CDI-E: An Elastic Cloud Service for Data Engineering

Prakash Das
Informatica
pdas@informatica.com

Shivangi Srivastava
Informatica
ssrivastava@informatica.com

Valentin Moskovich
Informatica
vmoskovich@informatica.com

Anmol Chaturvedi
Informatica
achaturvedi@informatica.com

Anant Mittal
Informatica
amittal@informatica.com

Yongqin Xiao
Informatica
yxiao@informatica.com

Mosharaf Chowdhury
University of Michigan
mosharaf@umich.edu

ABSTRACT

We live in the gilded age of data-driven computing. With public clouds offering virtually unlimited amounts of compute and storage, enterprises collecting data about every aspect of their businesses, and advances in analytics and machine learning technologies, data driven decision making is now timely, cost-effective, and therefore, pervasive. Alas, only a handful of power users can wield today's powerful data engineering tools. For one thing, most solutions require knowledge of specific programming interfaces or libraries. Furthermore, running them requires complex configurations and knowledge of the underlying cloud for cost-effectiveness.

We decided that a fundamental redesign is in order to democratize data engineering for the masses at cloud scale. The result is Informatica Cloud Data Integration - Elastic (CDI-E). Since the early 1990s, Informatica has been a pioneer and industry leader in building no-code data engineering tools. Non-experts can express complex data engineering tasks using a graphical user interface (GUI). Informatica CDI-E is built to incorporate the simplicity of GUI in the design layer with an elastic and highly scalable run time to handle data in any format without little to no user input using automated optimizations. Users upload their data to the cloud in any format and can immediately use them in conjunction with their data management and analytic tools of choice using CDI-E GUI. Implementation began in the Spring of 2017, and Informatica CDI-E has been generally available since the Summer of 2019. Today, CDI-E is used in production by a growing number of small and large enterprises to make sense of data in arbitrary formats.

In this paper, we describe the architecture of Informatica CDI-E and its novel no-code data engineering interface. The paper highlights some of the key features of CDI-E: simplicity without loss in productivity and extreme elasticity. It concludes with lessons we learned and an outlook of the future.

PVLDB Reference Format:
Prakash Das, Shivangi Srivastava, Valentin Moskovich, Anmol Chaturvedi, Anant Mittal, Yongqin Xiao, and Mosharaf Chowdhury. CDI-E: An Elastic Cloud Service for Data Engineering. PVLDB, 15(12): 3319 - 3331, 2022.
doi:10.14778/3554821.3554825

Proceedings of the VLDB Endowment, Vol. 15, No. 12 ISSN 2150-8097.
doi:10.14778/3554821.3554825

1 INTRODUCTION

The confluence of an exponential growth in data volumes and the rapid rise of distributed cloud computing has fundamentally transformed the computing and data management landscape over the past two decades. The industry as a whole is moving away from software running on local servers toward software-as-a-service (SaaS) offerings running on shared cloud data centers managed by infrastructure-as-a-service (IaaS) providers to deal with increasingly large and varied data stored in elastic cloud storage. Naturally, SaaS data systems have to address multiple foundational challenges to take advantage of economies of scale, scalability, and availability of cloud computing and elastic storage.

On the one hand, there's a need to support a wide range of jobs that generate business insights from business activity data. These *data engineering* jobs feed data into analytic systems, other enterprise applications, and machine learning systems with both batch and near real-time timing requirements (Figure 1). To scale out these jobs to the highly parallel cloud computing resources, many distributed computation frameworks [5, 9, 39] have gained traction in recent years.

On the other hand, there's a need for data engineering jobs to handle a large volume of data in a variety of data formats residing in on-premise sources as well as in cloud storage. While data in transactional database systems are often structured, file formats used in the cloud can be semi-structured or even unstructured. Data records in these formats are often hierarchical, represented through nested data structures. These file formats are also open in nature, not always bound to a specific application system, allowing these files to be used more freely in different enterprise compute activities. A cloud data engineering service should support data processing across these wide range of formats and allow combining them for timely insights. At the same time, data volume is growing rapidly. With a large data volume, there is a need to use a larger number of compute servers for timely processing. A cloud data engineering service should elastically scale up or down to match data volume.

Of course, none of this is news. Existing solutions, however, fall short in *democratizing such tools to the non-experts* in an enterprise. In almost all cases, complex data engineering tools require expert-level knowledge to tackle the challenges of mixing and analyzing varied data formats at different scales. Often, the IT department of an enterprise holds the key to gaining timely business insights.

Since mid-1990s, Informatica has pioneered data engineering tools that enterprises use to perform these jobs in their on-premise data center using a *no-code*, graphical user interface (GUI) environment. With cloud computing becoming more prevalent, enterprises want these applications to operate as a cloud service. Some enterprises no longer want to manage these applications and compute servers even though underlying business processes allow them to do so. Users of these applications want to use similar GUI tools as a web application to run similar jobs that they were running on their private data centers, but on the cloud as an elastic SaaS solution.

Over the past decade, Informatica has transformed its no-code GUI solution for data engineering to be a cloud data engineering service for enterprise customers with the following characteristics:

- ***No-code* computing environment.** The service manages user-authored jobs on its own cloud storage. Users with a web browser can author data engineering jobs using GUI widgets, visually explore data files, run, monitor and debug jobs, without using any low-level programming language.
- **Efficient parallel processing.** The service translates user-authored jobs into Apache Spark [9] programs, which can execute in parallel, without users being aware of the underlying environment. Performance of the generated Spark programs is comparable to ones hand-coded by experts.
- **Hierarchical data processing.** The service supports a GUI for constructing semi-structured files, processing of semi-structured data, and making hierarchy processing available to non-programmers. This can be challenging even for programmers well-versed in writing Spark programs.
- **Automatic selection of tuning parameters.** The service automatically chooses a reasonable set of tuning parameters for a generated Spark program by performing experimental trial runs of the program, relieving lesser-skilled users from having to do the same manually.
- **Elastic compute cluster.** It expands and shrinks a compute cluster based on demands from the ongoing Spark jobs.
- **Data security and isolation.** The service runs jobs of different enterprise customers in isolated compute clusters. Customers own their input and intermediate data at all times.
- **Cost-efficient.** The service-managed computing cluster is ephemeral, benefiting from automatic selection of tuning parameters of jobs, automatic expansion/shrinking of compute nodes, with support for different types of node instances, including GPU and spot instances, provided by cloud vendors.

Outline. Section 2 provides a quick background on the challenges faced by a cloud data engineering service, including example use cases and design goals of our service. Section 3 describes the no-code GUI programming constructs as well as the overall architecture of the service. Section 4 highlights performance characteristics and distinguishing features: support for heterogeneous data sources, hierarchical data processing, auto-tuning and elastic scaling of jobs, optimization of data pipeline branching, and data isolation for enterprises. Section 5 discusses related work. Section 6 concludes the paper with lessons learned and a vision for the road ahead.

2 BACKGROUND

Informatica is an enterprise cloud data management company founded in 1993 focusing on data engineering and data warehousing. While the data warehousing industry at that time was mostly RDBMS-based with a SQL interface, Informatica pioneered an intuitive GUI-based tool for data engineering across diverse systems.

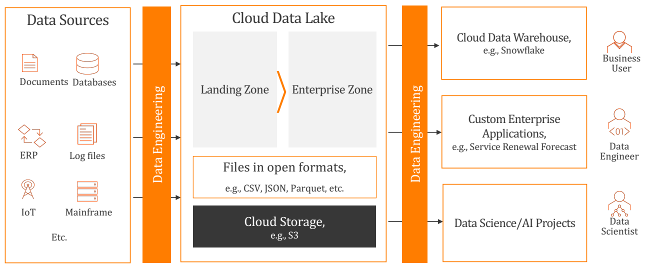


Figure 1: Cloud Data Lake usage scenario.

As the industry evolved to distributed processing and eventually to Hadoop [5, 41], Informatica retained the same GUI for customers while changing the underlying engine to Hadoop, Hive [6, 59], and eventually Apache Spark [9, 61, 62]. As more customers migrated to the cloud, Informatica presented them with a managed Cloud Data Integration – Elastic (CDI-E) offering described in this paper.

2.1 Data Lakes and Serverless Computing

With the availability of cloud storage such as AWS S3, Azure Blob/ADLS, and Google Storage [2, 12, 18], enterprises are building data lakes. Data in these storage areas are in open standard file formats, ranging from generic formats such as JSON, Parquet, ORC, Avro, and XML [3, 7, 8, 21, 27] to health industry-specific formats such as HIPAA-mandated [23] HL7 [24] to financial industry-specific format such as SWIFT-EDI [36]. Data lakes decouple data for further processing from the application that produces it. Data in these storage areas might have been populated through various means such as: (1) collecting web application's voluminous log data, traffic logs, or security audits; or (2) extracting data in a database of a custom business transaction system, application systems in finance, enterprise resource planning (ERP), or human resource (HR) management. Data in various formats needs either further processing for use in interactive analytics or building predictive machine learning models. Sometimes data processed in the cloud flows back to business applications. Data lake architecture facilitates distributed computing through disaggregated/serverless computing, allowing jobs to scale on demand as data volumes grow.

