
- ETL should validate inputs, not trust them blindly
- Silent API changes are high-risk events

Core Principle Reinforced

If an API can change, your pipeline must detect and adapt—not fail silently.

Scenario 9

Schema Versioning Not Enforced

Problem Statement

An upstream producer publishes messages using an **unversioned schema**. A later change introduces an **incompatible modification**, which immediately **breaks multiple downstream consumers**. Because no schema versioning is enforced, consumers cannot distinguish old data from new, incompatible data. The **SLA is 1 hour, multiple consumers depend on this pipeline, and data integrity is critical.**

Key Details

- Producer emits schema without versioning
- Incompatible change introduced
- Consumers fail to deserialize/process data
- Multiple downstream systems impacted
- SLA: 1 hour

Expected vs Actual Behavior

Expected	Actual
Schema evolution is controlled	Incompatible change deployed
Consumers handle changes safely	Consumers break immediately
Data integrity preserved	Data loss / backlog risk
SLA protected	SLA missed

This is a **schema governance and contract enforcement failure,** not an ETL or infrastructure issue.

Why This Problem Is High Risk

Because:

- Unversioned schemas remove consumer safety nets
- Incompatible changes propagate instantly
- Multiple consumers amplify blast radius
- Rollbacks become chaotic under pressure

Common but risky reactions:

- Retrying ETL or consumers
- Rolling back producers urgently
- Ignoring failures temporarily

But **without versioning, systems cannot evolve safely.**

Clarifying Questions

A senior data engineer asks:

- Is schema versioning explicitly enforced?
- What compatibility guarantees exist?
- Which consumers are affected and how?
- Can multiple schema versions coexist?
- How are breaking changes detected before deploy?

These questions focus on **preventing recurrence,** not short-term firefighting.

Confirmed Facts & Assumptions

After investigation:

- Producer does not publish schema versions
- Consumers assume a single implicit schema
- Incompatible change introduced without safeguards
- Retrying consumers does not resolve incompatibility
- Enforcing versioning is feasible immediately

Interpretation:
This is a **lack of schema contract discipline**, not a transient failure.

What Teams Often Assume vs Reality

Assumption	**Reality**
Schemas won't change often	They always do
Consumers can adapt automatically	They cannot without versions
Rollback is fastest	It disrupts upstream systems
Compatibility is implied	It must be enforced

Senior engineers **make schema evolution explicit, not implicit.**

Root Cause Analysis

Step 1: Analyze Consumer Failures

Observed:

- Deserialization and parsing errors
- Failures start exactly After schema change

Conclusion:
Incompatible schema change is the trigger.

Step 2: Review Schema Management

Observed:

- No schema versions tracked
- No compatibility checks enforced

This confirms **absence of schema governance.**

Step 3: Conceptual Root Cause

The root cause is **missing schema versioning:**

- Producer publishes unversioned schema
- Consumers have no isolation from change

This is a **design and governance gap.**

Step 4 : Wrong Approach vs Right Approach

Wrong Approach

- Ignore failures
- Retry ETL jobs
- Roll back producer blindly

Right Approach

- Enforce explicit schema versioning
- Allow multiple versions to coexist
- Validate compatibility Before deploy

Senior engineers **control change through versioning, not hope.**

Step 5 : Validation of the Fix

To validate:

- Enforce schema versioning in producer
- Register versions centrally
- Restart consumers
- Confirm successful processing across versions

Outcome:
Consumers resume processing within SLA with data integrity preserved.

Step 6 : Corrective Actions

- Enforce schema versioning policy
- Block incompatible changes without new versions
- Add compatibility checks in CI/CD
- Educate teams on versioned evolution
- Monitor schema changes proactively

These steps prevent **future schema-driven outages.**

Step 7 : Result After Fix

Before	After
Consumers breaking	Stable multi-version support
SLA missed	SLA met
Firefighting	Governed evolution
Fragile pipelines	Resilient contracts

Final Resolution

- **Root Issue:** Unversioned schema allowed incompatible changes
- **Action Taken:** Enforced schema versioning and compatibility rules

Key Learnings

- Versioning is mandatory in distributed systems
- Schema changes are inevitable
- Implicit contracts always fail
- Governance enables safe evolution

Core Principle Reinforced

If a schema isn't versioned, it isn't safe to change.

