fastpath

Arm Limited

Mar 31, 2026

CONTENTS

CHAPTER

ONE

CONTENTS

1.1 Introduction

1.1.1 Overview of Fastpath

Fastpath is a command-line tool specifically designed for monitoring the performance of the Linux kernel by executing structured performance benchmarks on a diverse range of hardware platforms. It automates the collection, storage, and analysis of performance data to assess how Linux kernel modifications impact various workloads. By structuring execution plans in YAML and providing flexible result management, Fastpath enables developers and performance engineers to detect regressions, validate optimizations, and track long-term performance trends effectively.

1.1.2 Key Features & Capabilities

- **Benchmark Execution Automation**: Define and execute benchmark runs using structured plans.
- **Flexible Result Management**: Store results in CSV, SQLite, or MySQL formats for analysis.
- **Statistical Performance Analysis**: Compare multiple benchmark runs to detect performance regressions or improvements.
- **Command-Line Driven**: Lightweight and scriptable for CI/CD integration.
- **Multi-Architecture Support**: Native support for arm64 and x86_64.
- **Extensibility**: New benchmarks can be integrated without needing to modify the tool.
- **Containerized Benchmarks**: Benchmarks are deployed as containers so that all user-space code remains identical across different SUTs.
- **Interactive Dashboard**: Visualize results in web browser.
- **Performance Regression Bisection (Coming Soon!)**: Identifies the specific Git commit responsible for performance regressions.

1.1.3 Supported Workflows & Use Cases

Fastpath is used in various scenarios, including:

- **Kernel Performance Tracking**: Monitoring Linux kernel updates and their impact on workloads.
- **Hardware Comparison**: Comparing different systems under test (SUTs) to evaluate performance.
- **Automated CI/CD Benchmarking**: Running benchmarks as part of a continuous integration process.

1.1.4 Links

- Documentation is available at: https://fastpath.docs.arm.com
- Source Code is available at: https://gitlab.arm.com/tooling/fastpath
- Container Images are available at: https://gitlab.arm.com/tooling/fastpath/container_registry

1.1.5 Repository Structure

Key directories within the repository:

Directory	Description
./benchmarks	Standard benhmark library. Includes all benchmark description yamls.
./containers	Definitions for all container images containing benchmarks.
./documentation	Source for this documentation.
./fastpath	CLI tool implementation.

1.1.6 Repository License

The software is provided under an MIT license (more details in *License*).

Contributions to the project should follow the same license.

1.1.7 Contributions and Bug Reports

Contributions are accepted under the MIT license. Only submit contributions where you have authored all of the code.

If you're hitting an error/bug and need help, it's best to raise an issue in GitLab.

1.1.8 Maintainer(s)

- Ryan Roberts <ryan.roberts@arm.com>
- Aishwarya TCV <Aishwarya.TCV@arm.com>
- Aishwarya Rambhadran <Aishwarya.Rambhadran@arm.com>

1.2 User Guide

1.2.1 Installation & Setup

System Requirements

Fastpath requires the following dependencies:

- **A Linux system with Python 3.x** is required to run the tool.
- **A separate Linux system as the SUT** is required, which must be configured according to the requirements at *Setup & Configure a SUT*.
- **SSH access** is necessary for communication between the host system and the SUT.

Installing Fastpath

Installing from Source

First, optionally create and activate a Python virtual environment to install all the Python package dependencies into:

```
python3 -m venv ~/venv
source ~/venv/bin/activate
```

To install Fastpath from the source repository, run:

```
git clone https://git.gitlab.arm.com/tooling/fastpath.git
pip install -r fastpath/fastpath/requirements.txt
export PATH=$PWD/fastpath/fastpath:$PATH
```

Verifying Installation

Run the following command to verify installation:

```
fastpath --help
```

If Fastpath is installed correctly, this command will display available options and supported commands.

Fastpath commands are all of the form:

`fastpath [global-options] <noun> <verb> [command-options]`

So you can additionally get help on any specific noun (e.g. `plan`) or verb (e.g. `exec`):

```
fastpath plan --help
fastpath plan exec --help
```

Configuring User Preferences

Fastpath has an extensible user preference system that allows users to customize certain aspects. User preferences can be stored in the INI file at `~/.fastpathpreferences` or can be specified as environment variables. If a preference is specified in both places, the environment variable takes precedence.

Each preference has a category and a name, which maps to a string. To set and get preferences in `~/.fastpathpreferences`, use the `fastpath preference` noun:

```
$ fastpath preference set --category default --name resultstore mysql://
↪fpuser:MyPassword@my.sql.server.com/fastpath-db
default.resultstore: mysql://fpuser:MyPassword@my.sql.server.com/fastpath-db

$ fastpath preference get --category default --name resultstore
default.resultstore: mysql://fpuser:MyPassword@my.sql.server.com/fastpath-db
```

When specified as an environment variable, concatenate the upper case category and name as `FASTPATH_<CATEGORY>_<NAME>`:

```
$ export FASTPATH_DEFAULT_RESULTSTORE=mysql://fpuser:MyPassword@my.sql.server.com/
↪fastpath-db
```

The following user preferences are currently defined:

category	name	default	description
default	resultstore	<None>	For a command that requires a resultstore, the default resultstore to use if one is not specified explicitly on the command line.
global	library	<None>	Colon-separated list of paths to custom benchmark libraries. Benchmark includes are searched for relative to the current directory as well as relative to these paths.

1.2.2 Setup & Configure a SUT

The System Under Test (SUT) is the system on which benchmarks are executed. Fastpath communicates with the SUT over SSH to configure it (e.g. install the kernel to be benchmarked, reboot, configure kernel command line and sysctls, etc). Then it invokes benchmarks by pulling and running Docker containers on the SUT.

Fastpath's requirements for the SUT are as follows:

- Must use grub to boot the kernel and kernel must be stored at /boot
- A minimal set of shell utilities must be available
- Docker must be installed
- A user account must be available which is a member of the "docker" and "sudo" groups
- The user account must have its public key setup such that it can SSH into the SUT without the need for a password
- The user account must be configured for passwordless sudo usage

Ubuntu is well tested with Fastpath and is known to meet the first 2 requirements out of the box: Using Ubuntu as the SUT's operating system is recommended for this reason.

Install Docker

This example assumes Ubuntu is installed on the SUT. Refer to distribution documentation if using a different distro. The apt version of Docker is preferred as issues have been observed with the snap version:

```
sudo apt-get install docker.io
sudo groupadd docker
sudo usermod -aG docker $USER
```

Create User Account

This example assumes Ubuntu is installed on the SUT. Refer to distribution documentation if using a different distro. This shows how to create a dedicated user account on the SUT that can be used by fastpath. You are free to use an existing account as long as it meets the stated requirements.

First, on the SUT, create a group and user account, which is a member of the docker and sudo groups. Then configure the user for passwordless sudo. Finally set a password for the user:

```
sudo groupadd fpuser
sudo useradd -g fpuser -m fpuser
sudo usermod --shell /bin/bash fpuser
sudo usermod -a -G docker fpuser
sudo usermod -a -G sudo fpuser
echo "fpuser ALL=(ALL) NOPASSWD: ALL" | sudo tee /etc/sudoers.d/fpuser
sudo passwd fpuser
<enter desired password>
```

Next, on the system that will run Fastpath, create a public/private key pair for the user and install the public key on the SUT. This can be performed with a combination of `ssh-keygen` and `ssh-copy-id`.

Warning: The private key must be kept secure as it can be used to access the SUT with root privileges.

You can validate the setup by running the Fastpath `fingerprint` command. If it successfully outputs all the HW and SW details, you have succeeded:

```
fastpath sut fingerprint --user fpuser [--port <id>] [--keyfile <filename>] <SUT␣
↪hostname or IP>
```

1.2.3 Build a Suitable Kernel

Fastpath will install on the SUT any kernel you provide to it and reboot into that kernel. There are few requirements for the features that must be enabled in the kernel, but it must have the required drivers to be able to boot the SUT (either built-in or as modules, as appropriate), and it must be able to support running Docker. Note that the arm64 defconfig does not support Docker.

A full kernel config can be used in place of defconfig when building the kernel. This can be helpful if you need to reproduce the exact environment of a SUT or have a known-good config you want to use. In some cases, such as with cloud VMs, defconfig may be missing required options for the system to boot or operate properly. If only a few additional options are needed, they can be added using a config fragment instead.

For Arm's internal usage of Fastpath, we predominantly build kernels with defconfig + the Kconfigs described in the fastpath.frag Kconfig fragment. This provides all the features required by Docker and all the drivers for the various HW we run on internally. The fragment is provided as an example only.

Fastpath requires a kernel image (see *swprofile object*), which may be compressed or uncompressed. If using modules, you will also provide a tarball of the modules in the plan's swprofile section.

There are many ways to build the Linux kernel, but tuxmake is a convenient option for this case because it includes all required tools in a container and outputs both the kernel image and the modules in the required formats. Assuming your linux source code is at `./linux`:

```
tuxmake \
      --directory linux \
      --output-dir my-kernel \
      --target-arch arm64 \
      --kconfig defconfig
      --kconfig-add kconfigs/fastpath.frag \
      --runtime docker \
      --fail-fast \
      config kernel modules
```

On completion, `my-kernel/Image.gz` and `my-kernel/modules.tar.xz` provide files that Fastpath requires.

