REDISCOVER

RedisCDC

Platform Overview

Last Updated 08/10/2020

redislabs
HOME OF REDIS

1 Introduction to RedisCDC

1.1 Technical Overview

Redis Enterprise Changed Data Capture ("RedisCDC") is a distributed platform that enables near real-time replication and transformation (data pipelines) of row-level changed-data events (Create, Update, Delete operations) from heterogeneous platforms to Redis Enterprise databases/modules. RedisCDC Jobs migrate source-database operations in the same time-order they were committed using event-driven workflows. It has a modular, extendable, and configurable architecture which provides the flexibility to deploy in a variety of topologies and cover multiple use-cases.

RedisCDC can also perform the function of event-sourcing, acting as both the message broker and event-store, so time-ordered history of changed-data events are captured for auditing and/or replay purposes. In the occurrence of target-database downtime, changed-data events will not be lost but instead resume replication, upon recovery, from the last committed checkpoint.

RedisCDC has a cloud-native shared-nothing architecture which allows any cluster node (RedisCDC Instance) to perform either/both Job Management and Job Execution functions. It is implemented and compiled in JAVA, which deploys on a platform-independent JVM, allowing RedisCDC instances to be agnostic of the underlying operating system (Linux, Windows, Docker Containers, etc.) Its lightweight design and minimal use of infrastructure-resources avoids complex dependencies on other distributed platforms such as Kafka and ZooKeeper. In fact, most uses of RedisCDC will only require the deployment of a few JVMs to handle Job Execution with high-availability.

Integration with source and target databases is handled by an extendable connector framework. Each RedisCDC Connector uniquely interfaces with its source database's supported and built-in change data capture process. While there are multiple patterns to implement change data capture (polling-publisher, dual-writes, transaction-log tailing, etc.), RedisCDC connectors prioritize integration with source-database transaction-logs (when supported) to avoid impacting performance and availability.

1.2 Use-cases

1.2.1 Database Migration – Blue-Green Deployment

Technology modernization has become commonplace as enterprises move more workloads to the Cloud. While stateless applications can sometimes lift-and-shift, it is less likely that their non-cloud native databases are as portable without replatforming. This creates a major problem, specifically for mission critical applications, since the operational-risk of an outage can be unacceptable to the business. To mitigate operational-risk, enterprises often rely on a blue-green deployment in which the lift-and-shifted stateless application can continue to rely on the "blue" legacy database until enough testing has been performed on the new "green" cloud-native database and a cut-over can be planned with the assurance of a rollback. For this use-case, enterprises often rely on change-data-capture to maintain consistency between the legacy and new cloud-native databases until some level of confidence is reached with the new deployment.

1.2.2 Database Migration – Multi-Phase Replatforming | Strangler Application

Applications that are too complex, mission-critical, and/or large for a blue-green deployment sometimes require a multi-phase migration plan which spans months, or in some cases years, to complete (i.e. MIPS/Mainframe Offload). Typically, these projects are executed by migrating small targeted workloads, supported by legacy application(s) and their dependent database tables, to newly created modular application(s) or more recently microservice(s). In architectural terms this is referred to as the Strangler Application pattern. Since the legacy database will continue to act as the system of record, for its legacy cross-modular application dependencies, it will need to remain consistent with the database(s) serving the new microservices/applications. For this use-case, change-data-capture can be used on a single, or small set of tables, to replicate data between the legacy database and new database(s), in some cases bi-laterally, in order to keep them consistent for an extended duration.

1.2.3 Database Read-Replica | Cache Prefetching

Investments in digital and mobile applications continue to increase as major industries begin to personalize their customer experience to the expectations of millennials and generation-z. Since the digital experience often generates workloads that are different in terms of volume, velocity, and variability, traditional databases become too expensive or incapable to support them. While some enterprises choose to migrate away from legacy databases, using blue-green deployments or multi-phased replatforming, many decide to keep them as a system-of-record to support their existing operations while, at the same time, leverage read-replicas, or cache prefetching, to support new digital and mobile applications. For this use-case, change-data-capture is commonly used to replicate data indefinitely from the system-of-record to a new database, or cache, that is better suited to more demanding SLAs.

1.2.4 Hybrid-Cloud Deployment

Legacy enterprise ecosystem technologies, compliance risks, and operational concerns can sometimes impede enterprises from migrating their databases to the cloud. However, less mission-critical and/or green-field applications, are more often approved to be built/migrated on the cloud even though they may be dependent on some of the on-premises system-of-record data. For this use-case, change-data-capture can be used to replicate data indefinitely from the system-of-record to either a message-broker, or database, that is capable of replicating workloads, in some cases bi-laterally, between on-premises and cloud environments.

