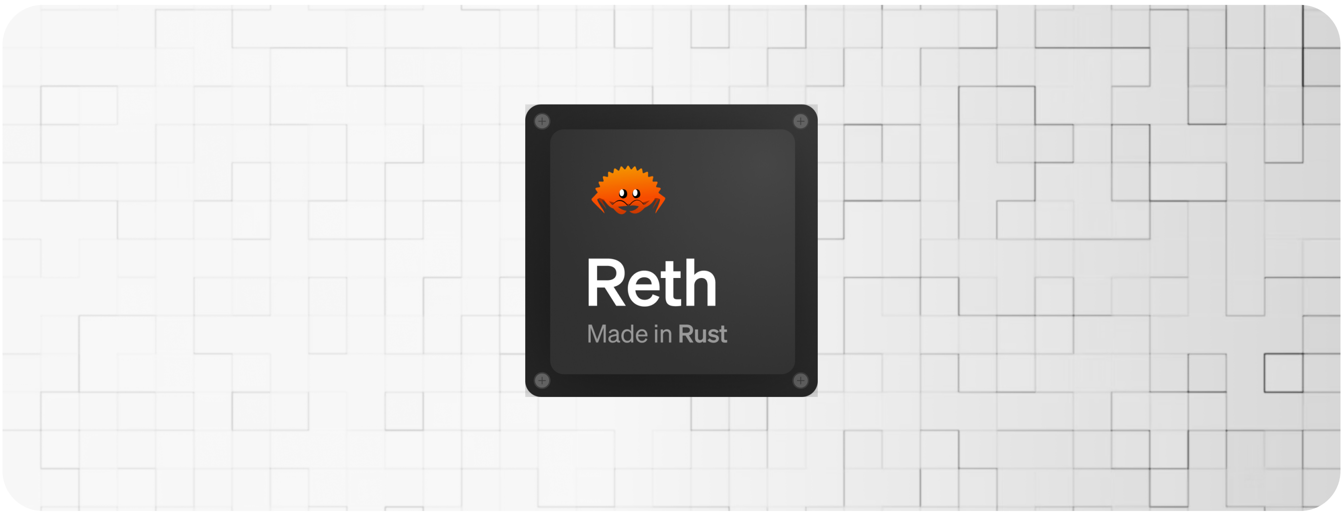
Reth Book

Documentation for Reth users and developers.

chat 172 Online

Reth (short for Rust Ethereum, pronunciation) is an **Ethereum full node implementation that is focused on being user-friendly, highly modular, as well as being fast and efficient.**



What is this about?

Reth is an execution layer (EL) implementation that is compatible with all Ethereum consensus layer (CL) implementations that support the Engine API.

It is originally built and driven forward by Paradigm, and is licensed under the Apache and MIT licenses.

As a full Ethereum node, Reth allows users to connect to the Ethereum network and interact with the Ethereum blockchain.

This includes sending and receiving transactions, querying logs and traces, as well as accessing and interacting with smart contracts.

Building a successful Ethereum node requires creating a high-quality implementation that is both secure and efficient, as well as being easy to use on consumer hardware. It also requires building a strong community of contributors who can help support and improve the software.

What are the goals of Reth?

1. Modularity

Every component of Reth is built to be used as a library: well-tested, heavily documented and benchmarked. We envision that developers will import the node's crates, mix and match, and innovate on top of them.

Examples of such usage include, but are not limited to, spinning up standalone P2P networks, talking directly to a node's database, or "unbundling" the node into the components you need.

To achieve that, we are licensing Reth under the Apache/MIT permissive license.

2. Performance

Reth aims to be fast, so we used Rust and the Erigon staged-sync node architecture.

We also use our Ethereum libraries (including ethers-rs and revm) which we’ve battle-tested and optimized via Foundry.

3. Free for anyone to use any way they want

Reth is free open source software, built for the community, by the community.

By licensing the software under the Apache/MIT license, we want developers to use it without being bound by business licenses, or having to think about the implications of GPL-like licenses.

4. Client Diversity

The Ethereum protocol becomes more antifragile when no node implementation dominates. This ensures that if there's a software bug, the network does not finalize a bad block. By building a new client, we hope to contribute to Ethereum's antifragility.

5. Used by a wide demographic

We want to solve for node operators that care about fast historical queries, but also for hobbyists who cannot operate on large hardware.

We also want to support teams and individuals who want both sync from genesis and via "fast sync".

We envision that Reth will be configurable enough for the tradeoffs that each team faces.

Who is this for?

Reth is a new Ethereum full node that allows users to sync and interact with the entire blockchain, including its historical state if in archive mode.

- Full node: It can be used as a full node, which stores and processes the entire blockchain, validates blocks and transactions, and participates in the consensus process.
- Archive node: It can also be used as an archive node, which stores the entire history of the blockchain and is useful for applications that need access to historical data.

As a data engineer/analyst, or as a data indexer, you'll want to use Archive mode. For all other use cases where historical access is not needed, you can use Full mode.

Is this secure?

Reth implements the specification of Ethereum as defined in the ethereum/execution-specs repository. To make sure the node is built securely, we run the following tests:

1. EVM state tests are run on every Revm Pull Request
2. Hive tests are run every 24 hours in the main Reth repository.
3. We regularly re-sync multiple nodes from scratch.
4. We operate multiple nodes at the tip of Ethereum mainnet and various testnets.
5. We extensively unit test, fuzz test and document all our code, while also restricting PRs with aggressive lint rules.

We intend to also audit / fuzz the EVM & parts of the codebase. Please reach out if you're interested in collaborating on securing this codebase.

Sections

Here are some useful sections to jump to:

- Install Reth by following the guide.
- Sync your node on any official network.
- View statistics and metrics about your node.
- Query the JSON-RPC using Foundry's `cast` or `curl`.
- Set up your development environment and contribute!

📖 **About this book**

The book is continuously rendered here! You can contribute to this book on GitHub.

Installation

Reth runs on Linux and macOS (Windows tracked).

There are three core methods to obtain Reth:

- Pre-built binaries
- Docker images
- Building from source.

Hardware Requirements

The hardware requirements for running Reth depend on the node configuration and can change over time as the network grows or new features are implemented.

The most important requirement is by far the disk, whereas CPU and RAM requirements are relatively flexible.

	Archive Node	Full Node
Disk	At least 2.2TB (TLC NVMe recommended)	At least 1TB (TLC NVMe recommended)
Memory	8GB+	8GB+
CPU	Higher clock speed over core count	Higher clock speeds over core count
Bandwidth	Stable 24Mbps+	Stable 24Mbps+

QLC and TLC

It is crucial to understand the difference between QLC and TLC NVMe drives when considering the disk requirement.

QLC (Quad-Level Cell) NVMe drives utilize four bits of data per cell, allowing for higher storage density and lower manufacturing costs. However, this increased density comes at the expense of performance. QLC drives have slower read and write speeds compared to TLC drives. They also have a lower endurance, meaning they may have a shorter lifespan and be less suitable for heavy workloads or constant data rewriting.

TLC (Triple-Level Cell) NVMe drives, on the other hand, use three bits of data per cell. While they have a slightly lower storage density compared to QLC drives, TLC drives offer faster performance. They typically have higher read and write speeds, making them more suitable for demanding tasks such as data-intensive applications, gaming, and multimedia editing. TLC drives also tend to have a higher endurance, making them more durable and longer-lasting.

Prior to purchasing an NVMe drive, it is advisable to research and determine whether the disk will be based on QLC or TLC technology. An overview of recommended and not-so-recommended NVMe boards can be found at here.

Disk

There are multiple types of disks to sync Reth, with varying size requirements, depending on the syncing mode. As of October 2023 at block number 18.3M:

- Archive Node: At least 2.2TB is required
- Full Node: At least 1TB is required

NVMe drives are recommended for the best performance, with SSDs being a cheaper alternative. HDDs are the cheapest option, but they will take the longest to sync, and are not recommended.

As of July 2023, syncing an Ethereum mainnet node to block 17.7M on NVMe drives takes about 50 hours, while on a GCP "Persistent SSD" it takes around 5 days.

> **Note**
>
> It is highly recommended to choose a TLC drive when using NVMe, and not a QLC drive. See the note above. A list of recommended drives can be found here.

CPU

Most of the time during syncing is spent executing transactions, which is a single-threaded operation due to potential state dependencies of a transaction on previous ones.

As a result, the number of cores matters less, but in general higher clock speeds are better. More cores are better for parallelizable stages (like sender recovery or bodies downloading), but these stages are not the primary bottleneck for syncing.

Memory

It is recommended to use at least 8GB of RAM.

Most of Reth's components tend to consume a low amount of memory, unless you are under heavy RPC load, so this should matter less than the other requirements.

Higher memory is generally better as it allows for better caching, resulting in less stress on the disk.

Bandwidth

A stable and dependable internet connection is crucial for both syncing a node from genesis and for keeping up with the chain's tip.

Note that due to Reth's staged sync, you only need an internet connection for the Headers and Bodies stages. This means that the first 1-3 hours (depending on your internet connection) will be online, downloading all necessary data, and the rest will be done offline and does not require an internet connection.

Once you're synced to the tip you will need a reliable connection, especially if you're operating a validator. A 24Mbps connection is recommended, but you can probably get away with less. Make sure your ISP does not cap your bandwidth.

What hardware can I get?

If you are buying your own NVMe SSD, please consult this hardware comparison which is being actively maintained. We recommend against buying DRAM-less or QLC devices as these are noticeably slower.

All our benchmarks have been produced on Latitude.sh, a bare metal provider. We use `c3.large.x86` boxes, and also recommend trying the `s2.small.x86` box for pruned/full nodes. So far our experience has been smooth with some users reporting that the NVMEs there outperform AWS NVMEs by 3x or more. We're excited for more Reth nodes on Latitude.sh, so for a limited time you can use `RETH400` for a $250 discount. Run a node now!

Binaries

Archives of precompiled binaries of reth are available for Windows, macOS and Linux. They are static executables. Users of platforms not explicitly listed below should download one of these archives.

If you use **macOS Homebrew** or **Linuxbrew**, you can install Reth from Paradigm's homebrew tap:

```
brew install paradigmxyz/brew/reth
```

If you use **Arch Linux** you can install stable Reth from the AUR using an AUR helper (paru as an example here):

```
paru -S reth # Stable
paru -S reth-git # Unstable (git)
```

Docker

There are two ways to obtain a Reth Docker image:

1. GitHub
2. Building it from source

Once you have obtained the Docker image, proceed to Using the Docker image.

Note

Reth requires Docker Engine version 20.10.10 or higher due to missing support for the `clone3` syscall in previous versions.

GitHub

Reth docker images for both x86_64 and ARM64 machines are published with every release of reth on GitHub Container Registry.

You can obtain the latest image with:

```
docker pull ghcr.io/paradigmxyz/reth
```

Or a specific version (e.g. v0.0.1) with:

```
docker pull ghcr.io/paradigmxyz/reth:v0.0.1
```

You can test the image with:

```
docker run --rm ghcr.io/paradigmxyz/reth --version
```

If you can see the latest Reth release version, then you've successfully installed Reth via Docker.

Building the Docker image

To build the image from source, navigate to the root of the repository and run:

```
docker build . -t reth:local
```

The build will likely take several minutes. Once it's built, test it with:

```
docker run reth:local --version
```

Using the Docker image

There are two ways to use the Docker image:

1. Using Docker
2. Using Docker Compose

Using Plain Docker

To run Reth with Docker, run:

```
docker run \
    -v rethdata:/root/.local/share/reth/mainnet/db \
    -d \
    -p 9001:9001 \
    -p 30303:30303 \
    -p 30303:30303/udp \
    --name reth \
    reth:local \
    node \
    --metrics 0.0.0.0:9001
```

The above command will create a container named `reth` and a named volume called `rethdata` for data persistence. It will also expose the `30303` port (TCP and UDP) for peering with other nodes and the `9001` port for metrics.

It will use the local image `reth:local`. If you want to use the GitHub Container Registry remote image, use `ghcr.io/paradigmxyz/reth` with your preferred tag.

Using Docker Compose

To run Reth with Docker Compose, run the following command from a shell inside the root directory of this repository:

```
./etc/generate-jwt.sh
docker compose -f etc/docker-compose.yml -f etc/lighthouse.yml up -d
```

Note

If you want to run Reth with a CL that is not Lighthouse:

- The JWT for the consensus client can be found at `etc/jwttoken/jwt.hex` in this repository, after the `etc/generate-jwt.sh` script is run
- The Reth Engine API is accessible on `localhost:8551`

To check if Reth is running correctly, run:

```
docker compose -f etc/docker-compose.yml -f etc/lighthouse.yml logs -f reth
```

The default `docker-compose.yml` file will create three containers:

- Reth
- Prometheus
- Grafana

The optional `lighthouse.yml` file will create two containers:

- Lighthouse
- `ethereum-metrics-exporter`

Grafana will be exposed on `localhost:3000` and accessible via default credentials (username and password is `admin`), with two available dashboards:

- reth
- Ethereum Metrics Exporter (works only if Lighthouse is also running)

Interacting with Reth inside Docker

To interact with Reth you must first open a shell inside the Reth container by running:

```
docker exec -it reth bash
```

If Reth is running with Docker Compose, replace `reth` with `reth-reth-1` in the above command

Refer to the CLI docs to interact with Reth once inside the Reth container.

Build from Source

You can build Reth on Linux, macOS, Windows, and Windows WSL2.

Note

Reth does **not** work on Windows WSL1.

Dependencies

First, **install Rust** using rustup:

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

The rustup installer provides an easy way to update the Rust compiler, and works on all platforms.

Tips

- During installation, when prompted, enter `1` for the default installation.
- After Rust installation completes, try running `cargo version` . If it cannot be found, run `source $HOME/.cargo/env` . After that, running `cargo version` should return the version, for example `cargo 1.68.2` .
- It's generally advisable to append `source $HOME/.cargo/env` to `~/.bashrc` .

With Rust installed, follow the instructions below to install dependencies relevant to your operating system:

- **Ubuntu**: `apt-get install libclang-dev pkg-config build-essential`
- **macOS**: `brew install llvm pkg-config`
- **Windows**: `choco install llvm` or `winget install LLVM.LLVM`

These are needed to build bindings for Reth's database.

Build Reth

With Rust and the dependencies installed, you're ready to build Reth. First, clone the repository:

```
git clone https://github.com/paradigmxyz/reth
cd reth
```

Then, install Reth into your `PATH` directly via:

```
cargo install --locked --path bin/reth --bin reth
```

The binary will now be accessible as `reth` via the command line, and exist under your default `.cargo/bin` folder.

Alternatively, you can build yourself with:

```
cargo build --release
```

This will place the reth binary under `./target/release/reth`, and you can copy it to your directory of preference after that.

Compilation may take around 10 minutes. Installation was successful if `reth --help` displays the command-line documentation.

If you run into any issues, please check the Troubleshooting section, or reach out to us on Telegram.

Update Reth

You can update Reth to a specific version by running the commands below.

The `reth` directory will be the location you cloned reth to during the installation process.

`${VERSION}` will be the version you wish to build in the format `vX.X.X`.

```
cd reth
git fetch
git checkout ${VERSION}
cargo build --release
```

Optimizations

Profiles

You can customise the compiler settings used to compile Reth via Cargo profiles.

Reth includes several profiles which can be selected via the Cargo flag `--profile`.

- `release` : default for source builds, enables most optimisations while not taking too long to compile.
- `maxperf` : default for binary releases, enables aggressive optimisations including full LTO. Although compiling with this profile improves some benchmarks by around 20% compared to `release` , it imposes a *significant* cost at compile time and is only recommended if you have a fast CPU.

Rust compiler flags

You can also use `RUSTFLAGS="-C target-cpu=native"` to enable CPU-specific optimisations. In order to get the highest performance out of your build:

```
RUSTFLAGS="-C target-cpu=native" cargo build --profile maxperf
```

Features

Finally, some features may improve performance on your system, most notably `jemalloc` , which replaces the default memory allocator used by reth.