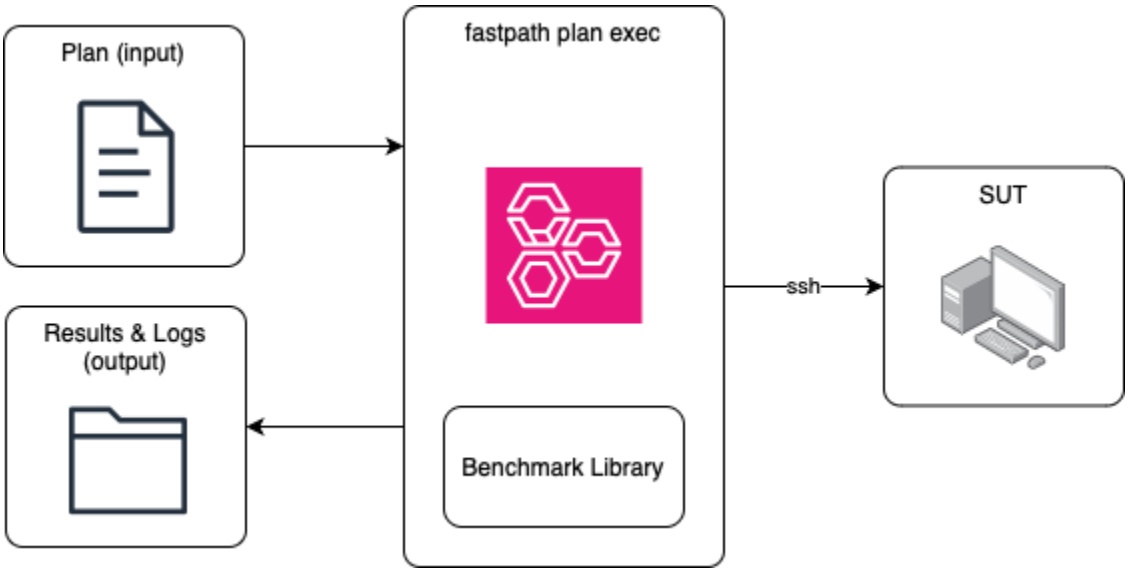
1.2.4 Define & Execute a Plan

Introduction

A plan is a yaml file that defines:

- The connection to the SUT that will run all the benchmarks
- A list of software profiles for which the benchmarks will be run
- A list of benchmark to run

The user creates a plan and passes it to Fastpath for execution. Fastpath configures the SUT for each software profile in turn, by installing the specified kernel and booting it with the specified command line parameters, etc, and runs the specified benchmarks, collating the results and logs into an output "resultstore".



The plan may refer to supplementary benchmark yaml files which are part of a Benchmark Library. Fastpath ships with a default Benchmark Library. Users may additionally set up their own, private Benchamrk Libraries.

See *Plan Schema* for a full description of the plan and benchmark library.

Build a Simple Plan

1. Define the `sut` section:

```
sut:
  name: my-fastpath-sut
  connection:
    method: SSH
    params:
      user: fpuser
      host: <sut hostname>
      port: <ssh port>
      keyfile: <private key>
```

This section defines the parameters to connect to the SUT over SSH. Here we assume you're using the `fpuser` account as per the guide at *Setup & Configure a SUT*. The port will usually be 22, and in that case there is no need to specify it since that's the default. The keyfile is the private key associated with the user account which will allow logging in to the SUT via SSH as described at *Setup & Configure a SUT*.

Alternatively, you could define all of this as part of your ~/.ssh/config then refer to the config name using the only host field:

```
sut:
  name: my-fastpath-sut
  connection:
    method: SSH
    params:
      host: my-fastpath-sut
```

Warning: The private key must be kept secure as it can be used to access the SUT with root privileges.

2. Define the `swprofiles` under which the benchmarks will execute:

```
swprofiles:
  - name: without-mthp
    kernel: /path/to/Image.gz
    modules: /path/to/modules.tar.xz
    cmdline:
      - thp_anon=64K:never;2M:madvise
  - name: with-mthp
    kernel: /path/to/Image.gz
    modules: /path/to/modules.tar.xz
    cmdline:
      - thp_anon=64K:always;2M:madvise
```

Each swprofile must contain the path to the kernel image, and if modules are needed then the path to the modules tarball must be provided as well. See *Build a Suitable Kernel* for a guide to building an appropriate kernel.

This example defines 2 swprofiles, both using the same kernel but one has 64K multi-size THP disabled and the other has it enabled. This demonstrates how to configure the kernel command line. We give each swprofile a convenient name so that we can easily refer to them when doing analysis on the results.

3. Define the `benchmarks` to execute:

```
benchmarks:
  - include: speedometer/v2.1.yaml
  - include: mmtests/kernbench.yaml
```

Here we will run 2 benchmarks, Speedometer v2.1 (Javascript benchmark) and kernbench (Time to compile the linux kernel). The actual benchmark definitions are in the yaml files within the Benchmark Library.

4. Add some miscellaneous parameters:

```
defaults:
  benchmark:
    warmups: 1
    repeats: 3
    sessions: 2
    timeout: 1h
```

Each benchmark may specify some optional parameters, including how many times to run each benchmark and a per-iteration timeout. In cases where the values are not explicitly provided for a benchmark, Fastpath will fallback to default values, which can be specified globally here. In this case, we request 2 sessions meaning the system will be booted twice. Then in each session we will run 1 warmup run, where the results are discarded, and 3 normal repeats where the results are kept. See *Plan Schema* for more info.

Putting it all together, we have the following which should be saved as demo.yaml:

```
sut:
  name: my-fastpath-sut
  connection:
    method: SSH
    params:
      host: my-fastpath-sut
swprofiles:
  - name: without-mthp
    kernel: /path/to/Image.gz
    modules: /path/to/modules.tar.xz
    cmdline:
      - thp_anon=64K:never;2M:madvise
  - name: with-mthp
    modules: /path/to/modules.tar.xz
    cmdline:
      - thp_anon=64K:always;2M:madvise
benchmarks:
  - include: speedometer/v2.1.yaml
  - include: mmtests/kernbench.yaml
defaults:
  benchmark:
    warmups: 1
    repeats: 3
    sessions: 2
    timeout: 1h
```

Let's validate the plan with the following command:

```
fastpath plan show demo.yaml
```

This will parse, normalise, flatten, and validate the plan. If your plan is buggy, this command will tell you why. If your plan is well-formed, it will output the plan in its final form:

```
%YAML 1.2
---
defaults:
  benchmark:
    repeats: 3
    sessions: 2
    warmups: 1
    timeout: 1h
sut:
  name: my-fastpath-sut
  connection:
    method: SSH
    params:
      host: my-fastpath-sut
      user: null
      port: null
      keyfile: null
swprofiles:
- name: without-mthp
  pkgtype: null
  kernel: /data_nvme0n1/ryarob01/fastpath/demo/assets/Image.gz
  modules: /data_nvme0n1/ryarob01/fastpath/demo/assets/modules.tar.xz
  cmdline:
  - thp_anon=64K:never;2M:madvise
  sysctl: []
  bootscript: []
  gitsha: null
- name: with-mthp
  pkgtype: null
  kernel: /data_nvme0n1/ryarob01/fastpath/demo/assets/Image.gz
  modules: /data_nvme0n1/ryarob01/fastpath/demo/assets/modules.tar.xz
  cmdline:
  - thp_anon=64K:always;2M:madvise
  sysctl: []
  bootscript: []
  gitsha: null
benchmarks:
- suite: speedometer
  name: v2.1
  type: browser
  params: {}
  image: registry.gitlab.geo.arm.com/software/linux-arm/fastpath/containers/
↪speedometer:v1.0
  repeats: 3
  sessions: 2
  warmups: 1
  timeout: 10m
```

(continues on next page)

(continued from previous page)

```
- suite: mmtests
  name: kernbench
  type: system
  params:
    config: |-
      export KERNBENCH_ITERATIONS=1
      export KERNBENCH_VERSION=6.12
    config-base: configs/config-workload-kernbench-max
    performance-governor: 'False'
  image: registry.gitlab.geo.arm.com/software/linux-arm/fastpath/containers/mmtests:v1.0
  repeats: 3
  sessions: 2
  warmups: 1
  timeout: 1h
```

Execute the Plan

Execute the plan, placing the results in the results directory:

```
fastpath plan exec --output results/ demo.yaml
```

The plan will execute, showing a progress as benchmarks are completed:

```
Executing demo.yaml...
  3%|                                                                    | 1/32␣
↪[05:01<2:35:48, 301.57s/it]
```

Upon completion, the results can be managed and analysed using various Fastpath commands. See *Store & Analyse Results* for more information.

1.2.5 Store & Analyse Results

Introduction

All results generated by benchmarks, along with metadata about those benchmarks, suts and swprofiles, are stored in a "resultstore". Fastpath supports 3 formats for resultstores. Where ever fastpath expects a resultstore, you can pass a format-specific URL as follows:

- **CSV**: `csv:///<directory>`
- **SQLite**: `sqlite:  sqlite:///<dbfile>`
- **MySQL**: `mysql://<user>:<password>@<host>:<port>/<dbname>`

Note: If the protocol is omitted, the URL is assumed to be pointing to a directory containing all the CSV files for the CSV format.

Note: When using a MySQL resultstore, it is sufficient for a user to have RO access when performing operations that only require reading data (e.g. `fastpath result list`, `fastpath result show` or when the resultstore is a

source for `fastpath result merge`). For operations that write data to the resultstore, the user must have read-write access.

All 3 formats are interchangable; all data can be represented with equal fidelity in all 3 formats, and it is possible to move data between the formats and merge data selectively from multiple resultstores into a single unified resultstore.

List Contained Objects

A resultstore catlogues 3 types of top-level objects; `sut`, `swprofile` and `benchmark`. Each object instance in the resultstore has an ID, which is used to refer to that specific object in many contexts; for example when requesting that `fastpath result show` or `fastpath result merge` filters certain objects.

The following command lists all objects of the specified type, listing their IDs and metadata specific to each object. This example lists swprofiles:

```
fastpath result list results/ --object swprofile
```

Additionally you can provide an ID glob to filter the listed objects:

```
fastpath result list results/ --object swprofile --id *without*
```

Analyse Results

Fastpath's `fastpath result show` command can perform a range of analysis on results in a resultstore. Let's use the resultstore generated from the tutorial in *Define & Execute a Plan* to demonstrate some of the features.

