Unknown date | Unknown author

Data Contracts: How They Work, Importance, & Best Practices

Your data pipeline breaks at 3 AM. Again. The marketing team's dashboard shows nonsense because someone changed a field name upstream without telling anyone. Sound familiar?

This scenario plays out daily at companies everywhere. Teams build dependencies on data they don't control, then scramble when inevitable changes break their systems. The traditional approach of hoping everyone communicates perfectly doesn't scale.

The problem gets worse as organizations grow. More teams produce data. More systems consume it. More critical decisions depend on it. Yet most companies still rely on informal agreements and good intentions to keep everything running. When those fail, data teams spend their time firefighting instead of building.

There's a better way to manage these dependencies. One that catches breaking changes before they hit production. One that makes expectations explicit and enforceable. One that actually speeds up development by eliminating the guesswork.

In this article, we'll cover everything you need to know about data contracts and how to implement them effectively. You'll learn the essential components, how to roll them out incrementally, and how to overcome the real challenges teams face. We'll share proven practices from organizations that have deployed contracts at scale, plus the tools and frameworks that make implementation straightforward.

Whether you're a data engineer tired of broken pipelines, a software developer who wants to ship changes safely, or a leader looking to improve data reliability, this article will show you exactly how to get started with data contracts.

Table of Contents

What is a data contract?

A data contract is an agreement between a service provider and data consumers. It refers to the management and intended usage of data between different organizations, or sometimes within a single company. But here's what makes it powerful: this agreement is implemented in code, not just documented in prose.

Despite the name, it's not a physical contract or legal SLA. Instead, it's a set of defined rules and technical measures that automatically enforce how data should look and behave. A data contract might specify the exact schema (fields and data types) that a service will output, the range of values those fields should contain, and the expected refresh frequency of the data. When a marketing analytics team relies on customer event data from the product team, their data contract ensures both sides understand exactly what's being delivered and when.

Why are data contracts required?

First things first, let's think about why we need data contracts.

Data teams find themselves reliant on systems and services, often internal, that emit production data that lands the data in the data warehouse and becomes part of different downstream processes. However, the software engineers in charge of these systems are not tasked with maintaining and are often unaware of these data dependencies. So when they make an update to their service that results in a schema change, these tightly coupled data systems crash.

Another equally important use case is downstream data quality issues. These arise when the data being brought into the data warehouse isn't in a format that is usable by data consumers. A data contract that enforces certain formats, constraints and semantic meanings can mitigate such instances.

We know that organizations are dealing with more data than ever before, and responsibilities for that data are frequently distributed between domains; that's one of the key principles of a data mesh approach.

The name can be a bit deceiving in this case – data contracts aren't in-depth legal documents, but a process to help data producers and data consumers get on the same page.

The more widely distributed data becomes, the more important it is to have a solution in place that ensures transparency and builds trust between teams using data that isn't their own.

What's in a data contract?

For those who've never seen one, the idea of creating data contracts might be daunting. Fortunately, using them well is more about effective implementation than creating long and complex schemas. Once you've settled on an interchange format for a data contract, ensuring maximum readability, actually creating one can be as simple as a few lines of text.

```
{
  local dcs = import 'utopia/utopia/dataContract/contract.libsonnet',
  local ddrSubject = dcs.ddrSubject,

  contract: dcs.v1.new() {
    metadata+: {
      name: 'creditor_details',
      dataset_name: 'creditor_details',
      description: 'Creditor details',
    },
    schema+: {
      versions: [
        dcs.v1.new_version(
          '1',
          dcs.anonymisation_strategy.overwrite,
          [
            dcs.field(
              'creditor_id',
              'Creditor ID',
              dcs.data_types.string,
              dcs.field_category.gocardless_internal,
              dcs.is_personal_data.yes,
              dcs.personal_data_identifier.indirect,
              fieldAnonymisationStrategy=dcs.field_anonymisation_strategy.none,
              required=true,
            ),

            dcs.field(
              'creditor_type',
              'Creditor type (Company/Individual/Charity/Trust/Partnership)',
              dcs.data_types.string,
              dcs.field_category.gocardless_internal,
              dcs.is_personal_data.no,
              dcs.personal_data_identifier.none,
              fieldAnonymisationStrategy=dcs.field_anonymisation_strategy.none,
              required=true,
            ),


          ],
          [
            ddrSubject('creditor', "creditor_id"),
          ],
        ) + dcs.v1.withPubSub() + dcs.v1.withBigQuery(),
      ],
    },
  },
}
```

An example abridged data contract in JSON. Courtesy of Andrew Jones

For a closer look at an actual data contract template you can access the YAML file PayPal has open sourced on GitHub.