2.2 Challenges in Democratizing Data Engineering

Traditional solutions force users to interact with data from varied sources using high-level programming languages (e.g., Scala, R, Python) or SQL, which restrict data-driven decision making abilities to a small pool of experts. Just to explore data records and understand the content of a new data source often requires non-trivial coding. For instance, users might need to learn how to use vendor-provided libraries to extract and populate a cloud storage. They might also need to use libraries written in different languages and deal with variety of data formats. These solutions often do not handle issues such as data errors, data type conversion, and job failures. Library API used in handling data sources might be deprecated and replaced by a newer version, often because of functionality enhancements, performance, and security fixes that would require modification to or rebuilding of these programs. Programmers need to ensure that the coded solution scales with data volume during execution and that appropriate tuning parameter values are chosen based on data volume and compute resource availability. Considerable DevOps efforts are needed to take advantage of compute elasticity available from cloud computing vendors. Long story short, it is non-trivial for non-experts to incorporate data-driven decisions in business intelligence.

Customer Use Cases. The following use cases demonstrate a range of scenarios where Informatica customers successfully use CDI-E.

A publicly traded **investment management firm** receives a wide variety of data products such as Index Data, Reference Data, Sustainability Data, Fund Data, and Market Data from multiple data vendors in various formats, both structured and semi-structured, numbering 150,000+ complex JSON and XML files. The firm plans to modernize the data platform by moving some of vendor data processing into a cloud native infrastructure.

A **telecom company** that provides fixed-line broadband and mobile services with operations around 180 countries needs to read their main OLTP database and identify all the subscriber accounts that changed in the previous day, in order to prepare reports on the changed accounts. Their data processing jobs require subscriber data ingestion, without any modification to the operational database setup, to a cloud storage, support of a SQL *join* operation on tables of large size, the two largest tables being about 475GB and 140GB. The company needs to generate these reports in an hour or two, replacing the existing data processing job, implemented by programmers in a specialized vendor-specific scripting language.

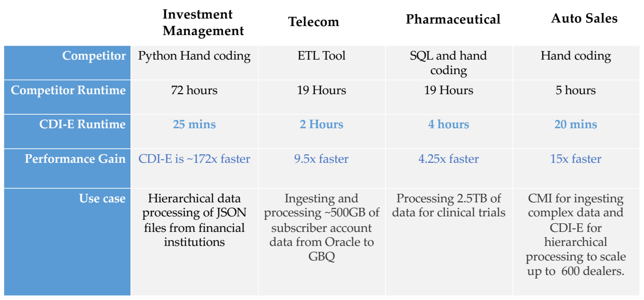
An American multinational **pharmaceutical and biotechnology company** needs to process clinical data on a disease. Data size resulting from clinical treatment in confirmed and unconfirmed disease cases is about 2.5TB. The company needs to process the data in a few hours for identifying new cases among unconfirmed and any red flag in confirmed cases because of the clinical treatment. The company prefers a no-code data processing solution.

An **auto sales company** provides centralized and remote car sales support to national rental and online purchasing platforms. As part of building their first Business-to-Consumer (B2C) car buying marketplace, as they grow with time, they want to scale the vehicle data ingestion process that include photos, to support larger number of dealers and vehicles in inventory. The company needs to process the ingested large volume of hierarchical data quickly to keep their inventory reasonably up-to-date.

An **oil company**, operating hundreds of wells and numerous drilling platforms, needs to ingest data from various systems into a cloud storage, storing it in an open-source format such as Parquet, forming a data lake. This includes ingesting time series data from Predictive Maintenance and Service (PdMS) of SAP, an Enterprise Resource Planning system; weather systems; and Supervisory control and data acquisition (SCADA) systems. The ingested data will be enriched with data from other databases and processed, feeding into analytical systems such as data warehouse to provide health checks on their rotating equipment to avoid unplanned events.

A **fast food restaurant company**, operating many brands and each brand having world-wide locations, needs to consolidate customer activity data into a cloud storage location dictated by local privacy laws for building predictive AI models and to improve productivity of model builders. Further consolidation to a central cloud storage is needed after anonymizing the data using custom logic for AI model building activity. The company does not want to own and manage any computing infrastructure for computing jobs.

Table 1: Representative CDI-E customer success stories.



	Investment Management	Telecom	Pharmaceutical	Auto Sales
Competitor	Python Hand coding	ETL Tool	SQL and hand coding	Hand coding
Competitor Runtime	72 hours	19 Hours	19 Hours	5 hours
CDI-E Runtime	**25 mins**	**2 Hours**	**4 hours**	**20 mins**
Performance Gain	CDI-E is ~172x faster	9.5x faster	4.25x faster	15x faster
Use case	Hierarchical data processing of JSON files from financial institutions	Ingesting and processing ~500GB of subscriber account data from Oracle to GBQ	Processing 2.5TB of data for clinical trials	CMI for ingesting complex data and CDI-E for hierarchical processing to scale up to 600 dealers.

Table 1 summarizes four of the above CDI-E customer stories.

2.3 Design Goals

When designing CDI-E, we focused on providing one solution to address the use cases above, and more, guided by four design goals.

- **Simplicity:** Data engineering jobs are getting more and more complex, but we want our solution to be approachable by experts and non-experts alike. We want the users to be able to express their jobs as easily as possible, and we want to make it simple to execute and maintain the jobs as well.
- **Productivity:** We want highly productive users. We want our solution to be efficient both during design time and run time, so that they can accomplish more in a given time frame.
- **Elasticity:** Data Engineering is constantly growing, and we want to make sure our solution can scale well in terms of data volume, number of concurrent jobs, number of users, cost incurred, as well as other scaling dimensions.
- **Reduced Cost:** Often problems can be solved by adding more hardware, but that comes with additional cost. So our focus is to optimize performance while trying to reduce cost.

3 INFORMATICA CDI-E

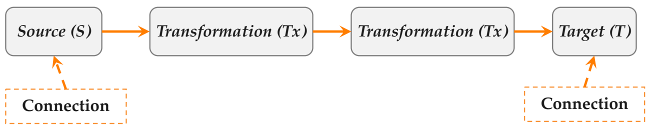
Informatica Cloud Data Integration – Elastic (CDI-E) is a GUI-based web application that users, developers, and administrators access through a web browser. Users create and edit CDI-E objects via drag-and-drop and by filling in templates to create executable data flow jobs that can be submitted to the CDI-E execution service.

3.1 CDI-E Abstractions

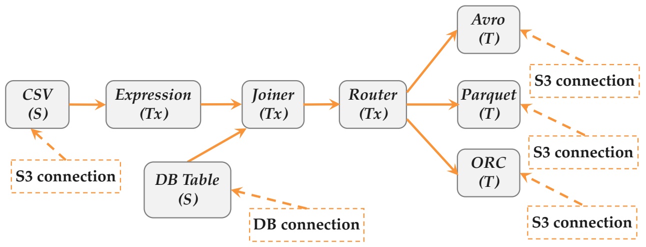
Users learn two sets of abstractions: (1) design-time abstractions to specify what a data engineering task does; and (2) runtime abstractions to specify parameter values and compute resources for the designed task when it executes.

3.1.1 Design Time Abstractions.

Mapping. A mapping is the central abstraction that users encounter. Users develop a mapping, which is a directed-acyclic-graph



(a) Simple mapping.



(b) Complex mapping with many sources and targets.

Figure 2: Examples of CDI-E mapping objects.

(DAG) of transformation objects, to describe the logic that an ETL-style job should perform. A mapping contains one or more sources to read data from, data transformation logic, and one or more targets to write data to after the specified logic transforms the data. Figure 2 depicts a few examples. The execution semantic of a mapping is similar to that of a SQL statement processing. The mapping transformations, described next, process a set of rows using a data model independent of actual underlying data sources.

Transformation. A transformation represents how a set of input rows are transformed into a different set of output rows. A Source transformation produces output rows, representing an underlying concrete data source that can be read. A Target transformation accepts input rows, representing an underlying data source to which rows can be written. A mapping must have at least one source and one target. Depending on the transformation, the number of output rows of a transformation might not be same as input rows. A transformation can accept multiple sets of input rows for an operation like a `Joiner`, or it can output multiple sets of output rows for an operation like a `Router`. Figure 2 shows types of available transformations. A set of transformations that process a distinct set of rows is called a *data pipeline*, or *pipeline*. If users want to leverage their own transformation logic, CDI-E provides a *Custom Transformation* similar to User-Defined-Functions (UDFs) in traditional RDBMS. However, the Custom Transformation API allows provisioning of additional associated metadata and code, like any other CDI-E provided transformation, that enables its participation in optimization and code generation stages of a mapping.