You can enable features by passing them to the `--features` Cargo flag.

> **Note**
>
> The `jemalloc` feature is unstable on Windows due to jemallocator itself.

Troubleshooting

Command is not found

Reth will be installed to `CARGO_HOME` or `$HOME/.cargo` . This directory needs to be on your `PATH` before you can run `$ reth` .

See "Configuring the `PATH` environment variable" for more information.

Compilation error

Make sure you are running the latest version of Rust. If you have installed Rust using rustup, simply run `rustup update` .

If you can't install the latest version of Rust you can instead compile using the Minimum

Supported Rust Version (MSRV) which is listed under the `rust-version` key in Reth's Cargo.toml.

If compilation fails with `(signal: 9, SIGKILL: kill)`, this could mean your machine ran out of memory during compilation. If you are on Docker, consider increasing the memory of the container, or use a pre-built binary.

If compilation fails with `error: linking with cc failed: exit code: 1`, try running `cargo clean`.

(Thanks to Sigma Prime for this section from their Lighthouse book!)

Bus error (WSL2)

In WSL 2 on Windows, the default virtual disk size is set to 1TB.

You must increase the allocated disk size for your WSL2 instance before syncing reth.

You can follow the instructions here: how to expand the size of your WSL2 virtual hard disk.

Building for ARM devices

Reth can be built for and run on ARM devices, but there are a few things to take into considerations before.

CPU Architecture

First, you must have a 64-bit CPU and Operating System, otherwise some of the project dependencies will not be able to compile or be executed.

Memory Layout on AArch64

Then, you must setup the virtual memory layout in such a way that the user space is sufficiently large. From the Linux Kernel documentation, you can see that the memory layout with 4KB pages and a level-3 translation table limits the user space to 512GB, which is too low for Reth to sync on Ethereum mainnet.

ARM Board Virtual Memory Limitation

Issue Description

Some ARM boards are equipped with only 3-level paging, which imposes a virtual memory limitation of 256GB for user space on Linux. This limitation can be a challenge for running applications like "reth", as the MDBX (Memory-mapped Database eXtreme) library requires a larger virtual memory allocation by design.

Understanding the Limitation

To determine if a specific ARM board is affected by this virtual memory limitation:

1. **Check Specifications:** When considering an ARM board, review its specifications for information on paging levels. Boards with 3-level paging may have a 256GB virtual memory limit.
2. **Manufacturer Documentation:** Consult the official ARM board documentation for details on supported paging levels.

3. **Community Discussions:** Search online ARM and Linux forums for insights into virtual memory limitations of specific boards.

Additional Context

According to MDBX documentation, changing this upper bound, which dictates the maximum size the database can reach, is a costly operation. Therefore, a reasonably large value was chosen. Given that the upper bound is currently set to 4TB, the assumption was that growth to 3TB might occur relatively soon. If the upper bound size is set to only 342GB, then "reth" cannot store more than 342GB of data, which is insufficient for a full sync.

It's worth noting that on x86_64 architecture, there is a 48-bit address space divided in half between user space and the kernel, providing each with 128TB of address space. In contrast, AArch64 architecture features a user space address space of 512GB and a kernel address space of 256TB.

Some newer versions of ARM architecture offer support for Large Virtual Address space, but enabling this requires running with a 64KB page size. The specifics of how to enable this functionality might vary.

Additional Resources

- ARM developer documentation
- ARM Community Forums

Build Reth

If both your CPU architecture and the memory layout are valid, the instructions for building Reth will not differ from the standard process.

Troubleshooting

> If you ever need to recompile the Linux Kernel because the official OS images for your ARM board don't have the right memory layout configuration, you can use the Armbian build framework.

Failed to open database

This error is documented here.

This error is raised whenever MBDX can not open a database due to the limitations imposed by the memory layout of your kernel. If the user space is limited to 512GB, the database will not be able to grow below this size.

You will need to recompile the Linux Kernel to fix the issue.

A simple and safe approach to achieve this is to use the Armbian build framework to create a new image of the OS that will be flashed to a storage device of your choice - an SD card for example - with the following kernel feature values:

- **Page Size**: 64 KB
- **Virtual Address Space Size**: 48 Bits

To be able to build an Armbian image and set those values, you will need to:

- Clone the Armbian build framework repository

```
git clone https://github.com/armbian/build
cd build
```

- Run the compile script with the following parameters:

```
./compile.sh \
BUILD_MINIMAL=yes \
BUILD_DESKTOP=no \
KERNEL_CONFIGURE=yes \
CARD_DEVICE="/dev/sdX" # Replace sdX with your own storage device
```

- From there, you will be able to select the target board, the OS release and branch. Then, once you get in the **Kernel Configuration** screen, select the **Kernel Features options** and set the previous values accordingly.
- Wait for the process to finish, plug your storage device into your board and start it. You can now download or install Reth and it should work properly.

Update Priorities

When publishing releases, reth will include an "Update Priority" section in the release notes, in the same manner Lighthouse does.

The "Update Priority" section will include a table which may appear like so:

User Class	Priority
Payload Builders	Medium Priority
Non-Payload Builders	Low Priority

To understand this table, the following terms are important:

- *Payload builders* are those who use reth to build and validate payloads.
- *Non-payload builders* are those who run reth for other purposes (e.g., data analysis, RPC or applications).
- *High priority* updates should be completed as soon as possible (e.g., hours or days).
- *Medium priority* updates should be completed at the next convenience (e.g., days or a week).
- *Low priority* updates should be completed in the next routine update cycle (e.g., two weeks).

Run a Node

Congratulations, now that you have installed Reth, it's time to run it!

In this chapter we'll go through a few different topics you'll encounter when running Reth, including:

1. Running on mainnet or official testnets
2. Logs and Observability
3. Configuring reth.toml
4. Transaction types
5. Pruning & Full Node
6. Ports
7. Troubleshooting

In the future, we also intend to support the OP Stack, which will allow you to run Reth as a Layer 2 client. More there soon!

Running Reth on Ethereum Mainnet or testnets

Reth is an *execution client*. After Ethereum's transition to Proof of Stake (aka the Merge) it became required to run a *consensus client* along your execution client in order to sync into any "post-Merge" network. This is because the Ethereum execution layer now outsources consensus to a separate component, known as the consensus client.

Consensus clients decide what blocks are part of the chain, while execution clients only validate that transactions and blocks are valid in themselves and with respect to the world state. In other words, execution clients execute blocks and transactions and check their validity, while consensus clients determine which valid blocks should be part of the chain. Therefore, running a consensus client in parallel with the execution client is necessary to ensure synchronization and participation in the network.

By running both an execution client like Reth and a consensus client, such as Lighthouse 🦀 (which we will assume for this guide), you can effectively contribute to the Ethereum network and participate in the consensus process, even if you don't intend to run validators.

Client	Role
Execution	Validates transactions and blocks
	(checks their validity and global state)
Consensus	Determines which blocks are part of the chain
	(makes consensus decisions)

Running the Reth Node

First, ensure that you have Reth installed by following the installation instructions.

Now, to start the archive node, run:

```
RUST_LOG=info reth node
```

And to start the full node, run:

```
RUST_LOG=info reth node --full
```

On differences between archive and full nodes, see Pruning & Full Node section.

> Note that these commands will not open any HTTP/WS ports by default. You can change this by adding the `--http`, `--ws` flags, respectively and using the `--http.api` and `--ws.api` flags to enable various JSON-RPC APIs. For more commands, see the `reth node` CLI reference.

The EL <> CL communication happens over the Engine API, which is by default exposed at `http://localhost:8551`. The connection is authenticated over JWT using a JWT secret which is auto-generated by Reth and placed in a file called `jwt.hex` in the data directory, which on Linux by default is `$HOME/.local/share/reth/` (`/Users/<NAME>/Library/Application Support/reth/mainnet/jwt.hex` in Mac).

You can override this path using the `--authrpc.jwtsecret` option. You MUST use the same JWT secret in BOTH Reth and the chosen Consensus Layer. If you want to override the address or port, you can use the `--authrpc.addr` and `--authrpc.port` options, respectively.

So one might do:

```
RUST_LOG=info reth node \
    --authrpc.jwtsecret /path/to/secret \
    --authrpc.addr 127.0.0.1 \
    --authrpc.port 8551
```

At this point, our Reth node has started discovery, and even discovered some new peers. But it will not start syncing until you spin up the consensus layer!

Running the Consensus Layer

First, make sure you have Lighthouse installed. Sigma Prime provides excellent installation and node operation instructions.

Assuming you have done that, run:

```
RUST_LOG=info lighthouse bn \
    --checkpoint-sync-url https://mainnet.checkpoint.sigp.io \
    --execution-endpoint http://localhost:8551 \
    --execution-jwt /path/to/secret
```

If you don't intend on running validators on your node you can add:

```
  --disable-deposit-contract-sync
```

The `--checkpoint-sync-url` argument value can be replaced with any checkpoint sync endpoint from a community maintained list.

Your Reth node should start receiving "fork choice updated" messages, and begin syncing the chain.

Verify the chain is growing

You can easily verify that by inspecting the logs, and seeing that headers are arriving in Reth. Sit back now and wait for the stages to run! In the meantime, consider setting up observability to monitor your node's health or test the JSON RPC API.

Running without a Consensus Layer

We provide a method for running Reth without a Consensus Layer via the `--debug.tip <HASH>` parameter. If you provide that to your node, it will simulate sending a `engine_forkChoiceUpdated` message *once* and will trigger syncing to the provided block hash. This is useful for testing and debugging purposes, but in order to have a node that can keep up with the tip you'll need to run a CL alongside it. At the moment we have no plans of including a Consensus Layer implementation in Reth, and we are open to including light clients other methods of syncing like importing Lighthouse as a library.

Run Reth in a private testnet using Kurtosis

For those who need a private testnet to validate functionality or scale with Reth.

Using Docker locally

This guide uses Kurtosis' ethereum-package and assumes you have Kurtosis and Docker installed and have Docker already running on your machine.

- Go here to install Kurtosis
- Go here to install Docker

The `ethereum-package` is a package for a general purpose Ethereum testnet definition used for instantiating private testnets at any scale over Docker or Kubernetes, locally or in the cloud. This guide will go through how to spin up a local private testnet with Reth various CL clients locally. Specifically, you will instantiate a 2-node network over Docker with Reth/Lighthouse and Reth/Teku client combinations.

To see all possible configurations and flags you can use, including metrics and observability tools (e.g. Grafana, Prometheus, etc), go here.

Genesis data will be generated using this genesis-generator to be used to bootstrap the EL and CL clients for each node. The end result will be a private testnet with nodes deployed as Docker containers in an ephemeral, isolated environment on your machine called an enclave. Read more about how the `ethereum-package` works by going here.

Step 1: Define the parameters and shape of your private network

First, in your home directory, create a file with the name `network_params.json` with the following contents:

```
{
  "participants": [
    {
      "el_client_type": "reth",
      "el_client_image": "ghcr.io/paradigmxyz/reth",
      "cl_client_type": "lighthouse",
      "cl_client_image": "sigp/lighthouse:latest",
      "count": 1
    },
    {
      "el_client_type": "reth",
      "el_client_image": "ghcr.io/paradigmxyz/reth",
      "cl_client_type": "teku",
      "cl_client_image": "consensys/teku:latest",
      "count": 1
    }
  ],
  "launch_additional_services": false
}
```

Step 2: Spin up your network

Next, run the following command from your command line:

```
kurtosis run github.com/kurtosis-tech/ethereum-package --args-file
~/network_params.json
```

Kurtosis will spin up an enclave (i.e an ephemeral, isolated environment) and begin to configure and instantiate the nodes in your network. In the end, Kurtosis will print the services running in your enclave that form your private testnet alongside all the container ports and files that were generated & used to start up the private testnet. Here is a sample output:

```
INFO[2023-08-21T18:22:18-04:00]
=====================================================
INFO[2023-08-21T18:22:18-04:00] ||          Created enclave: silky-swamp
||
INFO[2023-08-21T18:22:18-04:00]
=====================================================
Name:             silky-swamp
UUID:             3df730c66123
Status:           RUNNING
Creation Time:    Mon, 21 Aug 2023 18:21:32 EDT

========================================== Files Artifacts
==========================================
UUID           Name
c168ec4468f6   1-lighthouse-reth-0-63
61f821e2cfd5   2-teku-reth-64-127
e6f94fdac1b8   cl-genesis-data
e6b57828d099   el-genesis-data
1fb632573a2e   genesis-generation-config-cl
b8917e497980   genesis-generation-config-el
6fd8c5be336a   geth-prefunded-keys
6ab83723b4bd   prysm-password

========================================== User Services
==========================================
UUID           Name                                           Ports
Status
95386198d3f9   cl-1-lighthouse-reth                           http: 4000/tcp ->
http://127.0.0.1:64947      RUNNING
                                                              metrics: 5054/tcp
-> http://127.0.0.1:64948
                                                              tcp-discovery:
9000/tcp -> 127.0.0.1:64949
                                                              udp-discovery:
9000/udp -> 127.0.0.1:60303
5f5cc4cf639a   cl-1-lighthouse-reth-validator                 http: 5042/tcp ->
127.0.0.1:64950             RUNNING
                                                              metrics: 5064/tcp
-> http://127.0.0.1:64951
27e1cfaddc72   cl-2-teku-reth                                 http: 4000/tcp ->
127.0.0.1:64954             RUNNING
                                                              metrics: 8008/tcp
-> 127.0.0.1:64952
                                                              tcp-discovery:
9000/tcp -> 127.0.0.1:64953
                                                              udp-discovery:
9000/udp -> 127.0.0.1:53749
b454497fbec8   el-1-reth-lighthouse                           engine-rpc:
8551/tcp -> 127.0.0.1:64941       RUNNING
                                                              metrics: 9001/tcp
-> 127.0.0.1:64937
                                                              rpc: 8545/tcp ->
127.0.0.1:64939
                                                              tcp-discovery:
30303/tcp -> 127.0.0.1:64938
                                                              udp-discovery:
```

```
30303/udp -> 127.0.0.1:55861
                                                          ws: 8546/tcp ->
127.0.0.1:64940
03a2ef13c99b   el-2-reth-teku                             engine-rpc:
8551/tcp -> 127.0.0.1:64945       RUNNING
                                                          metrics: 9001/tcp
-> 127.0.0.1:64946
                                                          rpc: 8545/tcp ->
127.0.0.1:64943
                                                          tcp-discovery:
30303/tcp -> 127.0.0.1:64942
                                                          udp-discovery:
30303/udp -> 127.0.0.1:64186
                                                          ws: 8546/tcp ->
127.0.0.1:64944
5c199b334236   prelaunch-data-generator-cl-genesis-data   <none>
RUNNING
46829c4bd8b0   prelaunch-data-generator-el-genesis-data   <none>
RUNNING
```

Great! You now have a private network with 2 full Ethereum nodes on your local machine over Docker - one that is a Reth/Lighthouse pair and another that is Reth/Teku. Check out the Kurtosis docs to learn about the various ways you can interact with and inspect your network.

Using Kurtosis on Kubernetes

Kurtosis packages are portable and reproducible, meaning they will work the same way over Docker or Kubernetes, locally or on remote infrastructure. For use cases that require a larger scale, Kurtosis can be deployed on Kubernetes by following these docs here.

Running the network with additional services

The `ethereum-package` comes with many optional flags and arguments you can enable for your private network. Some include:

- A Grafana + Prometheus instance
- A transaction spammer called `tx-fuzz`
- A network metrics collector
- Flashbot's `mev-boost` implementation of PBS (to test/simulate MEV workflows)

Questions?

Please reach out to the Kurtosis discord should you have any questions about how to use the `ethereum-package` for your private testnet needs. Thanks!

Observability with Prometheus & Grafana

Reth exposes a number of metrics, which are listed here. We can serve them from an HTTP endpoint by adding the `--metrics` flag:

```
RUST_LOG=info reth node --metrics 127.0.0.1:9001
```

Now, as the node is running, you can `curl` the endpoint you provided to the `--metrics` flag to get a text dump of the metrics at that time:

```
curl 127.0.0.1:9001
```

The response from this is quite descriptive, but it can be a bit verbose. Plus, it's just a snapshot of the metrics at the time that you `curl` ed the endpoint.

