Redis Data Integration

Hands-on workshop

Agenda

- RDI Architecture
  - The data pipeline
  - REST api
- Installing RDI
- Creating data pipelines
- Running RDI

Learning Objectives

By the end of this learning you will be able to...

- **Explain** what Redis Data Integration is and how it works
- **Install** Redis Data Integration in a virtual machine
- **Configure** Redis Data Integration for change data capture (CDC) ingestion
- **Run** and monitor Redis Data Integration

Workshop Format

This workshop will combine

discussion of the principles and practice of RDI

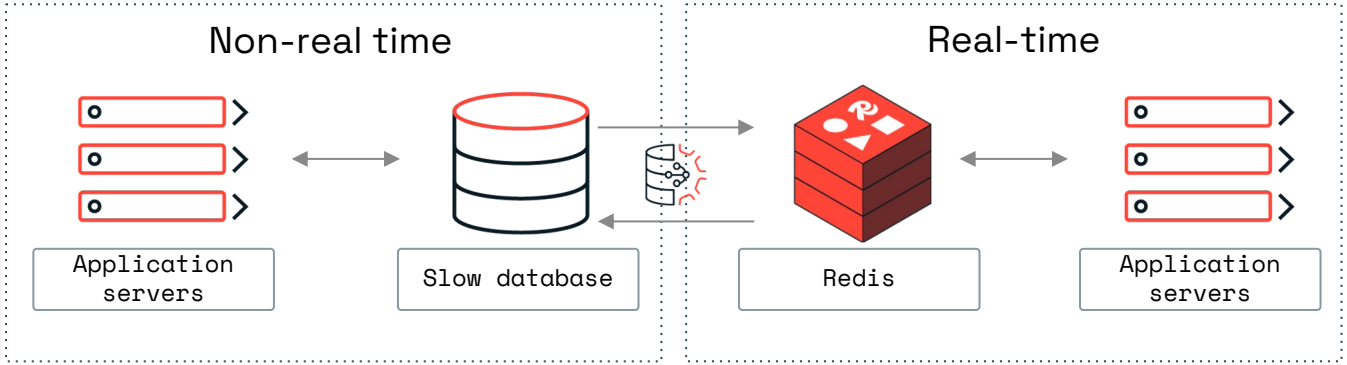
with hands-on exercises in a VM that we provide.

Redis Data Integration (RDI)

- Redis Data Integration (RDI) is a tool that runs alongside Redis Software
- It lets developers and architects synchronize existing databases with Redis Software to mirror application data
- It helps devs and architects save time on their data integration efforts

How RDI works

- It creates a data streaming pipeline that mirrors data from an existing database to Redis
- RDI's ingest flow capabilities include a Change Data Capture (CDC) platform to capture changes in the source database
- Within the RDI process, the data is filtered, transformed, and mapped to Redis data types such as Hash or JSON
- RDI then writes the data to the destination Redis database-and other applications can read from the Redis database without complicated workarounds



When to use RDI

You should use RDI when:

- You must use a slow database as the system of record for the app.
- The app must always *write* its data to the slow database.
- You already intend to use Redis for the app cache.
- The data changes frequently in small increments.
- Your app can tolerate *eventual* consistency of data in the Redis cache.

You should not use RDI when:

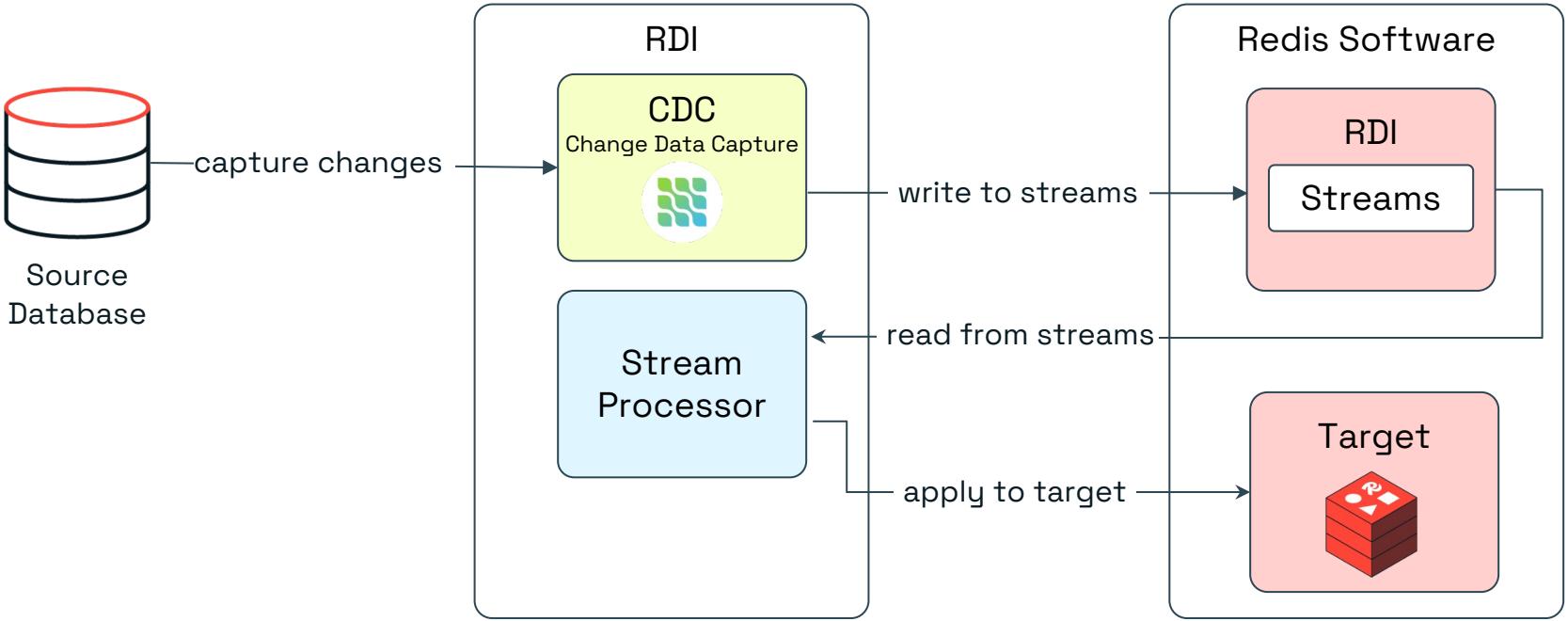
- You are migrating an existing data set into Redis only once.
- The data is updated infrequently and in big batches.
- Your app needs *immediate* cache consistency rather than *eventual* consistency.
- The app must *write* data to the Redis cache, which then updates the source database.
- Your data set will only ever be small.

thank you.