We won't dive too deep into data contract architecture here, as we've covered that before; the article we just linked has some great insights from GoCardless's Data Team Lead on how they implemented data contracts there.

We will, however, reiterate that data contracts might cover things like:

- What data is being extracted
- Ingestion type and frequency
- Details of data ownership/ingestion, whether individual or team
- Levels of data access required
- Information relating to security and governance (e.g. anonymization)
- How it impacts any system(s) that ingestion might impact

Because data contracts can differ substantially based on the type of data they refer to, as well as the type of organization they're being used in, we haven't yet seen a significant degree of standardization when it comes to data contract formats and content. A set of best practices may yet, however, emerge in the future, like we've seen with the OpenAPI Specification.

Key elements of a data contract

Data contracts contain several essential components that work together to create a complete agreement. Each element serves a specific purpose in ensuring data reliability and preventing downstream failures.

Schema definition

The core of any data contract is a precise schema. This includes the format and structure (such as Avro, JSON, or YAML), the list of fields or columns, data types, and structure of nested data when applicable. The contract explicitly states what data is provided.

If it's an event stream, the contract might specify each attribute of the event: user_id as an integer, event_time as a timestamp in UTC, and so on. This acts as the blueprint that producers must adhere to and consumers can rely on. No

more guessing whether that timestamp includes timezone information or if that ID field is a string or integer.

Data constraints and validation rules

Contracts establish data quality standards through rules and metrics that ensure data accuracy, completeness, and consistency. This includes rules like email addresses having a valid format or numerical fields falling within specific ranges.

A percentage must be 0-100. A status can only be active or inactive. These data quality rules are embedded directly in the contract. They ensure semantics are understood. A field representing a date won't ever be null or contain a negative number. If certain fields are identifiers, the contract requires them to be unique or non-null. This is where schema validation proves its worth: contracts enable automatic validation checks so any data violating the contract is flagged or rejected before it corrupts downstream systems.

Metadata and context

A good data contract includes metadata that clarifies the meaning and intended use of data elements and fields. This ensures a shared understanding among all parties involved. It clearly identifies the data owner or producer team and lists consumers or stakeholders relying on it. The contract maps upstream systems that produce the data and downstream systems that consume it.

Business meaning matters too. The contract defines what customer_id actually represents, eliminating ambiguity. Including ownership is important for accountability. When something goes wrong or a change is needed, everyone knows who to contact instead of sending desperate Slack messages to entire channels.

Service level expectations

Contracts specify commitments regarding data freshness, availability, and latency. For critical datasets, these SLA or SLO elements might guarantee that

"this data will be updated by 6am daily" or "no more than 0.1% of records will contain errors."

While not every contract needs this level of detail, setting expectations on data timeliness and quality levels prevents disappointment. These data SLAs transform vague promises into measurable commitments that teams can monitor and improve.

Data governance and compliance

Contracts outline rules and guidelines for data management, including access controls, privacy regulations (like GDPR or HIPAA), and data lifecycle management. They stipulate compliance measures for sensitive data, specifying that PII fields will be hashed or removed, defining data classification levels, and outlining access restrictions.

This isn't bureaucracy for its own sake. It's about using data safely and responsibly. When a contract clearly states security requirements, teams can build compliant pipelines from the start rather than scrambling to fix issues during an audit.

Ownership and accountability

Contracts clearly define who is responsible for the data, including maintenance, updates, and addressing issues. This goes deeper than just naming teams. It specifies who handles schema changes, who monitors data quality, and who responds when things break.

Clear ownership eliminates the confusion that plagues many data initiatives. It transforms finger-pointing sessions into productive conversations about maintaining data quality and resolving issues quickly.

Versioning and evolution

Data needs change over time. Contracts establish a process for managing changes to the data schema and contract terms, ensuring backward compatibility and smooth transitions. This might include version numbers, deprecation notices, and migration paths.

Without data versioning, a simple field addition can break dozens of downstream consumers. With it, teams can evolve their data products while giving consumers time to adapt.

These components combined form a “contract” that both sides agree to. Producers know exactly what they need to deliver. Consumers know exactly what they can expect. The guesswork disappears, replaced by clear, enforceable agreements that keep data pipelines running smoothly.

Who is responsible for data contracts?

Although they won’t necessarily be the ones implementing them, the decision to run with data contracts lies with data leaders. It’s worth pointing out, however, that they require input and buy-in from **all** stakeholders involved with the consumption of data.