First, let's look at the summary statistics for all the benchmarks of a single swprofile for a single sut. There is only a single sut in the resultstore so it is implicitly used. We specify the single swprofile we would like to view results for and specify that we want to see the summary statistics relative to the mean. The resulting table shows min/mean/max as well as the confidence interval bounds (95% by default but this can be overridden), the coefficient of variation (that's the standard deviation expressed as a percentage of the mean) and the number of samples:

```
fastpath result show results/ --swprofile without-mthp --relative
```

```
Results for SUT my-fastpath-sut and SW Profile without-mthp:
```

Benchmark	Result Class	min	ci95min	mean	ci95max	max	cv	count
mmtests/kernbench	elsp-64 (seconds)	-0.20%	-0.19%	389.80	0.19%	0.21%	0.18%	6
	syst-64 (seconds)	-0.58%	-0.51%	2155.99	0.51%	0.54%	0.48%	6
	user-64 (seconds)	-0.14%	-0.15%	20595.16	0.15%	0.14%	0.14%	6
speedometer/v2.1	score (runs/min)	-1.23%	-1.16%	162.00	1.16%	1.23%	1.10%	6

Now let's compare results between 2 swprofiles. This time we get a summary table that compares the means, and if there are any statistically significant changes, regressions are highlighted in red and improvements are highlighted in green. Here we can see that enabling 64K mTHP in the kernel improved the kernbench benchmark by 5.17% on average. The speedometer benchmark also improved by 1.65% on average, but this is not deemed statistically significant so it is not highlighted.

Note: A change is considered statistically significant if the confidence intervals (95% by default but this can be overridden) for the 2 results being compared do not overlap. Additionally the comparison mean must be beyond `noise-threshold` percent of the baseline mean. The `noise-threshold` can be specified but defaults to 1%. Whether

a change is considered an improvement or regression depends on the measurement. For example times usually improve when they get smaller, rates usually improve when they get bigger.

```
fastpath result show results/ --swprofile without-mthp --swprofile with-mthp --relative
```

Results for SUT my-fastpath-sut:

Benchmark	Result Class	without-mthp	with-mthp
mmtests/kernbench	elsp-64 (seconds)	389.80	-5.17%
	syst-64 (seconds)	2155.99	-43.83%
	user-64 (seconds)	20595.16	-1.46%
speedometer/v2.1	score (runs/min)	162.00	1.65%

`fastpath result show` supports a number of ancillary options. For example, we can print the table using only ascii characters. In this case, only ascii characters are used and instead of coloring the output for improvements and regressions, the symbols (I) and (R) are displayed. This can be useful for plain text email and for people with color blindness. Additionally you can filter the results to show only improvements (See `fastpath result show --help` for more options):

```
fastpath result show result/ --swprofile without-mthp --swprofile with-mthp --relative --
↪format ascii --display improvement
```

```
Results for SUT my-fastpath-sut:

+------------------+------------------+----------------+-------------+
| Benchmark        | Result Class     |   without-mthp |   with-mthp |
+==================+==================+================+=============+
| mmtests/kernbench | elsp-64 (seconds) |         389.80 |  (I) -5.17% |
|                  | syst-64 (seconds) |        2155.99 | (I) -43.83% |
+------------------+------------------+----------------+-------------+
```

Copy and Merge Data

Fastpath supports the `fastpath result merge` command to copy data from one or more source resultstores into a destination resultstore. If the destination does not exist, it will be created. It is already exists, data is merged with any existing data; the `--append` option is required for the latter scenario.

It is also possible to filter the data to be copied/merged based on sut, swprofile or benchmark ids.

See the command help for more information:

```
fastpath result merge --help
```

1.2.6 Dashboard

Note: The Fastpath Dashboard is currently under active development. Features and interface may change in the future.

The Fastpath Dashboard provides a web-based interface for visualizing and comparing benchmark results stored in your result store. The dashboard is built using Streamlit and runs as a local web server.

Starting Dashboard

To start the dashboard server:

```
fastpath dashboard start <resultstore url>
```

The dashboard will automatically open in your default web browser at `http://localhost:8501`.

If you have configured a default result store in your preferences (see *Installation & Setup*), you can omit `<resultstore url>` and the default will be used.

Using Dashboard

Filter and View Results

The dashboard allows you to interactively filter and visualize your benchmark results:

1. **Select SUT(s)**: Choose one or more Systems Under Test
2. **Select SW Profile(s)**: Choose SW Profiles/Profile
3. **Select Benchmark**: Pick the benchmark to analyze
4. **Select Result Class**: Choose the specific metric to visualize

Note: For comparative analysis, select either:

- **One SUT + Multiple SW Profiles**: Compare different SW Profiles on the same hardware
- **Multiple SUTs + One SW Profile**: Compare different hardware with the same SW Profile

Main Chart

The main chart displays:

- **95% Confidence Interval** (thick blue line): The statistical range where the true mean likely falls
- **Mean value** (white tick mark): Average of all measurements
- **Raw data points** (crosses): Individual measurement values from each test session
- **Hover tooltips**: Detailed statistics including min, max, mean, CI bounds, standard deviation, and sample count

Visualization Controls

- **Pan and zoom**: Click and drag to pan, scroll wheel to zoom
- **Session colors**: Each test session is shown in a different color (up to 20 colors)

Deviation Chart

The deviation chart shows relative performance from your baseline:

- **Baseline**: The first selected item (SUT or SW Profile) serves as the reference
- **Comparison**: All other selected items are compared against this baseline
- **Color coding**:
  - **Green bars**: Performance improvement over baseline
  - **Red bars**: Performance regression from baseline
  - **Gray bars**: No significant change (within 1% noise threshold)
- **Toggle view**: Use the sidebar checkbox to switch between relative (%) and absolute differences

---

Note: A change is classified as "improvement" or "regression" (green/red) only if **both** conditions are met: (1) the 95% confidence intervals do not overlap, and (2) the difference exceeds the 1% noise threshold. Otherwise, it is classified as "no significant change" (gray).

View Metadata

Expand the metadata sections at the bottom to see detailed information about:

- Selected SUT hardware specifications
- Selected SW Profiles
- Benchmark parameters

Aggregated Results Table

Click the **"Aggregated Results Table"** expander below the main chart to view:

- SUT and SW Profile identifiers
- Statistical summary: min, ci95min, mean, ci95max, max
- Standard deviation and sample count

This table shows the same data as the chart in tabular format.

Result Caching

The dashboard uses **Streamlit caching** with a 10-minute TTL (time-to-live) for result store data. This improves performance by avoiding repeated data loads.

To manually refresh the data before the cache expires, click the **" Refresh Data"** button in the sidebar. This clears the cache and reloads data from the result store.

Stopping Dashboard

Press `Ctrl+C` in the terminal where the dashboard is running to stop the Streamlit server.

1.2.7 Kernel Regression Bisection

Fastpath provides automated kernel regression bisection to identify which specific kernel commit introduced a performance regression. This uses Git bisection integrated with Fastpath's benchmarking and analysis capabilities.

Overview

The bisection process:

1. **Prepare** - Identify good/bad kernels and create bisection context (fastpath bisect start)
2. **Execute** - Git bisect automatically builds, tests, and evaluates each commit (git bisect run)
3. **Analyze** - Git bisect identifies the first bad commit (git bisect log)

Prerequisites

- A result store with benchmark results from both:
  - **Good swprofile** - Known working version with acceptable performance
  - **Bad swprofile** - Version showing the regression
- Both swprofiles must have been tested with:
  - Same SUT (System Under Test)
  - Same benchmark
  - Identical configuration (cmdline, sysctl, bootscript)
  - Only difference should be the kernel git SHA
- SSH access to the SUT for running tests
- Kernel source repository for building commits

Prepare Context

Create a bisection context file with `fastpath bisect start`:

```
# Basic command structure
fastpath bisect start \
  --host <hostname> --user <user> --port <port> --keyfile <keyfile> \
  --sut <sut-id> \
  --good-swprofile <good-id> --bad-swprofile <bad-id> \
  --benchmark <suite/name> --resultclass <metric> \
  --resultstore <url> --context <output.yaml>
```

Example

```
fastpath bisect start \
  --host test-server --user root --keyfile ~/.ssh/id_rsa \
  --sut "Ampere Altra Max" \
  --good-swprofile "6.8.0-baseline" --bad-swprofile "6.9.0-regression" \
  --benchmark "sysbench/thread" --resultclass "sysbenchthread-110" \
  --context ./bisection_context.yaml
```

This validates baseline results exist, profiles match (except `kernel_git_sha`), and SUT is single-node. Then creates a temporary result store with baseline copies and generates `bisection_context.yaml`.

Note: Result Store Isolation: Original result store remains read-only. A temporary store holds baseline copies plus new bisection results.

To configure additional sampling when the initial 1/1/1 run is ambiguous, use `--warmups`, `--repeats`, and `--sessions`. These are stored in the plan but only used by bisect for the followup pass if needed.

Generated Context File:

The `bisection_context.yaml` contains:

- Test plan (SUT connection, benchmark config, shared profile fields)
- Baseline profile names and kernel git SHAs
- Resultclass for performance evaluation
- Result store paths (original and temporary)
- Benchmark run counts for any follow-up collection

```yaml
plan:
  sut:
    name: "Ampere Altra Max"
    connection:
      method: SSH
      params: {...}
  swprofiles:
    - cmdline: [...]
      sysctl: []
      bootscript: []
  benchmarks:
    - suite: sysbench
      name: thread
      ...
defaults:
  benchmark:
    warmups: 1   # used only if initial 1/1/1 run is ambiguous
    repeats: 2
    sessions: 2
good-swprofile: "6.8.0-baseline"
good_sha: "a1b2c3d4e5f6"
bad-swprofile: "6.9.0-regression"
bad_sha: "f6e5d4c3b2a1"
resultclass: "sysbenchthread-110"
resultstore: "mysql://..."
output-resultstore: "/tmp/bisect-resultstore-abc123/"
```

Swprofile Naming:

During bisection, swprofiles are named as follows:

- **Baseline swprofiles**: Retain their original names from the resultstore (e.g., `6.8.0-baseline`, `6.9.0-regression`)
- **Test swprofiles**: Initially named `bisect-<sha>` where `<sha>` is the first 12 characters of the kernel git SHA being tested (e.g., `bisect-a1b2c3d4e5f6`). After evaluation, they are relabeled to `<iteration>-<verdict>-<sha>` format (e.g., `01-b-a1b2c3d4` for iteration 1, bad verdict, 8-char SHA). Verdict codes: `g` (good), `b` (bad), `s` (skip), `e` (error).