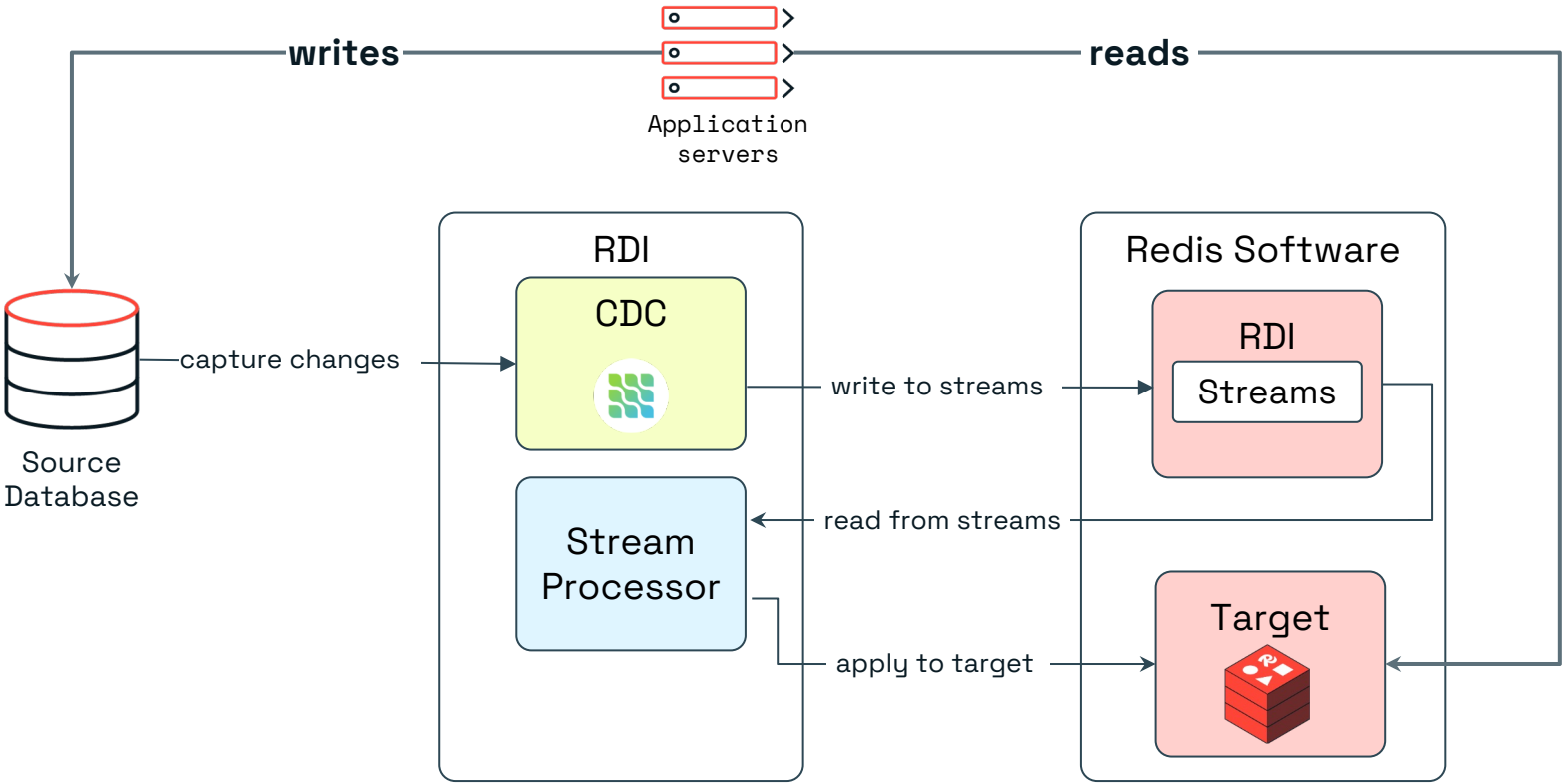
Architecture Overview

RDI deployment

Data plane

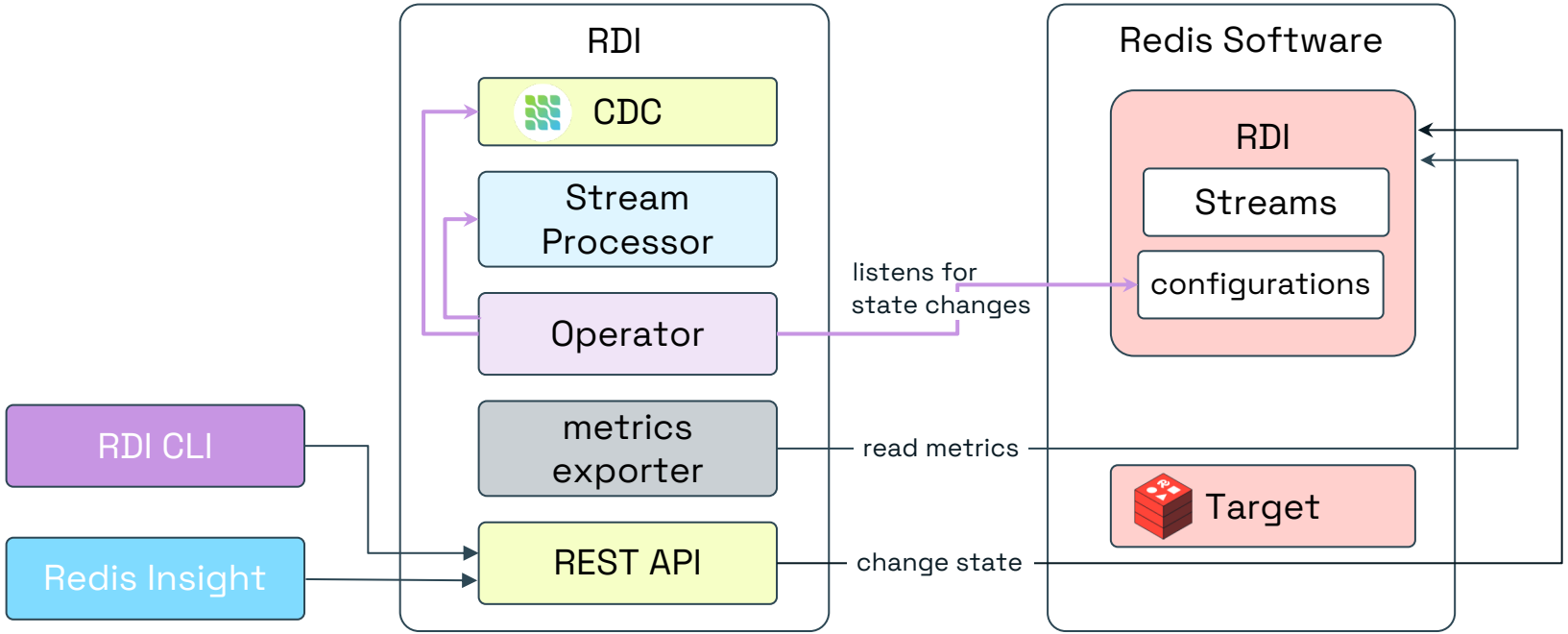


Redis Data Integration with your apps



RDI Deployment

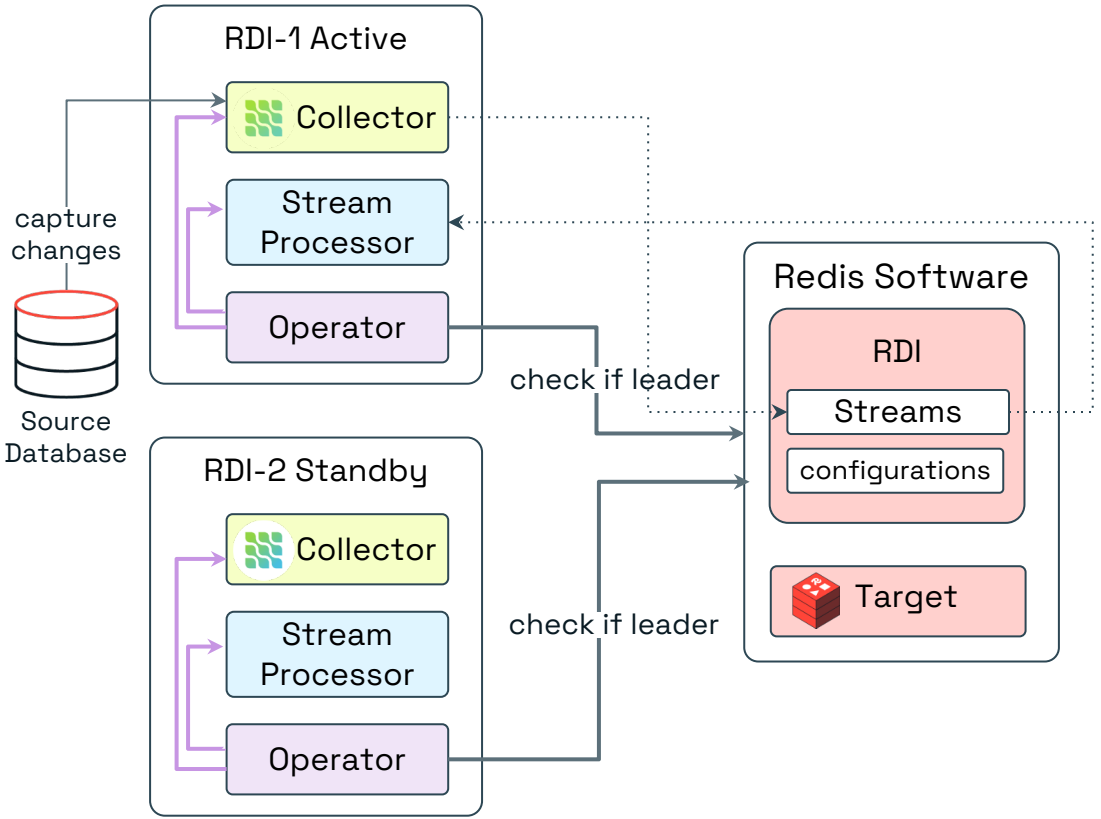
Control plane



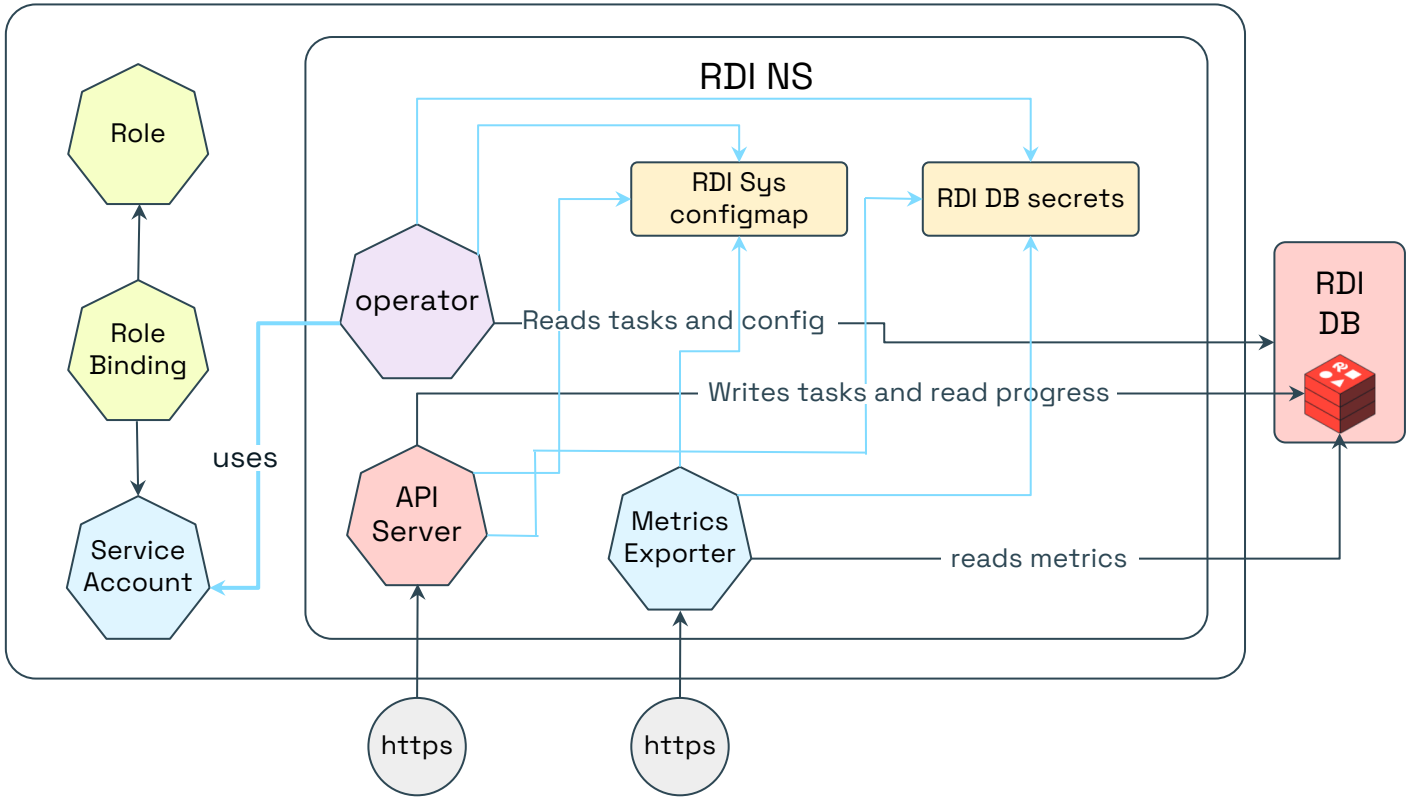
RDI on VMs

High availability (HA) deployment

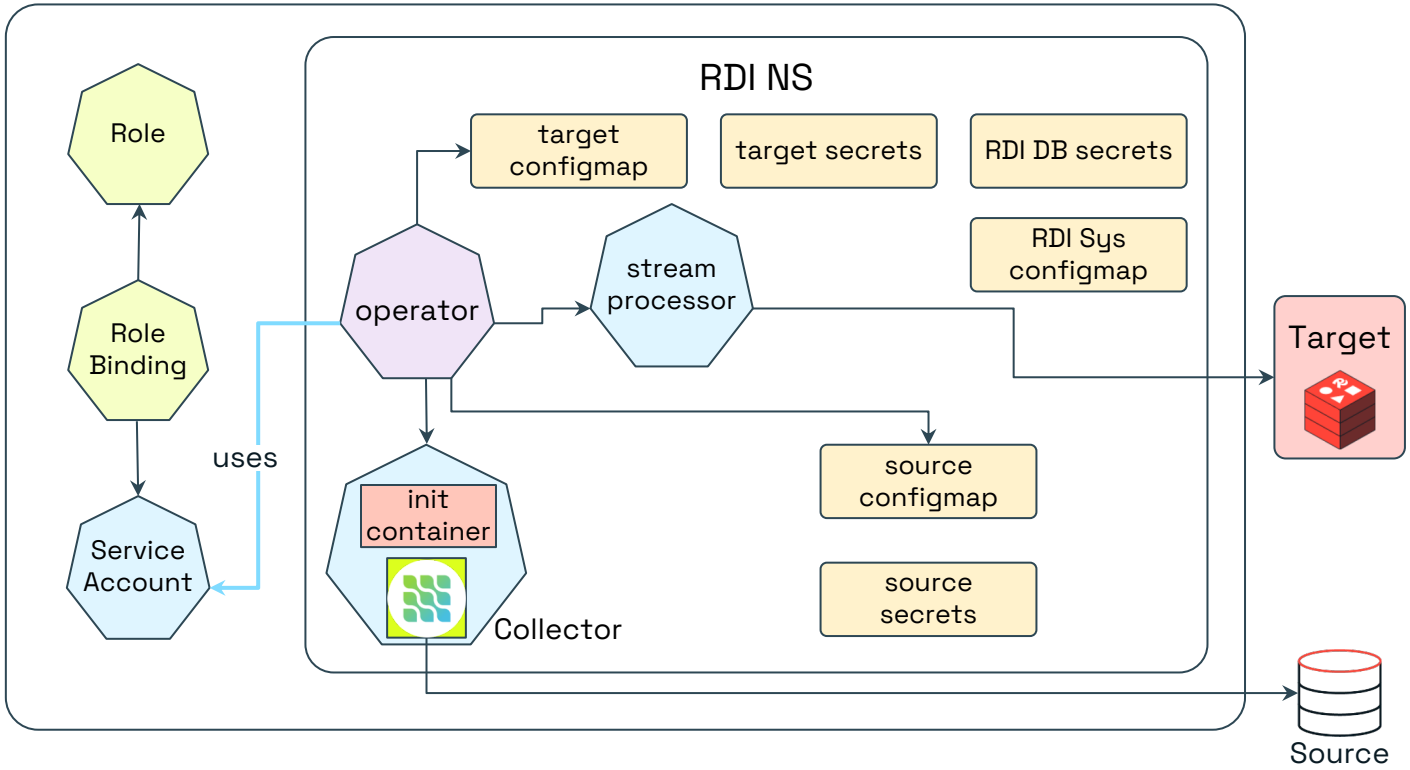
- The operators run on both VMs and use an algorithm to decide which is the active one (the "leader").
- The collector and stream processor are active on one VM and the other is a standby to provide HA.
- You can also install on just one VM if you don't need HA, such as in dev.



RDI control plane - K8s



RDI data plane - K8s



thank you.

Prepare Databases

Before installation

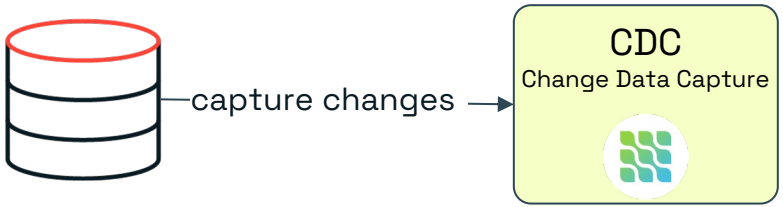
Complete the following steps before you can successfully install/configure RDI:

- Create the RDI database on your Redis Enterprise cluster.
- Create a user for the RDI database if you prefer not to use the default password (see Access control for more information).
- Prepare the source database (e.g. Postgres)

Preparing the source DB

The first step is to connect CDC to the source DB

- A CDC collector captures changes to the source database.
  - RDI currently uses an open source collector called Debezium for this step.
- The Debezium implementation depends on the source DB. Examples include:
  - Mining the source commit log
  - Querying binlog events
  - Monitoring change publications via plugin
  - Using a stored procedure in the source db



Instructions for supported connections

- Oracle and Oracle RAC for RDI
- MySQL/MariaDB for RDI
- PostgreSQL for RDI
- SQL Server for RDI

Preparing Source for the RDI Collector

This is where most problems are created!

- You need to verify the following.
  - Create the debezium DB user with the right permissions for the right db/schema
  - Enable the CDC mechanism (different between db types) for the schema/required tables
  - There is sufficient space for the snapshot and the commit log
- Debezium needs unique row id to generate CDC events.
  - Make sure tables have a PK or unique constraint
  - If one doesn't, you can configure config.yaml for a composite key for this table
    - Use `message.key.columns` in config.yaml to set the expression

thank you.

Installing RDI

Before you start...

Important information

- Download the appropriate RDI tar file from the download center.
- You can't install RDI on a host where a Redis Enterprise cluster is also installed, due to incompatible network rules.
- The supported OS versions for RDI are
  - RHEL 8 & 9
  - Ubuntu 18.04 & 20.04
- Helm charts are supported
- For VMs, you must run the RDI installer as a privileged user because it installs containerd and registers services. (You don't need any special privileges to run RDI processes for normal operation.)
- Additional requirements and considerations can be found in the documentation
  - Doc for installing on VMs
  - Doc for installing on Kubernetes

Installation steps

Follow the steps below for each of your VMs:

- Download the RDI installer from the Redis download center (under the “Modules, Tools & Integration” dropdown)
