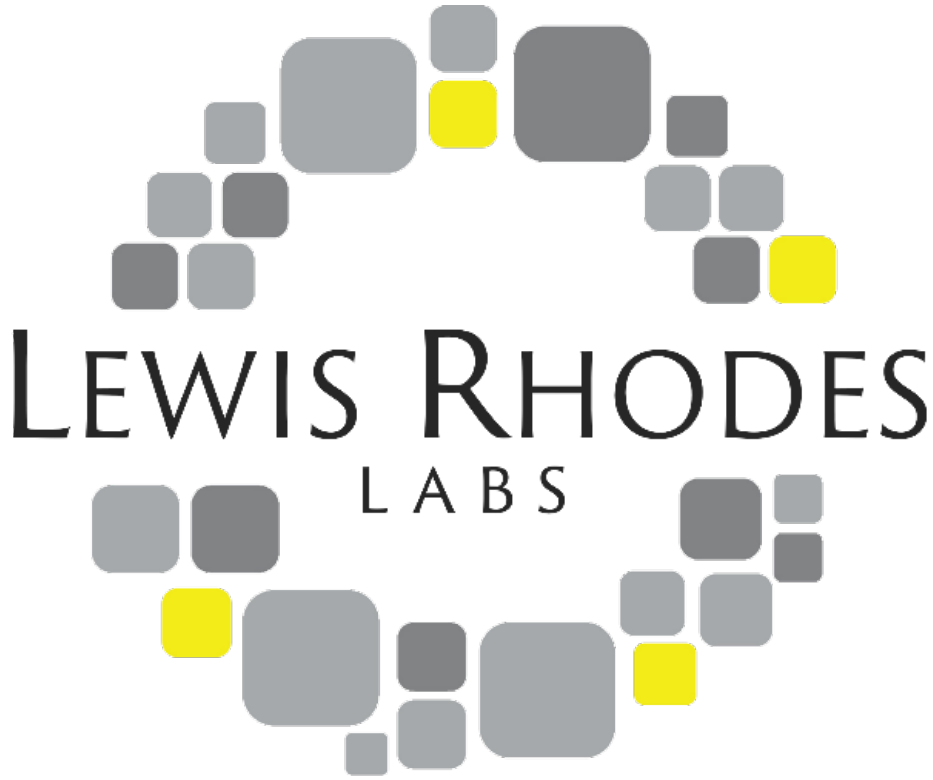

Extreme Search™ Manual

Version 2024.04

https://lewis-rhodes.com

support@lewis-rhodes.com

---

2 Architecture

2.1 Hardware Architecture

Each NPU*search* Computational Storage Device (CSD) consists of two types of components — npusearch devices and commodity NVMe SSDs — behind a peer-to-peer capable PCIe switch. A SmartSSD based NPU*search* CSD contains a single npusearch device and a single 4TB SSD. A Kuona NPU*search* CSD contains 4 npusearch devices, along with up to 8 SSDs.

Each npusearch device contains a region of on device memory — used for data, control and results buffers — and a collection of Processing Elements (PEs). The PEs of each npusearch device may be divided uniformly between a configurable number of "chains," with each chain processing data independently in parallel. The npusearch device implemented on the SmartSSD supports a maximum of 12 chains. The npusearch device on the Kuona card supports a maximum of 16.

Each chain scans data from buffers in the device memory at a fixed bandwidth, with each chain of a device scanning different data in parallel. The maximum scan bandwidth of a device, running with a given number of chains, is thus the product of the number of chains in use times the fixed per-chain rate of the device.

Each chain of an npusearch device processes files sequentially, scanning the entirety of one file before moving on to the next. Each file must be scanned by exactly one chain of one npusearch device. As such, scan performance is in general maximized when each device is asked to scan a large enough number of files such that every chain is active most of the time.

2.2 Software Architecture

In default operation, with automatic decompression disabled, data is provided to the npusearch device by `open(2)`ing files with flags including `O_DIRECT` and `pread(2)`ing from them into the device memory. This relies on the operating system's filesystem and NVMe driver stack generating p2p reads from the SSD into device memory, which will happen correctly on all supported operating systems (linux 5.4 and higher) when the SSDs are formatted using `XFS` and `ext4` filesystems. It is likely that other filesystems would support this as well, but they have not been tested. The details of these transfers are left entirely up to the OS's discretion.

When decompression is enabled, the first 64KB or more of every file is initially read into host memory and probed for compression. When a compressed file is detected, its contents are read into host memory, decompressed using `libarchive(3)`, and the resulting decompressed contents are moved to the NPUsearch device. When compression is not detected, the remaining handling of that file is the same as with decompression disabled.

As a result, decompression performance is dependent on CPU, memory bandwidth, and decompression library performance. This also means that when running with decompression enabled, every non compressed file sees an additional overhead on the order of 10s of microseconds. For files with sizes in the 10s of MB range or larger this has almost no observable impact, but for small files the performance impact can be noticable or even significant.