1.2.5 Cloud Bursting | Disaster Recovery | Business Continuity

Many industries are seeing an unprecedented increase in transaction volumes, reduction in brand loyalty, increase in competition, and extreme expectations on availability (in some cases regulated). Whether an enterprise intends to provision extra infrastructure to handle seasonal traffic peaks, invest more in data warehousing/analytics, recover quickly from a disaster, or maintain operational business continuity in the event of an entire data center failure, the common trend is to leverage the cloud for geo-distributed on-demand infrastructure. For this use-case, change-data-capture can be used, in the same way as covered in the Hybrid Cloud use-case, to replicate data from the source however, once on the cloud, is commonly complimented with further data-replication across multiple regions.

1.2.6 Microservice Pattern – Command Query Responsibility Segregation (CQRS)

Adoption of the Microservice Architecture continues to grow as more applications are replatformed and deployed on the Cloud. Since domain-driven design, isolation, and eventual consistency are at the heart of this architectural style, the need for different design patterns and a breakaway from traditional referential integrity (ACID constraints) is required. One of the more popular microservice design patterns is called Command Query Responsibility Segregation, commonly referred to by its acronym CQRS. In this pattern, different data-structures (commonly stored on different databases) are used to optimize for writes (command) and independently optimize for reads (query). Implementing this pattern is not trivial, since one of the main goals of a microservice is to be decoupled from the rest of the architecture, however, still maintain consistency and transform the replicated-data between the command and query data structures. For this use-case, change-data-capture is commonly used to implement CQRS between heterogenous databases by replicating and transforming data from the command to query data structures in near-real-time.

1.2.7 Data Integration | Materialized Views | Real-Time ETL | Search Engines | Multi-Model

Not all architectures are migrating away from their existing system-of-record(s), replatforming to a microservice architecture, moving to the cloud, nor have extreme business continuity requirements, however, still need to replicate their data to a different database. Data integration and the formation of materialized views require data consolidation from multiple heterogenous databases. Similarly, databases which were not designed for full-text searching must replicate their data to a capable search engine. While ETL (Extract, Transform, Load) and Search solutions have existed for years, the need for real-time performance and a reduction in operational expenditure (OpEx) is driving enterprises to seek alternatives. For this use-case, change-data-capture from multiple heterogenous databases to a multi-model database, such as Redis Enterprise, can support the demanding SLAs of systems requiring real-time ETL and/or real-time search (i.e. real-time analytics).