- Go to the installation folder:
  ```
  cd rdi_install/$RDI_VERSION
  ```
- Run the installer as a privileged user:
  ```
  sudo ./install.sh
  ```
- The installer gives you instructions to help you create secrets and create your pipeline.
- You can use the `-f` option to supply answers to the installer’s questions automatically using properties from a TOML file:
  ```
  ./install.sh -f silent.toml
  ```

Using TLS or mTLS

Sample file snippet for `silent.toml`

```
[rdi.database]
host = "localhost"
port = 12001
username = "username"
password = "password"
ssl = true
```

```
# if the RDI database uses TLS/mTLS.
[rdi.database.certificates]
ca = "/home/ubuntu/rdi/certs/ca.crt"
cert = "/home/ubuntu/rdi/certs/client.crt"
key = "/home/ubuntu/rdi/certs/client.key"
passphrase = "foobar"
```

continued in next column

SSL (true/false)? - If ssl = false then RDI will ignore the settings in the rdi.database.certificates section.

K8s Installation

Use the RDI Helm chart to install on Kubernetes

The installation creates the following K8s objects:

- A K8s namespace named rdi.
- Deployments for the RDI operator, metrics exporter, and API server.
- A service account along with a role and role binding for the RDI operator.
- A Configmap for the different components with RDI Redis database details.
- Secrets with the RDI Redis database credentials and TLS certificates.

Installation steps

Install the RDI Helm chart

- Decompress the tar file:
  `tar -xvf rdi-<rdi-tag>.tar.gz`
- Open the `values.yaml` file and set the appropriate values for your installation.
  - See the RDI Kubernetes doc for details.
- Start the installation:
  `helm install <The logical chart name> ./rdi --create-namespace -n rdi`
- To check the installation, run the following command:
  `helm list -n monitoring -n rdi`
- You can verify that the RDI API works by adding the server in Redis Insight.

Installation Troubleshooting

Common pitfalls

- Install didn't run as `sudo (VM installation)`
- Problem accessing the RS cluster (credentials, network, DNS)
- Not enough disk space on VM
  - RDI needs several GBs on top of the OS.
  - Ubuntu is recommended to have a minimal disk space of 25GB.

Installation Troubleshooting

How to troubleshoot

- RDI should provide a clear indication what went wrong
- Try network commands, such as these, to test connectivity and see what's happening in the network.
  - telnet
  - nslookup
  - traceroute
- Check disk availability
- Troubleshooting RDI components:
  - Look at `/opt/rdi/logs` to see the components' logs in case one or more are not ready
  - Run `kubectl -n rdi get all` to ensure all deployments have their pods ready

Uninstall RDI

If you want to remove your RDI installation, go to the installation folder and run the uninstall script as a privileged user:

```
sudo ./uninstall.sh
```

The script will verify you are sure before you proceed:

```
This will uninstall RDI and its dependencies, are you sure? [y, N]
```

If you type anything other than "y" here, the script will abort without making any changes to RDI or your source database.

thank you.