You can run the following command in a separate terminal to periodically poll the endpoint, and just print the values (without the header text) to the terminal:

```
while true; do date; curl -s localhost:9001 | grep -Ev '^(#|$)' | sort; echo;
sleep 10; done
```

We're finally getting somewhere! As a final step, though, wouldn't it be great to see how these metrics progress over time (and generally, in a GUI)?

Prometheus & Grafana

We're going to be using Prometheus to collect metrics off of the endpoint we set up, and use Grafana to scrape the metrics from Prometheus and define a dashboard with them.

Let's begin by installing both Prometheus and Grafana, which one can do with e.g. Homebrew:

```
brew update
brew install prometheus
brew install grafana
```

Then, kick off the Prometheus and Grafana services:

```
brew services start prometheus
brew services start grafana
```

This will start a Prometheus service which by default scrapes itself about the current instance. So you'll need to change its config to hit your Reth nodes metrics endpoint at `localhost:9001` which you set using the `--metrics` flag.

You can find an example config for the Prometheus service in the repo here: `etc/prometheus/prometheus.yml`

Depending on your installation you may find the config for your Prometheus service at:

- OSX: `/opt/homebrew/etc/prometheus.yml`
- Linuxbrew: `/home/linuxbrew/.linuxbrew/etc/prometheus.yml`
- Others: `/usr/local/etc/prometheus/prometheus.yml`

Next, open up "localhost:3000" in your browser, which is the default URL for Grafana. Here, "admin" is the default for both the username and password.

Once you've logged in, click on the gear icon in the lower left, and select "Data Sources". Click on "Add data source", and select "Prometheus" as the type. In the HTTP URL field, enter `http://localhost:9090`. Finally, click "Save & Test".

As this might be a point of confusion, `localhost:9001`, which we supplied to `--metrics`, is the endpoint that Reth exposes, from which Prometheus collects metrics. Prometheus then exposes `localhost:9090` (by default) for other services (such as Grafana) to consume Prometheus metrics.

To configure the dashboard in Grafana, click on the squares icon in the upper left, and click on "New", then "Import". From there, click on "Upload JSON file", and select the example file in `reth/etc/grafana/dashboards/overview.json`. Finally, select the Prometheus data source you just created, and click "Import".

And voilá, you should see your dashboard! If you're not yet connected to any peers, the dashboard will look like it's in an empty state, but once you are, you should see it start populating with data.

Conclusion

In this runbook, we took you through starting the node, exposing different log levels, exporting metrics, and finally viewing those metrics in a Grafana dashboard.

This will all be very useful to you, whether you're simply running a home node and want to keep an eye on its performance, or if you're a contributor and want to see the effect that your (or others') changes have on Reth's operations.

Configuring Reth

Reth places a configuration file named `reth.toml` in the data directory specified when starting the node. It is written in the TOML format.

The default data directory is platform dependent:

- Linux: `$XDG_DATA_HOME/reth/` or `$HOME/.local/share/reth/`
- Windows: `{FOLDERID_RoamingAppData}/reth/`
- macOS: `$HOME/Library/Application Support/reth/`

The configuration file contains the following sections:

- `[stages]` -- Configuration of the individual sync stages
  - `headers`
  - `total_difficulty`
  - `bodies`
  - `sender_recovery`
  - `execution`
  - `account_hashing`
  - `storage_hashing`
  - `merkle`
  - `transaction_lookup`
  - `index_account_history`
  - `index_storage_history`
- `[peers]`
  - `connection_info`
  - `reputation_weights`
  - `backoff_durations`
- `[sessions]`
- `[prune]`

The `[stages]` section

The stages section is used to configure how individual stages in reth behave, which has a direct impact on resource utilization and sync speed.

The defaults shipped with Reth try to be relatively reasonable, but may not be optimal for your specific set of hardware.

headers

The headers section controls both the behavior of the header stage, which download historical headers, as well as the primary downloader that fetches headers over P2P.

```toml
[stages.headers]
# The minimum and maximum number of concurrent requests to have in flight at
a time.
#
# The downloader uses these as best effort targets, which means that the
number
# of requests may be outside of these thresholds within a reasonable degree.
#
# Increase these for faster sync speeds at the cost of additional bandwidth
and memory
downloader_max_concurrent_requests = 100
downloader_min_concurrent_requests = 5
# The maximum number of responses to buffer in the downloader at any one
time.
#
# If the buffer is full, no more requests will be sent until room opens up.
#
# Increase the value for a larger buffer at the cost of additional memory
consumption
downloader_max_buffered_responses = 100
# The maximum number of headers to request from a peer at a time.
downloader_request_limit = 1000
# The amount of headers to persist to disk at a time.
#
# Lower thresholds correspond to more frequent disk I/O (writes),
# but lowers memory usage
commit_threshold = 10000
```

total_difficulty

The total difficulty stage calculates the total difficulty reached for each header in the chain.

```toml
[stages.total_difficulty]
# The amount of headers to calculate the total difficulty for
# before writing the results to disk.
#
# Lower thresholds correspond to more frequent disk I/O (writes),
# but lowers memory usage
commit_threshold = 100000
```

bodies

The bodies section controls both the behavior of the bodies stage, which download historical block bodies, as well as the primary downloader that fetches block bodies over P2P.

```toml
[stages.bodies]
# The maximum number of bodies to request from a peer at a time.
downloader_request_limit = 200
# The maximum amount of bodies to download before writing them to disk.
#
# A lower value means more frequent disk I/O (writes), but also
# lowers memory usage.
downloader_stream_batch_size = 1000
# The size of the internal block buffer in bytes.
#
# A bigger buffer means that bandwidth can be saturated for longer periods,
# but also increases memory consumption.
#
# If the buffer is full, no more requests will be made to peers until
# space is made for new blocks in the buffer.
#
# Defaults to around 2GB.
downloader_max_buffered_blocks_size_bytes = 2147483648
# The minimum and maximum number of concurrent requests to have in flight at
a time.
#
# The downloader uses these as best effort targets, which means that the
number
# of requests may be outside of these thresholds within a reasonable degree.
#
# Increase these for faster sync speeds at the cost of additional bandwidth
and memory
downloader_min_concurrent_requests = 5
downloader_max_concurrent_requests = 100
```

sender_recovery

The sender recovery stage recovers the address of transaction senders using transaction signatures.

```toml
[stages.sender_recovery]
# The amount of transactions to recover senders for before
# writing the results to disk.
#
# Lower thresholds correspond to more frequent disk I/O (writes),
# but lowers memory usage
commit_threshold = 100000
```

execution

The execution stage executes historical transactions. This stage is generally very I/O and memory intensive, since executing transactions involves reading block headers, transactions, accounts and account storage.

Each executed transaction also generates a number of changesets, and mutates the current state of accounts and storage.

For this reason, there are two ways to control how much work to perform before the results are written to disk.

```toml
[stages.execution]
# The maximum amount of blocks to execute before writing the results to disk.
max_blocks = 500000
# The maximum amount of account and storage changes to collect before writing
# the results to disk.
max_changes = 5000000
```

Either one of `max_blocks` or `max_changes` must be specified, and both can also be specified at the same time:

- If only `max_blocks` is specified, reth will execute (up to) that amount of blocks before writing to disk.
- If only `max_changes` is specified, reth will execute as many blocks as possible until the target amount of state transitions have occurred before writing to disk.
- If both are specified, then the first threshold to be hit will determine when the results are written to disk.

Lower values correspond to more frequent disk writes, but also lower memory consumption. A lower value also negatively impacts sync speed, since reth keeps a cache around for the entire duration of blocks executed in the same range.

`account_hashing`

The account hashing stage builds a secondary table of accounts, where the key is the hash of the address instead of the raw address.

This is used to later compute the state root.

```toml
[stages.account_hashing]
# The threshold in number of blocks before the stage starts from scratch
# and re-hashes all accounts as opposed to just the accounts that changed.
clean_threshold = 500000
# The amount of accounts to process before writing the results to disk.
#
# Lower thresholds correspond to more frequent disk I/O (writes),
# but lowers memory usage
commit_threshold = 100000
```

storage_hashing

The storage hashing stage builds a secondary table of account storages, where the key is the hash of the address and the slot, instead of the raw address and slot.

This is used to later compute the state root.

```
[stages.storage_hashing]
# The threshold in number of blocks before the stage starts from scratch
# and re-hashes all storages as opposed to just the storages that changed.
clean_threshold = 500000
# The amount of storage slots to process before writing the results to disk.
#
# Lower thresholds correspond to more frequent disk I/O (writes),
# but lowers memory usage
commit_threshold = 100000
```

merkle

The merkle stage uses the indexes built in the hashing stages (storage and account hashing) to compute the state root of the latest block.

```
[stages.merkle]
# The threshold in number of blocks before the stage starts from scratch
# and re-computes the state root, discarding the trie that has already been
built,
# as opposed to incrementally updating the trie.
clean_threshold = 50000
```

transaction_lookup

The transaction lookup stage builds an index of transaction hashes to their sequential transaction ID.

```
[stages.transaction_lookup]
# The maximum number of transactions to process before writing the results to
disk.
#
# Lower thresholds correspond to more frequent disk I/O (writes),
# but lowers memory usage
commit_threshold = 5000000
```

index_account_history

The account history indexing stage builds an index of what blocks a particular account changed.

```
[stages.index_account_history]
# The maximum amount of blocks to process before writing the results to disk.
#
# Lower thresholds correspond to more frequent disk I/O (writes),
# but lowers memory usage
commit_threshold = 100000
```

`index_storage_history`

The storage history indexing stage builds an index of what blocks a particular storage slot changed.

```
[stages.index_storage_history]
# The maximum amount of blocks to process before writing the results to disk.
#
# Lower thresholds correspond to more frequent disk I/O (writes),
# but lowers memory usage
commit_threshold = 100000
```

The `[peers]` section

The peers section is used to configure how the networking component of reth establishes and maintains connections to peers.

In the top level of the section you can configure trusted nodes, and how often reth will try to connect to new peers.

```
[peers]
# How often reth will attempt to make outgoing connections,
# if there is room for more peers
refill_slots_interval = '1s'
# A list of ENRs for trusted peers, which are peers reth will always try to
connect to.
trusted_nodes = []
# Whether reth will only attempt to connect to the peers specified above,
# or if it will connect to other peers in the network
connect_trusted_nodes_only = false
# The duration for which a badly behaving peer is banned
ban_duration = '12h'
```

`connection_info`

This section configures how many peers reth will connect to.

```toml
[peers.connection_info]
# The maximum number of outbound peers (peers we connect to)
max_outbound = 100
# The maximum number of inbound peers (peers that connect to us)
max_inbound = 30
```

`reputation_weights`

This section configures the penalty for various offences peers can commit.

All peers start out with a reputation of 0, which increases over time as the peer stays connected to us.

If the peer misbehaves, various penalties are exacted to their reputation, and if it falls below a certain threshold (currently `50 * -1024`), reth will disconnect and ban the peer temporarily (except for protocol violations which constitute a permanent ban).

```toml
[peers.reputation_weights]
bad_message = -16384
bad_block = -16384
bad_transactions = -16384
already_seen_transactions = 0
timeout = -4096
bad_protocol = -2147483648
failed_to_connect = -25600
dropped = -4096
```

`backoff_durations`

If reth fails to establish a connection to a peer, it will not re-attempt for some amount of time, depending on the reason the connection failed.

```toml
[peers.backoff_durations]
low = '30s'
medium = '3m'
high = '15m'
max = '1h'
```

The `[sessions]` section

The sessions section configures the internal behavior of a single peer-to-peer connection.

You can configure the session buffer sizes, which limits the amount of pending events (incoming messages) and commands (outgoing messages) each session can hold before it will start to ignore messages.

> **Note**
>
> These buffers are allocated *per peer*, which means that increasing the buffer sizes can have large impact on memory consumption.

```toml
[sessions]
session_command_buffer = 32
session_event_buffer = 260
```

You can also configure request timeouts:

```toml
[sessions.initial_internal_request_timeout]
secs = 20
nanos = 0

# The amount of time before the peer will be penalized for
# being in violation of the protocol. This exacts a permaban on the peer.
[sessions.protocol_breach_request_timeout]
secs = 120
nanos = 0
```

The `[prune]` section

The prune section configures the pruning configuration.

You can configure the pruning of different segments of the data independently of others. For any unspecified segments, the default setting is no pruning.

Default config

No pruning, run as archive node.

Example of the custom pruning configuration

This configuration will:

- Run pruning every 5 blocks
- Continuously prune all transaction senders, account history and storage history before the block `head-100_000`, i.e. keep the data for the last `100_000` blocks
- Prune all receipts before the block 1920000, i.e. keep receipts from the block

1920000

```
[prune]
# Minimum pruning interval measured in blocks
block_interval = 5

[prune.parts]
# Sender Recovery pruning configuration
sender_recovery = { distance = 100_000 } # Prune all transaction senders
before the block `head-128`, i.e. keep transaction senders for the last 129
blocks

# Transaction Lookup pruning configuration
transaction_lookup = "full" # Prune all TxNumber => TxHash mappings

# Receipts pruning configuration. This setting overrides
`receipts_log_filter`.
receipts = { before = 1920000 } # Prune all receipts from transactions before
the block 1920000, i.e. keep receipts from the block 1920000

# Account History pruning configuration
account_history = { distance = 100_000 } # Prune all historical account
states before the block `head-128`

# Storage History pruning configuration
storage_history = { distance = 100_000 } # Prune all historical storage
states before the block `head-128`
```

We can also prune receipts more granular, using the logs filtering:

```
# Receipts pruning configuration by retaining only those receipts that
contain logs emitted
# by the specified addresses, discarding all others. This setting is
overridden by `receipts`.
[prune.parts.receipts_log_filter]
# Prune all receipts, leaving only those which:
# - Contain logs from address `0xa0b86991c6218b36c1d19d4a2e9eb0ce3606eb48`,
starting from the block 17000000
# - Contain logs from address `0xdac17f958d2ee523a2206206994597c13d831ec7` in
the last 1001 blocks
"0xa0b86991c6218b36c1d19d4a2e9eb0ce3606eb48" = { before = 17000000 }
"0xdac17f958d2ee523a2206206994597c13d831ec7" = { distance = 1000 }
```

Transaction types

Over time, the Ethereum network has undergone various upgrades and improvements to enhance transaction efficiency, security, and user experience. Three significant transaction types that have evolved are:

- Legacy Transactions,
- EIP-2930 Transactions,
- EIP-1559 Transactions.

Each of these transaction types brings unique features and improvements to the Ethereum network.

Legacy Transactions

Legacy Transactions (type `0x0`), the traditional Ethereum transactions in use since the network's inception, include the following parameters:

- `nonce`,
- `gasPrice`,
- `gasLimit`,
- `to`,
- `value`,
- `data`,
- `v`,
- `r`,
- `s`.

These transactions do not utilize access lists, which specify the addresses and storage keys to be accessed, nor do they incorporate EIP-1559 fee market changes.

EIP-2930 Transactions

Introduced in EIP-2930, transactions with type `0x1` incorporate an `accessList` parameter alongside legacy parameters. This `accessList` specifies an array of addresses and storage keys that the transaction plans to access, enabling gas savings on cross-contract calls by pre-declaring the accessed contract and storage slots. They do not include EIP-1559 fee market changes.

EIP-1559 Transactions

EIP-1559 transactions (type `0x2`) were introduced in Ethereum's London fork to address network congestion and transaction fee overpricing caused by the historical fee market. Unlike traditional transactions, EIP-1559 transactions don't specify a gas price (`gasPrice`). Instead, they use an in-protocol, dynamically changing base fee per gas, adjusted at each block to manage network congestion.