Connection. A connection represents a concrete data source, such as a relational database system like MySQL, a directory of JSON files, or a Salesforce application connection. During mapping development, users interact with data sources using connection objects to preview data in a data source on a web browser. This allows the developer to understand the row structure of the underlying data and data types associated with each field, aiding rapid prototype development. Developers then map these fields to the data types that the mapping data model supports by associating a connection object with either a source or target, which converts data types to the data types of the mapping data model when necessary.

Parameterized object. The parameterized object abstraction allows developers to create objects with fields that contain only placeholder marker values, known as parameter fields. They are bound with actual values at run time via late binding. A parameterized field can be part of any of the type of objects that users encounter. We elaborate more on this later when we describe mapping tasks.

This parameterization concept extends to late binding of input and output row structure of transformations associated with a mapping, leading to a concept known as a *dynamic mapping*. To apply logic such as grouping or aggregation, transformation functionality needs to refer only to specific fields of input and output rows. The transformation can be very flexible about the rest of the fields comprising a row. When users create a transformation object, they can choose to define how to treat additional fields as part of input and output row structures. For example, an available rule is to pass additional fields as-is in a pass-through manner. The actual structure of input and output rows through each transformation is determined at run time. The schema of a Target transformation might also be affected, which is handled by connection's run time associated with that Target transformation.

This parameterized row structure functionality also promotes reusability of a mapping object that encapsulates row processing logic. The same dynamic mapping object that processes a JSON file can also be reused to process a relational table as long as the fields that transformation refers to exist in the row by name.

3.1.2 Runtime Abstractions.

Mapping task. A mapping task represents an executable mapping by binding all parameterized objects to actual values, including binding all sources and targets to connection objects. This abstraction provides the capability to bind connection objects associated with a mapping so that the same mapping logic can be run in a different execution environment. For example, a developer typically has access to a sample test data set but not to the actual data set of an enterprise due to security implications. This parameterization allows an IT system administrator, with access only to the actual data, to substitute those connection objects with the actual enterprise production environment's connection object.

Compute cluster. Users create a compute cluster object to run all mapping tasks in a shared cluster. They configure properties such as the type of compute nodes available from cloud vendors and the minimum and maximum number of compute nodes, preventing runaway compute costs. A compute cluster, associated with a cluster object, starts and shuts down based on demand for running the specified tasks. Development and production compute clusters are separate, thereby providing flexibility to manage costs. For example, users can specify the type of nodes in a cluster with GPU support, compute- or memory-optimized nodes, as well as cost-optimized nodes such as spot, on-demand, reserved, etc.

3.2 User Workflow

To perform a data processing job, users create a mapping object and a compute cluster object, which can be shared, to run it on. To

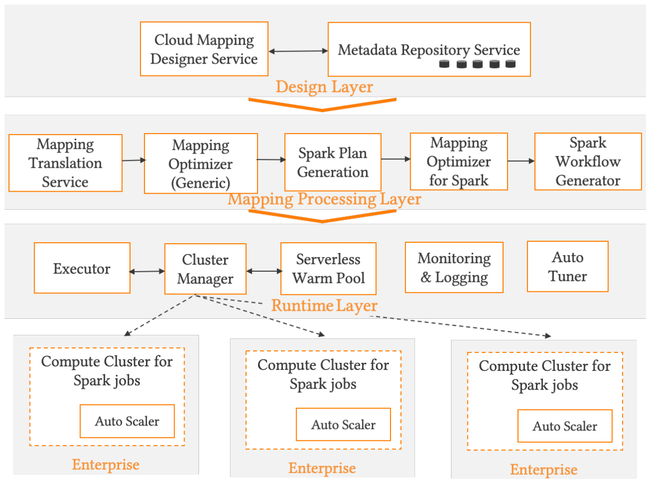


Figure 3: CDI-E Architecture: The top layer manages design and runtime abstractions. The middle layer processes abstractions into executable code. The bottom layer manages job executions on a cluster.

create a mapping, users associate connection objects with Source and Target transformations that represent data sources. Between sources and targets, users add transformations that reflect what they need a data processing job to do. During this design phase, users get information through the GUI about mapping validity, error conditions, and corrective actions. During design-time, users can preview data associated with a transformation in a mapping. When they preview data, the runtime engine processes the mapping up to and including the selected preview point and displays the configured number of rows to view. Using the same GUI, the user can monitor the progress of a mapping and can access to detailed execution plan and log files of that execution. Typically, an administrative user creates a compute cluster object to share with users who develop mappings. There can be multiple such cluster objects of an enterprise as needed. No user intervention is required to manage the life cycle of a cluster unless there are problems in cloud infrastructure, which is handled by the administrative user. Users creates a mapping task when a mapping is ready to deploy to production using parameters such as connection objects.

3.3 Architecture

Our architecture consists of design, mapping processing, and runtime layers. The design layer allows end users to express their requirements through a graphical interface. The mapping processing layer performs parameter resolution for design time objects, translation to underlying execution engine's execution plan and optimization. Finally, the runtime layer elastically runs the job.

3.3.1 Design Layer.

Cloud Mapping Designer Service (CMD). Users interact with this component in a web browser to author and edit CDI-E objects introduced in Section 3.1. These are serializable, modeled objects that are amenable for interactive editing and fast persistence. CMD also supports multiple concurrent users and permissions assignment.

Metadata Repository Service. All objects authored using CMD are persisted in a system-managed repository that is shared among all our enterprise customers. Enterprises typically have different functional computing environments such as development (Dev), quality-assurance (QA), and production (Prod). This service supports usability of objects across computing environments by allowing the export and import of objects across environments. Parameters, part of the parameterized objects, allow different parameter values to be bound, as per the computing environment's need, making parameterized objects reusable across computing environments. This repository service supports versioning and maintains the life cycle of design objects.

3.3.2 Mapping Processing Layer.

When a user or scheduler initiates execution of a mapping task, the components of this layer process the objects for semantic analysis, optimization, and code generation, analogous to how a SQL execution engine typically processes a SQL query. The Mapping Processing Layer can process native and custom logic guided by additional metadata associated with each transformation such as: (1) whether execution logic has side-effects, (2) whether it is a 1-to-1 or m-to-n transformation; or (3) whether it can be partitioned.

Mapping Translation Service. Parameter fields are fully resolved, known as materialization, using values that are part of an execution request, achieving late-binding capability. After materialization, Source transformations are bound to specific data sources, making the fields of a row fully defined for a mapping that uses the mapping template. The representation of a materialized mapping, still a modeled object, undergoes conversion into an internal data structure, amenable for semantic analysis and optimization, independent of any specific execution engine.

Mapping Optimizer (Generic). The underlying execution engine-agnostic optimizations performed by this component aim to reduce execution time to process rows. Examples include pushing predicates used in mapping transformations, such as a Filter transformation, closer to the Source transformations. This optimization reduces the number of rows to be processed earlier (*early selection*), removes unused fields from a row as soon as subsequent transformations stop using the fields (*early projection*), and evaluates constant expression used in a transformation, for example, field1 = 5 + field2 + 7 to be field1 = field2 + 12 (*constant folding*).

Spark Plan Generation. The execution service uses Apache Spark for mapping execution. For some mappings, such as ones with a Router transformation, mapping execution might require more than one Spark job to achieve the intent. Thus, this component converts mapping into a Spark-ASG, analogous to an Abstract Syntax Graph (ASG). Each node in this graph represents an operation available in Spark, as a preparation for Spark engine-specific optimization and Spark program generation.

Mapping Optimizer for Spark. Next, the Spark-ASG is traversed and Spark execution engine-specific CDI-E optimizations are applied. For example, intermediate data staging points between Spark jobs are added only when absolutely necessary, due to a Router transformation branching scenario (elaborated in Section 4.6).

Table 2: Network boundary options for a Compute Cluster.

Option	**Cluster Location**	**Managed**	**Elastic**	**Cluster Control Access**	**Enterprise Data Access**
Enterprise VPC	Enterprise VPC	Yes	Yes	HTTPS tunnel	Same VPC
Serverless Warmpool	CDI-E VPC	Yes	Yes	Same VPC	Special VNIC

Spark Work Flow (WF) Generator. After Spark engine-specific optimization, this component renders a Spark-graph into a DAG of Spark programs that use *dataframe* operations that utilize spark-native row representation in Scala. Each node in the Spark-ASG is rendered into Scala code. The DAG, representing a mapping task in execution, is submitted to the runtime layer.

3.3.3 Runtime Layer. Each Spark job in a DAG is executed on a Kubernetes cluster using a modified version of Spark-on-Kubernetes [35] resource controller and other components described below.

Executor. The executor acts as a coordinator to run a mapping task as a DAG of Spark jobs. It interacts with the Cluster Manager to set up the cluster for execution before submitting a Spark job. It tracks individual execution monitoring statistics and status of each Spark job, as well as overall status and statistics for a mapping task.