Data pipeline

Ingest Overview

Capture & Stream Changes

- CDC for most popular databases out of the box
- Can integrate with any CDC product / Streaming data source
- Modes of work:
  - Initial baseline snapshot
  - Streaming changes
- At least once guarantee

Transform & Denormalize

- Automatically converts from source types to Redis strings
- Declarative transformations can be applied per table
- Handles Many to one relationships as JSON
- Handles One to one relationships as Hash or JSON

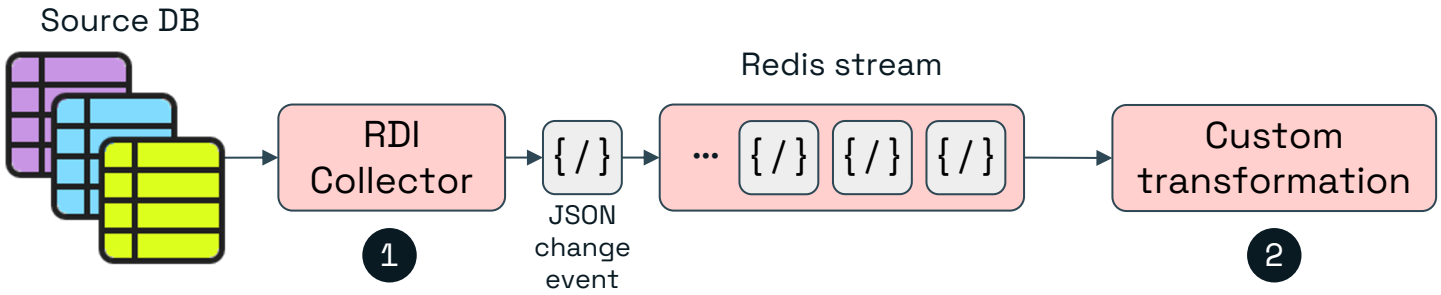
Handle Delivery

- Supports one source entry to many Redis keys
- Supports Hash, Json, Set, Sorted Set, Stream
- At least once guaranteed
- Malformed data saved to DLQ
- Auto recovers disconnects
- Data provenance

Capture and stream data

Capture & stream changes

Steps



A. Initial snapshot *(everything is a change)*
B. Streaming changes

At least once delivery guarantee

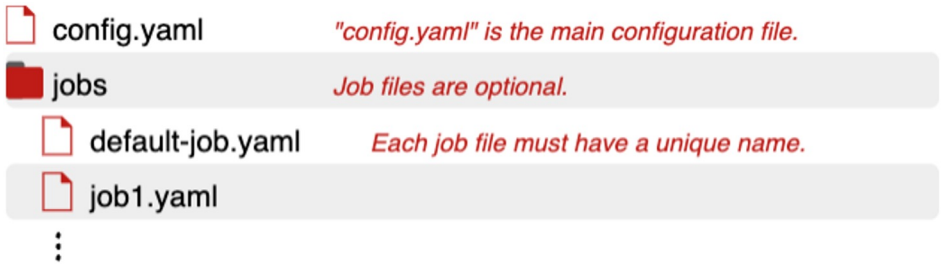
The data transformation involves two separate stages.

1. First, the data ingested during CDC is automatically transformed to a JSON format.
2. Then, this JSON data gets passed on to your custom transformation for further processing.

Pipeline configuration

Specify the full name of the source table in the job file (or '*' for `default-job.yaml`)

RDI uses a set of YAML files to configure each pipeline. The following diagram shows the folder structure:



- The main configuration for the pipeline is in the `config.yaml` file. This specifies:
  - Connection details for the source database
  - Queries that RDI will use to extract the required data.
- You should place job configurations in the `jobs` folder to specify your own data transformations.
- `default-job.yaml` will be used for any table that doesn't have its own `.yaml`

config.yaml

Part 1: sources

The sources section

- A unique name for each source
- type: currently cdc is the only option
- Logging level is up to you
- connection: supply the necessary info to connect to the source
- Include db credentials and TLS/mTLS secrets if applicable

```
sources:
  my_name:
    type: cdc
    logging:
      level: info
    connection:
      type: mysql
      host: ${RDI_REDIS_HOST}
      port: 13000
      database: redislabscdc
      user: ${SOURCE_DB_USERNAME}
      password: ${SOURCE_DB_PASSWORD}
```

config.yaml

Part 2: sources continued

The sources section for TLS/mTLS

- The names of the following properties should match the ones you used when setting the TLS/mTLS secrets.
- Set only `cacert` if you are using TLS.
- Set all of them if you are using mTLS.

```
connection:
 (continuation from previous slide)
  key: ${SOURCE_DB_KEY}
  cert: ${SOURCE_DB_CERT}
  cacert: ${SOURCE_DB_CACERT}
  key_password: ${SOURCE_DB_KEY_PASSWORD}
```

config.yaml

Part 3: sources continued

The sources section - tables

- The name of the this table query
- snapshot_sql: A query that selects the table(s) to include in the dataset
  - The default is **all** tables in the db
- columns: which ones you want to include in the target
  - The default is **all** columns
- keys: A list of primary keys, one for each table
  - or a unique composite if no column has the PRIMARY KEY or a UNIQUE constraint

```
tables:
      emp:
        snapshot_sql: "SELECT * from
redislabscdc.emp WHERE empno < 1000"
     # the preceding is all one line
        columns:
          - empno
          - fname
          - lname
        keys:
          - empno
```

config.yaml

Part 4: target

The targets section

- A unique name for each target
- connection: info for the target
  - type: presumably redis
  - host: and port: as usual
  - connection credentials and TLS/mTLS secrets if applicable

```
targets:
  my-redis:
    connection:
      type: redis
      host: localhost
      port: 12000
      user: ${TARGET_DB_USERNAME}
      password: ${TARGET_DB_PASSWORD}
```

config.yaml

Part 5: target continued

The target section for TLS/mTLS

- The names of the following properties should match the ones you used when setting the TLS/mTLS secrets.
- Set only ‘cacert’ if you are using TLS.
- Set all of them if you are using mTLS.

```
targets:
 (continuation from previous slide)
  key: ${TARGET_DB_KEY}
  cert: ${TARGET_DB_CERT}
  cacert: ${TARGET_DB_CACERT}
  key_password: ${TARGET_DB_KEY_PASSWORD}
```

Transform & denormalize

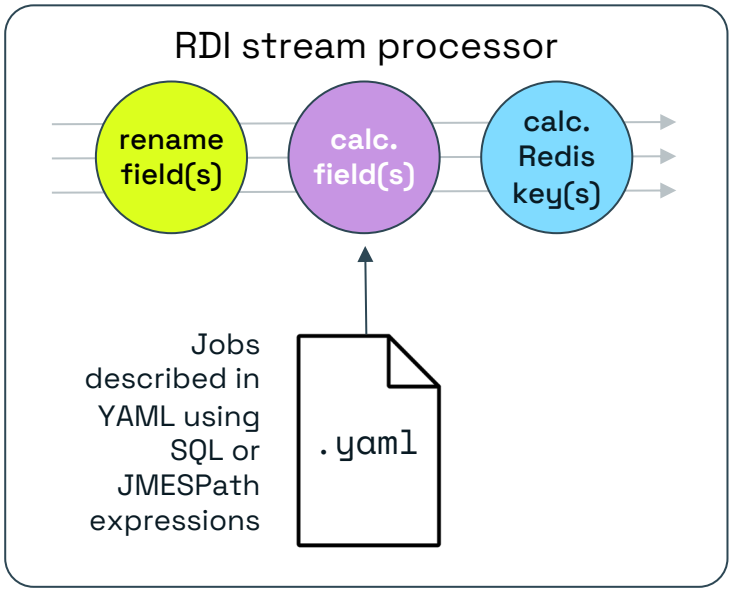
Single table transformation

Pipeline captures change data records from the source db, and transforms them into Redis data structures. It writes each of these new structures to a Redis target database under its own key.

You can provide your own custom transformation jobs for each source table, using your own data mapping and key pattern.

As part of the transformation, you can specify whether you want to store the data in Redis as JSON objects, hashes, sets, streams, sorted sets, or strings.

Transformation steps



Transformation

3 main sections of a `job.yaml`

Sample

- `source:` *(required section)*
- `table:` Database table name.
  - One of the table names you supplied in config.yaml.
  - Use "*" for the default job only.
- `row_format:` Format of the data to be transformed. Options are:
  - `data_only` *(default)*
  - `full` *(See the doc for details of the extra data you can access when you use the full option.)*

Explanation

```
source:
  table: "*"
  row_format: full
transform:
    Discussed in next slides
output:
    Discussed after transform
```

Transformation

How to transform data from source to output

Sample

```
transform: (optional section)
  - uses: (see options in next column)
    with: (options depend on use)
```

Explanation

- `transform:` *(optional section)*
- `- uses:` *(note the dash)*
  - `add_field`
  - `remove_field`
  - `rename_field`
  - `filter`
  - `map`
- `with:` *same indentation as* `uses`*, not as the dash*

Adding a field to the output

You can add one or many fields

Sample

```
transform:
  - uses: add_field
    with:
      fields:
        - field: name.first_name
          language: jmespath
          expression: concat( ... )
        - field: country
          language: sql
          expression: c_code || ...
```

Explanation

- `- uses: add_field`
- `with:` *(required)*
- `fields:` *skip if only adding 1 field*
- `- field:` name of added field in the correct format for the target data type *(skip dash if only 1 field)*
- `language:` of the expression. Options are `jmespath` and `sql`.
- `expression:` that specifies the value of the new field

Removing a field from the output

You can remove one or many fields

Sample

```
transform:
  - uses: remove_field
    with:
      fields:
        - field: credit_card
        - field: name.middlename
```

Explanation

- `- uses: remove_field`
- `with:` *(required)*
- `fields:` *skip if only removing 1 field*
- `- field:` name of field to remove in the correct format for the target data type *(skip dash if only 1 field)*

Rename a field from the output

All other fields remain unchanged

Sample

```
transform:
  - uses: rename_field
    with:
      fields:
        - from_field: name.lname
          to_field: name.last_name
        - from_field: name.fname
          to_field: name.first_name
```

Explanation

- `- uses: rename_field`
- `with:` *(required)*
- `fields:` *skip if only changing 1 field*
- `- from_field:` default name of field *(skip dash if only 1 field)*
- `to_field:` new name of field *(aligned with from_ field; no dash ever)*