Alongside the `accessList` parameter and legacy parameters (except `gasPrice`), EIP-1559 transactions include:

- `maxPriorityFeePerGas`, specifying the maximum fee above the base fee the sender is willing to pay,
- `maxFeePerGas`, setting the maximum total fee the sender is willing to pay.

The base fee is burned, while the priority fee is paid to the miner who includes the transaction, incentivizing miners to include transactions with higher priority fees per gas.

Pruning & Full Node

> Pruning and full node are new features of Reth, and we will be happy to hear about your experience using them either on GitHub or in the Telegram group.

By default, Reth runs as an archive node. Such nodes have all historical blocks and the state at each of these blocks available for querying and tracing.

Reth also supports pruning of historical data and running as a full node. This chapter will walk through the steps for running Reth as a full node, what caveats to expect and how to configure your own pruned node.

Basic concepts

- Archive node – Reth node that has all historical data from genesis.
- Pruned node – Reth node that has its historical data pruned partially or fully through a custom configuration.
- Full Node – Reth node that has the latest state and historical data for only the last 10064 blocks available for querying in the same way as an archive node.

The node type that was chosen when first running a node **can not** be changed after the initial sync. Turning Archive into Pruned, or Pruned into Full is not supported.

Modes

Archive Node

Default mode, follow the steps from the previous chapter on how to run on mainnet or official testnets.

Pruned Node

To run Reth as a pruned node configured through a custom configuration, modify the `reth.toml` file and run Reth in the same way as archive node by following the steps from the previous chapter on how to run on mainnet or official testnets.

Full Node

To run Reth as a full node, follow the steps from the previous chapter on how to run on mainnet or official testnets, and add a `--full` flag. For example:

```
RUST_LOG=info reth node \
    --full \
    --authrpc.jwtsecret /path/to/secret \
    --authrpc.addr 127.0.0.1 \
    --authrpc.port 8551
```

Size

All numbers are as of October 2023 at block number 18.3M for mainnet.

Archive Node

Archive node occupies at least 2.14TB.

You can track the growth of Reth archive node size with our public Grafana dashboard.

Pruned Node

Different segments take up different amounts of disk space. If pruned fully, this is the total freed space you'll get, per segment:

Segment	Size
Sender Recovery	75GB
Transaction Lookup	150GB
Receipts	250GB
Account History	240GB
Storage History	700GB

Full Node

Full node occupies at least 950GB.

Essentially, the full node is the same as following configuration for the pruned node:

```toml
[prune]
block_interval = 5

[prune.parts]
sender_recovery = { distance = 10_064 }
# transaction_lookup is not pruned
receipts = { before = 11052984 } # Beacon Deposit Contract deployment block:
https://etherscan.io
/tx/0xe75fb554e433e03763a1560646ee22dcb74e5274b34c5ad644e7c0f619a7e1d0
account_history = { distance = 10_064 }
storage_history = { distance = 10_064 }

[prune.parts.receipts_log_filter]
# Prune all receipts, leaving only those which contain logs from address
`0x00000000219ab540356cbb839cbe05303d7705fa`,
# starting from the block 11052984. This leaves receipts with the logs from
the Beacon Deposit Contract.
"0x00000000219ab540356cbb839cbe05303d7705fa" = { before = 11052984 }
```

Meaning, it prunes:

- Account History and Storage History up to the last 10064 blocks
- Sender Recovery up to the last 10064 blocks. The caveat is that it's pruned gradually after the initial sync is completed, so the disk space is reclaimed slowly.
- Receipts up to the last 10064 blocks, preserving all receipts with the logs from Beacon Deposit Contract

Given the aforementioned segment sizes, we get the following full node size:

```
Archive Node - Receipts - AccountHistory - StorageHistory = Full Node
```

```
2.14TB - 250GB - 240GB - 700GB = 950GB
```

RPC support

As it was mentioned in the pruning configuration chapter, there are several segments which can be pruned independently of each other:

- Sender Recovery
- Transaction Lookup
- Receipts
- Account History
- Storage History

Pruning of each of these segments disables different RPC methods, because the historical data or lookup indexes become unavailable.

Full Node

The following tables describe RPC methods available in the full node.

`debug` namespace

RPC	Note
`debug_getRawBlock`	
`debug_getRawHeader`	
`debug_getRawReceipts`	Only for the last 10064 blocks and Beacon Deposit Contract
`debug_getRawTransaction`	
`debug_traceBlock`	Only for the last 10064 blocks
`debug_traceBlockByHash`	Only for the last 10064 blocks
`debug_traceBlockByNumber`	Only for the last 10064 blocks
`debug_traceCall`	Only for the last 10064 blocks
`debug_traceCallMany`	Only for the last 10064 blocks
`debug_traceTransaction`	Only for the last 10064 blocks

`eth` namespace

RPC / Segment	Note
`eth_accounts`	
`eth_blockNumber`	
`eth_call`	Only for the last 10064 blocks
`eth_chainId`	
`eth_createAccessList`	Only for the last 10064 blocks
`eth_estimateGas`	Only for the last 10064 blocks
`eth_feeHistory`	
`eth_gasPrice`	
`eth_getBalance`	Only for the last 10064 blocks
`eth_getBlockByHash`	
`eth_getBlockByNumber`	
`eth_getBlockReceipts`	Only for the last 10064 blocks and Beacon Deposit Contract
`eth_getBlockTransactionCountByHash`	

RPC / Segment	Note
`eth_getBlockTransactionCountByNumber`	
`eth_getCode`	
`eth_getFilterChanges`	
`eth_getFilterLogs`	Only for the last 10064 blocks and Beacon Deposit Contract
`eth_getLogs`	Only for the last 10064 blocks and Beacon Deposit Contract
`eth_getStorageAt`	Only for the last 10064 blocks
`eth_getTransactionByBlockHashAndIndex`	
`eth_getTransactionByBlockNumberAndIndex`	
`eth_getTransactionByHash`	
`eth_getTransactionCount`	Only for the last 10064 blocks
`eth_getTransactionReceipt`	Only for the last 10064 blocks and Beacon Deposit Contract
`eth_getUncleByBlockHashAndIndex`	
`eth_getUncleByBlockNumberAndIndex`	
`eth_getUncleCountByBlockHash`	
`eth_getUncleCountByBlockNumber`	
`eth_maxPriorityFeePerGas`	
`eth_mining`	
`eth_newBlockFilter`	
`eth_newFilter`	
`eth_newPendingTransactionFilter`	
`eth_protocolVersion`	
`eth_sendRawTransaction`	
`eth_sendTransaction`	
`eth_sign`	
`eth_signTransaction`	
`eth_signTypedData`	
`eth_subscribe`	
`eth_syncing`	
`eth_uninstallFilter`	
`eth_unsubscribe`	

`net` namespace

RPC / Segment
`net_listening`
`net_peerCount`
`net_version`

`trace` namespace

RPC / Segment	Note
`trace_block`	Only for the last 10064 blocks
`trace_call`	Only for the last 10064 blocks
`trace_callMany`	Only for the last 10064 blocks
`trace_get`	Only for the last 10064 blocks
`trace_rawTransaction`	Only for the last 10064 blocks
`trace_replayBlockTransactions`	Only for the last 10064 blocks
`trace_replayTransaction`	Only for the last 10064 blocks
`trace_transaction`	Only for the last 10064 blocks

`txpool` namespace

RPC / Segment
`txpool_content`
`txpool_contentFrom`
`txpool_inspect`
`txpool_status`

Pruned Node

The following tables describe the requirements for prune segments, per RPC method:

- ✅ – if the segment is pruned, the RPC method still works
- ❌ - if the segment is pruned, the RPC method doesn't work anymore

`debug` namespace

RPC / Segment	Sender	Transaction	Receipts	Accou

	Recovery	Lookup		Histo
`debug_getRawBlock`	✅	✅	✅	✅
`debug_getRawHeader`	✅	✅	✅	✅
`debug_getRawReceipts`	✅	✅	❌	✅
`debug_getRawTransaction`	✅	❌	✅	✅
`debug_traceBlock`	✅	✅	✅	❌
`debug_traceBlockByHash`	✅	✅	✅	❌
`debug_traceBlockByNumber`	✅	✅	✅	❌
`debug_traceCall`	✅	✅	✅	❌
`debug_traceCallMany`	✅	✅	✅	❌
`debug_traceTransaction`	✅	✅	✅	❌

`eth` namespace

RPC / Segment	Sender Recovery	Transaction Lookup	Rec
`eth_accounts`	✅	✅	✅
`eth_blockNumber`	✅	✅	✅
`eth_call`	✅	✅	✅
`eth_chainId`	✅	✅	✅
`eth_createAccessList`	✅	✅	✅
`eth_estimateGas`	✅	✅	✅
`eth_feeHistory`	✅	✅	✅
`eth_gasPrice`	✅	✅	✅
`eth_getBalance`	✅	✅	✅
`eth_getBlockByHash`	✅	✅	✅
`eth_getBlockByNumber`	✅	✅	✅
`eth_getBlockReceipts`	✅	✅	❌
`eth_getBlockTransactionCountByHash`	✅	✅	✅
`eth_getBlockTransactionCountByNumber`	✅	✅	✅
`eth_getCode`	✅	✅	✅
`eth_getFilterChanges`	✅	✅	✅
`eth_getFilterLogs`	✅	✅	❌
`eth_getLogs`	✅	✅	❌
`eth_getStorageAt`	✅	✅	✅

RPC / Segment	Sender Recovery	Transaction Lookup	Rec
`eth_getTransactionByBlockHashAndIndex`	✅	✅	✅
`eth_getTransactionByBlockNumberAndIndex`	✅	✅	✅
`eth_getTransactionByHash`	✅	❌	✅
`eth_getTransactionCount`	✅	✅	✅
`eth_getTransactionReceipt`	✅	❌	❌
`eth_getUncleByBlockHashAndIndex`	✅	✅	✅
`eth_getUncleByBlockNumberAndIndex`	✅	✅	✅
`eth_getUncleCountByBlockHash`	✅	✅	✅
`eth_getUncleCountByBlockNumber`	✅	✅	✅
`eth_maxPriorityFeePerGas`	✅	✅	✅
`eth_mining`	✅	✅	✅
`eth_newBlockFilter`	✅	✅	✅
`eth_newFilter`	✅	✅	✅
`eth_newPendingTransactionFilter`	✅	✅	✅
`eth_protocolVersion`	✅	✅	✅
`eth_sendRawTransaction`	✅	✅	✅
`eth_sendTransaction`	✅	✅	✅
`eth_sign`	✅	✅	✅
`eth_signTransaction`	✅	✅	✅
`eth_signTypedData`	✅	✅	✅
`eth_subscribe`	✅	✅	✅
`eth_syncing`	✅	✅	✅
`eth_uninstallFilter`	✅	✅	✅
`eth_unsubscribe`	✅	✅	✅

`net` namespace

RPC / Segment	Sender Recovery	Transaction Lookup	Receipts	Account History	Stor Hist
`net_listening`	✅	✅	✅	✅	✅
`net_peerCount`	✅	✅	✅	✅	✅
`net_version`	✅	✅	✅	✅	✅

`trace` namespace

RPC / Segment	Sender Recovery	Transaction Lookup	Receipts
`trace_block`	✅	✅	✅
`trace_call`	✅	✅	✅
`trace_callMany`	✅	✅	✅
`trace_get`	✅	❌	✅
`trace_rawTransaction`	✅	✅	✅
`trace_replayBlockTransactions`	✅	✅	✅
`trace_replayTransaction`	✅	❌	✅
`trace_transaction`	✅	❌	✅

`txpool` namespace

RPC / Segment	Sender Recovery	Transaction Lookup	Receipts	Account History
`txpool_content`	✅	✅	✅	✅
`txpool_contentFrom`	✅	✅	✅	✅
`txpool_inspect`	✅	✅	✅	✅
`txpool_status`	✅	✅	✅	✅

Ports

This section provides essential information about the ports used by the system, their primary purposes, and recommendations for exposure settings.

Peering Ports

- **Port:** 30303
- **Protocol:** TCP and UDP
- **Purpose:** Peering with other nodes for synchronization of blockchain data. Nodes communicate through this port to maintain network consensus and share updated information.
- **Exposure Recommendation:** This port should be exposed to enable seamless interaction and synchronization with other nodes in the network.

Metrics Port

- **Port:** 9001
- **Protocol:** TCP
- **Purpose:** This port is designated for serving metrics related to the system's performance and operation. It allows internal monitoring and data collection for analysis.
- **Exposure Recommendation:** By default, this port should not be exposed to the public. It is intended for internal monitoring and analysis purposes.

HTTP RPC Port

- **Port:** 8545
- **Protocol:** TCP
- **Purpose:** Port 8545 provides an HTTP-based Remote Procedure Call (RPC) interface. It enables external applications to interact with the blockchain by sending requests over HTTP.
- **Exposure Recommendation:** Similar to the metrics port, exposing this port to the public is not recommended by default due to security considerations.

WS RPC Port

- **Port:** 8546
- **Protocol:** TCP
- **Purpose:** Port 8546 offers a WebSocket-based Remote Procedure Call (RPC) interface. It allows real-time communication between external applications and the blockchain.
- **Exposure Recommendation:** As with the HTTP RPC port, the WS RPC port should not be exposed by default for security reasons.

Engine API Port

- **Port:** 8551
- **Protocol:** TCP
- **Purpose:** Port 8551 facilitates communication between specific components, such as "reth" and "CL" (assuming their definitions are understood within the context of the system). It enables essential internal processes.
- **Exposure Recommendation:** This port is not meant to be exposed to the public by default. It should be reserved for internal communication between vital components of the system.

Troubleshooting

As Reth is still in alpha, while running the node you can experience some problems related to different parts of the system: pipeline sync, blockchain tree, p2p, database, etc.

This page tries to answer how to deal with the most popular issues.

Database

Database write error

If you encounter an irrecoverable database-related errors, in most of the cases it's related to the RAM/NVMe/SSD you use. For example:

```
Error: A stage encountered an irrecoverable error.

Caused by:
   0: An internal database error occurred: Database write error code: -30796
   1: Database write error code: -30796
```

or

```
Error: A stage encountered an irrecoverable error.

Caused by:
   0: An internal database error occurred: Database read error code: -30797
   1: Database read error code: -30797
```

1. Check your memory health: use memtest86+ or memtester. If your memory is faulty, it's better to resync the node on different hardware.
2. Check database integrity:

   ```
   git clone https://github.com/paradigmxyz/reth
   cd reth
   make db-tools
   db-tools/mdbx_chk $(reth db path)/mdbx.dat | tee mdbx_chk.log
   ```

   If `mdbx_chk` has detected any errors, please open an issue and post the output from the `mdbx_chk.log` file.

JSON-RPC

You can interact with Reth over JSON-RPC. Reth supports all standard Ethereum JSON-RPC API methods.

JSON-RPC is provided on multiple transports. Reth supports HTTP, WebSocket and IPC (both UNIX sockets and Windows named pipes). Transports must be enabled through command-line flags.

The JSON-RPC APIs are grouped into namespaces, depending on their purpose. All method names are composed of their namespace and their name, separated by an underscore.

Each namespace must be explicitly enabled.

Namespaces

The methods are grouped into namespaces, which are listed below:

Namespace	Description	Sensitive
`eth`	The `eth` API allows you to interact with Ethereum.	Maybe
`web3`	The `web3` API provides utility functions for the web3 client.	No
`net`	The `net` API provides access to network information of the node.	No
`txpool`	The `txpool` API allows you to inspect the transaction pool.	No
`debug`	The `debug` API provides several methods to inspect the Ethereum state, including Geth-style traces.	No
`trace`	The `trace` API provides several methods to inspect the Ethereum state, including Parity-style traces.	No
`admin`	The `admin` API allows you to configure your node.	**Yes**
`rpc`	The `rpc` API provides information about the RPC server and its modules.	No

Note that some APIs are sensitive, since they can be used to configure your node (`admin`), or access accounts stored on the node (`eth`).

Generally, it is advisable to not expose any JSONRPC namespace publicly, unless you know what you are doing.