This naming scheme makes it easy to identify bisection runs in the result store, track which kernel commit each test corresponds to, and see the outcome at a glance.

Run Bisection

Run automated bisection in your kernel source repository:

```
cd /path/to/kernel/source

# Extract SHAs and start git bisect
export GOOD_SHA=$(python3 -c "import yaml; print(yaml.safe_load(open('bisection_context.
↪yaml'))['good_sha'])")
export BAD_SHA=$(python3 -c "import yaml; print(yaml.safe_load(open('bisection_context.
↪yaml'))['bad_sha'])")

git bisect start
git bisect good $GOOD_SHA
git bisect bad $BAD_SHA

# Run automated bisection
git bisect run /path/to/fastpath/scripts/execute_bisection.sh \
  /path/to/bisection_context.yaml
```

The bisection script will:

1. Build the kernel for each commit tested
2. Create a unique swprofile named `bisect-<sha>` (first 12 chars of SHA)
3. Execute the benchmark on the SUT
4. Compare results against good/bad baselines using confidence intervals
5. Report to git bisect: GOOD (0), BAD (1), SKIP (125), or ERROR (128)

Git will automatically test commits until it identifies the first bad commit.

For more control over each bisection step:

```
# Manual bisection loop
cd /path/to/kernel/source

# Extract SHAs and start git bisect
export GOOD_SHA=$(python3 -c "import yaml; print(yaml.safe_load(open('bisection_context.
↪yaml'))['good_sha'])")
export BAD_SHA=$(python3 -c "import yaml; print(yaml.safe_load(open('bisection_context.
↪yaml'))['bad_sha'])")
```

(continues on next page)

(continued from previous page)

```
git bisect start
git bisect good $GOOD_SHA
git bisect bad $BAD_SHA

# For each commit picked by git bisect, repeat until done:

# 1. Build the kernel
./scripts/build_local_kernel.sh
source ./scripts/.env

# 2. Test and evaluate
fastpath bisect run \
  --context bisection_context.yaml \
  --kernel $KERNEL_PATH \
  --modules $MODULES_PATH \
  --gitsha $GITSHA

# 3. Mark commit based on exit code (0=good, 1=bad, 125=skip)
git bisect good   # if exit code is 0
git bisect bad    # if exit code is 1
git bisect skip   # if exit code is 125

# Git bisect picks next commit and repeats until first bad commit found
```

Bisection Output

During each bisection step:

```
Building kernel for current commit...
Executing plan.yaml...
Result: REGRESSION detected for resultclass 'sysbenchthread-110'
        comparing 'bisect-a1b2c3d4e5f6' vs '6.8.0-baseline'.
fastpath bisect run exited with status 1
```

Final bisection result:

```
a1b2c3d4e5f6 is the first bad commit
commit a1b2c3d4e5f6
Author: Developer Name <dev@example.com>
Date:   Mon Nov 1 10:00:00 2025 +0000

    Subject line of the problematic commit
```

Understanding Results

Each commit is tested and classified by comparing results against good baseline:

- **GOOD (0)**: Performance matches or exceeds good baseline
- **BAD (1)**: Performance regression detected
- **SKIP (125)**: Overlapping confidence intervals
- **ERROR (128)**: Fatal error, abort bisection (environment/infrastructure failure)

Adaptive Testing: Tests with 1 sample initially (1/1/1). If inconclusive (gap <2× the 99% confidence interval width), runs additional sampling using the configured warmups, repeats, and sessions (defaults: 1/2/2, yielding 4 more samples for 5 total).

Limitations: Single-node SUTs only.

1.2.8 Plan Schema

Top-Level plan object

A plan describes the set of benchmarks which should be run against a set of swprofiles (which includes a specific kernel) on a single SUT. The following is the set of top-level keys that may be defined in a plan:

key	type	required	default	description
sut	dict	true	N/A	Encapsulates all required information about the system under test (SUT). See "sut object".
swprofiles	list	true	N/A	A list of swprofile objects, each of which describes a software profile that the SUT will be configured into, and for which the benchmarks will be run.
benchmarks	list	true	N/A	A list of benchmark objects, each describing a benchmark to run for each required software profile.
defaults	dict	false	default defaults object	Default values for various parameters. See "defaults object".

sut object

The sut dictionary describes The system under test (SUT):

key	type	required	default	description
sutclass	string	false	None	Name of sutclass to which the SUT belongs.
name	string	false	None	User-supplied friendly name to identify the system under test.
connection	dict	1 from 2	N/A	For a single-node SUT, describes how to connect to the node. See "node.connection object". Exactly one of "connection" and "nodes" is required.
nodes	list	1 from 2	N/A	A list of nodes objects, each of which describes a node (machine/device) from which the SUT is composed. Exactly one of "connection" and "nodes" is required.

node object

A node is a machine/device that runs a single instance of Linux. A SUT is composed of one or mode nodes. It contains the following keys:

key	type	required	default	description
name	string	false	None	User-supplied friendly name to identify the node. Must be unique amongst nodes.
connection	dict	true	N/A	Describes how to connect to the node. See "node.connection object".

node.connection object

The connection dictionary describes how to connect to the node that the benchmarks will run on. It contains the following keys:

key	type	required	default	description
method	enum	true	N/A	Method used to connect to the node. For now, only "SSH" is supported. In future, "LAVA" will be added.
params	dict	true	N/A	Method-specific dictionary of parameters. See "SSH-params" below.

node.connection.SSH-params object

The SSH-params dictionary describes how to connect to a node using the "SSH" method. It contains the following keys:

key	type	required	default	description
host	string	true	N/A	Host name or IP address of the connection, or name of Host in SSH config file.
user	string	false	None	Login user for the remote connection. When None, SSH uses its default configured user, which may be specified in the SSH config file if host is the name of a Host in the SSH config file.
port	int	false	None	Remote port to connect to. When None, SSH uses its default configured port, which may be specified in the SSH config file if host is the name of a Host in the SSH config file.
keyfile	string	false	None	Path of private key to use for connection. When None, SSH uses its default configured private key(s).

swprofile object

The top-level swprofiles key maps a list of swprofile dictionaries. Each swprofile represents a software profile that the benchmarks are run against. It contains the following keys:

key	type	required	default	description
name	string	false	None	User-supplied friendly name to identify the software profile.
pkgtype	enum	false	Inferred	Describes what "kernel" parameter points to. Either "RAW" (if a raw kernel Image) or "DEB" (if a Debian package). If omitted, value is inferred; "DEB" if kernel string ends with ".deb", or "RAW" otherwise.
kernel	string	true	N/A	The kernel package to be used by the SUT. Installation, reboot and uninstallation is managed by the automatically. May be a filesystem path or a URL.
modules	string	false	None	When pkgtype is RAW, modules to install, provided as tarball. Must be None for other pkgtypes. May be a filesystem path or a URL.
cmdline	list	false	[]	List of additional command line options to be appended to the kernel's command line. The list is joined with a space (" ") separator.
sysctl	list	false	[]	List of sysctls to be applied persistently prior to reboot. Then reverted after reboot to remove persistence.
bootscript	list	false	[]	Bash commands to be executed after reboot with the correct kernel.
gitsha	string	false	None	Full, 40 character Git SHA for the revision of the source code used to build the provided kernel, or None if not known.

benchmark object

The top-level benchmarks key maps a list of benchmark dictionaries. Each benchmark represents a benchmark that is run for each software profile on the sut. It contains the following keys:

key	type	required	default	description
include	string	false	N/A	Path to benchmark fragment yaml, within a benchmark library. See Benchmark Library.
suite	string	false	unknown	Benchmark suite name. Passed when invoking container image.
name	string	true	N/A	Benchmark name. Passed when invoking container image.
type	string	false	unknown	Descriptive type label for the benchmark. E.g. cpu, memory, io, system, etc.
params	dict	false	{}	Dictionary of parameters specific to the benchmark. All keys and values must be strings. Passed when invoking container image.
image	string	true	N/A	Container image to pull to the SUT and invoke in order to run the benchmark. See Container Interface.
repeats	int	false	defaults.benchmark.repeats	Number of times to repeat the benchmark per boot session.
sessions	int	false	defaults.benchmark.sessions	Number of times to reboot the SUT to repeat the benchmark.
warmups	int	false	defaults.benchmark.warmups	Number of times to run the benchmark at the start of a boot session to warm up the system before the real repeats are executed.
timeout	string	false	defaults.benchmark.timeout	Timeout after which to assume the benchmark has hung. Provided as a string with format "<integer><suffix>" where the suffix is 's' (seconds), 'm' (minutes), 'h' (hours) or 'd' days.
roles	list	false	[executer]	List of roles that the benchmark implements. When multiple roles are defined, each role is executed in parallel, possibly on different nodes. If not specified, the benchmark is assumed to have a single role.
rolemap	dict	false	{r: 0 for r in roles}	Dictionary mapping roles to nodes, where key is the role and value represents the node, either an integer index into sut.nodes or a node name if a string. Multiple roles may be mapped to the same node. Any unmapped roles default to sut.nodes[0].

defaults object

A dictionary that holds various default values. It contains the following keys:

key	type	required	default	description
benchmark	dict	false	default defaults.benchmark	Default values relating to the benchmark object.

defaults.benchmark object

A dictionary that holds various default values. It contains the following keys:

key	type	required	default	description
repeats	int	false	3	Default number of times to repeat a benchmark per boot session.
sessions	int	false	1	Number of times to reboot the SUT to repeat a benchmark.
warmups	int	false	1	Number of times to run the benchmark at the start of a boot session to warm up the system before the real repeats are executed.
timeout	string	false	1h	Timeout after which to assume the benchmark has hung. Provided as a string with format "<integer><suffix>" where the suffix is 's' (seconds), 'm' (minutes), 'h' (hours) or 'd' days.