Cluster Manager. This component starts up an ephemeral Kubernetes cluster, if not already running, specific to an enterprise as defined in a cluster object (Section 3.1), on a request from the executor component. The execution of multiple mapping tasks from the same enterprise uses the same cluster. Each enterprise has a compute cluster that is isolated from the compute clusters of all other enterprises. The Cluster Manager is responsible for the life cycle of the cluster, including scaling the cluster (both compute and storage), using the Auto Scaler and shutting it down when it is not in use.

As shown in Table 2, each customer has an option to choose a cluster's operating network boundary, i.e., *Virtual Private Cloud* (VPC). The choice does not affect functionality, but affects network setup for control and data channel messages. Enterprise customers who want to use cloud storage and databases but not manage anything related to data processing jobs prefer the *Serverless Warmpool Option*, for which the Cluster Manager manages compute nodes in a warm state in a common pool for a faster cluster startup time. Security aspects of both options are discussed in Section 4.7.

Auto Scaler. Auto Scaler is a logical grouping of interacting components that manage functionalities related to cluster scaling, Spark shuffle storage scaling, Spark job scheduling, and Spark job scaling. It assumes that the cluster is being used exclusively for jobs managed by this system. Modified open-source components used to achieve auto-scaling are discussed later in the paper.

Auto Tuner. Auto Tuner manages the tuning parameters of a Spark job that results from running a mapping task repeatedly on demand or at a scheduled interval. It records job completion time and associated Spark tuning parameters chosen by its algorithm for each run. The component uses a stochastic algorithm to explore the tuning space, detailed later in the paper, to set the tuning parameters for each run of a Spark job.

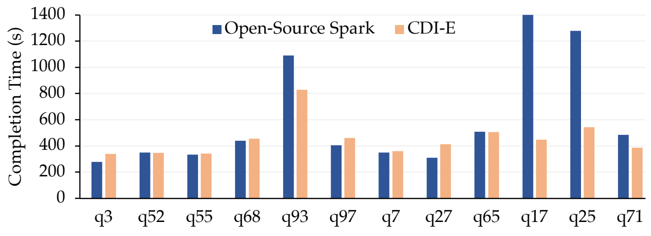


Figure 4: CDI-E performance on TPC-DS queries.

Monitoring and Logging. History of executed mapping task's execution logs and job statistics are made available to other components through this one. It also manages expiration and reclamation of storage associated with these artifacts.

4 FEATURE HIGHLIGHTS

We start by providing some end-to-end benchmark numbers that demonstrate how CDI-E can provide simplicity without efficiency loss. Throughout the rest of this section, we discuss selected flagship features of CDI-E that deals with complex targets and data formats, automated tuning and scaling, runtime optimizations, and data security and isolation. We conclude by highlighting some performance metric improvements observed by our customers.

4.1 Performance Comparison Using CDI-E

Using CDI-E, users can expect performance similar to that of manually writing Spark SQL queries. Figure 4 shows that CDI-E allows non-expert users to express complex TPC-DS benchmark queries using mapping objects using a GUI while achieving high performance. For the purpose of evaluation, a subset of TPC-DS[38] queries with different complexities were run by dividing them into three buckets:

- Simple (Q3, Q52, Q55, Q68, Q93, Q97): These queries have simple operations and require no shuffle.
- Medium (Q7, Q27, Q65): These queries require a single shuffle operation combined with other simple expressions.
- Complex (Q17, Q25, Q71): These queries have multiple shuffle operations and other complex expressions.

The benchmark was run with 1000GB of Parquet data on a 1 to 20 node 16 vCPU, 64GB (m5.4xLarge) auto-scaling cluster on AWS.

If a compute cluster has GPU support, users can leverage them to run mappings (Figure 5). In addition to performance improvements, users experienced up to 72% cost reductions as well. The evaluations were run using TPC-H[38] SF100 Lineitem data on a 10-node cluster. `g4dn.2xlarge` instances were used for GPU and `m5d.2xlarge` for CPU configurations.

Users can take further advantage of the spot instances to reduce execution costs by 56%, 70%, and 65% for `m5.2xlarge`, `t3.2xlarge`, and `c5.4xlarge`, compared with their on-demand counterparts.

4.2 Heterogeneous Data Sources and Targets

Connector. Being able to connect to a wide variety of data sources is critical for any Data Engineering product. For example, the oil company use case mentioned earlier required connectivity to SAP, various SQL databases, flat and hierarchical files, and real-time systems. CDI-E allows disparate data source systems to be added as a plug-in module known as *connector*. These data source systems range from on-premises database systems such as Oracle and DB2 to cloud analytic query systems such as Redshift, BigQuery, and Snowflake to files of different format on cloud storage system such as AWS S3, Google Storage, and Azure Blob. Each file format such as Parquet, ORC, JSON, XML, and CSV combined with a specific file storage system is treated as a different connector. CDI-E also supports its own *flatfile* format specification. Each plug-in connector library module, using either Spark Datasource v1 or v2 API, provides underlying runtime capability to read and write to a specific type of data source. Users do not use these connector library modules directly. Instead, an instance of the data source manifests as a connection object, typically associated with a source or target transformation, in a user application. Users create a connection object, providing URL, authentication information, and any other required metadata specific to a data source, such as character encoding.

Data source preview. Before any data transformation logic can be written, users need to understand the record structure of first few records during mapping development time. Users can view this data with just having access to a connection object during development of a mapping. Again, this is facilitated by a connector library, and it's accomplished without users having to code and run an explicit job for this purpose.

Data migration and file format conversion. With various connectors available, data migration from one data source to another or file format conversion is simple using a mapping. In order to migrate data from an Oracle DBMS to a Redshift instance, users simply create a mapping with a Source transformation pointing to an Oracle instance and Target transformation pointing to a Redshift instance. Users will create a similar mapping to convert a file containing records in JSON format to a file in Parquet or ORC format, which is more suitable for analytic query. Only the connection objects associated with Source and Target transformations will be different.

Optimization support. Each connector module, depending on an underlying data source's capability, allows pushing some transformation functionality of a mapping, such as filter, expression, or aggregation to the source system for optimized execution. This is possible because a mapping captures the user's logical intent instead of a physical execution plan. Sometimes, when both Source and Target transformations use the same relational system and mapping logic uses transformations found in SQL, the entire execution takes place within that data source system.

4.3 Hierarchical Data Processing

Hierarchical data processing is a common operation in large enterprise settings. As seen in a couple of customer use cases of Section 2, some data engineering use cases require processing of JSON, XML, Parquet, or industry-standard hierarchical files. Application-specific

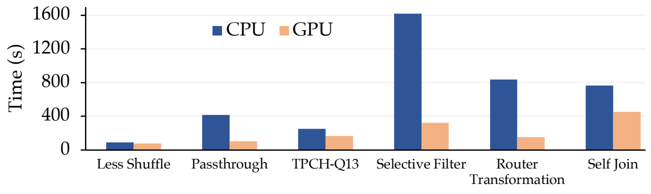


Figure 5: Mappings benefiting from GPU support in CDI-E.

hierarchical data standards such as HL7 or SWIFT EDI can be very complex. Users need visual help to understand data conforming to these standards, as well as data in files conforming to hierarchical structures such as Parquet, ORC, and Avro. Once a hierarchy structure is identified, the logic of the hierarchy processing job needs to be coded. CDI-E helps users to code such processing logic visually.

4.3.1 *Identifying Hierarchy Structure.*

When a user points a specific data source, such as a file in XML, Json, Avro, or Parquet format, to *Intelligent Structure Discovery* (ISD) component, which is integrated with the corresponding *Connector* for that data source, ISD applies an algorithm based on machine learning to generate a visual hierarchical model structure, and it infers name and data type of each element of the hierarchy. Users can edit the inferred data type, e.g, *number* instead of *string*, *date* instead of *long*, etc. Such visual representation is useful, especially for cases when a hierarchy has thousands of elements, which is common for large enterprise customers. Users can decide which elements they need, and they can refine the hierarchical structure of each row, out of which ISD captures the schema in an *Intelligent Structure Model* (ISM) object. This model object can then be associated with a *Source* transformation, fixing the row structure that subsequent transformations can operate on. The *Connector* run time leverages the information in the ISM object for parsing the input data to produce the desired row that Spark engine can process.

Schema drift support. The implied schema of a semi-structured file changes over time, for example, application logs in JSON format. Data may have newer fields that were missing in the sample data source. In this case, ISD passes along the additional data in a placeholder column instead of discarding it, and ISD continues processing the input instead of failing the job. Users can choose how to handle such additional data while designing the mapping.

4.3.2 *Processing Hierarchical Data.*

There are three main categories of jobs that require hierarchy processing logic: (a) Hierarchy building from relational tables (H2R); (b) Extracting relational tables from a hierarchy (R2H); and (c) Transforming one hierarchy into another hierarchy (H2H). Our system has a *Hierarchy Processor* transformation to aid users perform such processing in a mapping. It allows users to build a row structure that supports *Struct*, *Array*, and *Map*. At run time, this row structure translates to Spark's support for this type of row structure. The Hierarchy-to-Hierarchy transformation, such as auto-conversion of HL7 [24] message into FHIR [22] format, as required by one of our customers, is complex. For simplicity, we illustrate our hierarchy processing transformation's functionality with a relational-to-hierarchy building example. Let's say we are using the TPC-DS dataset with `Customer`, `Orders`, and `LineItem` tables. We want to convert this relational data into hierarchical format, as shown in inset of Figure 6.