Transports

Reth supports HTTP, WebSockets and IPC.

HTTP

Using the HTTP transport, clients send a request to the server and immediately get a response back. The connection is closed after the response for a given request is sent.

Because HTTP is unidirectional, subscriptions are not supported.

To start an HTTP server, pass `--http` to `reth node`:

```
reth node --http
```

The default port is `8545`, and the default listen address is localhost.

You can configure the listen address and port using `--http.addr` and `--http.port` respectively:

```
reth node --http --http.addr 127.0.0.1 --http.port 12345
```

To enable JSON-RPC namespaces on the HTTP server, pass each namespace separated by a comma to `--http.api`:

```
reth node --http --http.api eth,net,trace
```

You can pass the `all` option, which is a convenient wrapper for the all the JSON-RPC namespaces `admin,debug,eth,net,trace,txpool,web3,rpc` on the HTTP server:

```
reth node --http --http.api all
```

```
reth node --http --http.api All
```

You can also restrict who can access the HTTP server by specifying a domain for Cross-Origin requests. This is important, since any application local to your node will be able to access the RPC server:

```
reth node --http --http.corsdomain https://mycoolapp.rs
```

Alternatively, if you want to allow any domain, you can pass `*`:

```
reth node --http --http.corsdomain "*"
```

WebSockets

WebSockets is a bidirectional transport protocol. Most modern browsers support WebSockets.

A WebSocket connection is maintained until it is explicitly terminated by either the client or the node.

Because WebSockets are bidirectional, nodes can push events to clients, which enables clients to subscribe to specific events, such as new transactions in the transaction pool, and new logs for smart contracts.

The configuration of the WebSocket server follows the same pattern as the HTTP server:

- Enable it using `--ws`
- Configure the server address by passing `--ws.addr` and `--ws.port` (default `8546`)
- Configure cross-origin requests using `--ws.origins`
- Enable APIs using `--ws.api`

IPC

IPC is a simpler transport protocol for use in local environments where the node and the client exist on the same machine.

The IPC transport is enabled by default and has access to all namespaces, unless explicitly disabled with `--ipcdisable`.

Reth creates a UNIX socket on Linux and macOS at `/tmp/reth.ipc`. On Windows, IPC is provided using named pipes at `\\.\pipe\reth.ipc`.

You can configure the IPC path using `--ipcpath`.

Interacting with the RPC

One can easily interact with these APIs just like they would with any Ethereum client.

You can use `curl`, a programming language with a low-level library, or a tool like Foundry to interact with the chain at the exposed HTTP or WS port.

As a reminder, you need to run the command below to enable all of these APIs using an

HTTP transport:

```
RUST_LOG=info reth node --http --http.api
"admin,debug,eth,net,trace,txpool,web3,rpc"
```

This allows you to then call:

```
cast block-number
cast rpc admin_nodeInfo
cast rpc debug_traceTransaction
cast rpc trace_replayBlockTransactions
```

`eth` Namespace

Documentation for the API methods in the `eth` namespace can be found on ethereum.org.

`web3` Namespace

The `web3` API provides utility functions for the web3 client.

`web3_clientVersion`

Get the web3 client version.

Client	Method invocation
RPC	`{"method": "web3_clientVersion"}`

Example

```
// > {"jsonrpc":"2.0","id":1,"method":"web3_clientVersion","params":[]}
{"jsonrpc":"2.0","id":1,"result":"reth/v0.0.1/x86_64-unknown-linux-gnu"}
```

`web3_sha3`

Get the Keccak-256 hash of the given data.

Client	Method invocation
RPC	`{"method": "web3_sha3", "params": [bytes]}`

Example

```
// > {"jsonrpc":"2.0","id":1,"method":"web3_sha3","params":["rust is
awesome"]}
{"jsonrpc":"2.0","id":1,"result":"0xe421b3428564a5c509ac118bad93a3b84485ec3f927
```

`net` Namespace

The `net` API provides information about the networking component of the node.

`net_listening`

Returns a `bool` indicating whether or not the node is listening for network connections.

Client	Method invocation
RPC	`{"method": "net_listening", "params": []}`

Example

```
// > {"jsonrpc":"2.0","id":1,"method":"net_listening","params":[]}
{"jsonrpc":"2.0","id":1,"result":true}
```

`net_peerCount`

Returns the number of peers connected to the node.

Client	Method invocation
RPC	`{"method": "net_peerCount", "params": []}`

Example

```
// > {"jsonrpc":"2.0","id":1,"method":"net_peerCount","params":[]}
{"jsonrpc":"2.0","id":1,"result":10}
```

`net_version`

Returns the network ID (e.g. 1 for mainnet)

Client	Method invocation
RPC	`{"method": "net_version", "params": []}`

Example

```
// > {"jsonrpc":"2.0","id":1,"method":"net_version","params":[]}
{"jsonrpc":"2.0","id":1,"result":1}
```

`txpool` Namespace

The `txpool` API allows you to inspect the transaction pool.

`txpool_content`

Returns the details of all transactions currently pending for inclusion in the next block(s), as well as the ones that are being scheduled for future execution only.

See here for more details

Client	Method invocation
RPC	`{"method": "txpool_content", "params": []}`

`txpool_contentFrom`

Retrieves the transactions contained within the txpool, returning pending as well as queued transactions of this address, grouped by nonce.

See here for more details

Client	Method invocation
RPC	`{"method": "txpool_contentFrom", "params": [address]}`

`txpool_inspect`

Returns a summary of all the transactions currently pending for inclusion in the next block(s), as well as the ones that are being scheduled for future execution only.

See here for more details

Client	Method invocation
RPC	`{"method": "txpool_inspect", "params": []}`

`txpool_status`

Returns the number of transactions currently pending for inclusion in the next block(s), as well as the ones that are being scheduled for future execution only.

See here for more details

Client	Method invocation
RPC	`{"method": "txpool_status", "params": []}`

`debug` Namespace

The `debug` API provides several methods to inspect the Ethereum state, including Geth-style traces.

`debug_getRawHeader`

Returns an RLP-encoded header.

Client	Method invocation
RPC	`{"method": "debug_getRawHeader", "params": [block]}`

`debug_getRawBlock`

Retrieves and returns the RLP encoded block by number, hash or tag.

Client	Method invocation
RPC	`{"method": "debug_getRawBlock", "params": [block]}`

`debug_getRawTransaction`

Returns an EIP-2718 binary-encoded transaction.

Client	Method invocation
RPC	`{"method": "debug_getRawTransaction", "params": [tx_hash]}`

`debug_getRawReceipts`

Returns an array of EIP-2718 binary-encoded receipts.

Client	Method invocation
RPC	`{"method": "debug_getRawReceipts", "params": [block]}`

debug_getBadBlocks

Returns an array of recent bad blocks that the client has seen on the network.

Client	Method invocation
RPC	`{"method": "debug_getBadBlocks", "params": []}`

debug_traceChain

Returns the structured logs created during the execution of EVM between two blocks (excluding start) as a JSON object.

Client	Method invocation
RPC	`{"method": "debug_traceChain", "params": [start_block, end_block]}`

debug_traceBlock

The `debug_traceBlock` method will return a full stack trace of all invoked opcodes of all transaction that were included in this block.

This expects an RLP-encoded block.

Note

The parent of this block must be present, or it will fail.

Client	Method invocation
RPC	`{"method": "debug_traceBlock", "params": [rlp, opts]}`

debug_traceBlockByHash

Similar to `debug_traceBlock`, `debug_traceBlockByHash` accepts a block hash and will replay the block that is already present in the database.

Client	Method invocation

Client	Method invocation
RPC	`{"method": "debug_traceBlockByHash", "params": [block_hash, opts]}`

debug_traceBlockByNumber

Similar to `debug_traceBlockByHash`, `debug_traceBlockByNumber` accepts a block number and will replay the block that is already present in the database.

Client	Method invocation
RPC	`{"method": "debug_traceBlockByNumber", "params": [block_number, opts]}`

debug_traceTransaction

The `debug_traceTransaction` debugging method will attempt to run the transaction in the exact same manner as it was executed on the network. It will replay any transaction that may have been executed prior to this one before it will finally attempt to execute the transaction that corresponds to the given hash.

Client	Method invocation
RPC	`{"method": "debug_traceTransaction", "params": [tx_hash, opts]}`

debug_traceCall

The `debug_traceCall` method lets you run an `eth_call` within the context of the given block execution using the final state of parent block as the base.

The first argument (just as in `eth_call`) is a transaction request.

The block can optionally be specified either by hash or by number as the second argument.

Client	Method invocation
RPC	`{"method": "debug_traceCall", "params": [call, block_number, opts]}`

`trace` Namespace

The `trace` API provides several methods to inspect the Ethereum state, including Parity-style traces.

A similar module exists (with other debug functions) with Geth-style traces (`debug`).

The `trace` API gives deeper insight into transaction processing.

There are two types of methods in this API:

- **Ad-hoc tracing APIs** for performing diagnostics on calls or transactions (historical or hypothetical).
- **Transaction-trace filtering APIs** for getting full externality traces on any transaction executed by reth.

Ad-hoc tracing APIs

Ad-hoc tracing APIs allow you to perform diagnostics on calls or transactions (historical or hypothetical), including:

- Transaction traces (`trace`)
- VM traces (`vmTrace`)
- State difference traces (`stateDiff`)

The ad-hoc tracing APIs are:

- `trace_call`
- `trace_callMany`
- `trace_rawTransaction`
- `trace_replayBlockTransactions`
- `trace_replayTransaction`

Transaction-trace filtering APIs

Transaction trace filtering APIs are similar to log filtering APIs in the `eth` namespace, except these allow you to search and filter based only upon address information.

Information returned includes the execution of all contract creations, destructions, and calls, together with their input data, output data, gas usage, transfer amounts and success statuses.

The transaction trace filtering APIs are:

- `trace_block`
- `trace_filter`
- `trace_get`
- `trace_transaction`

`trace_call`

Executes the given call and returns a number of possible traces for it.

The first parameter is a transaction object where the `from` field is optional and the `nonce` field is ommitted.

The second parameter is an array of one or more trace types (`vmTrace` , `trace` , `stateDiff`).

The third and optional parameter is a block number, block hash, or a block tag (`latest` , `finalized` , `safe` , `earliest` , `pending`).

Client	Method invocation
RPC	`{"method": "trace_call", "params": [tx, type[], block]}`

Example

```
// > {"jsonrpc":"2.0","id":1,"method":"trace_call","params":[{},["trace"]}
{
    "id": 1,
    "jsonrpc": "2.0",
    "result": {
        "output": "0x",
        "stateDiff": null,
        "trace": [{
            "action": { ... },
            "result": {
                "gasUsed": "0x0",
                "output": "0x"
            },
            "subtraces": 0,
            "traceAddress": [],
            "type": "call"
        }],
        "vmTrace": null
    }
}
```

`trace_callMany`

Performs multiple call traces on top of the same block, that is, transaction `n` will be executed on top of a pending block with all `n - 1` transaction applied (and traced) first.

The first parameter is a list of call traces, where each call trace is of the form `[tx, type[]]` (see `trace_call`).

The second and optional parameter is a block number, block hash, or a block tag (`latest`, `finalized`, `safe`, `earliest`, `pending`).

Client	Method invocation
RPC	`{"method": "trace_call", "params": [trace[], block]}`

Example

```
// > {"jsonrpc":"2.0","id":1,"method":"trace_callMany","params":
[[[{"from":"0x407d73d8a49eeb85d32cf465507dd71d507100c1","to":"0xa94f5374fce5edb
["trace"]],
[{"from":"0x407d73d8a49eeb85d32cf465507dd71d507100c1","to":"0xa94f5374fce5edbc8
["trace"]]],"latest"]}
{
    "id": 1,
    "jsonrpc": "2.0",
    "result": [
        {
            "output": "0x",
            "stateDiff": null,
            "trace": [{
                "action": {
                    "callType": "call",
                    "from": "0x407d73d8a49eeb85d32cf465507dd71d507100c1",
                    "gas": "0x1dcd12f8",
                    "input": "0x",
                    "to": "0xa94f5374fce5edbc8e2a8697c15331677e6ebf0b",
                    "value": "0x186a0"
                },
                "result": {
                    "gasUsed": "0x0",
                    "output": "0x"
                },
                "subtraces": 0,
                "traceAddress": [],
                "type": "call"
            }],
            "vmTrace": null
        },
        {
            "output": "0x",
            "stateDiff": null,
            "trace": [{
                "action": {
                    "callType": "call",
                    "from": "0x407d73d8a49eeb85d32cf465507dd71d507100c1",
                    "gas": "0x1dcd12f8",
                    "input": "0x",
                    "to": "0xa94f5374fce5edbc8e2a8697c15331677e6ebf0b",
                    "value": "0x186a0"
                },
                "result": {
                    "gasUsed": "0x0",
                    "output": "0x"
                },
                "subtraces": 0,
                "traceAddress": [],
                "type": "call"
            }],
            "vmTrace": null
        }
    ]
}
```

`trace_rawTransaction`

Traces a call to `eth_sendRawTransaction` without making the call, returning the traces.

Client	Method invocation
RPC	`{"method": "trace_call", "params": [raw_tx, type[]]}`

Example

```
// > {"jsonrpc":"2.0","id":1,"method":"trace_rawTransaction","params":
["0xd46e8dd67c5d32be8d46e8dd67c5d32be8058bb8eb970870f072445675058bb8eb970870f07
["trace"]]}
{
    "id": 1,
    "jsonrpc": "2.0",
    "result": {
        "output": "0x",
            "stateDiff": null,
            "trace": [{
            "action": { ... },
            "result": {
                "gasUsed": "0x0",
                "output": "0x"
            },
            "subtraces": 0,
            "traceAddress": [],
            "type": "call"
        }],
            "vmTrace": null
    }
}
```

`trace_replayBlockTransactions`

Replays all transactions in a block returning the requested traces for each transaction.

Client	Method invocation
RPC	`{"method": "trace_replayBlockTransactions", "params": [block, type[]]}`

Example

```
// >
{"jsonrpc":"2.0","id":1,"method":"trace_replayBlockTransactions","params":
["0x2ed119",["trace"]]}
{
    "id": 1,
    "jsonrpc": "2.0",
    "result": [
        {
            "output": "0x",
            "stateDiff": null,
            "trace": [{
                "action": { ... },
                "result": {
                    "gasUsed": "0x0",
                    "output": "0x"
                },
                "subtraces": 0,
                "traceAddress": [],
                "type": "call"
            }],
            "transactionHash": "0x...",
            "vmTrace": null
        },
        { ... }
    ]
}
```

`trace_replayTransaction`

Replays a transaction, returning the traces.

Client	Method invocation
RPC	`{"method": "trace_replayTransaction", "params": [tx_hash, type[]]}`

Example

```
// > {"jsonrpc":"2.0","id":1,"method":"trace_replayTransaction","params":
["0x02d4a872e096445e80d05276ee756cefef7f3b376bcec14246469c0cd97dad8f",
["trace"]]}
{
    "id": 1,
    "jsonrpc": "2.0",
    "result": {
        "output": "0x",
        "stateDiff": null,
        "trace": [{
            "action": { ... },
            "result": {
                "gasUsed": "0x0",
                "output": "0x"
            },
            "subtraces": 0,
            "traceAddress": [],
            "type": "call"
        }],
        "vmTrace": null
    }
}
```

`trace_block`

Returns traces created at given block.

Client	Method invocation
RPC	`{"method": "trace_block", "params": [block]}`

Example

```
// > {"jsonrpc":"2.0","id":1,"method":"trace_block","params":["0x2ed119"]}
{
    "id": 1,
    "jsonrpc": "2.0",
    "result": [
        {
            "action": {
                "callType": "call",
                "from": "0xaa7b131dc60b80d3cf5e59b5a21a666aa039c951",
                "gas": "0x0",
                "input": "0x",
                "to": "0xd40aba8166a212d6892125f079c33e6f5ca19814",
                "value": "0x4768d7effc3fbe"
            },
            "blockHash":
"0x7eb25504e4c202cf3d62fd585d3e238f592c780cca82dacb2ed3cb5b38883add",
            "blockNumber": 3068185,
            "result": {
                "gasUsed": "0x0",
                "output": "0x"
            },
            "subtraces": 0,
            "traceAddress": [],
            "transactionHash":
"0x07da28d752aba3b9dd7060005e554719c6205c8a3aea358599fc9b245c52f1f6",
            "transactionPosition": 0,
            "type": "call"
        },
        ...
    ]
}
```

trace_filter

Returns traces matching given filter.