Data consumers tend to be the most motivated participants as data contracts clearly make their lives easier. Data producers such as software engineers may need some convincing to show them how data contracts can benefit the organization and improve data quality without too much additional effort.

To this end, it can be worth pointing out data contracts are typically fairly evergreen and don’t need much ongoing maintenance. Aside from occasional version control tweaks and updates to contact details etc., they shouldn’t create a significant burden once they’re up and running.

Best practices for effective data contracts

Data contracts work best when implemented thoughtfully. These practices come from teams who’ve successfully deployed contracts at scale and learned what actually moves the needle.

Keep contracts focused on critical data

It’s neither feasible nor necessary to put every dataset under contract. Start with high-impact data like critical dashboards and key ML inputs where reliability is paramount. This ensures effort goes where it matters most. Look

for datasets where failures trigger pages, block deployments, or cause executives to ask uncomfortable questions in Monday meetings.

The temptation to contract everything is real but counterproductive. Teams that try to boil the ocean typically burn out before seeing results. Instead, identify your top five most painful data failures from the last quarter. Those pipelines are your starting point. Once you've proven value with these critical datasets, you'll have the credibility and experience to expand.

Over time, expand coverage based on pain points and business value. A good rule of thumb is to add contracts when the cost of failure exceeds the cost of implementation. This might mean protecting revenue-impacting data first, then moving to operational dashboards, and finally addressing nice-to-have analytics. Focus wins over breadth every time.

Involve all stakeholders early

Data contracts require cooperation between producers and consumers. Get buy-in and input from both sides when defining the contract. Joint workshops or meetings help clarify schemas and SLAs that work for everyone.

When producers understand the value (fewer late-night emergencies) and consumers clearly state their needs, contracts succeed. Skip this step and you'll end up with contracts that nobody follows.

Automate enforcement as much as possible

Manual processes don't scale. Build automated schema checks into CI pipelines and set up alerts for breaches. This makes following the contract the path of least resistance.

Treat contracts like code. Store them in Git, review changes via pull requests, and write tests that validate compliance. When automation handles enforcement, humans can focus on improving data quality instead of policing it.

Version control and iterate

Maintain version history for all contracts. When requirements change, update the contract in a controlled way with new versions and backward compatibility checks. Document what changed so consumers can adapt. This isn't just about tracking changes. It's about enabling evolution without breaking trust.

Contract versioning follows the same principles as API versioning. Major versions signal breaking changes that require consumer updates. Minor versions add optional fields or capabilities. Patch versions fix bugs or clarify documentation. This semantic versioning approach helps teams understand the impact of changes at a glance. A change from version 2.1.3 to 3.0.0 immediately signals that consumers need to pay attention.

A good versioning strategy includes deprecation notices and migration windows. If you're removing a field, mark it deprecated in version 2.0, optional in 3.0, and remove it in 4.0. Give consumers at least one full release cycle to adapt. Document migration paths clearly. This practice prevents the chaos of sudden changes while still allowing data products to evolve with business needs.

Keep contracts lightweight and low maintenance

Contracts should enable teams, not burden them. Once set up, contracts are typically fairly evergreen and don't require constant work. Keep them as simple as possible while meeting actual needs.

Don't include dozens of rules that aren't truly necessary. Overly complex contracts discourage adoption and create maintenance headaches. Start minimal and add complexity only when real problems justify it.

Establish clear monitoring and ownership

Each contract needs an owner who's responsible for maintaining it. This isn't about creating bureaucracy. It's about ensuring someone gets paged when things break and someone has the authority to approve changes. The owner might be the producing team's tech lead or a dedicated data steward, but it must be a specific person or role, not a vague "the team."

Tag or group monitors by contract in your observability platform to easily track compliance. This makes it obvious when specific contracts are violated. Set up dashboards that show contract health at a glance. Green means all contracts are passing. Red means someone needs to investigate. Use consistent naming conventions like contract.user_events.schema_valid so related alerts naturally group together.

Review contract breaches in post-mortems or quarterly reviews. But don't just track failures. Celebrate successes too. When a contract prevents an outage, make sure both producers and consumers know about it. Use these sessions to identify patterns and improve both the contracts and the systems they protect. Ownership without monitoring is just wishful thinking. Monitoring without ownership is just noise.

Prioritize communication about changes

Create a process for communicating contract changes. If a producer needs to modify a schema, they should start discussions well in advance. This might result in an updated contract version with a scheduled release date.

Formalize this communication through tickets or change logs. Good communication prevents surprises and maintains trust between teams. The goal is boring, predictable data changes that everyone expects.