Manually Building Hierarchy Using Spark SQL. We first outline steps that such a Spark program would have. We omit detailed code due to space constraints, while trying to convey the laborious nature and cognitive complexity.

Step 1: Construct Spark *DataFrames* `order`, `customer`, `lineitem`, with columns of each dataframe cast to the appropriate data type. This can be laborious if most columns need to be converted to a data type with such an expression: `order.columns(7).cast(IntegerType)`.

Step 2: Perform *join* operations between `order`, `customer`, and `lineitem` DataFrames using `custkey` and `orderkey`, resulting in `ord_cust_litem` DataFrame that has repeating column values from `customer` and `order`. Only `lineitem` column values do not repeat, which need to be converted into a column of Spark `lineitem_struct` data type, done in the next step.

Step 3: Create `lineitem_struct` data type with 15 fields, each field having a defined Spark data type. Then construct a new DataFrame `ord_cust_litem1` with a new column using `struct()` constructor function with input from 15 `lineitem` columns of `ord_cust_litem`.

Step 4: Create a new data type `ord_wi_litem_struct_type` to nest line items (`lineitem_struct[]`) within each order, needed for aggregating result after performing *groupBy* using `orderkey`.

Step 5: Perform a *groupBy* operation on `ord_cust_litem1` dataframe using `orderkey` and aggregation function utilizing `last()` to repeat order column values and `collect_list()` to collect `ord_wi_litem_struct_type` values as one item. This resulting aggregate value is needed for `struct()` constructor function. On this resulting dataframe, apply `struct()`, and then `cast()` that to `ord_wi_litem_struct_type` to get the next dataframe. Let's call this DataFrame `ocl_grp_order_id_struct`.

Step 6: Perform a *groupBy* operation on `ocl_grp_order_id_struct` dataframe using `custkey` to create the final dataframe, suitable to be written in a format supporting hierarchy such as Parquet. The operations are similar to the previous step. For this step, `collect_list()` collects `ocl_grp_order_id_struct` column value, from which a new column of type `ocl_grp_order_id_struct[]` is created.

Automatically Building Hierarchy Using CDI-E. Figures 6 and 7 show the solution in CMD (§3.3.1). We omit the figure for building level 2 hierarchy, which is similar to Figure 7, for brevity. Users can utilize the `HierarchyProcessor` transformation to build the hierarchy structure from the fields of `customer`, `order`, `lineitem` tables, using UI actions such as drag-and-drop. Users concentrate on specifying the parent-child relationship of the hierarchy elements. Figure 7 shows how users specify the parent-child relationship of an *array* field. These pieces of information are captured in a model object of `HierarchyProcessor`, which is later translated into a Spark program, inferring *join* and *groupBy* expressions, similar to what we described earlier. Thus the performance of a mapping used in hierarchy processing can be same as that of a Spark program developed by a programmer. The *Hierarchy*

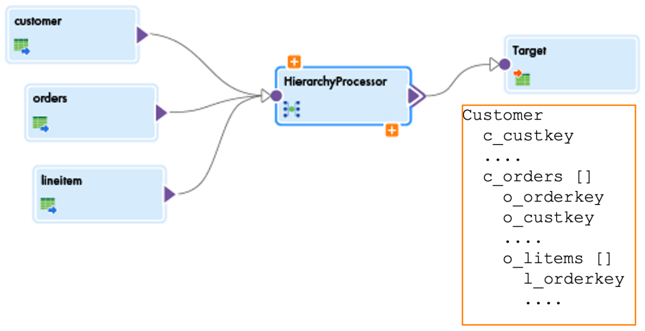
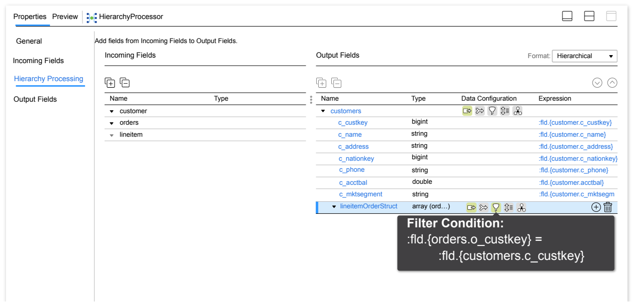


Figure 6: Hierarchy building mapping.


Properties | Preview | HierarchyProcessor

General
Incoming Fields
Hierarchy Processing
Output Fields

Add fields from Incoming Fields to Output Fields.

Incoming Fields

Name	Type
customer	
orders	
lineitem	

Output Fields — Format: Hierarchical

Name	Type	Data Configuration	Expression
customers			
c_custkey	bigint		:fld.{customer.c_custkey}
c_name	string		:fld.{customer.c_name}
c_address	string		:fld.{customer.c_address}
c_nationkey	bigint		:fld.{customer.c_nationkey}
c_phone	string		:fld.{customer.c_phone}
c_acctbal	double		:fld.{customer.acctbal}
c_mktsegment	string		:fld.{customer.c_mktsegm
lineitemOrderStruct	array (ord...)		

Filter Condition:
:fld.{orders.o_custkey} =
:fld.{customers.c_custkey}

Figure 7: Level 1 structure of hierarchy building mapping.

Processor transformation can also be used to filter and aggregate array elements of a hierarchical structure.

4.4 Auto Tuning

It is critical for production Data Engineering jobs to be tuned optimally. For example, in the investment management use case that was using hand-coded Python scripts, optimal tuning improved performance from dozens of hours to minutes. However, tuning a Spark job can be a challenging task since it is dependent on a large set of configuration parameters. An inefficient set of values can have adverse impacts, ranging from performance loss to failure of the executed jobs, such as running out of memory. Performance loss can often go unnoticed, leading to added costs in terms of resource usage and slower pipelines. Tuning is not a one-time operation either. As the engine evolves and adds features, more optimization opportunities are available, and some older configurations may not apply. Also, changes in data volumes impact configurations. It makes tuning a continuous process, and manually keeping up with many configuration options is challenging.

In order to avoid having to manually tune Spark jobs, we created Auto Tuner. Auto Tuner is a feature in the product that allows users to trigger automated tuning for any of their Spark jobs. It allows iterative generation of sets of job configurations which include Spark properties as well as properties specific to the Informatica Spark engine. These properties control resource parameters such as memory, and CPU cores as well as optimization parameters such as compression and persistence. The tuned Spark job is the original job with the best set of generated configurations attached to it.

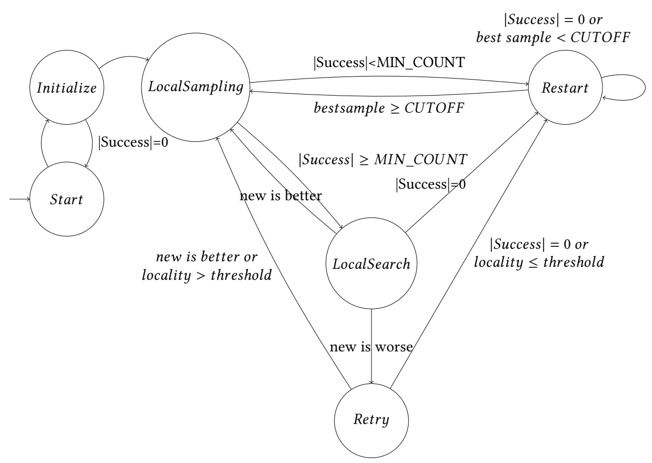


Figure 8: State machine diagram of Auto Tuner.

At its core, Auto Tuner runs a persisted state machine-based exploration process [60] to sample the configuration space and continuously identify beneficial configurations. It has a *Sampler* that can pick *n* samples from a given *configuration space* in *d* dimensions, where *d* is the number of configuration parameters under consideration. We use Latin Hypercube Sampling (LHS) [56] because it can cover all dimensions with a fixed number of samples. Given a few samples, Auto Tuner uses the *MinimaGenerator* to generate a new point in the search space, which is likely to be the minima for the target function. We use a quadratic minima generator, which assumes that the target function follows a quadratic curve for each of the configurations independently. For points close to each other, this works well. A quadratic function is fitted for each of the configurations and corresponding minima is identified and used as the value for that specific configuration.

4.4.1 *State Machine.*

Auto Tuner performs tuning using a state machine (Figure 8.) Every state generates a set of configurations. The framework executes the job with these configurations and gets the value of the target variable for each of these configuration sets. The values act as inputs to the next state.