Scenario 10

Deprecated Field Removed Without Grace Period

Problem Statement

An upstream producer **removes a deprecated field immediately,** without providing a grace period. Although the field was marked deprecated, **multiple downstream ETL pipelines still depend on it,** causing **job failures and incorrect data processing.** The **SLA is 1 hour, multiple consumers are affected, and data correctness is critical.**

Key Details

- Deprecated field removed abruptly
- Downstream ETL still references the field
- Multiple consumers impacted
- Data correctness critical
- SLA: 1 hour

Expected vs Actual Behavior

Expected	Actual
Deprecated field remains during transition	Field removed immediately
Consumers migrate gradually	ETL fails abruptly
Backward compatibility maintained	Pipelines break
SLA protected	SLA breached

This is a **deprecation and change-management failure,** not a processing or infrastructure issue.

Why This Problem Is High Risk

Because:

- Deprecation implies *temporary coexistence*, not removal
- Downstream consumers often migrate at different speeds
- Immediate removal causes cascading failures
- Business teams lose trust in data stability

Common but risky reactions:

- Ignoring failures to meet SLA
- Forcing consumers to hotfix under pressure
- Reprocessing data without fixing root cause

But **deprecation without a grace period is equivalent to a breaking change.**

Clarifying Questions

A senior data engineer asks:

- Was a deprecation timeline communicated?
- Which consumers still rely on the field?
- Is the field required for historical data?
- Can the field be retained temporarily?
- Is there a versioned alternative field available?

These questions focus on **safe migration,** not blame.

Confirmed Facts & Assumptions

After investigation:

- Field was removed immediately After deprecation
- Several consumers had not migrated
- Retrying ETL does not fix missing field
- Reprocessing data does not restore schema
- Keeping the field temporarily is feasible

Interpretation:

This is a **schema lifecycle governance issue**, not a consumer bug.

What Teams Often Assume vs Reality

Assumption	**Reality**
Deprecated means unused	Many consumers still rely on it
Notification is enough	Enforcement must be gradual
Consumers can hotfix quickly	They often cannot
Removal cleans up tech debt	It creates outages

Senior engineers **treat deprecation as a transition phase, not a delete command.**

Root Cause Analysis

Step 1: Inspect ETL Failures

Observed:

- Parsing and transformation errors
- Failures start immediately After field removal

Conclusion:
Field removal is the direct trigger.

Step 2: Review Deprecation Process

Observed:

- No grace period enforced
- No compatibility window provided

This confirms **improper deprecation handling.**

Step 3: Conceptual Root Cause

The root cause is **removing deprecated fields without backward compatibility:**

- Producer assumed deprecation = safe removal
- Consumers not protected

This is a **schema governance failure.**

Step 4 : Wrong Approach vs Right Approach

Wrong Approach

- Ignore failures
- Force consumers to hotfix
- Reprocess data blindly

Right Approach

- Keep deprecated field temporarily
- Allow consumers to migrate safely
- Remove field only After adoption

Senior engineers **optimize for system stability over cleanup speed.**

Step 5 : Validation of the Fix

To validate:

- Restore deprecated field temporarily
- Re-run ETL pipelines
- Verify downstream jobs succeed
- Confirm metrics correctness

Outcome:
Pipelines stabilize within SLA while consumers migrate.

Step 6 : Corrective Actions

- Enforce deprecation grace periods
- Track consumer migration status
- Use schema versioning for removals
- Communicate clear timelines
- Block immediate breaking removals

These steps prevent **future deprecation-related outages.**

Step 7 : Result After Fix

Before	**After**
ETL failures	Stable pipelines
SLA breaches	SLA met
Forced firefighting	Controlled migration
Fragile evolution	Governed lifecycle

Final Resolution

- **Root Issue:** Deprecated field removed without grace period
- **Action Taken:** Temporarily retained deprecated field to maintain backward compatibility

Key Learnings

- Deprecation ≠ deletion
- Backward compatibility requires time
- Schema cleanup must be planned
- Stability beats speed in shared systems

Core Principle Reinforced

A deprecated field is still a contract—until every consumer lets it go.