1.3 Supported Topologies

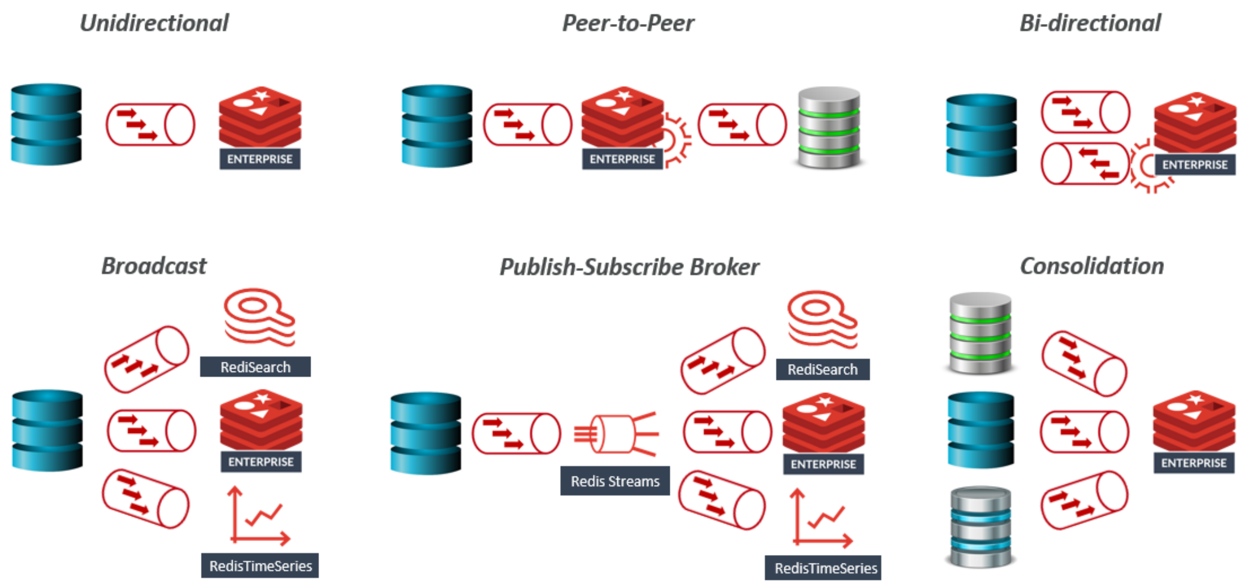


Diagram 1.1 – Topologies supported by RedisCDC

Note: *The following terms will be defined and used for readability -*

- *source - non-Redis Enterprise database (i.e. RDBMS, NoSQL, IMDG, etc.)*
- *target - Redis Enterprise database/module (i.e. RediSearch, RedisJSON, etc.)*

Unidirectional	**Peer-to-Peer**	**Bi-directional**
Replication from a source to a target. As an example, this topology could be applicable in a Database Migration use-case.	A sequenced cascading replication pattern which is performed between multiple sources and targets. As an example, this topology could be applicable in a Microservices-CQRS use-case.	Bi-lateral replication between a source and target. As an example, this topology could be applicable in a Hybrid-Cloud use-case.

Broadcast	**Consolidation**
A parallel cascading replication pattern from a source to multiple targets. In this topology, each RedisCDC Job would operate, end-to-end, in isolation between source and target. As an example, this topology could be applicable in a Read-Replica use-case in which disparate target databases might require the use of different Redis models without the need/overhead of brokered coordination.	Replication from multiple sources to a centralized target. In this topology, each RedisCDC Job would operate, end-to-end, in isolation between source and target. As an example, this topology could be applicable in a Materialized View use-case in which disparate sources would use Redis Enterprise as an in-memory view/cache for a workload that requires high throughput and real-time latency.

Publish-Subscribe Broker

A fan-out cascading replication pattern from a source to multiple targets with Redis Streams (*see section 1.5.6*) acting as an immutable ledger and message broker. In this topology, a single RedisCDC Job would replicate changed-data events from the source to a Redis Stream in order to record them, in order, for auditing/replay/de-duplication/etc. Subsequently, each target would have its own RedisCDC Job that uses a dedicated Redis Streams Consumer Group to read data at its preferred pace. As an example, this topology could be applicable in a Read-Replica use-case from a source that captures duplicate changed-data events, such as Apache Cassandra, which can be de-duped with the use of a message broker.

1.4 Supported Databases

Redis | Redis Enterprise

Redis is an open source in-memory NoSQL database which natively supports a variety of data structures and data models each uniquely designed for peak real-time performance and high-throughput use-cases across transactional, operational, and analytical workloads. It was originally popularized for its unparallel sub-millisecond performance, ease of use, embrace of well-known data structures, and more recently its extensibility via Redis Labs Modules.

Redis Enterprise picks up where Redis Open Source leaves off with full compliance to its RESP protocol including coverage of all data structures and operations; except for those that pertain to administration. That caveat exists because Redis Enterprise provides enterprise-grade administrative capabilities that are expected of a primary database including linear scalability, high-availability, durability, back-up and restore utility, security and unique capabilities such as multi-tenancy, active-active geo-replication, and tiered data storage.

Gemfire / Apache Geode

Apache Geode is a data management platform that provides real-time, consistent access to data-intensive applications. It was originally developed by GemStone Systems in 2002, commercially available as GemFire™, and eventually open-sourced as Apache Geode. Geode pools memory, CPU, network resources, and optionally local disk across multiple processes to manage application objects and behavior. It uses dynamic replication and data partitioning techniques to implement high availability, improved performance, scalability, and fault tolerance.

Microsoft SQL Server (MS SQL Server)

Microsoft SQL Server is a relational DBMS developed by Microsoft. Microsoft markets at least a dozen different editions of Microsoft SQL Server, aimed at different audiences and for workloads ranging from small single-machine applications to large internet-facing applications with many concurrent users. SQL Server Enterprise Edition includes both the core database engine and add-on services, with a range of tools for creating and managing a SQL Server cluster.