1.2.9 Benchmark Library

While the only required keys for a benchmark object are name and image, in practice a benchmark will usually have a number of parameters which can be set to vary its behaviour. It would be cumbersome to have to enter all these details for every benchmark in every plan, so to simplify this, a benchmark can be specified in its own yaml file fragment. And those yaml file fragments can be stored in a benchmark library, a well-known directory on the filesystem. Let's say we have this simple plan, with a single benchmark:

```
sut:
  connection:
    method: SSH
    params:
      host: my-server
swprofiles:
  - kernel: /path/to/Image.gz
    modules: /path/to/modules.tar.xz
benchmarks:
  - suite: my-suite
    name: my-benchmark
    type: system
    params:
      cpus: 4
      ram: 4G
    image: docker.io/fastpath/benchmarks/my-benchmark:latest
```

Instead of defining the benchmark directly, we could define it in a benchmark fragment, let's assume its called *my-suite/my-benchmark-default.yaml* and lives in the benchmark library:

```yaml
suite: my-suite
name: my-benchmark
type: system
params:
  cpus: 4
  ram: 4G
image: docker.io/fastpath/benchmarks/my-benchmark:latest
```

Now the plan do simply include the benchmark:

```yaml
sut:
  connection:
    method: SSH
    params:
      host: my-server
swprofiles:
  - kernel: /path/to/Image.gz
    modules: /path/to/modules.tar.xz
benchmarks:
  - include: my-suite/my-benchmark-default.yaml
```

We could even define a variant benchmark that inherits from the default one and overrides some parameters. Let's assume this is called *my-suite/my-benchmark-single.yaml*:

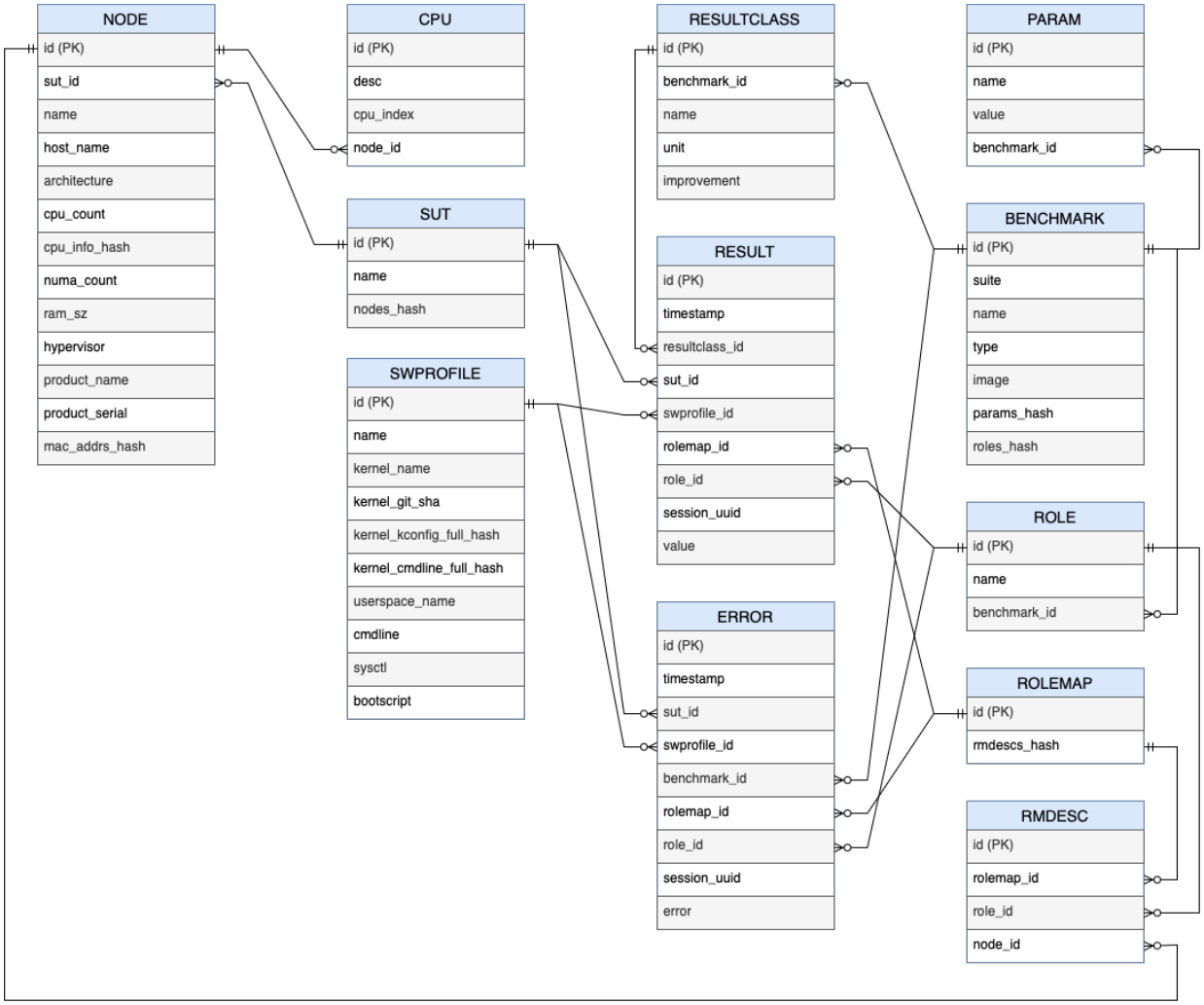
```yaml
include: my-suite/my-benchmark-default.yaml
params:
  cpus: 1
```

Now the plan can include both benchmarks:

```yaml
sut:
  connection:
    method: SSH
    params:
      host: my-server
swprofiles:
  - kernel: /path/to/Image.gz
    modules: /path/to/modules.tar.xz
benchmarks:
  - include: my-suite/my-benchmark-default.yaml
  - include: my-suite/my-benchmark-single.yaml
```

1.2.10 Resultstore Schema

Entity Relationships



BENCHMARK Table

One entity per Benchmark configuration. Describes a benchmark including its parameters.

key	type	description
id	bigserial (PK)	Primary key. Auto-incrementing integer
suite	text	Benchmark suite as it appears in the plan
name	text	Benchmark name as it appears in the plan
type	text	Benchmark type as it appears in the plan
image	text	Benchmark container image identifier as it appears in the plan
params_hash	char(64)	SHA256 hash of the params dictionary as they appear in the plan
roles_hash	char(64)	SHA256 hash of the roles dictionary as they appear in the plan

CPU Table

One entity per CPU contained within a node. Describes the set of CPUs that a node contains.

key	type	description
id	bigserial (PK)	Primary key. Auto-incrementing integer
desc	text	arch-specific descriptor, encoded in JSON
cpu_index	smallint	Logical CPU index
node_id	bigint (FK)	Node ID relates CPUs to node

ERROR Table

One entity per error that occured during running the benchmark. A single benchmark invocation generates 0 or 1 error entities. If an error is generated, no results are generated for the benchmark. Examples of possible errors include the container side rejecting benchmark parameters, or the benchmark timing out.

key	type	description
id	bigserial (PK)	Primary key. Auto-incrementing integer
timestamp	timestamp	Approximate time at which error was emitted. Errors are stamped when passed back to host and there is latency between generating the error on the SUT and passing it to the host
sut_id	bigint (FK)	SUT ID relates error to SUT
swprofile_id	bigint (FK)	swprofile ID relates error to swprofile
benchmark_id	bigint (FK)	Benchmark ID relates error to Benchmark
rolemap_id	bigint (FK)	Rolemap ID relates error to rolemap
role_id	bigint (FK)	Role ID relates error to role that generated the error
session_uuid	uuid	Universally Unique ID representing the boot session on the SUT that the result was generated during. Conceptually, a new UUID is generated each time the SUT is rebooted. Used to group results and errors from the same session
error	int	Error code

NODE Table

One entity per node. Describes a unique node within a SUT. A node is a machine or device that runs a single instance of Linux. A SUT may be composed of 1 or more nodes.

key	type	description
id	bigserial (PK)	Primary key. Auto-incrementing integer
sut_id	bigint (FK)	SUT that the node belongs to
name	text	Node friendly name as it appears in the plan
host_name	text	result of *uname -n*
architecture	text	result of *uname -m*
cpu_count	smallint	Number of CPUs in the system
cpu_info_hash	char(64)	SHA256 hash of all the CPU info structures
numa_count	smallint	Number of NUMA nodes in the system
ram_sz	bigint	Total RAM in the system in bytes
hypervisor	text	The name of the hypervisor if running in a virtual machine. (e.g. 'kvm', 'hyper-v' etc.) else empty
product_name	text	The DMI product_name, if available, else empty
product_serial	text	The DMI product_serial, if available, else empty
mac_addrs_hash	char(64)	SHA256 hash of the sorted list of permanent MAC addresses from physical devices attached to the node

PARAM Table

One entity per param specified for a benchmark. Describes the set of parameters that a benchmark is configured with.

key	type	description
id	bigserial (PK)	Primary key. Auto-incrementing integer
name	text	Param name as it appears in the plan
value	text	Param value as it appears in the plan
benchmark_id	bigint (FK)	Benchmark ID relates Param to Benchmark

RESULT Table

One entity per result. A single benchmark invocation returns 1 or more results.

key	type	description
id	bigserial (PK)	Primary key. Auto-incrementing integer
timestamp	timestamp	Approximate time at which result was collected. Results are stamped when passed back to host and there is latency between generating the result on the SUT and passing it to the host
resultclass_id	bigint (FK)	ResultClass ID relates Result to ResultClass
sut_id	bigint (FK)	SUT ID relates Result to SUT
swprofile_id	bigint (FK)	swprofile ID relates Result to swprofile
rolemap_id	bigint (FK)	Rolemap ID relates result to rolemap
role_id	bigint (FK)	Role ID relates result to role that generated the result
session_uuid	uuid	Universally Unique ID representing the boot session on the SUT that the result was generated during. Conceptually, a new UUID is generated each time the SUT is rebooted. Used to group results and errors from the same session
value	double	Numerical result value

RESULTCLASS Table

One entity per ResultClass. Every Result belongs to a single ResultClass.

key	type	description
id	bigserial (PK)	Primary key. Auto-incrementing integer
benchmark_id	bigint (FK)	Benchmark ID relates ResultClass to Benchmark
name	text	Result name (e.g. duration or memory)
unit	text	Unit of the result value (e.g. seconds, ops/sec, etc)
improvement	enum	Either 'bigger' or 'smaller': Determines whether a bigger or smaller value represents an improvement in performance