These practices aren't theoretical. They come from organizations that have successfully scaled data contracts across hundreds of pipelines. Start with one or two practices that address your biggest pain points, then expand as you see results.

When should data contracts be implemented?

You might assume that the answer to the question of when to implement data contracts would be "the sooner the better." But let's say that you're still working on getting organizational buy-in for a data mesh approach. Adding data contracts into the mix might complicate matters, and comes with a risk of stakeholders being overwhelmed.

It could be worth making sure you have all your ducks in a row – stable and reliable data pipelines that are working smoothly – before delving into data contracts. On the other hand, in the article we linked above, GoCardless's Andrew Jones suggests "if your team is pursuing any type of data meshy initiative, it's an ideal time to ensure data contracts are a part of it."

Of course, questions like "when should data contracts be implemented?" and "how long does it take to implement data contracts?" tend to have similar answers: in both cases, "it depends." Jones goes on to say that:

"As of this writing, we are 6 months into our initial implementation and excited by the momentum and progress. Roughly 30 different data contracts have been deployed which are now powering about 60% of asynchronous inter-service communication events."

In other words, this is not (nor does it have to be) an overnight process. And, when you do start, you can keep things simple. Once you're armed with the knowledge you've collected from team members and other stakeholders, you can begin to roll out data contracts.

How to implement data contracts

Implementing data contracts doesn't require a complete overhaul of your data infrastructure. You can start small, prove value, and expand systematically. Here's a practical roadmap that'll take your team from no contracts to enforceable agreements.

1. Identify critical data pipelines for contracts

Not every piece of data needs a contract on day one. Start with business-critical assets that feed important analytics or machine learning models. Look for pipelines where failures cause executive dashboards to break or where financial reports depend on accurate information.

Begin by gathering requirements for these datasets. Talk to both producers and consumers. What schema do consumers need? What's the source system currently emitting? Document the current schema and any known expectations, even if it starts in a simple design doc. You'll often find surprising

misalignments, like consumers expecting UTC timestamps while producers send local time.

The prioritization process itself builds organizational buy-in. When teams see their most painful data issues addressed first, they become advocates for expanding the approach. Start with one or two pipelines, demonstrate success, then tackle the next tier.

2. Define the contract using a standard format

Once requirements are clear, implement the contract in a machine-readable way. Choose a format like JSON Schema, Protobuf, or Avro. The key is consistency with your tech stack. If you're already using Avro for Kafka, stick with Avro for contracts.

Place the contract under version control in Git, just like code. Store it in a schema registry for easy access. As data evolves, contracts will have iterations. A registry helps manage versions and ensures backward compatibility, preventing the chaos of undocumented schema changes.

3. Enforce the contract in the data pipeline

Integration is where contracts prove their worth. Embed checks in the CI/CD pipeline of the data producer. If an engineer tries to deploy a change that violates the contract, automated tests should fail. Custom CI scripts can validate schemas during deployment.

Add circuit breakers in data ingestion too. If incoming data doesn't match the contract, stop it from flowing into the warehouse. Runtime checks compare schemas and halt the pipeline when there's a mismatch. This prevents bad data from corrupting your entire system.

4. Automate testing and validation

Implement tests that ensure data meets the contract. Use frameworks like dbt tests or Great Expectations to validate data in staging before it hits production. These tests should check schema compliance and data quality rules like value ranges and null constraints.

Your testing strategy needs multiple layers. Unit tests verify individual transformations respect the contract. Integration tests ensure end-to-end flows maintain compliance. Contract tests run automatically when either producer or consumer code changes, catching breaking changes early.

This proactive approach saves hours of debugging and builds deployment confidence. By shifting testing left in the development cycle, teams catch contract breaches before they impact production systems. The investment in test automation pays for itself through reduced on-call burden and faster feature delivery.

5. Monitor in production and iterate

Continuous monitoring catches what upfront enforcement might miss. Set up monitoring on key contracts to detect anomalies or drift over time. Use a data observability platform or custom scripts.

Monte Carlo can monitor schema changes, volume, and freshness. It provides APIs for contract-specific checks with alerts. When a breach occurs, the right people get notified immediately. Also establish a feedback loop. If requirements change or contracts prove too strict, iterate with a version bump and communicate changes to stakeholders.

Remember, implementing data contracts is a journey. Start with one critical pipeline, prove value, then expand. Each step reduces broken pipelines and increases deployment confidence. Teams that embrace this incremental approach find themselves with more reliable data and fewer emergency fixes.

Common challenges and how to overcome them

Implementing data contracts isn't always smooth sailing. Here are the real challenges teams face and practical ways to address them. The good news is that every organization encounters these same issues, and proven solutions exist. By understanding what to expect, you can avoid the common pitfalls instead of discovering them the hard way.