Filters are objects with the following properties:

- `fromBlock` : Returns traces from the given block (a number, hash, or a tag like `latest`).
- `toBlock` : Returns traces to the given block.
- `fromAddress` : Sent from these addresses
- `toAddress` : Sent to these addresses
- `after` : The offset trace number
- `count` : The number of traces to display in a batch

All properties are optional.

Client	Method invocation

Client	Method invocation
RPC	`{"method": "trace_filter", "params": [filter]}`

Example

```
// > {"jsonrpc":"2.0","id":1,"method":"trace_filter","params":
[{"fromBlock":"0x2ed0c4","toBlock":"0x2ed128","toAddress":
["0x8bbB73BCB5d553B5A556358d27625323Fd781D37"],"after":1000,"count":100}]}
{
    "id": 1,
    "jsonrpc": "2.0",
    "result": [
        {
            "action": {
                "callType": "call",
                "from": "0x32be343b94f860124dc4fee278fdcbd38c102d88",
                "gas": "0x4c40d",
                "input": "0x",
                "to": "0x8bbb73bcb5d553b5a556358d27625323fd781d37",
                "value": "0x3f0650ec47fd240000"
            },
            "blockHash":
"0x86df301bcdd8248d982dbf039f09faf792684e1aeee99d5b58b77d620008b80f",
            "blockNumber": 3068183,
            "result": {
                "gasUsed": "0x0",
                "output": "0x"
            },
            "subtraces": 0,
            "traceAddress": [],
            "transactionHash":
"0x3321a7708b1083130bd78da0d62ead9f6683033231617c9d268e2c7e3fa6c104",
            "transactionPosition": 3,
            "type": "call"
        },
        ...
    ]
}
```

`trace_get`

Returns trace at given position.

Client	Method invocation
RPC	`{"method": "trace_get", "params": [tx_hash,indices[]]}`

Example

```
// > {"jsonrpc":"2.0","id":1,"method":"trace_get","params":
["0x17104ac9d3312d8c136b7f44d4b8b47852618065ebfa534bd2d3b5ef218ca1f3",
["0x0"]]}
{
    "id": 1,
    "jsonrpc": "2.0",
    "result": {
        "action": {
            "callType": "call",
            "from": "0x1c39ba39e4735cb65978d4db400ddd70a72dc750",
            "gas": "0x13e99",
            "input": "0x16c72721",
            "to": "0x2bd2326c993dfaef84f696526064ff22eba5b362",
            "value": "0x0"
        },
        "blockHash":
"0x7eb25504e4c202cf3d62fd585d3e238f592c780cca82dacb2ed3cb5b38883add",
            "blockNumber": 3068185,
            "result": {
            "gasUsed": "0x183",
            "output":
"0x0000000000000000000000000000000000000000000000000000000000000001"
        },
        "subtraces": 0,
            "traceAddress": [
            0
        ],
        "transactionHash":
"0x17104ac9d3312d8c136b7f44d4b8b47852618065ebfa534bd2d3b5ef218ca1f3",
        "transactionPosition": 2,
        "type": "call"
    }
}
```

`trace_transaction`

Returns all traces of given transaction

Client	Method invocation
RPC	`{"method": "trace_transaction", "params": [tx_hash]}`

Example

```
// > {"jsonrpc":"2.0","id":1,"method":"trace_transaction","params":
["0x17104ac9d3312d8c136b7f44d4b8b47852618065ebfa534bd2d3b5ef218ca1f3"]}
{
    "id": 1,
    "jsonrpc": "2.0",
    "result": [
        {
            "action": {
                "callType": "call",
                "from": "0x1c39ba39e4735cb65978d4db400ddd70a72dc750",
                "gas": "0x13e99",
                "input": "0x16c72721",
                "to": "0x2bd2326c993dfaef84f696526064ff22eba5b362",
                "value": "0x0"
            },
            "blockHash":
"0x7eb25504e4c202cf3d62fd585d3e238f592c780cca82dacb2ed3cb5b38883add",
            "blockNumber": 3068185,
            "result": {
                "gasUsed": "0x183",
                "output":
"0x0000000000000000000000000000000000000000000000000000000000000001"
            },
            "subtraces": 0,
            "traceAddress": [
                0
            ],
            "transactionHash":
"0x17104ac9d3312d8c136b7f44d4b8b47852618065ebfa534bd2d3b5ef218ca1f3",
            "transactionPosition": 2,
            "type": "call"
        },
        ...
    ]
}
```

`admin` Namespace

The `admin` API allows you to configure your node, including adding and removing peers.

> **Note**
>
> As this namespace can configure your node at runtime, it is generally **not advised** to expose it publicly.

`admin_addPeer`

Add the given peer to the current peer set of the node.

The method accepts a single argument, the `enode` URL of the remote peer to connect to, and returns a `bool` indicating whether the peer was accepted or not.

Client	Method invocation
RPC	`{"method": "admin_addPeer", "params": [url]}`

Example

```
// > {"jsonrpc":"2.0","id":1,"method":"admin_addPeer","params":
["enode://a979fb575495b8d6db44f750317d0f4622bf4c2aa3365d6af7c284339968eef29b69a
{"jsonrpc":"2.0","id":1,"result":true}
```

`admin_removePeer`

Disconnects from a peer if the connection exists. Returns a `bool` indicating whether the peer was successfully removed or not.

Client	Method invocation
RPC	`{"method": "admin_removePeer", "params": [url]}`

Example

```
// > {"jsonrpc":"2.0","id":1,"method":"admin_removePeer","params":
["enode://a979fb575495b8d6db44f750317d0f4622bf4c2aa3365d6af7c284339968eef29b69a
{"jsonrpc":"2.0","id":1,"result":true}
```

admin_addTrustedPeer

Adds the given peer to a list of trusted peers, which allows the peer to always connect, even if there would be no room for it otherwise.

It returns a `bool` indicating whether the peer was added to the list or not.

Client	Method invocation
RPC	`{"method": "admin_addTrustedPeer", "params": [url]}`

Example

```
// > {"jsonrpc":"2.0","id":1,"method":"admin_addTrustedPeer","params":
["enode://a979fb575495b8d6db44f750317d0f4622bf4c2aa3365d6af7c284339968eef29b69a
{"jsonrpc":"2.0","id":1,"result":true}
```

admin_removeTrustedPeer

Removes a remote node from the trusted peer set, but it does not disconnect it automatically.

Returns true if the peer was successfully removed.

Client	Method invocation
RPC	`{"method": "admin_removeTrustedPeer", "params": [url]}`

Example

```
// > {"jsonrpc":"2.0","id":1,"method":"admin_removeTrustedPeer","params":
["enode://a979fb575495b8d6db44f750317d0f4622bf4c2aa3365d6af7c284339968eef29b69a
{"jsonrpc":"2.0","id":1,"result":true}
```

admin_nodeInfo

Returns all information known about the running node.

These include general information about the node itself, as well as what protocols it participates in, its IP and ports.

Client	Method invocation
RPC	`{"method": "admin_nodeInfo"}`

Example

```
// > {"jsonrpc":"2.0","id":1,"method":"admin_nodeInfo","params":[]}
{
    "jsonrpc": "2.0",
    "id": 1,
    "result": {
        "enode":
"enode://44826a5d6a55f88a18298bca4773fca5749cdc3a5c9f308aa7d810e9b31123f3e7c5fb
            "id":
"44826a5d6a55f88a18298bca4773fca5749cdc3a5c9f308aa7d810e9b31123f3e7c5fba0b1d70a
            "ip": "::",
            "listenAddr": "[::]:30303",
            "name": "reth/v0.0.1/x86_64-unknown-linux-gnu",
            "ports": {
                "discovery": 30303,
                "listener": 30303
        },
        "protocols": {
            "eth": {
                "difficulty": 17334254859343145000,
                "genesis":
"0xd4e56740f876aef8c010b86a40d5f56745a118d0906a34e69aec8c0db1cb8fa3",
                "head":
"0xb83f73fbe6220c111136aefd27b160bf4a34085c65ba89f24246b3162257c36a",
                "network": 1
            }
        }
    }
}
```

admin_peerEvents, admin_peerEvents_unsubscribe

Subscribe to events received by peers over the network.

Like other subscription methods, this returns the ID of the subscription, which is then used in all events subsequently.

To unsubscribe from peer events, call `admin_peerEvents_unsubscribe`

Client	Method invocation
RPC	`{"method": "admin_peerEvents"}`

Example

```
// > {"jsonrpc":"2.0","id":1,"method":"admin_peerEvents","params":[]}
// responds with subscription ID
{"jsonrpc": "2.0", "id": 1, "result": "0xcd0c3e8af590364c09d0fa6a1210faf5"}
```

`rpc` Namespace

The `rpc` API provides methods to get information about the RPC server itself, such as the enabled namespaces.

`rpc_modules`

Lists the enabled RPC namespaces and the versions of each.

Client	Method invocation
RPC	`{"method": "rpc_modules", "params": []}`

Example

```
// > {"jsonrpc":"2.0","id":1,"method":"rpc_modules","params":[]}
{"jsonrpc":"2.0","id":1,"result":{"txpool":"1.0","eth":"1.0","rpc":"1.0"}}
```

Handling Responses During Syncing

When interacting with the RPC server while it is still syncing, some RPC requests may return an empty or null response, while others return the expected results. This behavior can be observed due to the asynchronous nature of the syncing process and the availability of required data. Notably, endpoints that rely on specific stages of the syncing process, such as the execution stage, might not be available until those stages are complete.

It's important to understand that during pipeline sync, some endpoints may not be accessible until the necessary data is fully synchronized. For instance, the `eth_getBlockReceipts` endpoint is only expected to return valid data after the execution stage, where receipts are generated, has completed. As a result, certain RPC requests may return empty or null responses until the respective stages are finished.

This behavior is intrinsic to how the syncing mechanism works and is not indicative of an issue or bug. If you encounter such responses while the node is still syncing, it's recommended to wait until the sync process is complete to ensure accurate and expected RPC responses.

CLI Reference

The Reth node is operated via the CLI by running the `reth node` command. To stop it, press `ctrl-c`. You may need to wait a bit as Reth tears down existing p2p connections or other cleanup tasks.

However, Reth has more commands than that:

```
reth --help
```

Some of the most useful commands as a node developer are:

- `reth node`: Starts the Reth node's components, including the JSON-RPC.
- `reth init`: Initialize the database from a genesis file.
- `reth import`: This syncs RLP encoded blocks from a file.
- `reth db`: Administrative TUI to the key-value store.
- `reth stage`: Runs a stage in isolation. Useful for testing and benchmarking.
- `reth p2p`: P2P-related utilities
- `reth test-vectors`: Generate Test Vectors
- `reth config`: Write config to stdout
- `reth debug`: Various debug routines

See below for the full list of commands.

Commands

```
$ reth --help
Reth

Usage: reth [OPTIONS] <COMMAND>

Commands:
  node          Start the node
  init          Initialize the database from a genesis file
  import        This syncs RLP encoded blocks from a file
  db            Database debugging utilities
  stage         Manipulate individual stages
  p2p           P2P Debugging utilities
  test-vectors  Generate Test Vectors
  config        Write config to stdout
  debug         Various debug routines
  recover       Scripts for node recovery
  help          Print this message or the help of the given subcommand(s)

Options:
      --chain <CHAIN_OR_PATH>
          The chain this node is running.

          Possible values are either a built-in chain or the path to a chain
specification file.

          Built-in chains:
          - mainnet
          - goerli
          - sepolia
          - holesky

          [default: mainnet]

      --instance <INSTANCE>
          Add a new instance of a node.

          Configures the ports of the node to avoid conflicts with the
defaults. This is useful for running multiple nodes on the same machine.

          Max number of instances is 200. It is chosen in a way so that it's
not possible to have port numbers that conflict with each other.

          Changes to the following port numbers: - DISCOVERY_PORT: default +
`instance` - 1 - AUTH_PORT: default + `instance` * 100 - 100 - HTTP_RPC_PORT:
default - `instance` + 1 - WS_RPC_PORT: default + `instance` * 2 - 2

          [default: 1]

  -h, --help
          Print help (see a summary with '-h')

  -V, --version
          Print version

Logging:
      --log.directory <PATH>
```

```
          The path to put log files in

          [default: /reth/logs]

      --log.max-size <SIZE>
          The maximum size (in MB) of log files

          [default: 200]

      --log.max-files <COUNT>
          The maximum amount of log files that will be stored. If set to 0,
background file logging is disabled

          [default: 5]

      --log.journald
          Log events to journald

      --log.filter <FILTER>
          The filter to use for logs written to the log file

          [default: error]

      --color <COLOR>
          Sets whether or not the formatter emits ANSI terminal escape codes
for colors and other text formatting

          [default: always]

          Possible values:
          - always: Colors on
          - auto:   Colors on
          - never:  Colors off

Display:
  -v, --verbosity...
          Set the minimum log level.

          -v      Errors
          -vv     Warnings
          -vvv    Info
          -vvvv   Debug
          -vvvvv  Traces (warning: very verbose!)

  -q, --quiet
          Silence all log output
```