Map a row to a Redis data type

Map a record into a new output based on expressions

```
transform:
  - uses: map
    with:
      expression: |
        {
          "FirstName": first_name,
          "LastName": last_name,
          "Location":
          {
            "Street": address,
            "City": city,
            … etc. …
          }
        }
      language: jmespath
```

Explanation

- `- uses: map`
- `with:` *(required)*
- `language:` of the expression. Options are `jmespath` and `sql`.
- `expression:` that specifies the name of the target field and the column for its value

Filter records on chosen criteria

Only cache relevant information

Sample

```
transform:
  - uses: filter
    with:
      language: sql
      from_field: name.lname
      expression: age>20
```

Explanation

- `- uses: filter`
- `with:` *(required)*
- `language:` of the expression. Options are `jmespath` and `sql`.
- `expression:` that specifies the criteria for which rows to include

Output location, type, and keyname

- `output:` *(required)*
- `- uses: redis.write` *(required)*
- `  with:` *(required)*
- `connection:` *(optional)* name of target defined in `config.yaml`
- `data_type:` target structure *(optional)* choices are:
  - `hash`, `json`, `set`, `sorted_set`, `stream`, and `string`
- `key:` *(optional)* override default key naming for target data
- `expression:` how to form key
- `language: [sql | jmespath]`

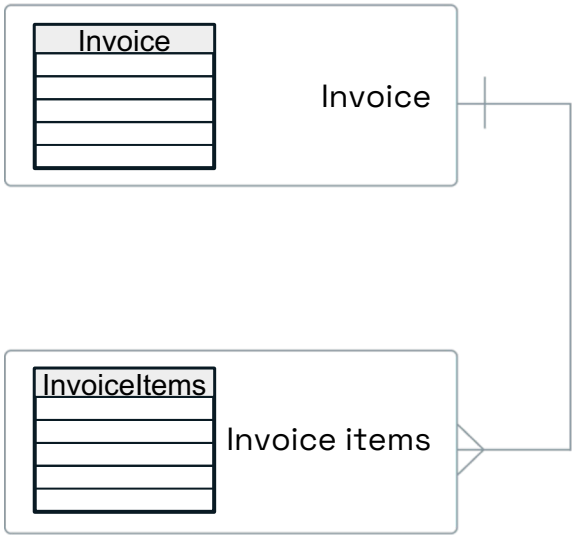
Sample

```
output:
  - uses: redis.write
    with:
      connection: target
      key:
        expression:
concat(['emp:fname:',fname,':lname:',lname])
 # preceding “expression:” is all one line
        language: jmespath
```

Denormalization

Multi-table transformation with denormalization

Source relational database



Many-to-one relationship is translated into a single parent JSON with a map of child objects

*..1

Redis

```
Invoice:InvoiceID:233 →
{
 ...
  -   InvoiceLineItems{
              1244: {}
              1156: {}
              ...
      }
}
```

Sample job for nesting

When normal isn't an option

When you use nesting:

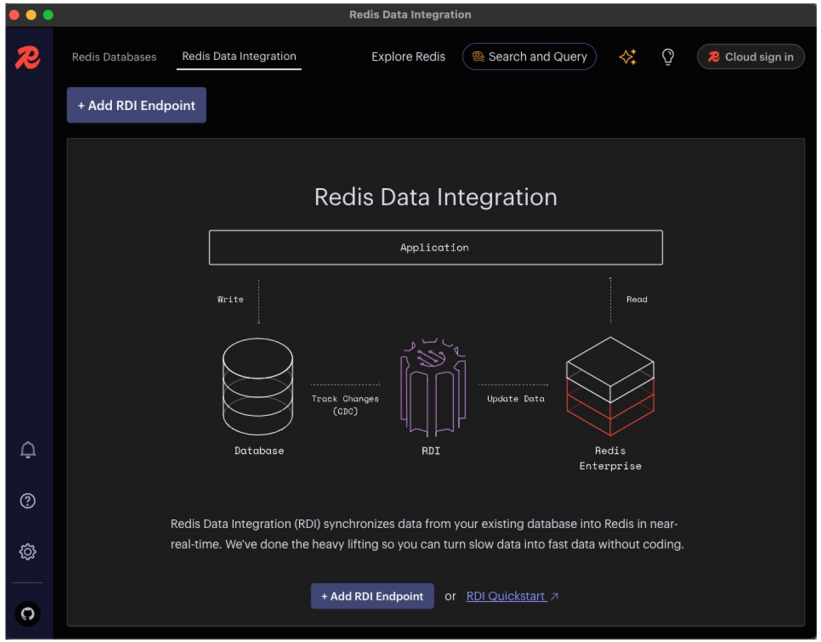
- You must also set the `data_type` attribute to `json` and the `on_update` attribute to `merge` in the `output` block.
- Key expressions are **not** supported for the nest `output` blocks.
- You can only use one level of nesting.

```
source:
  table: InvoiceLine
output:
  - uses: redis.write
    with:
      nest: # cannot co-exist with other parameters
        parent:
          table: Invoice
        nesting_key: InvoiceLineId # can't be composite
        parent_key: InvoiceId # cannot be composite
        path: $.InvoiceLineItems # must start from root
        structure: map # only map supported for now
      on_update: merge # only merge supported for now
      data_type: json # only json supported for now
```

Deploy a pipeline

There are two options for deploying a pipeline

Redis Insight



redis-di CLI

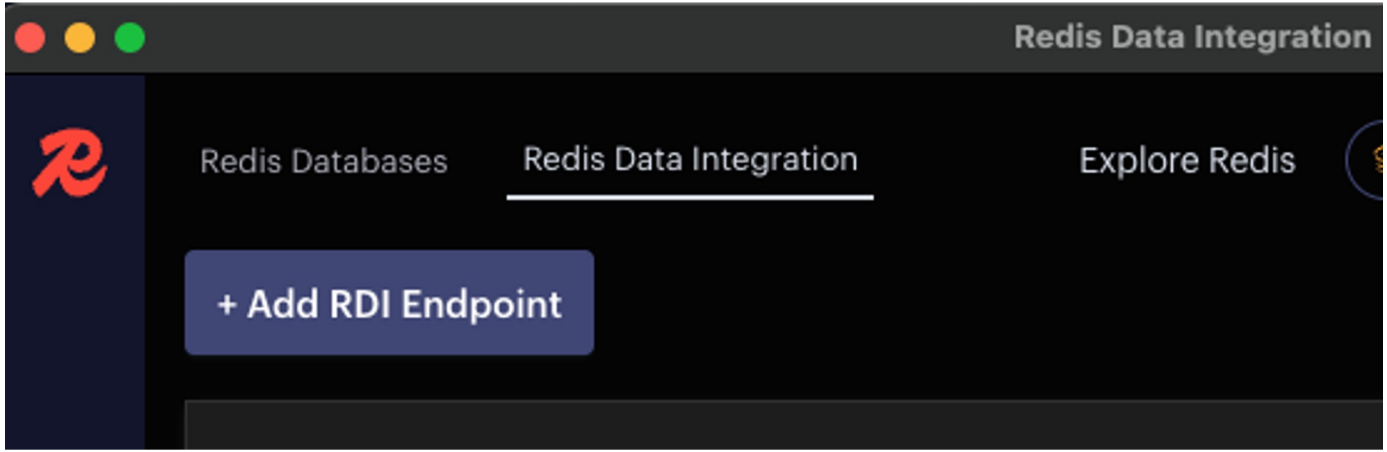
```
redis-di set-secret SOURCE_DB_USERNAME
myUserName
# set remaining secrets as needed
redis-di deploy --dir <path to pipeline folder>
```