- **Start**: Use Sampler to generate, say, 5 samples from the complete configuration space. Set the state as *Initialize*.
- **Initialize**: If all the previous samples failed, go back to *Start*. Otherwise, set the best one as the `currentBestSample`. Set `localitySize` to 50% of the total configuration space. Create a locality around the `currentBestSample` of size `localitySize`. Generate `SAMPLE_COUNT`, currently set to 5, samples from the locality using the Sampler. Set the state as *Local Sampling*.
- **Local Sampling**: If the number of successful samples is less than `MIN_COUNT`, go to state *Restart*. The value of `MIN_COUNT` is 3 was chosen based on the requirements of the MinimaGenerator used, but it can be varied. Otherwise, generate a sample using the MinimaGenerator with the successful samples. Set the state as *Local Search*.
- **Local Search**: If there are no successful samples then go to *Restart*. Otherwise, check if the new sample is better than previous successful samples. If it is, then set the new sample as `currentBestSample` and create a locality around the `currentBestSample` of size `localitySize`. Generate 5 samples from the locality using the Sampler. Set the state as *Local Sampling*. If the new sample is not better than previous samples, then add it to the set of previous successful samples and generate a sample using the MinimaGenerator with these successful samples. Set the state as *Retry*.
- **Retry**: If there are no successful samples, then go to *Restart*. Otherwise, check if the new sample is better than previous successful samples. If it is, then set the new sample as `currentBestSample` and create a locality around the `currentBestSample` of size `localitySize`. Generate 5 samples from the locality using the Sampler. Set the state as *Local Sampling*. If the new sample is not better, then reduce the `localitySize` to 80% of its size. If locality size is less than 10% of the total configuration space, then go to *Restart*. Otherwise create a locality around the `currentBestSample` of size `localitySize`. Generate samples from the locality using the Sampler. Set the state as *Local Sampling*.
- **Restart**: If there were successful samples in the last run and the best of those is better than `CUTOFF` of all the samples seen so far, then set it as the `currentBestSample`. Set `localitySize` to 50% of the total configuration space. The value of `CUTOFF` is chosen to be 80% after experimental analysis. Create a locality around the `currentBestSample` of size `localitySize`. Generate five samples from the locality using the Sampler. Set the state as *Local Sampling*. Otherwise, generate samples using Sampler with 50% of the total configuration space. Set the state as *Restart*.

4.4.2 *Performance Benefit.*

In order to see the real-world benefit of Auto Tuner, we compared it against manually tuning Spark jobs by Spark experts. A critical criterion for practical success was future proofing. Auto Tuner was initially tuned using a specific set of configurations. During testing, a larger and partially different set of configurations was used to evaluate the performance gain.

In order to be suitable for practical use, a requirement for Auto Tuner is to be able to give a decent prediction with a small number of samples. Hence, it was tested allowing only 10 samples here with a relatively small data set of SF1 (~1GB). The benefit increases with increasing sample and data sizes. Evaluation on a large variety of scenarios and benchmark queries showed sizable benefits (Figure 9).

4.4.3 *Cost Analysis.*

For the jobs that get a reduction in run time after tuning, it takes a few runs of the Spark job for it to become profitable. This is to recover the initial cost spent in running the iterative jobs spawned by Auto Tuner. We can represent the threshold of number of runs it takes after which Auto Tuned jobs start becoming profitable as follows:

$$Threshold\ Runs = \frac{Initial\ Runs}{1 - Reduction\ Factor}$$

A simplifying assumption in this formula is that the initial runs on an average take the same time as an Spark job run prior to tuning. In practice, however, they generally take much shorter time as the tuning happens incrementally so the threshold is even better.

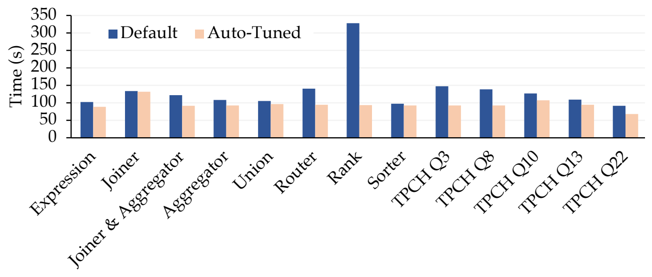


Figure 9: Performance improvement with Auto Tuner.

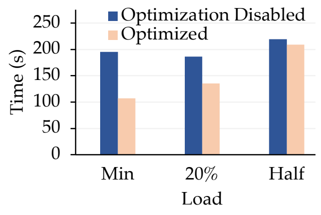


Figure 10: Performance benefit of dynamic partitioning.

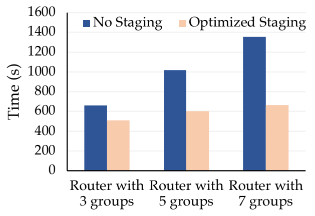


Figure 11: Performance benefit of optimized staging.

For example, let us consider the Rank job in Figure 9. Auto Tuner reduced the job completion time by around 28%. This gives us a reduction factor of 0.28, and it took Auto Tuner 10 iterations to achieve this, which gives us the threshold of 14 runs. Meaning, after running 14 scheduled iterations (or fewer) of the Spark job, we would have already recovered the initial cost spent.

4.4.4 Future Work. Going forward, we plan to do inter-job optimization by identifying similar processing subsets across jobs and start from an optimum point in the tuning space.

4.5 Managed Elastic Scaling

Auto Scaler performs infrastructure scaling and job scaling for optimizing performance and cost without users having to do upfront capacity planning. Many Data Engineering use cases such sales and retail have vastly varying workloads at different dates and times, so being able to scale to fit the workload becomes critical.

4.5.1 Infrastructure Scaling. Automatically scaling the infrastructure involves scaling both the compute and storage resources. In addition, we include a scheduler to do both in tandem.

Compute Scaler. We use a modified version of Kubernetes' cluster auto-scaler program to scale a cluster up and down. This scaler relies on the resource requirements, both memory and CPU to determine if more nodes are required for processing. The node scaling algorithm ensures the correctness of job execution. During downscaling, it checks for nodes where no jobs run, and it tracks where the intermediate shuffle data for a node might be stored. If a node hosts intermediate data, then the node is not removed from the cluster until the subsequent stage of a Spark job has consumed the data. This ensures that no Spark task needs to be rerun.

Storage Scaler. Spark jobs may use disk storage to store intermediate data such as shuffle service data and persisted Spark RDDs. It is hard to predict the maximum storage size needed by each compute node due to unpredictable job size and data skew during shuffle stage. Because lack of storage on a compute node leads to job failure, it is perhaps a more critical resource than a compute node, lack of which results only in slow execution. Besides, storage costs can be high – on AWS, one `t2.2xlarge` with 1TB general-purpose SSDs (EBS-backed) can cost around $1,500 per month. Storage scaler reduces overall storage cost without user intervention.

It constantly monitors the disk usage of each node and dynamically attaches additional storage volumes as need arises. This is always bounded by the configured maximum storage size per node. The increase of storage is achieved by leveraging the logical volume manager (LVM) [29][30] feature from the Linux kernel with no noticeable impact on job performance.

To further save storage costs, it also downscales a node's storage when its disk usage shrinks for an extended period of time. During downscaling, since disks are unmounted, the scaler needs to wait for an uninterrupted idle time period. For this, the scaler indicates to Kubernetes not to schedule any pods on the node being downscaled using Kubernetes' taints[37] mechanism.

Scheduler. We use a modified version of Kubernetes' default scheduler [28] for pod scheduling. It assumes that it is scheduling only Spark jobs on the cluster. The default scheduler schedules Kubernetes pods across all nodes in a uniform manner. Instead, our scheduler packs Spark Drivers or Executors (pods) on a node for maximum utilization before choosing another node without sacrificing performance. This allows aggressive downscaling.

4.5.2 Job Scaling. Parallelism in a Spark job is determined by the number of partitions in its source data. Our custom Source Partitioner dynamically determines the number of partitions. First, it takes into account the compute cluster's idle resource capacity: with more idle capacity, it increases the number of partitions by reducing size of each partition from Spark's default 128MB. Second, it handles a very large number of small files by batching them into a single partition by applying a lower bound on each partition size. Fewer Spark Executors also allow aggressive downscaling of a compute cluster. Users do have the option of manually tuning a Spark job's partition count for desired resource use, allowing for aggressive up-scaling of a compute cluster.

Figure 10 shows the comparison between default job scaler that decides partition size based on total cluster capacity and Informatica Job Scaler with dynamic partitioning. The evaluations were run on AWS using a 10 node `m5.2xlarge` compute cluster for jobs requiring less than 20% total cluster capacity, 20% cluster capacity, and 50% cluster capacity.

4.5.3 Future work. We are working on features like forecasting the need to speed up cluster startup and scaling times, allowing users to have warm pool of instances, and allowing heterogeneous instances and custom schedulers to suit their specific use-cases. Additionally, while the current scheduler works with Spark jobs, we plan to expand to different types of jobs.