`reth node`

Start the node

```
$ reth node --help

Usage: reth node [OPTIONS]

Options:
      --datadir <DATA_DIR>
          The path to the data dir for all reth files and subdirectories.

          Defaults to the OS-specific data directory:

          - Linux: `$XDG_DATA_HOME/reth/` or `$HOME/.local/share/reth/`
          - Windows: `{FOLDERID_RoamingAppData}/reth/`
          - macOS: `$HOME/Library/Application Support/reth/`

          [default: default]

      --config <FILE>
          The path to the configuration file to use.

      --chain <CHAIN_OR_PATH>
          The chain this node is running.

          Possible values are either a built-in chain or the path to a chain
specification file.

          Built-in chains:
          - mainnet
          - goerli
          - sepolia
          - holesky
          - dev

          [default: mainnet]

      --instance <INSTANCE>
          Add a new instance of a node.

          Configures the ports of the node to avoid conflicts with the
defaults. This is useful for running multiple nodes on the same machine.

          Max number of instances is 200. It is chosen in a way so that it's
not possible to have port numbers that conflict with each other.

          Changes to the following port numbers: - DISCOVERY_PORT: default +
`instance` - 1 - AUTH_PORT: default + `instance` * 100 - 100 - HTTP_RPC_PORT:
default - `instance` + 1 - WS_RPC_PORT: default + `instance` * 2 - 2

          [default: 1]

      --trusted-setup-file <PATH>
          Overrides the KZG trusted setup by reading from the supplied file

  -h, --help
          Print help (see a summary with '-h')

Metrics:
      --metrics <SOCKET>
          Enable Prometheus metrics.

          The metrics will be served at the given interface and port.

Networking:
  -d, --disable-discovery
          Disable the discovery service

      --disable-dns-discovery
          Disable the DNS discovery

      --disable-discv4-discovery
          Disable Discv4 discovery

      --discovery.port <DISCOVERY_PORT>
          The UDP port to use for P2P discovery/networking. default: 30303

      --trusted-peers <TRUSTED_PEERS>
          Target trusted peer enodes --trusted-peers
enode://abcd@192.168.0.1:30303

      --trusted-only
          Connect only to trusted peers

      --bootnodes <BOOTNODES>
          Bootnodes to connect to initially.

          Will fall back to a network-specific default if not specified.

      --peers-file <FILE>
          The path to the known peers file. Connected peers are dumped to
this file on nodes
          shutdown, and read on startup. Cannot be used with `--no-persist-
peers`.

      --identity <IDENTITY>
          Custom node identity

          [default: reth/VERSION/PLATFORM]

      --p2p-secret-key <PATH>
          Secret key to use for this node.

          This will also deterministically set the peer ID. If not specified,
it will be set in the data dir for the chain being used.

      --no-persist-peers
          Do not persist peers.

      --nat <NAT>
          NAT resolution method (any|none|upnp|publicip|extip:<IP>)

          [default: any]

      --port <PORT>
          Network listening port. default: 30303
      --max-outbound-peers <MAX_OUTBOUND_PEERS>
          Maximum number of outbound requests. default: 100

      --max-inbound-peers <MAX_INBOUND_PEERS>
          Maximum number of inbound requests. default: 30

RPC:
      --http
          Enable the HTTP-RPC server

      --http.addr <HTTP_ADDR>
          Http server address to listen on

          [default: 127.0.0.1]

      --http.port <HTTP_PORT>
          Http server port to listen on

          [default: 8545]

      --http.api <HTTP_API>
          Rpc Modules to be configured for the HTTP server

          [possible values: admin, debug, eth, net, trace, txpool, web3, rpc,
reth, ots]

      --http.corsdomain <HTTP_CORSDOMAIN>
          Http Corsdomain to allow request from

      --ws
          Enable the WS-RPC server

      --ws.addr <WS_ADDR>
          Ws server address to listen on

          [default: 127.0.0.1]

      --ws.port <WS_PORT>
          Ws server port to listen on

          [default: 8546]

      --ws.origins <ws.origins>
          Origins from which to accept WebSocket requests

      --ws.api <WS_API>
          Rpc Modules to be configured for the WS server

          [possible values: admin, debug, eth, net, trace, txpool, web3, rpc,
reth, ots]

      --ipcdisable
          Disable the IPC-RPC  server

      --ipcpath <IPCPATH>
          Filename for IPC socket/pipe within the datadir
```

```
                [default: /tmp/reth.ipc]

            --authrpc.addr <AUTH_ADDR>
                Auth server address to listen on

                [default: 127.0.0.1]

            --authrpc.port <AUTH_PORT>
                Auth server port to listen on

                [default: 8551]

            --authrpc.jwtsecret <PATH>
                Path to a JWT secret to use for authenticated RPC endpoints

            --rpc-max-request-size <RPC_MAX_REQUEST_SIZE>
                Set the maximum RPC request payload size for both HTTP and WS in
      megabytes

                [default: 15]

            --rpc-max-response-size <RPC_MAX_RESPONSE_SIZE>
                Set the maximum RPC response payload size for both HTTP and WS in
      megabytes

                [default: 115]
                [aliases: --rpc.returndata.limit]

            --rpc-max-subscriptions-per-connection
      <RPC_MAX_SUBSCRIPTIONS_PER_CONNECTION>
                Set the the maximum concurrent subscriptions per connection

                [default: 1024]

            --rpc-max-connections <COUNT>
                Maximum number of RPC server connections

                [default: 500]

            --rpc-max-tracing-requests <COUNT>
                Maximum number of concurrent tracing requests

                [default: 25]

            --rpc-max-logs-per-response <COUNT>
                Maximum number of logs that can be returned in a single response

                [default: 20000]

            --rpc-gas-cap <GAS_CAP>
                Maximum gas limit for `eth_call` and call tracing RPC methods

                [default: 50000000]

      Gas Price Oracle:
            --gpo.blocks <BLOCKS>
```

```
          Number of recent blocks to check for gas price

          [default: 20]

      --gpo.ignoreprice <IGNORE_PRICE>
          Gas Price below which gpo will ignore transactions

          [default: 2]

      --gpo.maxprice <MAX_PRICE>
          Maximum transaction priority fee(or gasprice before London Fork) to
be recommended by gpo

          [default: 500000000000]

      --gpo.percentile <PERCENTILE>
          The percentile of gas prices to use for the estimate

          [default: 60]

      --block-cache-len <BLOCK_CACHE_LEN>
          Maximum number of block cache entries

          [default: 5000]

      --receipt-cache-len <RECEIPT_CACHE_LEN>
          Maximum number of receipt cache entries

          [default: 2000]

      --env-cache-len <ENV_CACHE_LEN>
          Maximum number of env cache entries

          [default: 1000]

TxPool:
      --txpool.pending_max_count <PENDING_MAX_COUNT>
          Max number of transaction in the pending sub-pool

          [default: 10000]

      --txpool.pending_max_size <PENDING_MAX_SIZE>
          Max size of the pending sub-pool in megabytes

          [default: 20]

      --txpool.basefee_max_count <BASEFEE_MAX_COUNT>
          Max number of transaction in the basefee sub-pool

          [default: 10000]

      --txpool.basefee_max_size <BASEFEE_MAX_SIZE>
          Max size of the basefee sub-pool in megabytes

          [default: 20]

      --txpool.queued_max_count <QUEUED_MAX_COUNT>
```

```
          Max number of transaction in the queued sub-pool

          [default: 10000]

      --txpool.queued_max_size <QUEUED_MAX_SIZE>
          Max size of the queued sub-pool in megabytes

          [default: 20]

      --txpool.max_account_slots <MAX_ACCOUNT_SLOTS>
          Max number of executable transaction slots guaranteed per account

          [default: 16]

      --txpool.pricebump <PRICE_BUMP>
          Price bump (in %) for the transaction pool underpriced check

          [default: 10]

      --blobpool.pricebump <BLOB_TRANSACTION_PRICE_BUMP>
          Price bump percentage to replace an already existing blob
transaction

          [default: 100]

Builder:
      --builder.extradata <EXTRADATA>
          Block extra data set by the payload builder

          [default: reth/VERSION/OS]

      --builder.gaslimit <GAS_LIMIT>
          Target gas ceiling for built blocks

          [default: 30000000]

      --builder.interval <SECONDS>
          The interval at which the job should build a new payload after the
last (in seconds)

          [default: 1]

      --builder.deadline <SECONDS>
          The deadline for when the payload builder job should resolve

          [default: 12]

      --builder.max-tasks <MAX_PAYLOAD_TASKS>
          Maximum number of tasks to spawn for building a payload

          [default: 3]

Debug:
      --debug.continuous
          Prompt the downloader to download blocks one at a time.

          NOTE: This is for testing purposes only.
      --debug.terminate
          Flag indicating whether the node should be terminated after the
pipeline sync

      --debug.tip <TIP>
          Set the chain tip manually for testing purposes.

          NOTE: This is a temporary flag

      --debug.max-block <MAX_BLOCK>
          Runs the sync only up to the specified block

      --debug.print-inspector
          Print opcode level traces directly to console during execution

      --debug.hook-block <HOOK_BLOCK>
          Hook on a specific block during execution

      --debug.hook-transaction <HOOK_TRANSACTION>
          Hook on a specific transaction during execution

      --debug.hook-all
          Hook on every transaction in a block

Database:
      --db.log-level <LOG_LEVEL>
          Database logging level. Levels higher than "notice" require a debug
build

          Possible values:
          - fatal:   Enables logging for critical conditions, i.e. assertion
failures
          - error:   Enables logging for error conditions
          - warn:    Enables logging for warning conditions
          - notice:  Enables logging for normal but significant condition
          - verbose: Enables logging for verbose informational
          - debug:   Enables logging for debug-level messages
          - trace:   Enables logging for trace debug-level messages
          - extra:   Enables logging for extra debug-level messages

Dev testnet:
      --dev
          Start the node in dev mode

          This mode uses a local proof-of-authority consensus engine with
either fixed block times
          or automatically mined blocks.
          Disables network discovery and enables local http server.
          Prefunds 20 accounts derived by mnemonic "test test test test test
test test test test test
          test junk" with 10 000 ETH each.

      --dev.block-max-transactions <BLOCK_MAX_TRANSACTIONS>
          How many transactions to mine per block

      --dev.block-time <BLOCK_TIME>
```

```
          Interval between blocks.

          Parses strings using [humantime::parse_duration]
          --dev.block_time 12s

Pruning:
      --full
          Run full node. Only the most recent 10064 block states are stored.
This flag takes priority over pruning configuration in reth.toml

Logging:
      --log.directory <PATH>
          The path to put log files in

          [default: /reth/logs]

      --log.max-size <SIZE>
          The maximum size (in MB) of log files

          [default: 200]

      --log.max-files <COUNT>
          The maximum amount of log files that will be stored. If set to 0,
background file logging is disabled

          [default: 5]

      --log.journald
          Log events to journald

      --log.filter <FILTER>
          The filter to use for logs written to the log file

          [default: error]

      --color <COLOR>
          Sets whether or not the formatter emits ANSI terminal escape codes
for colors and other text formatting

          [default: always]

          Possible values:
          - always: Colors on
          - auto:   Colors on
          - never:  Colors off

Display:
  -v, --verbosity...
          Set the minimum log level.

          -v      Errors
          -vv     Warnings
          -vvv    Info
          -vvvv   Debug
          -vvvvv  Traces (warning: very verbose!)

  -q, --quiet
```

```
          Silence all log output
```

`reth init`

Initialize the database from a genesis file

```
$ reth init --help

Usage: reth init [OPTIONS]

Options:
      --datadir <DATA_DIR>
          The path to the data dir for all reth files and subdirectories.

          Defaults to the OS-specific data directory:

          - Linux: `$XDG_DATA_HOME/reth/` or `$HOME/.local/share/reth/`
          - Windows: `{FOLDERID_RoamingAppData}/reth/`
          - macOS: `$HOME/Library/Application Support/reth/`

          [default: default]

      --chain <CHAIN_OR_PATH>
          The chain this node is running.

          Possible values are either a built-in chain or the path to a chain
specification file.

          Built-in chains:
          - mainnet
          - goerli
          - sepolia
          - holesky

          [default: mainnet]

      --instance <INSTANCE>
          Add a new instance of a node.

          Configures the ports of the node to avoid conflicts with the
defaults. This is useful for running multiple nodes on the same machine.

          Max number of instances is 200. It is chosen in a way so that it's
not possible to have port numbers that conflict with each other.

          Changes to the following port numbers: - DISCOVERY_PORT: default +
`instance` - 1 - AUTH_PORT: default + `instance` * 100 - 100 - HTTP_RPC_PORT:
default - `instance` + 1 - WS_RPC_PORT: default + `instance` * 2 - 2

          [default: 1]

  -h, --help
          Print help (see a summary with '-h')

Database:
      --db.log-level <LOG_LEVEL>
          Database logging level. Levels higher than "notice" require a debug
build

          Possible values:
          - fatal:   Enables logging for critical conditions, i.e. assertion
failures
```

```
          - error:   Enables logging for error conditions
          - warn:    Enables logging for warning conditions
          - notice:  Enables logging for normal but significant condition
          - verbose: Enables logging for verbose informational
          - debug:   Enables logging for debug-level messages
          - trace:   Enables logging for trace debug-level messages
          - extra:   Enables logging for extra debug-level messages

Logging:
      --log.directory <PATH>
          The path to put log files in

          [default: /reth/logs]

      --log.max-size <SIZE>
          The maximum size (in MB) of log files

          [default: 200]

      --log.max-files <COUNT>
          The maximum amount of log files that will be stored. If set to 0,
background file logging is disabled

          [default: 5]

      --log.journald
          Log events to journald

      --log.filter <FILTER>
          The filter to use for logs written to the log file

          [default: error]

      --color <COLOR>
          Sets whether or not the formatter emits ANSI terminal escape codes
for colors and other text formatting

          [default: always]

          Possible values:
          - always: Colors on
          - auto:   Colors on
          - never:  Colors off

Display:
  -v, --verbosity...
          Set the minimum log level.

          -v      Errors
          -vv     Warnings
          -vvv    Info
          -vvvv   Debug
          -vvvvv  Traces (warning: very verbose!)

  -q, --quiet
          Silence all log output
```

`reth import`

This syncs RLP encoded blocks from a file

```
$ reth import --help

Usage: reth import [OPTIONS] <IMPORT_PATH>

Options:
      --config <FILE>
          The path to the configuration file to use.

      --datadir <DATA_DIR>
          The path to the data dir for all reth files and subdirectories.

          Defaults to the OS-specific data directory:

          - Linux: `$XDG_DATA_HOME/reth/` or `$HOME/.local/share/reth/`
          - Windows: `{FOLDERID_RoamingAppData}/reth/`
          - macOS: `$HOME/Library/Application Support/reth/`

          [default: default]

      --chain <CHAIN_OR_PATH>
          The chain this node is running.

          Possible values are either a built-in chain or the path to a chain
specification file.

          Built-in chains:
          - mainnet
          - goerli
          - sepolia
          - holesky

          [default: mainnet]

      --instance <INSTANCE>
          Add a new instance of a node.

          Configures the ports of the node to avoid conflicts with the
defaults. This is useful for running multiple nodes on the same machine.

          Max number of instances is 200. It is chosen in a way so that it's
not possible to have port numbers that conflict with each other.

          Changes to the following port numbers: - DISCOVERY_PORT: default +
`instance` - 1 - AUTH_PORT: default + `instance` * 100 - 100 - HTTP_RPC_PORT:
default - `instance` + 1 - WS_RPC_PORT: default + `instance` * 2 - 2

          [default: 1]

  -h, --help
          Print help (see a summary with '-h')

Database:
      --db.log-level <LOG_LEVEL>
          Database logging level. Levels higher than "notice" require a debug
build
```

```
          Possible values:
          - fatal:   Enables logging for critical conditions, i.e. assertion
failures
          - error:   Enables logging for error conditions
          - warn:    Enables logging for warning conditions
          - notice:  Enables logging for normal but significant condition
          - verbose: Enables logging for verbose informational
          - debug:   Enables logging for debug-level messages
          - trace:   Enables logging for trace debug-level messages
          - extra:   Enables logging for extra debug-level messages

  <IMPORT_PATH>
          The path to a block file for import.

          The online stages (headers and bodies) are replaced by a file
import, after which the
          remaining stages are executed.

Logging:
      --log.directory <PATH>
          The path to put log files in

          [default: /reth/logs]

      --log.max-size <SIZE>
          The maximum size (in MB) of log files

          [default: 200]

      --log.max-files <COUNT>
          The maximum amount of log files that will be stored. If set to 0,
background file logging is disabled

          [default: 5]

      --log.journald
          Log events to journald

      --log.filter <FILTER>
          The filter to use for logs written to the log file

          [default: error]

      --color <COLOR>
          Sets whether or not the formatter emits ANSI terminal escape codes
for colors and other text formatting

          [default: always]

          Possible values:
          - always: Colors on
          - auto:   Colors on
          - never:  Colors off

Display:
  -v, --verbosity...
          Set the minimum log level.
          -v      Errors
          -vv     Warnings
          -vvv    Info
          -vvvv   Debug
          -vvvvv  Traces (warning: very verbose!)

  -q, --quiet
          Silence all log output
```