Deploying with Redis Insight

First connect to the RDI server

While on Redis Data Integration page

Click **+Add RDI Endpoint**



Connect to RDI

Fill in the relevant values

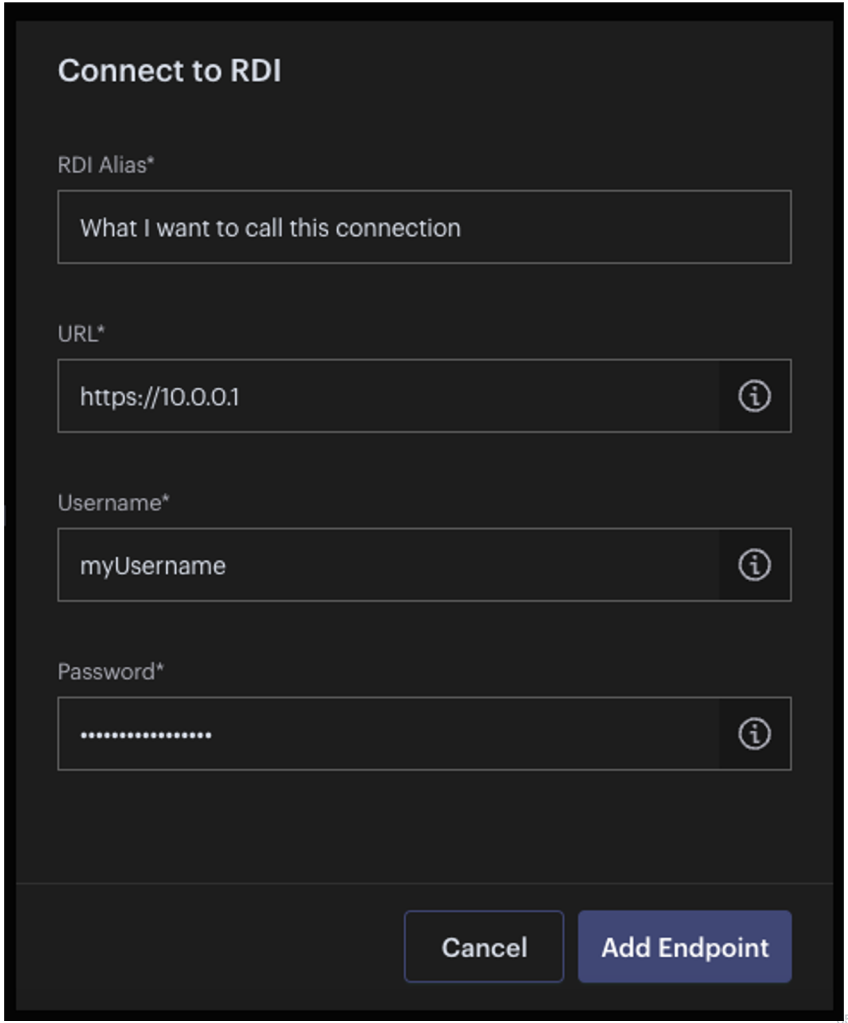
RDI Alias: whatever you want

URL: of the RDI server, not the target db and not the source db

Username: to the RDI database that you made

Password: for the RDI database that you made

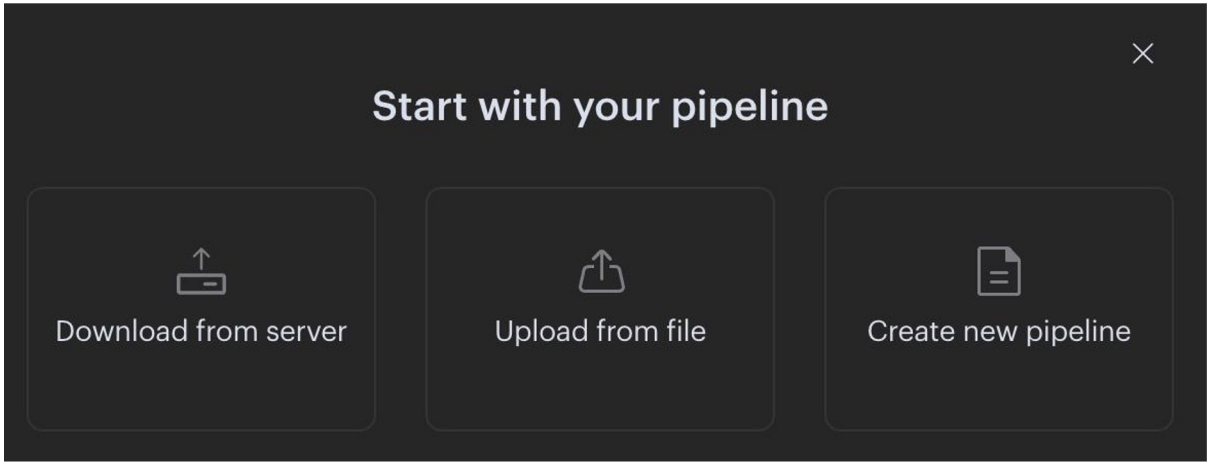
Click the **Add Endpoint** button



Add the pipeline

You can

- Download from server
- Upload from file
- Start from scratch



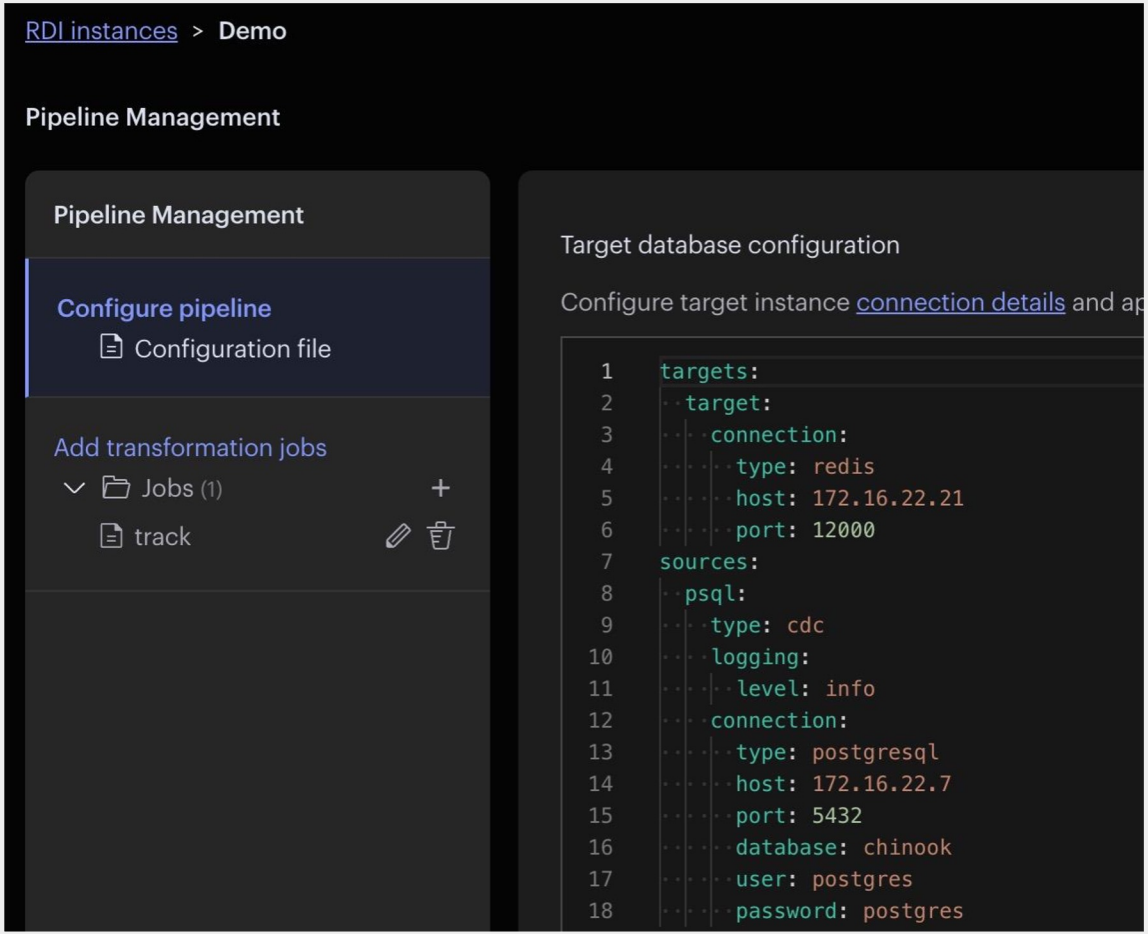
This is where you add `config.yaml` when using Redis Insight.

Pipeline management

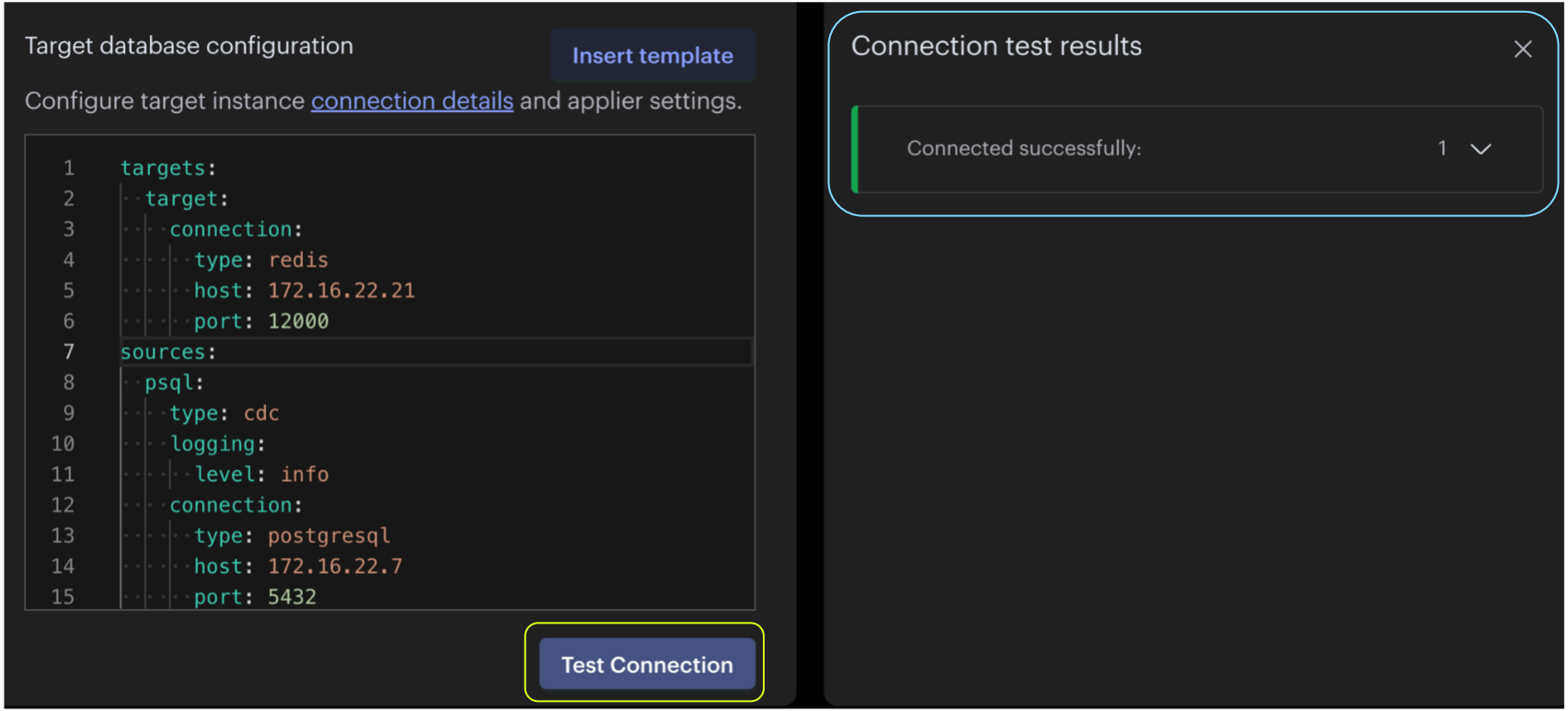
Sample `config.yaml`

Enter the target and source database information

This shows Redis Insight, but any editor can be used as long as the directory structure is correct.