Apache Cassandra | DataStax

Apache Cassandra is a free and open-source, distributed, wide column store, NoSQL DBMS designed to handle large amounts of data across many commodity servers, providing high availability with no single point of failure. Cassandra offers robust support for clusters spanning multiple datacenters, with asynchronous masterless replication allowing low latency operations for all clients. A company called DataStax went on to create its own proprietary version of Cassandra, a NoSQL database called DataStax Enterprise (DSE).

1.5 Architecture

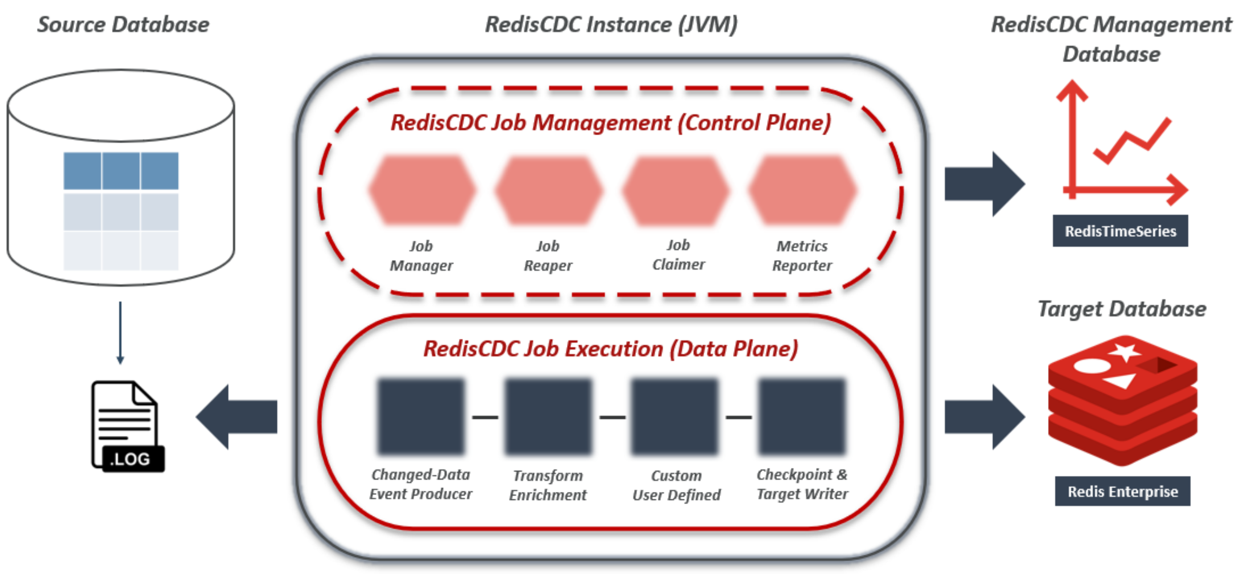


Diagram 1.2 – RedisCDC high-level Architecture

RedisCDC has a cloud-native shared-nothing architecture which allows any cluster node (RedisCDC Instance) to perform either/both Job Management and Job Execution functions. It is implemented and compiled in JAVA, which deploys on a platform-independent JVM, allowing RedisCDC instances to be agnostic of the underlying operating system (Linux, Windows, Docker Containers, etc.) Its lightweight design and minimal use of infrastructure-resources avoids complex dependencies on other distributed platforms such as Kafka and ZooKeeper. In fact, most uses of RedisCDC will only require the deployment of a few JVMs to handle Job Execution and Job Management with high-availability.

On their own RedisCDC instances are stateless therefore require Redis to manage Job Management and Job Execution state – such as checkpoints, claims, optional intermediary data storage, etc. With this design, RedisCDC instances can fail/failover without risking data loss, duplication, and/or order. As long as another RedisCDC instance is actively available to claim responsibility for Job Execution, or can be recovered, it will pick up from the last recorded checkpoint.

Note: For mission-critical production deployments, RedisCDC should use a Redis Enterprise database to support scalability, high-availability, recovery, geo-redundancy, monitoring/metrics, and security.

1.5.1 Job Execution

RedisCDC Jobs are configurable event-driven workflows (state machine) which transition state across multiple event-processors (workflow stages). Each event-processor is pinned to a dedicated thread and processes only its assigned workflow stage. Workflow instances are executed, end-to-end, within the same RedisCDC instance (JVM) to reduce network I/O, context-switching, and JVM garbage-collection pauses. This design allows for predictable resource utilization, optimized performance, and clean failover between RedisCDC instances.