ROLE Table

Describes a role that a benchmark performs.

key	type	description
id	bigserial (PK)	Primary key. Auto-incrementing integer
name	text	Role name as it appears in the plan
benchmark_id	bigint (FK)	Benchmark ID relates role to benchmark

ROLEMAP Table

Describes the mapping of a benchmark's roles to the nodes they execute on. Composed of 1 or more rmdescs.

key	type	description
id	bigserial (PK)	Primary key. Auto-incrementing integer
rmdescs_hash	char(64)	SHA256 hash of all the rmdesc structures

RMDESC Table

Describes the mapping of a single benchmark role to the node it executes on.

key	type	description
id	bigserial (PK)	Primary key. Auto-incrementing integer
rolemap_id	bigint (FK)	Rolemap that the rmdesc belongs to
role_id	bigint (FK)	Role that is being mapped
node_id	bigint (FK)	Node that the role is mapped to

SUT Table

One entity per SUT. Describes a unique SUT independent of its SW profile. A SUT is composed of 1 of more nodes.

key	type	description
id	bigserial (PK)	Primary key. Auto-incrementing integer
name	text	SUT friendly name as it appears in the plan
nodes_hash	char(64)	SHA256 hash of all the node structures
sutclass_id	bigint (FK)	sutclass that the SUT belongs to (may be NULL)

SUTCLASS Table

One entity per SUTCLASS. Describes the class of a SUT. Multiple SUT instances may exist for the same class.

key	type	description
id	bigserial (PK)	Primary key. Auto-incrementing integer
name	text	sutclass name

SWPROFILE Table

One entity per software profile. Describes a software configuration including the kernel under test.

key	type	description
id	bigserial (PK)	Primary key. Auto-incrementing integer
name	text	swprofile friendly name as it appears in the plan
kernel_name	text	Output from *uname -r*
kernel_git_sha	char(40)	Git SHA for kernel source (if available)
kernel_kconfig_full_hash	char(64)	SHA256 hash of kernel config (e.g. from /boot/config-$(uname -r), or /proc/config.gz) (if available)
kernel_cmdline_full_hash	char(64)	SHA256 hash of /proc/cmdline
userspace_name	text	Output from *lsb_release -d*
cmdline	text	Alphabetically sorted command line parameters, as specified in plan, as a newline separated string
sysctl	text	Alphabetically sorted applied sysctl settings, as specified in plan, as a newline separated string
bootscript	text	Bootscript passed as it appears in plan

1.2.11 Using the Scheduler

Introduction

The scheduler is the Fastpath service that manages access to SUT classes and dispatches queued work onto concrete SUTs as they become available. It supports two main workflows:

- **Plan execution**: submit a plan that targets a SUT class and let the scheduler split and run it across all available SUTs.
- **Manual leasing**: acquire a SUT from a SUT class for interactive use, then release it when finished.

This guide walks through setting up a small demo scheduler instance, saving the scheduler URL and user token in Fastpath preferences, and then using the scheduler CLI verbs to submit and manage plans and leases.

Note: For low-level integration details and the REST interface itself, see *Scheduler REST API*.

Start a Demo Scheduler

For a simple demo, use the built-in `static` SUT provider. Create a file called `sutclasses.yaml`:

```
- sutclass: demo
  type: static
  params:
    suts:
      - name: demo-01
        connection:
          method: SSH
          params:
            host: demo-01
```

(continues on next page)

(continued from previous page)

```
            user: fpuser
            keyfile: /path/to/id_ed25519
      - name: demo-02
        connection:
          method: SSH
          params:
            host: demo-02
            user: fpuser
            keyfile: /path/to/id_ed25519
```

This defines a SUT class called `demo` containing two static SUTs.

Start the scheduler:

```
fastpath sched run \
  --host 127.0.0.1 \
  --port 5000 \
  --database sqlite:///scheduler.db \
  --storage ./sched_storage \
  --sutclasses ./sutclasses.yaml
```

If you want scheduled plans to automatically publish into a resultstore, also provide `--resultstore`:

```
fastpath sched run \
  --host 127.0.0.1 \
  --port 5000 \
  --database sqlite:///scheduler.db \
  --storage ./sched_storage \
  --sutclasses ./sutclasses.yaml \
  --resultstore sqlite:///scheduler-results.db
```

On first start the scheduler creates a default admin user and prints its token. The admin user has all capabilities, including `admin`:

```
****
Created default admin user 'admin' with token: <token>
****
```

Keep that token. It is needed to create additional scheduler users.

Save the Scheduler URL and Token

The scheduler CLI verbs take `--url` and `--token` options, but it is much more convenient to store them as Fastpath preferences:

```
fastpath preference set --category sched --name url http://127.0.0.1:5000
fastpath preference set --category sched --name token <admin-token>
```

Now scheduler commands can be run without repeating either argument; if –url or –token is omitted, they default to the values saved as preferences. For example:

```
fastpath sched sutclass-list
```

You can inspect the saved values with:

```
fastpath preference get --category sched --name url
fastpath preference get --category sched --name token
```

Create Scheduler Users

Create users with the `sched user-create` verb. Capabilities are passed as a comma-separated list of lower-case names.

For example, create a user that can execute plans and query scheduler state:

```
fastpath sched user-create alice --capabilities exec,query
```

Or create a user that can manually lease SUTs and query scheduler state:

```
fastpath sched user-create bob --capabilities lease,query
```

Each command prints the created user's token. If you want to work as that user, update the saved `sched.token` preference to the new token (or pass it with –token).

Inspect the Scheduler State

List known SUT classes:

```
fastpath sched sutclass-list
```

This shows the SUT class id, name, whether it is enabled, current usage, and creation time.

To report usage over a specific time window, provide `--from` and/or `--to`:

```
fastpath sched sutclass-list --from 2026-03-01
fastpath sched sutclass-list --from 2026-03-01 --to 2026-04-01
```

Submit a Plan

Plans submitted to the scheduler must target a SUT class rather than a specific SUT. For example:

```yaml
sut:
  sutclass: demo
swprofiles:
  - name: baseline
    kernel: /path/to/Image
    modules: /path/to/modules.tar.xz
benchmarks:
  - include: speedometer/v2.1.yaml
  - include: mmtests/kernbench.yaml
defaults:
  benchmark:
    warmups: 1
    repeats: 3
    sessions: 3
    timeout: 1h
```

Submit it to the scheduler:

```
fastpath sched plan-exec demo-plan.yaml
```

Optionally set a priority:

```
fastpath sched plan-exec --priority 8 demo-plan.yaml
```

If the scheduler was started with `--resultstore`, scheduled plan results are published there by default. To queue a plan without publishing its results:

```
fastpath sched plan-exec --no-publish demo-plan.yaml
```

If the plan references local files (e.g. kernel or modules), the scheduler client automatically uploads them along with the plan payload.

Track and Manage Plans

List all plans:

```
fastpath sched plan-list --all-users
```

Inspect specific plan by id:

```
fastpath sched plan-list 12
```

The plan listing shows:

- the owning user
- the target SUT class
- plan priority
- progress as `completed_jobs/total_jobs`
- plan status
- queued, started, and completed timestamps

Filter plan listings:

```
fastpath sched plan-list --states running
fastpath sched plan-list --users alice
fastpath sched plan-list --all-users --all-states
```

By default, `plan-list` filters to the authenticated user and to plans in the `queued` or `running` states. Use `--all-users` and `--all-states` to remove those default filters.

Update the priority of a queued or active plan:

```
fastpath sched plan-update 12 --priority 3
```

Cancel a plan:

```
fastpath sched plan-cancel 12
```

Cancellation is best-effort. Queued child jobs are cancelled immediately. Already-running child jobs are allowed to finish naturally.

Acquire a Lease

To request a manual lease on a SUT from a SUT class:

```
fastpath sched lease-acquire demo
```

To wait until the lease becomes active and a concrete SUT has been assigned:

```
fastpath sched lease-acquire demo --wait
```

The command prints the lease details, including the assigned SUT once it is available.

Manage Leases

List all leases:

```
fastpath sched lease-list --all-users
```

Inspect a specific lease by id:

```
fastpath sched lease-list 7
```

Filter lease listings:

```
fastpath sched lease-list --states acquired
fastpath sched lease-list --users bob
fastpath sched lease-list --all-users --all-states
```

By default, `lease-list` filters to the authenticated user and to leases in the `queued` or `acquired` states. Use `--all-users` and `--all-states` to remove those default filters.

Release a lease when finished:

```
fastpath sched lease-release 7
```

1.3 Developer Guide

1.3.1 Integrate Benchmark to Fastpath

This guide explains how to create and integrate a benchmark with Fastpath. A benchmark integration typically consists of:

1. A **benchmark YAML fragment** under `fastpath/benchmarks/<suite>/`, defines the benchmark identity & configuration
2. A **container Dockerfile** under `fastpath/containers/<suite>/`, defines the runtime environment
3. A **container entrypoint script** (`exec.py`) under `fastpath/containers/<suite>/`, for benchmark execution & result processing

Once created, the container image must be built and referred to in the benchmark YAML fragment. The benchmark then becomes available for execution via `fastpath plan exec`.

See *Define & Execute a Plan* for a full description of the plan creation & execution.

High-level Concepts

Fastpath executes benchmarks over Docker containers on the SUT node(s). Each benchmark container will receive the benchmark configuration (`benchmark.yaml`) generated by Fastpath in a shared directory (`/fastpath-share`). This `benchmark.yaml` is constructed from the benchmark object in the plan, which may either be defined inline in the plan or (more commonly) *included* from a benchmark YAML fragment stored under `fastpath/benchmarks/<suite>/` (see *Plan Schema*, Benchmark Library section). The container entrypoint (`exec.py`) then:

- reads the benchmark params from `benchmark.yaml`
- validates those input parameters
- initiates the benchmark workload (directly or through helper scripts)
- writes `results.csv` back into `/fastpath-share` for Fastpath to collect

A benchmark can be:

- **single role** (Fastpath assigns default `executer` role), or
- **multi role** (multiple roles such as `server`, `client`, `monitor`).