Redis Insight lets you test the connection immediately



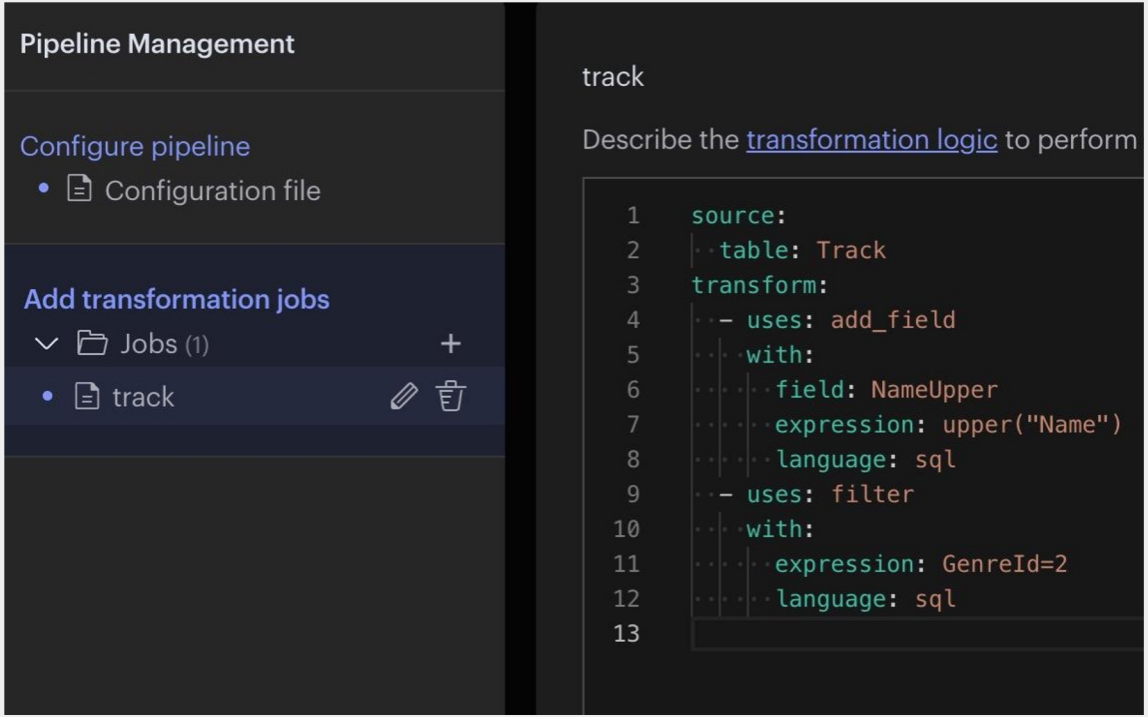
Transformation jobs

Add as many transformation files as needed for your implementation.

None are required.

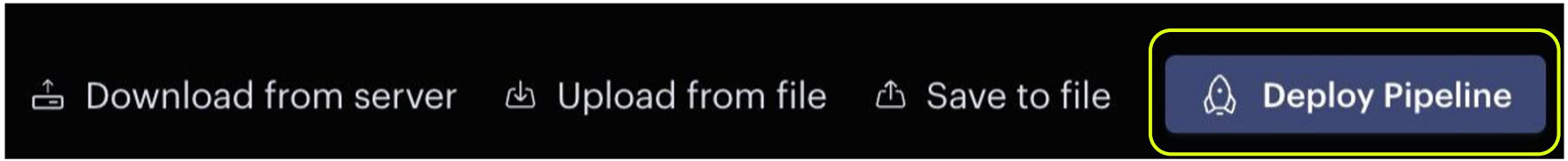
By default, RDI transforms the source data into hashes or JSON objects for the target with a standard data mapping and a standard format for the key.

This shows Redis Insight, but any editor can be used as long as the directory structure is correct.



Deploy the pipeline

Using Redis Insight:



Using CLI:

```
redis-di deploy --dir <path to pipeline folder>
```

thank you.

Observability

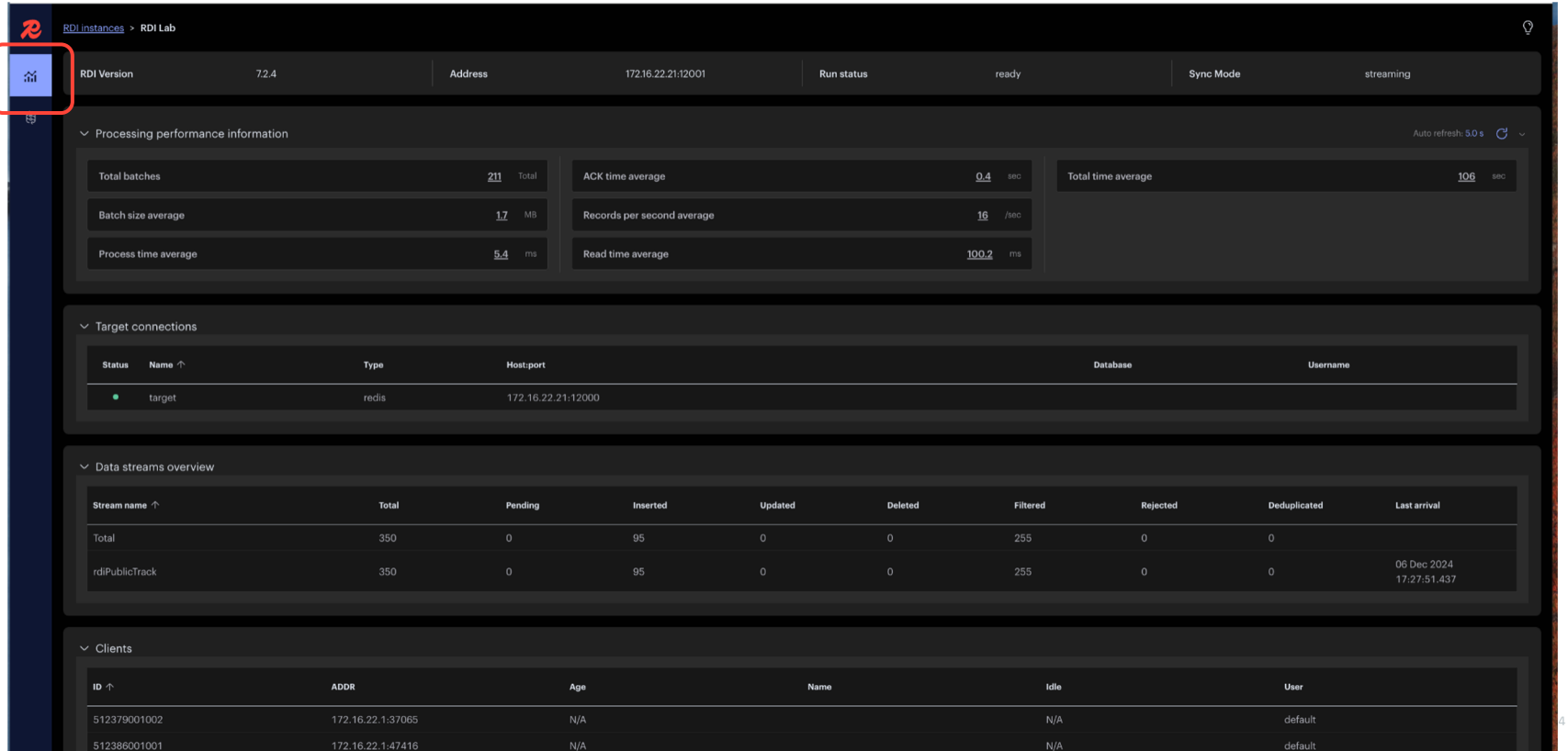
Monitoring RDI

There are three general categories

- Metrics
  - RDI exposes two endpoints, one for CDC collector metrics and another for stream processor metrics.
- Logs
  - By default, RDI stores logs in the host VM file system at `/opt/rdi/logs`. The logs are recorded at the minimum INFO level and get rotated when they reach a size of 100MB.
- Support package “dump”
  - For a comprehensive set of forensics data you can run `redis-di dump-support-package` from the CLI.

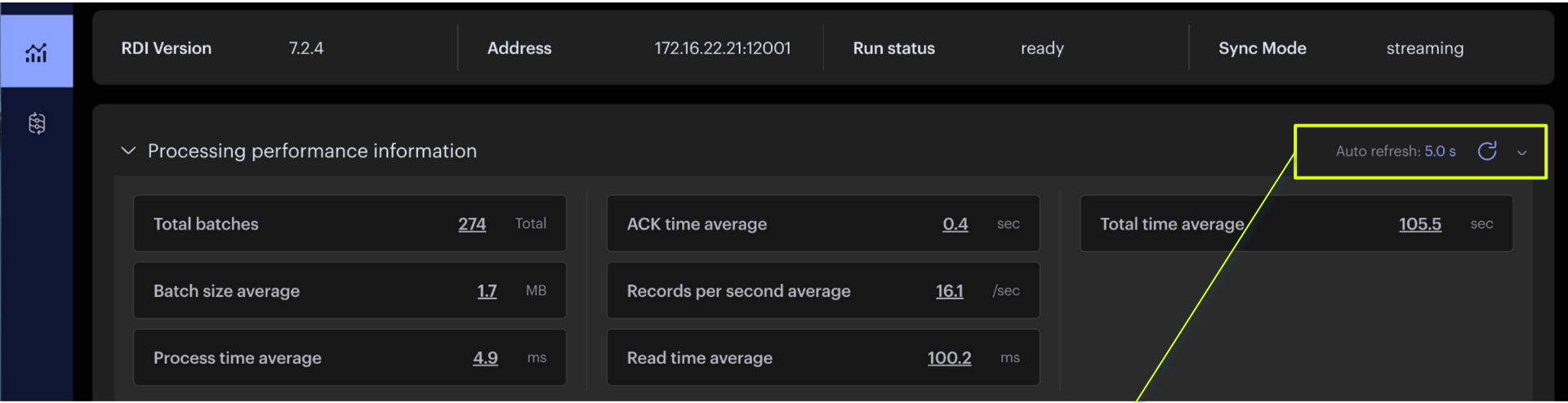
Pipeline status

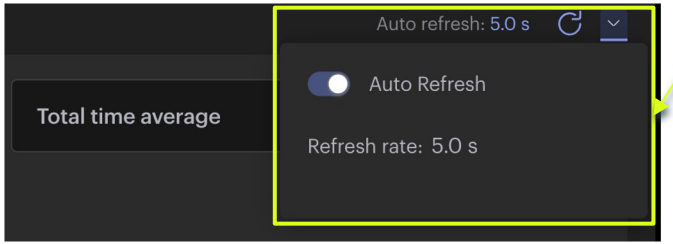
Redis Insight has a built in RDI dashboard