As depicted in *Diagram 1.2*, a typical workflow only requires a few stages to complete end-to-end, which means that only a few threads are required to resource Job execution. However, since RedisCDC is a highly-configurable and extendable framework, users can add additional (sequenced or parallel) workflow stages to take advantage of built-in functions, deploy different topologies, and/or create their own user-defined stage-implementations.

The initial stage of each workflow is a changed-data event producer which, unlike the remaining stages, might utilize multiple threads since its function requires network I/O. The final workflow stage is typically a target-writer which is responsible for committing changed-data events and/or updating the checkpoint (*see section 1.5.4*) to one or more Redis Enterprise database(s). Once all workflow stages are completed, the changed-data event producer's barrier is unblocked to initiate the next batch of changed-data events into the data pipeline. Although communication with Redis Enterprise requires network I/O, Redis' sub-millisecond performance avoids the final stage from becoming the workflow's bottleneck.

1.5.2 Job Management

RedisCDC Job Management services are encapsulated within the same RedisCDC instances that can facilitate Job Execution. Job Management is broken up into separate services – each handling a distinct operational concern, supported by their own thread pool, and pre-configured via YAML files.

During the deployment process, RedisCDC instances can be enabled to join a pool of eligible Job Management members. For reliability and distributed consensus, each RedisCDC instance was designed to be stateless and therefore is dependent on a centralized Job Management Database (*see section 1.5.3*) to maintain critical state such as distributed locks, leases, and claims.

1.5.3 Job Management Database

RedisCDC is dependent on Redis to maintain critical Job Management metadata and act as a natural resolution mechanism for potential collisions. RedisCDC Job Management services take full advantage of Redis' data modeling flexibility by leveraging its data structures including Strings, Hashes, Sets, Sorted Sets, Lists, and Streams.

Note: *It is highly recommended to use Redis Enterprise for production environments*

Redis Enterprise provides enterprise-grade features, which extend the capabilities of Redis OSS, to act as a highly-available, scalable, resilient, performant, geo-redundant, and secure production database. It's supported extensibility allows RedisCDC to take advantage of Redis Modules, such as RedisTimeSeries, for metrics storage and querying.

1.5.4 Checkpoints

Checkpoints are used for offset management which is required to maintain continuity, order, delivery guarantees, and resiliency in the event of failure during Job Execution. Checkpoints are typically represented as a set of unique IDs/timestamps and are used to mark the last-used offset within the source-database transaction-log or outbox-table. Upon each iteration of Job Execution, a workflow stage will commit the new offset to a Redis database.

It's common for each source-database to implement its transaction-log differently, or with some variant, which may require tracking more than one position-marker to uniquely identify the offset. For this reason, RedisCDC connectors (*see section 1.5.5*) embed specialized logic to traverse and identify their source-database's position-maker(s) so offset management remains seamless to the user-experience.

Once a RedisCDC instance claims responsibility for Job Execution, it must determine the offset from which to begin its changed-data event producer. If the Job were claimed as a consequence of failover then the last-committed checkpoint would already be populated in Redis so Job Execution would pick up exactly where it had left off. However, for initial deployments the offset would not be automatically prepopulated, therefore the user can choose to add one manually to Redis before deploying RedisCDC. In the case, that offset position-marker(s) do not match those within the transaction-log then RedisCDC will start at the top.

A typical Job would conclude its workflow with a stage that either independently commits the checkpoint or combines this function with also performing the target-writer stage within a transactional scope. In either case, RedisCDC connectors design checkpoints to commit the offset after receiving an acknowledgment that all changed-data events have been successfully committed to the target-database.

Note: *Checkpoints are typically committed to the target-database. Not the Job Management database*

1.5.5 Connector Framework

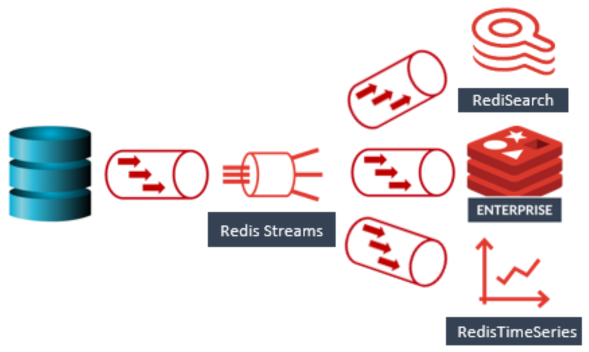
RedisCDC comes with its own connector framework that supports integration with a growing set of heterogenous databases including RDBMS, NoSQL, IMDG, and more. (*see Section 1.4 for complete list*). RedisCDC connectors encapsulate both source (read from source-database) and sink (write to target-database) integrations. The connector framework is extendable so while we encourage users to leverage existing connectors, it is technically possible to build a custom connector from scratch and integrate it with RedisCDC. (*contact RedisLabs for more information*).