Disabling decompression on a per-scan basis avoids this additional overhead.

2.2.1 Backend Management

Each npusearch device is driven by an instance of a platform dependant "backend" application — currently found at either `/opt/lrl/lib/npusearch/smartssd-serve-generic-shared` or `/opt/lrl/lib/npusearch/kuona-serve-generic-shared`. Each backend is launched with an npusearch device and a list of directories it is responsible for.

Typically, the backends on one server will all be started using a management script, found at either `/opt/lrl/bin/npusearch_startup_smartssd.sh` or `/opt/lrl/bin/npusearch_startup_kuona.sh`. These scripts enumerate all npusearch devices and all mounted partitions of all npusearch SSDs. The scripts then assign each SSD to one npusearch device, and launch a backend per device passing it all mount points for its corresponding SSDs.

The npusearch startup scripts are, in turn, typically run under systemd, with the unit `npusearch.service`. The systemd unit loads the npusearch config file, verifies that redis is reachable and that backends are not running on the machine it would bring up backends on, launches the startup script, and restarts the backends if they exit unless they exit with code `0`. The state of the backends may generally be inspected using `systemctl` and `journalctl`, and they may be started, stopped, and restarted using `systemctl` like other systemd services.

In rare cases it may be desirable to run the startup scripts manually. In this case, care **must** be taken to ensure that backends are not already running. Configuration may be passed to the startup script by setting environment variables, as found in `/opt/lrl/etc/npusearch.conf`, as discussed below.

2.2.2 Client Access (Frontend)

Access to the npusearch capabilities is provided by the `npusearch` python package. The `npusearch` package provides a client access object, `npusearch.NPUGlusterClient`, which encapsulates communication with the backends, as well as optionally gluster. An `NPUGlusterClient` can be used to scan files, retrieve information

on backends and shutdown some or all backend. It can also, as a convenience, be used to list all files matching a glob pattern in a distributed fashion, optionally including their sizes or size, extent, mode, uid and gid as returned by `stat(2)`.

The `NPUGlusterClient` communicates with the backends via redis pubsub. Each backend chooses a unique request channel, subscribes to pubsub messages from redis on that channel, and registers itself by inserting the directories it is responsible under that channel in a redis hash at a key derived from the registry name. Each backend registers itself on startup after it has begun listening for messages, and re-registers itself periodically thereafter.

Each client object chooses a unique response channel for itself, and subscribes to redis pubsub messages on that channel. To issue commands, a client looks up the request channels for backends subscribed under the registry it was configured with and sends a message to each of those channels containing a request id, its response channel, the command, and any additional arguments the command requires. If a message intended for a given backend has no subscribers, delivery to that backend is assumed to have failed; otherwise, delivery of the request is assumed successful. The client then waits for messages on its response channel, and processes them as they come in. Each backend will respond with one or more messages, with all but the last marked "partial". The client will continue listening for response messages until either every backend which was listening responds with a non-partial message or a timeout is exceeded.

Each backend can only run one `scan` operation at a time, as well as one non scan operation such as an `info`, `ls`, `sizes` or `stat` request. All `scan` operations which will inspect at least one file are queued in the backend and may be cancelled up until they have completed. All other operations are queued only in the redis pubsub output queue. Many or long running non-scan requests may in some cases delay `scan` requests from being submitted to the hardware. If many requests will be submitted in parallel, the redis `client-output-buffer-limit` for `pubsub` should be increased to some large value, or set to unlimited.

2.2.3 Gluster Integration

When the GlusterFS integration is being used, each backend must be run with permissions allowing it to read the files under `/var/lib/glusterd/vols`, and the client must be created passing in one or more gluster hosts from the same trusted pool (`gluster_hosts`) and a gluster volume name (`volname`). The client accesses gluster via the standard python wrapper for `libgfapi.so`. The backends look up the locations of gluster bricks they are responsible for by reading the files under `/var/lib/glusterd/vols` directly, and do not interact with the gluster daemon in any way.

When gluster is used and the backends receive a request specifying a `volname`, they will treat all file patterns specified in the request as relative to the gluster brick directories for that volume that are located under directories they are responsible for. The client will **only** interact with gluster directly for commands that ask for paths under gluster with a return type including `CHECKED`. In this case, each file storage path in every response from the backends is checked against gluster to verify that it is a canonical copy of the corresponding gluster volume.

If gluster paths are requested, but `CHECKED` is **not** requested, the conversion from storage path to corresponding gluster path is done via simple path rewriting and is not verified. This may, in some cases, result in matches reported for files which exist under gluster brick directories but which do not correspond to canonical copies of files under gluster — for example because gluster is unaware of the file, or because a brick contains a stale copy of a file in a replicated volume. Unchecked return types will never in and of themselves result in files not being matched when they should have been.

While in some cases using unchecked returns may result in files being reported which do not, as far as gluster is concerned, exist, the process of checking file locations with gluster can cause substantial slowdown when there is a very high match rate or a modest match rate with very many small files.