4.6 Pipeline Branching Optimization

Data is often sent to multiple downstream pipelines to apply different transformations, or to load the data into various targets. A Router transformation directs the data based on a conditional expression, in which downstream pipelines get a row sent to it.

Spark, a pull-based processing engine, is not designed to handle such use cases very efficiently due to its lazy evaluation technique. For example, Apache Spark executes a job with multiple branches to generate multiple Spark jobs, one for each loading target. Each job re-executes the shared upstream logic to run these jobs in sequential order. Let us consider a use case, where an organization wants to split their incoming data into three branches for non-GDPR requirement country, GDPR required European country and, GDPR required non-European country.

A representative code for this use case is shown below. Even for this simple use case, three Spark jobs are created.

```
val df = spark.read.parquet ("gdprdata.parquet")
df.where(df("gdpr") === "no")
        .write.parquet("nogdpr.parquet")
df.where(df("gdpr") === "yes" && df("eur") === "yes" )
        .write.parquet("gdpreur.parquet")
df.where(df("gdpr") === "yes" && df("eur") === "no" )
        .write.parquet("gdprnoeur.parquet")
```

There are two main concerns related to pipeline branching:

- This may result in sub-optimal performance due to repeated computation and sequential evaluation, especially when the computation cost of the common upstream pipeline is high.
- Data consistency can also be challenging if some data transformation in the common upstream pipeline does not produce deterministic output, such as random data or Universally Unique Identifier (UUID) generation per row.

Apache Spark Solution. A programmer can persist Spark RDD representing the common upstream pipeline to avoid repeated execution, but Spark program execution semantics does not guarantee it. For example, Spark can re-execute an RDD to manage the implicit persistence store space, leading to possibly inconsistent data silently. If the programmer uses an alternate staging area to avoid repeated execution, the performance of downstream pipelines is not optimal if each pipeline reads the entire staged data. Each downstream pipeline is interested only in subset of the entire staged data. The more downstream pipelines there are, the less efficient the solution is unless the programmer implements an optimized solution for this problem like our service does.

CDI-E Solution. This benefits our users without any extra effort from a user during design time. Mapping Processing Layer can detect downstream pipeline branches. Mapping Optimizer for Spark component in that layer optimizes Spark code generation for both scenarios. This component generates optimized intermediate staging data when it detects any built-in non-deterministic transformation expression, otherwise avoids staging.

No Staging. The optimizing component generates code for a thread pool allowing downstream RDDs, representing independent pipeline branches, to execute in parallel. This speeds up overall job completion time compared to code that executes downstream Spark RDDs sequentially.

Optimized Staging. The optimizing component generates code for staging data to a file in Parquet format in a manner that allows a downstream pipeline to read data it is interested in. It assigns an ID to each of the downstream pipelines. After evaluation of a conditional expression on a row, it is tagged with the ID of the recipient pipeline using an intermediate temporary column. The generated code for a downstream pipeline that reads the staged parquet file has an additional filter condition to identify the pipeline. The end result is that each downstream pipeline does not have to read the entire staged data. The parallel execution mode adopted for the no-staging case is used for this case also.

Figure 11 shows parallel job execution with detailed data staging versus parallel job execution with smart data staging for SF100 Lineitem data from TPC-H benchmark on a 10 node `m5.2xLarge` cluster. We observe an increasing gap with increasing complexity.

4.7 Data Security and Isolation

When a mapping task runs in a compute cluster (Table 2), it reads and writes to an enterprise's data source. It might produce intermediate temporary files. One notable aspect here is that such a job no longer maintains any persistent storage such as cache files once it terminates. Such jobs may also produce execution logs, which may contain data from data sources when logging level is verbose. Our multi-tenant service supporting multiple enterprises on the same infrastructure needs to isolate enterprises from each other with respect to data sources, temporary files, and log files.

Enterprise VPC Option. We simply piggyback on VPC isolation provided by the cloud vendors. In this case, a compute cluster runs entirely within the customer's enterprise VPC. An enterprise provides us with a cloud account credential that has all the required capabilities such as starting, expanding a compute cluster within that customer's VPC. The compute nodes of that cluster have access to the data sources within that VPC. Enterprise data never leaves the customer network. Intermediate persistent files are part of the ephemeral compute cluster. The execution log files are archived to a cloud storage location of the customer's choosing, specified through a compute cluster object. A developer can access these log files through cloud storage's web interface or associated tool. However, for easy browsing of these log files by a user, our service does offer a web UI, which is accessible on our service cloud after user provides an explicit consent, required to move the log files to the user's computer through our service network.

Serverless Warmpool Option. With a serverless warmpool, compute nodes are part of our service's VPC. We instantiate compute nodes with a configuration that allows access to data sources within an enterprise's VPC but isolated from other enterprise's VPC. These compute nodes are typically pre-allocated in a vanilla state based on aggregate utilization to improve scale-up time and are assigned once to join a compute cluster. These compute nodes, along with attached persistent disks, are never reused. That is, they are destroyed as soon as such a compute node is removed from a compute cluster after a single use. This compute node destruction step ensures that intermediate temporary data files are not leaked.

To enable network traffic to flow between the VPC of our service and the enterprise's VPC, we configure multi-homed compute nodes using *Virtual Network Interface Cards* (VNIC). Each VNIC belongs to a specific cloud account. We configure our enterprise customer owned VNIC within a compute node to handle network traffic to and from the customer VPC and nothing else. Any network interaction with our services from such a compute node uses the other VNIC that belongs to our service's cloud account. It is clear that the cloud vendor's security mechanism needs to facilitate the ability of a cloud account to use a VNIC of a different cloud account in this manner. For example, AWS provides such a mechanism through their *Cross Account Role* and *Security Group* primitives to implement the plan we describe.

Our customers benefit from this option with a simplified compute cluster configuration without having to go into cloud vendor-specific compute instance configuration and network configuration that is associated with a Kubernetes compute cluster. Our customers can simply specify CPU cores and memory in a generic manner.

4.8 Productivity Evaluation

Informatica CDI-E provides a number of key benefits for the customers. Nucleaus Research[32], a third-party company, did a comprehensive analysis of the ROI of using Informatica technology [33] and identified the following key benefits:

- An average ROI of 321% over a three-year period, with an average payback period of 4 months.
- 62% higher monthly revenue from streamlined processing.
- 20–70% reduction in ETL process time.
- 40–55% reduction in new integration job setup time.
- 15–35% less time spent on data quality assurance tasks.

5 RELATED WORK

Data Integration for the Cloud. Data integration has a storied history in the database literature [42, 43, 48, 50, 57]. We highlight a few recent advances for the cloud. On the one hand, there are cloud-based parallel database-inspired systems such as Amazon Redshift [1, 44], Google BigQuery [16], and Azure Synapse Analytics [15]. On the other hand, there are document stores such as MongoDB [31] and Apache Cassandra [4], as well as unstructured "Big Data" frameworks such as Apache Spark [9] and Facebook Presto [34]. Informatica CDI-E[26] combines the best of both worlds and exposes them through a simple interface.

No-Code Data Engineering. Interacting with data using a GUI is becoming increasingly popular in the research community [54]. However, most existing cloud services such as Amazon Glue [10], Azure Data Factory (ADF) [19], and Databricks [20] that perform large-scale data engineering jobs using Apache Spark primarily use (manual) programming language interfaces. ADF has introduced *Mapping data flow* which has a GUI and is similar to our mapping concept. Informatica pioneered the use of GUI for enterprises for ETL jobs in late 1990s. Unlike CDI-E, GUI-based Business Process Integration cloud services such as Azure Logic Apps[14], Informatica Cloud Application Integration[25] process integration data at message level, instead of large data blocks from data sources like Parquet files.

Serverless Data Analytics. All major cloud service providers have serverless or Functions-as-a-Service (FaaS) platforms [11, 13, 17]. Several research projects focused on large-scale data processing leveraging these native serverless platforms [46, 51–53, 55]. At the same time, many companies provide managed, serverless analytics solutions to their customers on top of Infrastructure-as-a-Service (IaaS) platforms. In contrast, Informatica provides this capability through a no-code programming environment for the users.

Elastic Scaling. Amazon Glue, Databricks, and others support scaling of jobs through addition of new computing nodes. Glue service supports scaling of compute node that is similar to our serverless warmpool component's functionality. Elastic scaling is also widely explored in databases [40, 45, 47, 49, 58].

6 CONCLUSIONS

CDI-E abstractions are proven. Although CDI-E is relatively new, its core abstractions (§3.1) have been in continuous use since late 1990s. With more than 5,500 satisfied customers, we did not need to change the programming interface, which helped their seamless transition to CDI-E in large data lakes.

Design time data exploration is crucial. When users design a mapping, they like to explore data visually in the data source's data type and display format. This helps them to understand necessary data type conversion and extraction of relevant fields of a record, especially in a hierarchy. A standard interface to integrate any new data source connector at design time is essential.

Users need easier semi-structured document processing support. From early 2000s prominence of XML documents to the now-dominant JSON format, user requirements keep changing. The constant for us is to provide an easier way to process these in batch jobs using our programming interface.