Organizational silos and resistance

The biggest challenges are organizational, not technical. Different teams have different priorities and may be siloed. When you ask producers to do extra work for data contracts, they'll naturally wonder "What's in it for me?"

Executive support and culture change are essential. Educate stakeholders on the cost of bad data by sharing incident post-mortems or quantifying data downtime. When leadership sees the real impact of data failures on business operations, support for contracts becomes much stronger.

Create incentives or recognition for teams that maintain good contracts. Include data quality metrics in engineering KPIs. When producers see that preventing downstream failures counts toward their performance reviews, they suddenly care a lot more about schema changes.

Defining the scope correctly

Finding the right balance is tricky. Too lax and the contract won't prevent issues. Too strict and it hinders development agility. Teams often struggle with this goldilocks problem.

Focus on the most essential schema and quality elements that truly impact downstream analysis. Leave out trivial constraints that don't add value. If a field is purely informational and nobody's dashboard breaks when it changes, it probably doesn't need strict validation.

Start with an MVP contract and iterate. Begin with core fields that absolutely cannot change without notice. Add quality rules only for fields where bad data has caused real problems. You can always tighten the contract later, but starting too strict kills adoption.

Ensuring all parties adhere

Even with contracts in place, people forget or bypass the process. This is partly a tooling issue but mostly a human issue. Early adoption is especially fragile when old habits die hard.

Make the correct path the easiest one. Provide simple libraries or templates that automatically format data to match the contract. If following the contract

requires less effort than not following it, compliance becomes natural. One team created a SDK that made contract-compliant data production literally one line of code.

Training and documentation matter too. Ensure new engineers learn about data contracts during onboarding. Include contract checks in your definition of done. When everyone understands the process from day one, contracts become part of the culture rather than an afterthought.

Dealing with evolving data

Data isn't static. New business requirements demand changes constantly. The challenge is avoiding contract sprawl or constant breaking changes that frustrate consumers.

Implement versioning and backward compatibility checks. Use schema registry features to ensure new versions don't break older consumers. A good registry will reject incompatible changes automatically, forcing teams to think through migrations.

Encourage additive changes over destructive ones. Adding new optional fields is usually safe. Removing or renaming fields breaks things. When breaking changes are unavoidable, plan carefully. Maintain two versions in parallel during a deprecation period. Give consumers time to migrate. Document the timeline clearly so nobody gets surprised.

Each of these challenges can be managed with the right approach. Teams that anticipate these issues and plan for them succeed. Those that expect smooth sailing from day one usually give up when reality hits. Set realistic expectations, start small, and build momentum through early wins.

What's next for data contracts?

Historically, data management within an organization has often been the responsibility of a dedicated team. Or, in some cases, the remit of just one plucky (and possibly overworked) data scientist. In such situations, data contracts weren't really necessary to maintain order.

As organizations move towards a data mesh approach – domain-driven architecture, self-serve data platforms, and federated governance – that's no longer the case. When data is viewed as a product, with different teams and sub-teams contributing to its upkeep, mechanisms to keep everything coupled and running smoothly are much more important.

The data contract is still a relatively new idea. They're an early attempt at improving the maintenance of data pipelines, and the issues that come from breaking down a monolith, so we'll probably see further iterations and other approaches emerge in the future.

For the moment, however, they might just be the best solution at our disposal for preventing data quality issues arising from unexpected schema changes.

+++
We highly encourage you to follow Andrew on LinkedIn and check out his website.

Thinking about data contracts also means thinking about how reliable your data is. To talk about data observability within your organization, schedule a time to talk with us below!

Our promise: we will show you the product.

Frequently Asked Questions

Why are data contracts important?

The most commonly cited use case for data contracts is to prevent cases where a software engineer updates a service in a way that breaks downstream data pipelines. For example, a code commit that changes how the data is output (the schema) in one micro-service, could break the data pipelines and/or other assets downstream. Having a solid contract in place and enforcing it could help prevent such cases.

Another equally important use case is downstream data quality issues. These arise when the data being brought into the data warehouse isn't in a format that

is usable by data consumers. A data contract that enforces certain formats, constraints and semantic meanings can mitigate such instances.

How do you use data contracts?

There are different data contract architectures and philosophies. One of the most effective implementations was built by Andrew Jones at GoCardless. The contract is in Jsonette, merged to Git by the data owner, dedicated BigQuery and PubSub resources are automatically deployed and populated with the requested data via a Kubernetes cluster and custom self-service infrastructure platform called Utopia.