Although some connectors leverage open-source libraries, RedisCDC is not limited functionally or dependent on any external process to operate. In fact, connectors are never deployed on their own as standalone processes. Changed-data event producers, transformers, data-mappers, file-managers, etc. can be added to custom-built connectors, however, will need to be packaged and deployed as part of a RedisCDC instance.

When change-data-capture is enabled on a source database, it will commonly begin an internal process to extract (capture) row-level changed-data events from a proprietary transactional log and replicate them to either an outbox table or persist them to a dedicated CDC write-ahead log (WAL). Each RedisCDC connector is implemented with compliance to its unique source-database's CDC process for versioning and continuous maintenance purposes. In addition, our preference to rely on a transactional-log-tailing integration pattern, whenever applicable, over polling-publisher integration with outbox tables take precedent for performance and stability.

1.5.6 Publish-Subscribe Broker

Redis Streams is an append-only immutable in-memory log data structure which can support concurrent publishers and subscribers to a given stream (a.k.a. topic or channel). In other words, in an abstract way, it behaves just like a write-ahead log (WAL) assuring that all published changed-data events remain in sequenced time-order, however, without disk-based limitations on performance. Subscribers (a.k.a. consumers) to a Redis Stream can be assigned to isolated consumer groups which can read changed-data events, at their own pace, without disruption or visibility by consumers outside of their group.



As described in *Section 1.3*, Redis Streams can be used as an intermediary publish-subscribe immutable ledger and message broker for changed-data events to solve for use-cases such as:

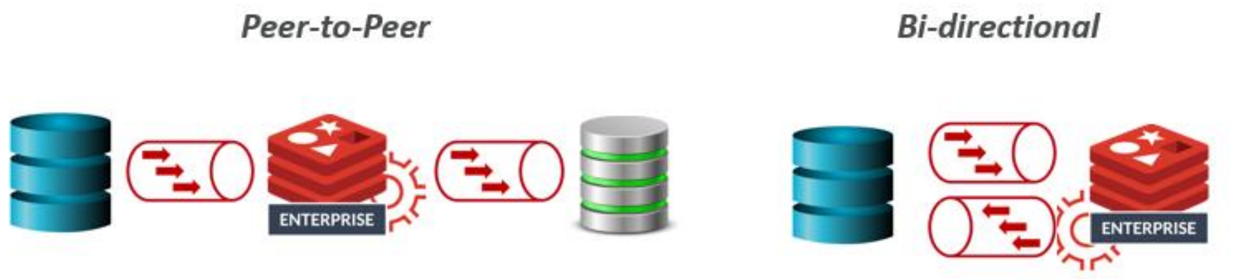
- fan-out
- de-duplication
- slow consumers
- data type channeling

In this topology, a single RedisCDC job would replicate changed-data from the source-database to a Redis Stream. Subsequently, each target-database would have its own RedisCDC Job that uses a dedicated Redis Streams Consumer Group to read data at its preferred pace. In other words, the consumer-side Jobs, would use Redis as the source-database, with each Job capable of having its own workflow choreography, stage-implementations, transformers, etc. To reduce overhead, Jobs can be run multi-tenant on the same RedisCDC instances.

1.5.7 Redis Durable Notifications and Write-Behind

RedisGears is a dynamic framework that enables developers to write and execute functions that implement data flows in Redis, while abstracting away the data's distribution and deployment. These capabilities enable efficient data processing using multiple models in Redis with infinite programmability, while remaining simple to use in any environment.

In this context, RedisGears can be used to provide durable notification of changed-data events within a Redis Database. In the same way that Redis Enterprise implements both stages to write-behind (capture and migration) to heterogenous databases, RedisGears can used to publish changed-data events to a Redis Stream for downstream consumption by a RedisCDC Job.



As shown in *Section 1.3,* RedisGears can be useful for peer-to-peer and bi-directional topologies. In both topologies, users can choose to either use Redis Enterprise's supported write-behind implementation or a RedisCDC job to replicate changed-data events out of Redis into a heterogenous database/S3/data-warehouse.

Note: *Special attention should be paid to avoid circular replication in a bi-directional topology*