For multi-role benchmarks, Fastpath starts a container per role on the node(s) (based on single-node/multi-node SUT selection & rolemap defined in plan) and runs them in parallel.

1. Add Benchmark YAML

Location
`fastpath/benchmarks/<suite>/<benchmark-name>.yaml`

Purpose
Defines a benchmark as a reusable **benchmark fragment** in the Fastpath benchmark library. It declares the benchmark identity and how Fastpath should execute it. Plans can reference this fragment (via `include`) from the Benchmark Library instead of repeating the benchmark definition inline, while still allowing per-plan overrides (for example, overriding `warmups`, `repeats` or other fields).

Typical fields

- `suite`: Benchmark suite name (folder name).
- `name`: Benchmark/workload name within the suite.
- `type`: Category of benchmark (e.g. scheduler, network, storage).
- `image`: Full image path for the docker container (registry or local).
- `params`: Configuration keys supported by the benchmark (optional).
- `roles`: Role names for multi-role benchmarks where each role performs a different task (optional).

Example

```
suite: repro-collection
name: mysql-workload
type: database
image: registry.gitlab.arm.com/tooling/fastpath/containers/repro_collection:v2.1
params:
  workload: mysql
  sut_nr_cpus: 16
```

If your benchmark requires multiple roles, include `roles` (example):

```
roles:
  - server
  - client
```

Fastpath will start one container per role and run them concurrently on selected SUT node(s) as per the rolemap (default or could be defined in the plan).

Sample plan snippet for multi-role multi-node benchmark with rolemap defined:

```
benchmark:
- include: repro-collection/mysql-workload.yaml
  rolemap:
    server: 0   #0 is the first node in the SUT definition
    client: 1   #1 is the second node in the SUT definition
```

2. Add Container Dockerfile

Location
`fastpath/containers/<suite>/Dockerfile`

Purpose
Defines the runtime environment for the benchmark. The resulting image must be built and deployed based on your deployment choice (registry or local). The deployed image path will be used in the benchmark YAML.

Typical responsibilities

- Install system dependencies (`apt-get install ...`).
- Install Python and set up a Python venv for pip packages.
- Install common Fastpath Python dependencies from `fastpath/requirements.txt`.
- Install benchmark-specific dependencies.
- Fetch the benchmark workload code (e.g. `git clone` and build).
- Copy any helper files/scripts needed by the benchmark to container's shared folder.
- Copy the Fastpath container entrypoint `exec.py` to shared folder.
- Set the entrypoint/command to run `exec.py`.

Example structure

```
FROM registry.gitlab.arm.com/tooling/fastpath/containers/base:latest

ENV DEBIAN_FRONTEND=noninteractive

RUN apt-get update && \
    apt-get install --assume-yes --no-install-recommends \
      python3 python3-pip python3-venv python3-dev \
      build-essential pkg-config

RUN python3 -m venv /pyvenv
ENV PATH="/pyvenv/bin:${PATH}"

COPY fastpath/requirements.txt /tmp/requirements.txt
RUN pip3 install -r /tmp/requirements.txt && rm -rf /tmp/requirements.txt
```

(continues on next page)

(continued from previous page)

```
# benchmark-specific dependencies
RUN apt-get update && \
apt-get install --assume-yes --no-install-recommends git

RUN mkdir /fastpath

# benchmark workload source (example)
RUN git clone https://example.org/bench.git /fastpath/bench && \
cd /fastpath/bench && make

ARG NAME
# Copy & compile helper scripts (example)
COPY containers/${NAME}/helper.c /tmp/helper.c
RUN gcc /tmp/helper.c -o /fastpath/bench
RUN rm -rf /tmp/helper.c

# Setup the entrypoint.
COPY containers/${NAME}/exec.py /fastpath/.
RUN chmod +x /fastpath/exec.py
CMD /fastpath/exec.py
```

Versioning
: The image tag used in the benchmark YAML (for example `:v1.0`) should match the container version you publish locally or in the registry. Update the tag whenever you change benchmark container behavior.

3. Implement exec.py

Location
: `fastpath/containers/<suite>/exec.py`

Purpose
: This script is the container entrypoint and the primary integration between Fastpath and the benchmark. It must initiate the benchmark workloads, parse and write results in Fastpath's Results schema to `/fastpath-share/results.csv`.

Container contract

- Input file: `/fastpath-share/benchmark.yaml`
- Output directory (recommended for logs): `/fastpath-share/output/`
- Output file: `/fastpath-share/results.csv`

Typical structure for exec.py

1. Read `benchmark.yaml`.
2. Validate suite/name and validate required parameters from the yaml.
3. Expand parameters into one or more test descriptors (if the benchmark runs a sweep).
4. Execute the benchmark command(s) (or invoke helper scripts) based on `rolemap`.
5. Parse output and convert to Fastpath results format (one resultclass per metric).
6. Catch and raise exceptions (optional).

7. Log the script output (debug logs) to a log file such as `/fastpath-share/output/exec.log` (optional).
8. Write results to `results.csv`.

Important requirements

- Always write `results.csv` (even on failure). On failure, write a row with an error code so Fastpath can report the failure consistently. (Example : `ERR_INVAL_BENCHMARK_FORMAT=2`)
- Prefer writing benchmark logs into `/fastpath-share/output` to make them available in Fastpath collected artifacts.

Minimal results format

Fastpath expects rows with fields like:

- `name` (taken as `resultclass`)
- `unit`
- `improvement` (`bigger` or `smaller`)
- `value`
- `error` (0 for success; non-zero for benchmark errors)

Helper files (optional)

You may add extra files under `fastpath/containers/<suite>/` (shell scripts, patch files, configs). The Dockerfile can copy them into the image, and `exec.py` can invoke them.

4. Build & Publish Container Image

Once all benchmark files (YAML, Dockerfile, exec.py) are ready, build and deploy the container image.

For Arm internal users (Fastpath tool CI)

1. Trigger the **Build Container** pipeline.
2. Provide the container folder name (the `<suite>` directory under `fastpath/containers/`).
3. Provide the container version/tag.
4. The pipeline produces and pushes an image to the GitLab Container Registry.

For external users (manual Docker build)

You can build the container image manually using standard Docker commands:

- **Build the container image:**

```
docker build \
  --build-arg NAME=<suite> \
  -t <image-name>:<version> \
  -f fastpath/containers/<suite>/Dockerfile \
  .
```

Replace `<suite>` with your benchmark suite name (e.g., `repro_collection`), `<image-name>` with your desired image name, and `<version>` with the version tag (e.g., `v1.0`).

- **Publish the container image:**
  - **If you have a registry**: Push the image and update the benchmark YAML.

```
docker tag <image-name>:<version> <registry>/<image-name>:<version>
docker push <registry>/<image-name>:<version>
```

Update the benchmark YAML `image: ...` field:

```
image: <registry>/<image-name>:<version>
```

– **If you do not have a registry**: Use the local image name in the benchmark YAML. In this case, ensure that the image is available on the SUT before execution.

```
image: <image-name>:<version>
```

5. Execute Benchmark

Benchmarks are executed through a plan (Refer *Define & Execute a Plan*).

```
fastpath plan exec --output <output-dir> full-plan.yaml
```

The final plan must include:

- the benchmark YAML file path
- role map of the benchmark (if applicable)
- any benchmark parameters you want to override from the benchmark YAML

Verify that the benchmark runs successfully and that results are visible.

```
fastpath results show --benchmark <suite/name> <output-dir>/results
```

Note: For role-based benchmarks (server/client), ensure final plan maps roles to the correct nodes (if multi-node SUT selected) and that `exec.py` implements role-specific behavior (for example, starting the server process on one node and the client process on another). See *Plan Schema* `benchmark object` section for details about role mapping.

Checklist for New Benchmarks

- [ ] Add benchmark YAML under `fastpath/benchmarks/<suite>/`.
- [ ] Add Dockerfile under `fastpath/containers/<suite>/`.
- [ ] Add exec.py under `fastpath/containers/<suite>/` with results.csv output.
- [ ] Ensure logs are written under `/fastpath-share/output`.
- [ ] Build and publish Docker image locally or to a registry.
- [ ] Update benchmark YAML image tag to the published version.
- [ ] Validate benchmark execution via `fastpath plan exec` and confirm results appear in the output/resultstore.

1.3.2 Compile Documentation Locally

To build the docs locally, the following packages need to be installed on the host:

```
sudo apt-get install python3-pip
pip3 install -r documentation/requirements.txt
```

To build and generate the documentation in html format, run:

```
sphinx-build -b html -a -W documentation public
```

To render and explore the documentation, simply open *public/index.html* in a web browser.

1.3.3 Scheduler REST API

The scheduler is a long-running service that manages access to SUT classes and dispatches queued work onto concrete SUT instances as they become available. It provides a REST API for:

- creating scheduler users and API tokens
- submitting plans for execution, tracking their progress, reprioritizing and cancelling them
- acquiring and releasing manual leases on SUTs
- querying SUT class metadata and usage

The main use cases are:

- **Plan execution**: submit a plan that targets a SUT class and let the scheduler split the plan and dispatch onto available SUTs in parallel.
- **SUT leasing**: request a lease on a SUT from a SUT class, wait for a SUT to become available, and release it when finished.
- **Operational reporting**: inspect leases, plans, and SUT class usage.

Note: Users are expected to interact with the scheduler via the CLI commands (see *Using the Scheduler*). Those commands wrap the API documented here. The API is documented to enable integration with other systems.

Authentication

All endpoints require a bearer token in the `Authorization` header:

```
Authorization: Bearer <token>
```

Tokens are created via `POST /users`.

Capabilities

The scheduler enforces capabilities on each endpoint:

- `CAP_ADMIN`: unrestricted administrative access
- `CAP_EXEC`: submit and modify plans
- `CAP_LEASE`: acquire and release manual leases
- `CAP_QUERY`: query scheduler state

An admin token bypasses ownership checks. Non-admin users are restricted to operating only on their own leases and plans.

Content Types

Most requests and all responses use JSON.