Processing performance

Includes these statistics

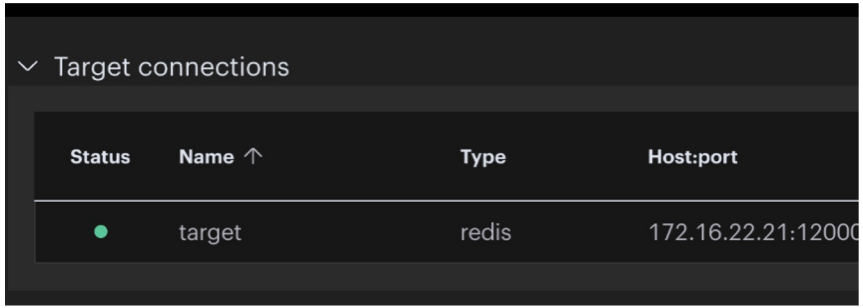




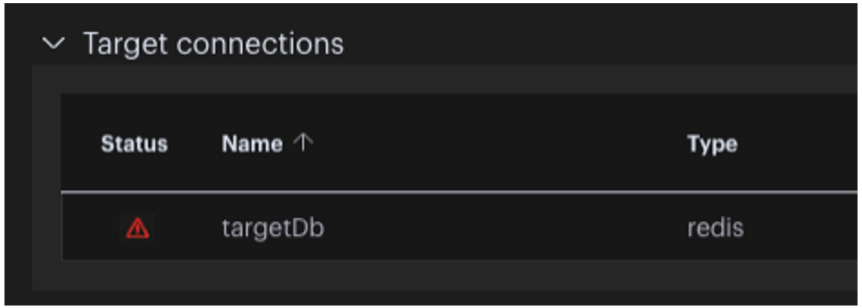
Can be set to static or auto refresh

Target connections

Connected



Disconnected



Data streams overview

Data streams overview

Stream name ↑	Total	Pending	Inserted	Updated	Deleted	Filtered	Rejected	Deduplicated	Last arrival
Total	2950	2	732	0	0	2218	0	0	
rdiPublicTrack	2950	2	732	0	0	2218	0	0	06 Dec 2024 17:41:02.838

Collector metrics

The endpoint for collector metrics is `https://<RDI_HOST>/metrics/collector-source`

These metrics are divided into three groups:

- Pipeline state: metrics about the pipeline mode and connectivity
- Data flow counters: counters for data breakdown per source table
- Processing performance: processing speed of RDI micro batches

Stream processor metrics

The endpoint for stream processor metrics is `https://<RDI_HOST>/metrics/rdi`

RDI reports metrics during the two main phases of the ingest pipeline, the snapshot phase and the change data capture (CDC) phase.

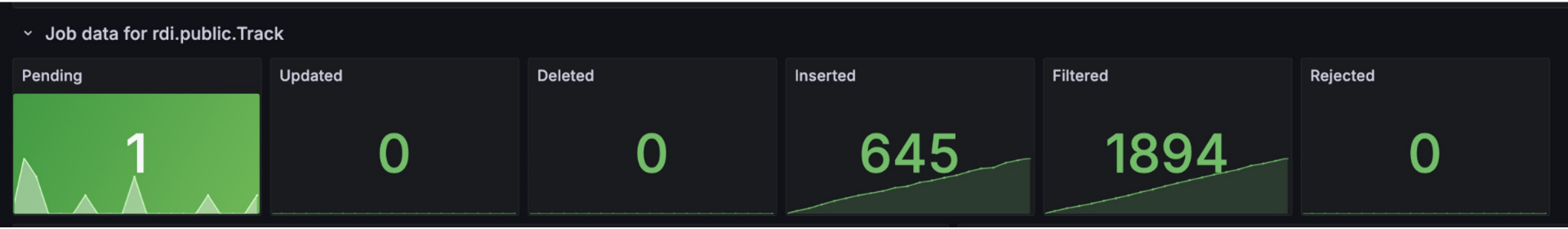
The full list is in the docs.

Grafana dashboard for RDI

Grafana dashboard for RDI

Job data

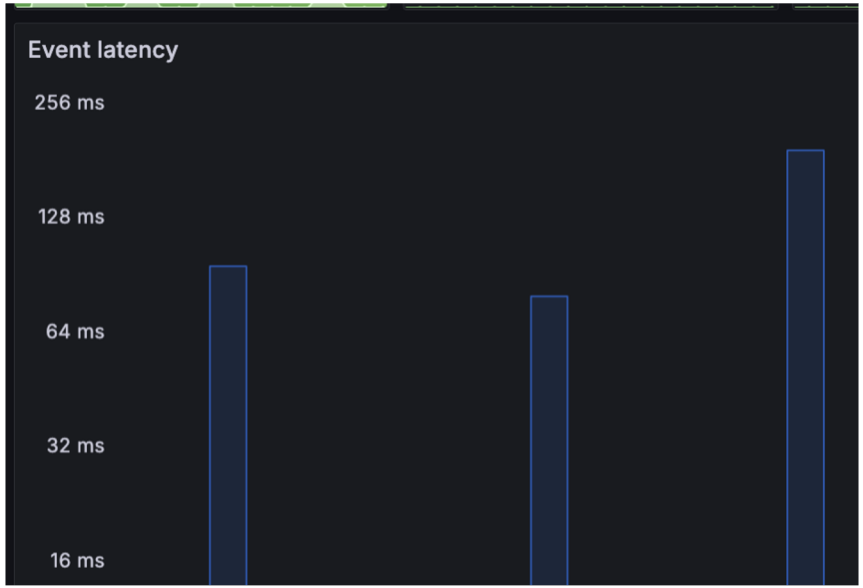
- Pending - how many records are waiting to be written to Redis?
- Updated - Number of updated key values (i.e. completed update operations)
- Deleted - Number of deleted keys
- Inserted - Number of inserted keys
- Filtered - Number of key changes skipped because they were filtered out
- Rejected - Number of errors - ideally, always zero in production



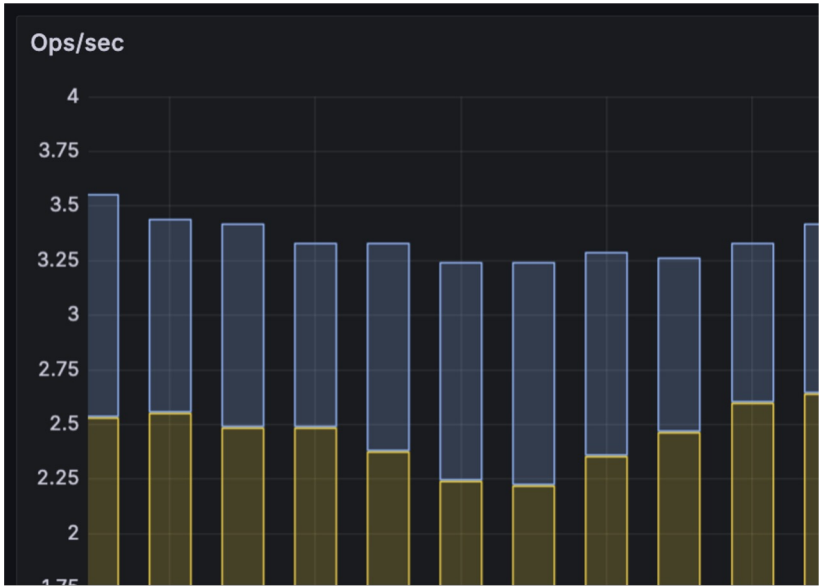
Grafana dashboard for RDI

You can also see latency and ops/sec

Event latency



Operations per second



thank you.

General troubleshooting

General troubleshooting

- `redis-di status` command - discover source, target, data flow (next slide)
- Logs (following slide)
- Debezium metrics - see the state of collection
- Support package - has everything
- If you want to run kubectl to get immediate components' status: use `-n rdi` with all Kubectl commands
  - Start with
    `kubectl -n rdi get all -a`
  - Then you can continue with
    `kubectl -n rdi describe [pod|configmap|secret]<object name>`

`redis-di status` *`args`* `[OPTIONS]`

These are the arguments and options

- args
  - `rdi_host` (REQUIRED)
  - `rdi_port` (REQUIRED)
  - `rdi_user` (if applicable)
  - `rdi_password` (if applicable)
- Options
  - `log_level [DEBUG, INFO, WARN, ERROR, CRITICAL]`
    - `redis-di status --log-level DEBUG`
  - `live` (for live data flow monitoring)
    - `redis-di status --live TRUE`
  - `redis-di status --help`

Logs

RDI uses fluentd and logrotator to ship and rotate logs at `/opt/rdi/logs`

Component	Log name
Operator	`/opt/rdi/logs/operator.log`
Stream Processor	`/opt/rdi/logs/processor.log`
Other logs	`/opt/rdi/logs/monitor.log`

Logs

"Other" logs, continued

Filename	Phase
rdi_collector-collector-initializer.log	Initializing the collector
rdi_collector-debezium-ssl-init.log	Establishing the SSL connections to the source and RDI database (if you are using SSL)
rdi_collector-collector-source.log	Collector change data capture (CDC) operations
rdi_rdi-rdi-operator.log	Main RDI control plane component
rdi_processor-processor.log	RDI stream processing

Metrics

Getting access to metrics

- Debezium collects metrics and we expose them through a dedicated Prometheus exporter
- The Stream Processor writes metrics to RDI database and the Metrics exporter provides these metrics
- The CLI `status` command can provide live or snapshot metrics
- There is an API to get metrics as well

Which metrics are available:

- Debezium has different metrics per phase of the pipeline:
  - Snapshot metrics
  - CDC/Streaming metrics
- The Stream Processor reports data counters per source table and performance metrics per micro batch

thank you.