Open-source software made implementation of CDI-E easier. High quality open-source software such as Apache Spark and Kubernetes are invaluable. For example, Apache Spark's support for nested data types in a row made it easier for us to support complex data formats. We were able to modify both Spark and Kubernetes to get the execution behavior we wanted without any surprises.

Looking forward. Our plans in the near future for CDI-E include (1) more cost savings across all fleets; (2) native serverless support across all cloud vendors; (3) improving user ability to mix GUI functionalities with custom code; (4) democratizing data sources and transformation logic through an open API; (5) automated cluster configuration (e.g., picking the best combination of VM instances); and (6) improved support for streaming and real-time data.

As the number of users using CDI-E grows, we will understand enterprise job patterns even more. With enterprises' consent, we can then leverage machine learning techniques to provide automatic recommendations to make users more productive.

ACKNOWLEDGMENTS

CDI-E is the work of many people at Informatica, and we would like to thank them all for their effort, dedication, incredibly hard work and all the great innovations. We would like to thank David Cao, Bala Kumaresan, Daphne Williams, Brett Roscoe, Sumeet Agrawal, Abhilash Mula, Oscar Najarro and all other reviewers of the paper for their great comments and feedback.

REFERENCES

[1] [n.d.]. Amazon Redshift. https://aws.amazon.com/redshift/.
[2] [n.d.]. Amazon Simple Storage Service (S3). https://aws.amazon.com/s3.
[3] [n.d.]. Apache Avro. https://avro.apache.org.
[4] [n.d.]. Apache Cassandra. https://cassandra.apache.org.
[5] [n.d.]. Apache Hadoop. https://www.hadoop.apache.org.
[6] [n.d.]. Apache Hive. https://www.hive.apache.org.
[7] [n.d.]. Apache ORC. https://orc.apache.org.
[8] [n.d.]. Apache Parquet. https://parquet.apache.org.
[9] [n.d.]. Apache Spark. https://spark.apache.org.
[10] [n.d.]. AWS Glue. https://aws.amazon.com/glue/.
[11] [n.d.]. AWS Lambda. https://docs.aws.amazon.com/lambda/.
[12] [n.d.]. Azure Blob Storage. https://docs.microsoft.com/en-us/azure/storage/blobs/storage-blobs-overview.
[13] [n.d.]. Azure Functions. https://azure.microsoft.com/en-us/services/functions/.
[14] [n.d.]. Azure Logic Apps. https://azure.microsoft.com/en-us/services/logic-apps/#overview.
[15] [n.d.]. Azure Synapse Analytics. https://azure.microsoft.com/en-us/services/synapse-analytics/.
[16] [n.d.]. BigQuery. https://cloud.google.com/bigquery.
[17] [n.d.]. Cloud Functions. https://cloud.google.com/functions.
[18] [n.d.]. Cloud Storage. https://cloud.google.com/storage.
[19] [n.d.]. Data Factory. https://azure.microsoft.com/en-us/services/data-factory/.
[20] [n.d.]. Databricks. https://databricks.com.
[21] [n.d.]. Extensible Markup Language (XML) - W3C. https://www.w3.org/xml.
[22] [n.d.]. Fast Healthcare Interoperability Resources. https://en.wikipedia.org/wiki/Fast_Healthcare_Interoperability_Resources.
[23] [n.d.]. Health Insurance Portability and Accountability Act. https://www.hhs.gov/hipaa.
[24] [n.d.]. HL7 International. https://hl7.org.
[25] [n.d.]. Informatica Cloud Application Integration. https://www.informatica.com/products/cloud-application-integration.html.
[26] [n.d.]. Informatica's Cost Optimization Engine. https://www.informatica.com/lp/informaticas-cost-optimization-engine_4257.html.
[27] [n.d.]. JavaScript Object Notation. https://www.json.org.
[28] [n.d.]. Kubernetes Scheduler. https://www.kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/.
[29] [n.d.]. LVM. https://www.tecmint.com/create-lvm-storage-in-linux/.
[30] [n.d.]. LVM1. https://www.tecmint.com/extend-and-reduce-lvms-in-linux/.
[31] [n.d.]. MongoDB. https://www.mongodb.com.
[32] [n.d.]. Nucleus Research. https://nucleusresearch.com/.
[33] [n.d.]. Nucleus Research Informatica. https://nucleusresearch.com/research/single/roi-guidebook-informatica/.
[34] [n.d.]. presto. https://prestodb.io.
[35] [n.d.]. Running Spark on Kubernetes. https://www.spark.apache.org/docs/latest/running-on-kubernetes.html.
[36] [n.d.]. SWIFT EDI Document Standard. https://www.edibasics.com/edi-resources/document-standards/swift/.
[37] [n.d.]. Taints and Tolerations. https://www.kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/.
[38] [n.d.]. TPC. https://www.tpc.org.
[39] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, et al. 2016. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data*. 215–226.
[40] Sudipto Das, Shoji Nishimura, Divyakant Agrawal, and Amr El Abbadi. 2011. Albatross: Lightweight elasticity in shared storage databases for the cloud using live data migration. *Proceedings of the VLDB Endowment* 4, 8 (2011), 494–505.
[41] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (jan 2008), 107–113. https://doi.org/10.1145/1327452.1327492
[42] AnHai Doan, Alon Y. Halevy, and Zachary G. Ives. 2012. *Principles of Data Integration.* Morgan Kaufmann. https://research.cs.wisc.edu/dibook/
[43] Xin Luna Dong and Divesh Srivastava. 2013. Big data integration. In *ICDE*. IEEE, 1245–1248.
[44] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. 2015. Amazon Redshift and the Case for Simpler Data Warehouses. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1917–1923. https://doi.org/10.1145/2723372.2742795
[45] Botong Huang, Matthias Boehm, Yuanyuan Tian, Berthold Reinwald, Shirish Tatikonda, and Frederick R Reiss. 2015. Resource elasticity for large-scale machine learning. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 137–152.
[46] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, and Ce Zhang. 2021. Towards demystifying serverless machine learning training. In *ACM SIGMOD*. 857–871.
[47] Jörn Kuhlenkamp, Markus Klems, and Oliver Röss. 2014. Benchmarking scalability and elasticity of distributed database systems. *Proceedings of the VLDB Endowment* 7, 12 (2014), 1219–1230.
[48] Maurizio Lenzerini. 2002. Data integration: A theoretical perspective. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 233–246.
[49] Edo Liberty, Zohar Karnin, Bing Xiang, Laurence Rouesnel, Baris Coskun, Ramesh Nallapati, Julio Delgado, Amir Sadoughi, Yury Astashonok, Piali Das, et al. 2020. Elastic machine learning algorithms in amazon sagemaker. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 731–737.
[50] Renée J Miller. 2018. Open data integration. *Proceedings of the VLDB Endowment* 11, 12 (2018), 2130–2139.
[51] Ingo Müller, Renato Marroquín, and Gustavo Alonso. 2020. Lambada: Interactive data analytics on cold data using serverless cloud infrastructure. In *ACM SIGMOD*. 115–130.
[52] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. 2020. Starling: A scalable query engine on cloud functions. In *ACM SIGMOD*. 131–141.
[53] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *USENIX NSDI*. 193–206.
[54] Protiva Rahman, Lilong Jiang, and Arnab Nandi. 2020. Evaluating interactive data systems. *The VLDB Journal* 29, 1 (2020), 119–146.
[55] Vaishaal Shankar, Karl Krauth, Qifan Pu, Eric Jonas, Shivaram Venkataraman, Ion Stoica, Benjamin Recht, and Jonathan Ragan-Kelley. 2018. numpywren: Serverless linear algebra. *arXiv preprint arXiv:1810.09679* (2018).
[56] Michael Stein. 1987. Large Sample Properties of Simulations Using Latin Hypercube Sampling. *Technometrics* 29, 2 (1987), 143–151. https://doi.org/10.1080/00401706.1987.10488205
[57] Michael Stonebraker, Ihab F Ilyas, et al. 2018. Data Integration: The Current Status and the Way Forward. *IEEE Data Engineering Bulletin* 41, 2 (2018), 3–9.
[58] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. 2014. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proceedings of the VLDB Endowment* 8, 3 (2014), 245–256.
[59] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. Hive: A Warehousing Solution over a Map-Reduce Framework. *Proc. VLDB Endow.* 2, 2 (aug 2009), 1626–1629. https://doi.org/10.14778/1687553.1687609
[60] Bowei Xi, Zhen Liu, Mukund Raghavachari, Cathy Xia, and Li Zhang. 2004. A Smart Hill-Climbing Algorithm for Application Server Configuration. *Thirteenth International World Wide Web Conference Proceedings, WWW2004* (04 2004). https://doi.org/10.1145/988672.988711
[61] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. 15–28.
[62] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing* (Boston, MA) *(HotCloud'10)*. USENIX Association, USA, 10.