Plan submission is the exception: `POST /plans` accepts either JSON or `multipart/form-data`. Multipart must be used when the plan references local files that need to be uploaded alongside the plan payload (e.g. kernel images).

Endpoints

Users

`POST /users`
Create a new scheduler user. Requires `CAP_ADMIN`.

Request body:

```
{
  "name": "alice",
  "capabilities": "lease,query,exec"
}
```

Response `201`:

```
{
  "status": "success",
  "user_id": 3,
  "token": "<bearer-token>"
}
```

Plans

`POST /plans`
Submit a plan for scheduler execution. Requires `CAP_EXEC`.

The scheduler validates the plan, stores any uploaded input files under the plan storage directory, stores a normalized `plan.yaml`, splits the plan into plan jobs, and queues those jobs for dispatch. If the scheduler was started with a configured resultstore, plans publish results there by default; this can be disabled per-submission with `publish`.

JSON request body:

```
{
  "plan": { ... plan object ... },
  "priority": 5,
  "publish": true
}
```

Multipart request:

- form field `payload` containing JSON with `plan`, `priority`, and optional `publish`
- repeated file field `files` for any local files referenced by the plan

Response `201`:

```
{
  "status": "success",
  "plan_id": 11
}
```

`GET /plans`
List plans. Requires `CAP_QUERY`.

Optional query parameters to filter the response. If not provided, acts as if all values were provided:

- `states=queued,running,success,failed,cancelled`
- `users=alice,bob`

The special user value `__current__` may be used to mean the authenticated user.

`GET /plans/<plan_id>`
Retrieve a single plan. Requires `CAP_QUERY`.

`PATCH /plans/<plan_id>`
Modify a plan. Requires `CAP_EXEC`. Non-admin users may only modify their own plans.

Supported fields:

- `priority`: reprioritize all non-completed child jobs
- `cancel`: boolean; when `true` requests plan cancellation

Either field may be provided on its own, or both together.

Request body:

```
{
  "priority": 8,
  "cancel": false
}
```

Response `200`:

```
{
  "status": "success"
}
```

Plan fields

`plan_id`
Plan id.

user
Object containing `id` and `name`.

sutclass
Object containing `id` and `name`.

priority
Plan priority.

state
Plan state. One of `queued`, `running`, `success`, `failed`, or `cancelled`.

total_jobs
Number of split plan jobs.

completed_jobs
Number of split plan jobs that completed successfully.

queued_at, **started_at**, **completed_at**
plan timestamps.

Leases

POST /leases
Queue a manual lease against a SUT class. Requires `CAP_LEASE`.

Request body:

```
{
  "sutclass": "myclass",
  "priority": 5
}
```

Response `201`:

```
{
  "status": "success",
  "lease_id": 7
}
```

PATCH /leases/<lease_id>
Release a lease. Requires `CAP_LEASE`. Non-admin users may only release their own leases.

Response `200`:

```
{
  "status": "success"
}
```

GET /leases
List leases. Requires `CAP_QUERY`.

Optional query parameters to filter the response. If not provided, acts as if all values were provided:

- `states=queued,acquired,released,failed,cancelled`
- `users=alice,bob`

The special user value `__current__` may be used to mean the authenticated user.

`GET /leases/<lease_id>`
Retrieve a single lease. Requires `CAP_QUERY`.

Lease fields

`lease_id`
Lease id.

`user`
Object containing `id` and `name`.

`job_id`
Backing job id.

`state`
Lease state. One of `queued`, `acquired`, `released`, `failed`, or `cancelled`.

`priority`
Lease priority.

`sutclass`
Object containing `id` and `name`.

`sut`
The acquired SUT description, or `null` if not yet acquired.

`queued_at`, `started_at`, `completed_at`
Timestamps for the backing job lifecycle.

SUT Classes

`GET /sutclasses`
List SUT classes known to the scheduler. Requires `CAP_QUERY`.

Optional query parameters:

- `from=<iso-datetime>`
- `to=<iso-datetime>`

When `from` and/or `to` are provided, the server calculates `usage` only within that time window. When neither is provided, `usage` is the scheduler's accumulated all-time usage counter.

SUT class fields

`id`
SUT class id.

`name`
SUT class name.

`enabled`
Boolean indicating whether the SUT class is enabled.

`usage`
Runtime usage in seconds.

`created_at`
Timestamp when the SUT class row was created.

Errors

Errors return JSON of the form `{"status": "<status>"}`.

The scheduler currently uses these status codes:

- `400 invalid`
- `400 resourcebusy`
- `401 unauthorized`
- `403 forbidden`
- `404 notfound`
- `409 exists`
- `413 toobig`
- `500 internalerror`

Examples

Create a scheduler user:

```
curl -X POST \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{"name":"alice","capabilities":"lease,query,exec"}' \
  http://localhost:5000/users
```

Acquire a lease:

```
curl -X POST \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{"sutclass":"myclass","priority":5}' \
  http://localhost:5000/leases
```

Submit a plan using multipart upload:

```
curl -X POST \
  -H "Authorization: Bearer $TOKEN" \
  -F 'payload={"plan": {...}, "priority": 5}' \
  -F 'files=@Image' \
  http://localhost:5000/plans
```

List SUT classes for a specific window:

```
curl -X GET \
  -H "Authorization: Bearer $TOKEN" \
  'http://localhost:5000/sutclasses?from=2026-03-01T00:00:00+00:00&to=2026-04-
↪01T00:00:00+00:00'
```

1.4 Regression Tracker

1.4.1 Introduction

Fastpath monitors the Linux kernel across a range of hardware platforms and workloads. When a statistically significant performance change is detected between two kernel versions, it is raised as a tracked regression entry and investigated. This page records all known findings together with their current status and, where available, links to upstream reports and fix commits.

Column descriptions:

- **ID** – Tracking identifier (e.g. `FP-619-02`).
- **Branch / Good / Bad** – The kernel tree and the two versions that bracket the regression.
- **SUT(s)** – System(s) Under Test on which the regression was observed.
- **Benchmark / Resultclass** – The specific workload and measured quantity.
- **Delta** – Measured change between the good and bad kernel versions.
- **Regressing Commit** – The first bad commit identified by bisection.
- **Status** – One of four states:
  - *Detected* – Regression observed; queued for investigation.
  - *Investigating* – Currently under active investigation.
  - *Reported* – Reported upstream; awaiting a fix.
  - *Fixed* – Fix merged upstream.
- **Report** – Link to the upstream mailing-list report or patch.
- **Fix Commit(s)** – Commit(s) that resolved the regression.

1.4.2 Regression History

ID	Branch	Good Kernel	Bad Kernel	SUT(s)	Benchmark / Resultclass	Delta	Regressing Commit	Status	Report	Fix Commit(s)
FP-614-01	mainline	6-13-0	6-14-0-rc4	aws-m7g.metal-01	mmtests/hackbench (process-pipes-30)	-10.69%	aaec5a95d596 *"pipe_read: don't wake up the writer if the pipe is still full"*	Fixed	Upstream report	
FP-615-01	mainline	6-15-0-rc1	6-15-0-rc2	aws-m7g.metal-01	pts/pgbench Scale: 100 Clients: 1000, Read Only (TPS)	-98.81%	7ab4f0e37a0f *"ACPI PPTT: Fix coding mistakes in a couple of sizeof() calls"*	Fixed	Upstream report	adfab6b39202 *"ACPI: PPTT: Fix processor subtable walk"*
FP-619-01	mainline	6-18-0	6-19-0-rc1	aws-m7g.metal-01	micromm/vmalloc fix_align_alloc_test: p:1, h:0, l:500000 (usec)	-40.57%	a06157804399 *"mm/vmalloc: request large order pages from buddy allocator"*	Reported	Upstream report	
FP-619-02	mainline	6-18-0	6-19-0-rc1	aws-m7g.metal-01 cesw-aarch64-ampereone-1s-a192-32x-02	schbench/thread-contention (-m 64 -t 4 -r 10 -s 1000), avg_rps (req/sec)	-17.41%	e837456fdca8 *"sched/fair: Reimplement NEXT_BUDDY to align with EEVDF goals"*	Fixed	Upstream report	4f70f106bca1 *"sched/fair: Disable scheduler feature NEXT_BUDDY"* 15257cc2f905 *"sched/fair: Revert force wakeup preemption" (merged in 6-19-0-rc7)*
FP-619-03	mainline	6-18-0	6-19-0-rc2	aws-m7g.metal-01	micromm/munmap p:1, d:10 (seconds)	-23.74%	7e44d00a13da *"memcg: use mod_node_page_state to update stats"*	Reported	Upstream report	
FP-700-01	mainline	6-19-0	7-0-0-rc1	aws-m7i.24xlarge-01 aws-m7g.metal-01 cesw-	micromm/vmalloc kvfree_rcu_2_arg_vmalloc_test: p:1, h:0, l:500000 (usec)	-9.26%	1e40 *"slab: add sheaves to most caches"*	Investigating		

1.5 License

The software is provided under the MIT license (below).

```
Copyright (c) 2024-2025 Arm Limited

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to
deal in the Software without restriction, including without limitation the
rights to use, copy, modify, merge, publish, distribute, sublicense, and/or
sell copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice (including the next
paragraph) shall be included in all copies or substantial portions of the
Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS
IN THE SOFTWARE.
```

1.5.1 SPDX Identifiers

Individual files contain the following tag instead of the full license text.

```
SPDX-License-Identifier: MIT
```

This enables machine processing of license information based on the SPDX License Identifiers that are here available: http://spdx.org/licenses/

1.5.2 Pre-Built Container Images

All Fastpath images are based on either Debian or Ubuntu. See the respective links for licensing information.

As with all Docker images, these likely also contain other software which may be under other licenses (such as Bash, etc from the base distribution, along with any direct or indirect dependencies of the primary software being contained). Additional license information can be found at /usr/share/doc/*/copyright within the image.

For images that contain benchmarking software, use of the benchmarking software is subject to any license it may provide. See Dockerfiles within containers directory for a list of such software added to each image.

As for any pre-built image usage, it is the image user's responsibility to ensure that any use of this image complies with any relevant licenses for all software contained within.