`reth db`

Database debugging utilities

```
$ reth db --help

Usage: reth db [OPTIONS] <COMMAND>

Commands:
  stats    Lists all the tables, their entry count and their size
  list     Lists the contents of a table
  diff     Create a diff between two database tables or two entire databases
  get      Gets the content of a table for the given key
  drop     Deletes all database entries
  clear    Deletes all table entries
  version  Lists current and local database versions
  path     Returns the full database path
  help     Print this message or the help of the given subcommand(s)

Options:
      --datadir <DATA_DIR>
          The path to the data dir for all reth files and subdirectories.

          Defaults to the OS-specific data directory:

          - Linux: `$XDG_DATA_HOME/reth/` or `$HOME/.local/share/reth/`
          - Windows: `{FOLDERID_RoamingAppData}/reth/`
          - macOS: `$HOME/Library/Application Support/reth/`

          [default: default]

      --chain <CHAIN_OR_PATH>
          The chain this node is running.

          Possible values are either a built-in chain or the path to a chain
specification file.

          Built-in chains:
          - mainnet
          - goerli
          - sepolia
          - holesky

          [default: mainnet]

      --instance <INSTANCE>
          Add a new instance of a node.

          Configures the ports of the node to avoid conflicts with the
defaults. This is useful for running multiple nodes on the same machine.

          Max number of instances is 200. It is chosen in a way so that it's
not possible to have port numbers that conflict with each other.

          Changes to the following port numbers: - DISCOVERY_PORT: default +
`instance` - 1 - AUTH_PORT: default + `instance` * 100 - 100 - HTTP_RPC_PORT:
default - `instance` + 1 - WS_RPC_PORT: default + `instance` * 2 - 2

          [default: 1]
  -h, --help
          Print help (see a summary with '-h')

Database:
      --db.log-level <LOG_LEVEL>
          Database logging level. Levels higher than "notice" require a debug
build

          Possible values:
          - fatal:   Enables logging for critical conditions, i.e. assertion
failures
          - error:   Enables logging for error conditions
          - warn:    Enables logging for warning conditions
          - notice:  Enables logging for normal but significant condition
          - verbose: Enables logging for verbose informational
          - debug:   Enables logging for debug-level messages
          - trace:   Enables logging for trace debug-level messages
          - extra:   Enables logging for extra debug-level messages

Logging:
      --log.directory <PATH>
          The path to put log files in

          [default: /reth/logs]

      --log.max-size <SIZE>
          The maximum size (in MB) of log files

          [default: 200]

      --log.max-files <COUNT>
          The maximum amount of log files that will be stored. If set to 0,
background file logging is disabled

          [default: 5]

      --log.journald
          Log events to journald

      --log.filter <FILTER>
          The filter to use for logs written to the log file

          [default: error]

      --color <COLOR>
          Sets whether or not the formatter emits ANSI terminal escape codes
for colors and other text formatting

          [default: always]

          Possible values:
          - always: Colors on
          - auto:   Colors on
          - never:  Colors off

Display:
  -v, --verbosity...
```

```
          Set the minimum log level.

          -v      Errors
          -vv     Warnings
          -vvv    Info
          -vvvv   Debug
          -vvvvv  Traces (warning: very verbose!)

  -q, --quiet
          Silence all log output
```

reth db clear

Deletes all table entries

```
$ reth db clear --help

Usage: reth db clear [OPTIONS] <TABLE>

Arguments:
  <TABLE>
          Table name
```

reth db diff

Create a diff between two database tables or two entire databases

```
$ reth db diff --help

Usage: reth db diff [OPTIONS] --secondary-datadir <SECONDARY_DATADIR>
--output <OUTPUT>

Options:
      --secondary-datadir <SECONDARY_DATADIR>
          The path to the data dir for all reth files and subdirectories.
```

reth db drop

Deletes all database entries

```
$ reth db drop --help

Usage: reth db drop [OPTIONS]

Options:
  -f, --force
          Bypasses the interactive confirmation and drops the database
directly
```

reth db get

Gets the content of a table for the given key

```
$ reth db get --help

Usage: reth db get [OPTIONS] <TABLE> <KEY>

Arguments:
  <TABLE>
          The table name

          NOTE: The dupsort tables are not supported now.

  <KEY>
          The key to get content for
```

reth db list

Lists the contents of a table

```
$ reth db list --help

Usage: reth db list [OPTIONS] <TABLE>

Arguments:
  <TABLE>
          The table name

Options:
  -s, --skip <SKIP>
          Skip first N entries

          [default: 0]

  -r, --reverse
          Reverse the order of the entries. If enabled last table entries are
read

  -l, --len <LEN>
          How many items to take from the walker

          [default: 5]

      --search <SEARCH>
          Search parameter for both keys and values. Prefix it with `0x` to
search for binary data, and text otherwise.

          ATTENTION! For compressed tables (`Transactions` and `Receipts`),
there might be missing results since the search uses the raw uncompressed
value from the database.

  -c, --count
          Returns the number of rows found

  -j, --json
          Dump as JSON instead of using TUI
```

reth db path

Returns the full database path

```
$ reth db path --help

Usage: reth db path [OPTIONS]
```

reth db stats

Lists all the tables, their entry count and their size

```
$ reth db stats --help

Usage: reth db stats [OPTIONS]
```

reth db version

Lists current and local database versions

```
$ reth db version --help

Usage: reth db version [OPTIONS]
```

`reth stage`

Manipulate individual stages

```
$ reth stage --help

Usage: reth stage [OPTIONS] <COMMAND>

Commands:
  run     Run a single stage
  drop    Drop a stage's tables from the database
  dump    Dumps a stage from a range into a new database
  unwind  Unwinds a certain block range, deleting it from the database
  help    Print this message or the help of the given subcommand(s)

Options:
      --chain <CHAIN_OR_PATH>
          The chain this node is running.

          Possible values are either a built-in chain or the path to a chain
specification file.

          Built-in chains:
          - mainnet
          - goerli
          - sepolia
          - holesky

          [default: mainnet]

      --instance <INSTANCE>
          Add a new instance of a node.

          Configures the ports of the node to avoid conflicts with the
defaults. This is useful for running multiple nodes on the same machine.

          Max number of instances is 200. It is chosen in a way so that it's
not possible to have port numbers that conflict with each other.

          Changes to the following port numbers: - DISCOVERY_PORT: default +
`instance` - 1 - AUTH_PORT: default + `instance` * 100 - 100 - HTTP_RPC_PORT:
default - `instance` + 1 - WS_RPC_PORT: default + `instance` * 2 - 2

          [default: 1]

  -h, --help
          Print help (see a summary with '-h')

Logging:
      --log.directory <PATH>
          The path to put log files in

          [default: /reth/logs]

      --log.max-size <SIZE>
          The maximum size (in MB) of log files

          [default: 200]

      --log.max-files <COUNT>
```

```
          The maximum amount of log files that will be stored. If set to 0,
background file logging is disabled

          [default: 5]

      --log.journald
          Log events to journald

      --log.filter <FILTER>
          The filter to use for logs written to the log file

          [default: error]

      --color <COLOR>
          Sets whether or not the formatter emits ANSI terminal escape codes
for colors and other text formatting

          [default: always]

          Possible values:
          - always: Colors on
          - auto:   Colors on
          - never:  Colors off

Display:
  -v, --verbosity...
          Set the minimum log level.

          -v      Errors
          -vv     Warnings
          -vvv    Info
          -vvvv   Debug
          -vvvvv  Traces (warning: very verbose!)

  -q, --quiet
          Silence all log output
```

reth stage drop

Drop a stage's tables from the database

```
$ reth stage drop --help

Usage: reth stage drop [OPTIONS] <STAGE>
```

reth stage dump

Dumps a stage from a range into a new database

```
$ reth stage dump --help

Usage: reth stage dump [OPTIONS] <COMMAND>

Commands:
  execution        Execution stage
  storage-hashing  StorageHashing stage
  account-hashing  AccountHashing stage
  merkle           Merkle stage
  help             Print this message or the help of the given subcommand(s)
```

reth stage dump execution

Execution stage

```
$ reth stage dump execution --help

Usage: reth stage dump execution [OPTIONS] --output-db <OUTPUT_PATH> --from
<FROM> --to <TO>

Options:
      --output-db <OUTPUT_PATH>
          The path to the new database folder.

  -f, --from <FROM>
          From which block

  -t, --to <TO>
          To which block

  -d, --dry-run
          If passed, it will dry-run a stage execution from the newly created
database right after dumping
```

reth stage dump storage-hashing

StorageHashing stage

```
$ reth stage dump storage-hashing --help

Usage: reth stage dump storage-hashing [OPTIONS] --output-db <OUTPUT_PATH>
--from <FROM> --to <TO>

Options:
      --output-db <OUTPUT_PATH>
          The path to the new database folder.

  -f, --from <FROM>
          From which block

  -t, --to <TO>
          To which block

  -d, --dry-run
          If passed, it will dry-run a stage execution from the newly created
database right after dumping
```

reth stage dump account-hashing

AccountHashing stage

```
$ reth stage dump account-hashing --help

Usage: reth stage dump account-hashing [OPTIONS] --output-db <OUTPUT_PATH>
--from <FROM> --to <TO>

Options:
      --output-db <OUTPUT_PATH>
          The path to the new database folder.

  -f, --from <FROM>
          From which block

  -t, --to <TO>
          To which block

  -d, --dry-run
          If passed, it will dry-run a stage execution from the newly created
database right after dumping
```

reth stage dump merkle

Merkle stage

```
$ reth stage dump merkle --help

Usage: reth stage dump merkle [OPTIONS] --output-db <OUTPUT_PATH> --from
<FROM> --to <TO>

Options:
      --output-db <OUTPUT_PATH>
          The path to the new database folder.

  -f, --from <FROM>
          From which block

  -t, --to <TO>
          To which block

  -d, --dry-run
          If passed, it will dry-run a stage execution from the newly created
database right after dumping
```

reth stage run

Run a single stage.

```
$ reth stage run --help

Note that this won't use the Pipeline and as a result runs stages assuming
that all the data can be held in memory. It is not recommended to run a stage
for really large block ranges if your computer does not have a lot of memory
to store all the data.

Usage: reth stage run [OPTIONS] --from <FROM> --to <TO> <STAGE>

Arguments:
  <STAGE>
          The name of the stage to run

          [possible values: headers, bodies, senders, execution, account-
hashing, storage-hashing, hashing, merkle, tx-lookup, account-history,
storage-history, total-difficulty]

Options:
      --config <FILE>
          The path to the configuration file to use.

      --metrics <SOCKET>
          Enable Prometheus metrics.

          The metrics will be served at the given interface and port.

      --from <FROM>
          The height to start at

  -t, --to <TO>
          The end of the stage

      --batch-size <BATCH_SIZE>
          Batch size for stage execution and unwind

  -s, --skip-unwind
          Normally, running the stage requires unwinding for stages that
already have been run, in order to not rewrite to the same database slots.

          You can optionally skip the unwinding phase if you're syncing a
block range that has not been synced before.
```

reth stage unwind to-block

Unwinds the database until the given block number (range is inclusive)

```
$ reth stage unwind to-block --help

Usage: reth stage unwind to-block [OPTIONS] <TARGET>

Arguments:
  <TARGET>
```

`reth stage unwind num-blocks`

Unwinds the given number of blocks from the database

```
$ reth stage unwind num-blocks --help

Usage: reth stage unwind num-blocks [OPTIONS] <AMOUNT>

Arguments:
  <AMOUNT>
```

`reth p2p`

P2P Debugging utilities

```
$ reth p2p --help

Usage: reth p2p [OPTIONS] <COMMAND>

Commands:
  header  Download block header
  body    Download block body
  help    Print this message or the help of the given subcommand(s)

Options:
      --config <FILE>
          The path to the configuration file to use.

      --chain <CHAIN_OR_PATH>
          The chain this node is running.

          Possible values are either a built-in chain or the path to a chain
specification file.

          Built-in chains:
          - mainnet
          - goerli
          - sepolia
          - holesky

          [default: mainnet]

      --datadir <DATA_DIR>
          The path to the data dir for all reth files and subdirectories.

          Defaults to the OS-specific data directory:

          - Linux: `$XDG_DATA_HOME/reth/` or `$HOME/.local/share/reth/`
          - Windows: `{FOLDERID_RoamingAppData}/reth/`
          - macOS: `$HOME/Library/Application Support/reth/`

          [default: default]

      --p2p-secret-key <PATH>
          Secret key to use for this node.

          This also will deterministically set the peer ID.

  -d, --disable-discovery
          Disable the discovery service

      --disable-dns-discovery
          Disable the DNS discovery

      --disable-discv4-discovery
          Disable Discv4 discovery

      --discovery.port <DISCOVERY_PORT>
          The UDP port to use for P2P discovery/networking. default: 30303

      --trusted-peer <TRUSTED_PEER>
```

```
          Target trusted peer

      --trusted-only
          Connect only to trusted peers

      --retries <RETRIES>
          The number of retries per request

          [default: 5]

      --instance <INSTANCE>
          Add a new instance of a node.

          Configures the ports of the node to avoid conflicts with the
defaults. This is useful for running multiple nodes on the same machine.

          Max number of instances is 200. It is chosen in a way so that it's
not possible to have port numbers that conflict with each other.

          Changes to the following port numbers: - DISCOVERY_PORT: default +
`instance` - 1 - AUTH_PORT: default + `instance` * 100 - 100 - HTTP_RPC_PORT:
default - `instance` + 1 - WS_RPC_PORT: default + `instance` * 2 - 2

          [default: 1]

      --nat <NAT>
          [default: any]

  -h, --help
          Print help (see a summary with '-h')

Database:
      --db.log-level <LOG_LEVEL>
          Database logging level. Levels higher than "notice" require a debug
build

          Possible values:
          - fatal:   Enables logging for critical conditions, i.e. assertion
failures
          - error:   Enables logging for error conditions
          - warn:    Enables logging for warning conditions
          - notice:  Enables logging for normal but significant condition
          - verbose: Enables logging for verbose informational
          - debug:   Enables logging for debug-level messages
          - trace:   Enables logging for trace debug-level messages
          - extra:   Enables logging for extra debug-level messages

Logging:
      --log.directory <PATH>
          The path to put log files in

          [default: /reth/logs]

      --log.max-size <SIZE>
          The maximum size (in MB) of log files

          [default: 200]
      --log.max-files <COUNT>
          The maximum amount of log files that will be stored. If set to 0,
background file logging is disabled

          [default: 5]

      --log.journald
          Log events to journald

      --log.filter <FILTER>
          The filter to use for logs written to the log file

          [default: error]

      --color <COLOR>
          Sets whether or not the formatter emits ANSI terminal escape codes
for colors and other text formatting

          [default: always]

          Possible values:
          - always: Colors on
          - auto:   Colors on
          - never:  Colors off

Display:
  -v, --verbosity...
          Set the minimum log level.

          -v      Errors
          -vv     Warnings
          -vvv    Info
          -vvvv   Debug
          -vvvvv  Traces (warning: very verbose!)

  -q, --quiet
          Silence all log output
```

reth p2p body

Download block body

```
$ reth p2p body --help

Usage: reth p2p body [OPTIONS] <ID>

Arguments:
  <ID>
          The block number or hash
```

reth p2p header

Download block header

```
$ reth p2p header --help

Usage: reth p2p header [OPTIONS] <ID>

Arguments:
  <ID>
          The header number or hash
```

`reth test-vectors`

Generate Test Vectors

```
$ reth test-vectors --help

Usage: reth test-vectors [OPTIONS] <COMMAND>

Commands:
  tables  Generates test vectors for specified tables. If no table is
specified, generate for all
  help    Print this message or the help of the given subcommand(s)

Options:
      --chain <CHAIN_OR_PATH>
          The chain this node is running.

          Possible values are either a built-in chain or the path to a chain
specification file.

          Built-in chains:
          - mainnet
          - goerli
          - sepolia
          - holesky

          [default: mainnet]

      --instance <INSTANCE>
          Add a new instance of a node.

          Configures the ports of the node to avoid conflicts with the
defaults. This is useful for running multiple nodes on the same machine.

          Max number of instances is 200. It is chosen in a way so that it's
not possible to have port numbers that conflict with each other.

          Changes to the following port numbers: - DISCOVERY_PORT: default +
`instance` - 1 - AUTH_PORT: default + `instance` * 100 - 100 - HTTP_RPC_PORT:
default - `instance` + 1 - WS_RPC_PORT: default + `instance` * 2 - 2

          [default: 1]

  -h, --help
          Print help (see a summary with '-h')

Logging:
      --log.directory <PATH>
          The path to put log files in

          [default: /reth/logs]

      --log.max-size <SIZE>
          The maximum size (in MB) of log files

          [default: 200]

      --log.max-files <COUNT>
          The maximum amount of log files that will be stored. If set to 0,
background file logging is disabled
```

```
          [default: 5]

      --log.journald
          Log events to journald

      --log.filter <FILTER>
          The filter to use for logs written to the log file

          [default: error]

      --color <COLOR>
          Sets whether or not the formatter emits ANSI terminal escape codes
for colors and other text formatting

          [default: always]

          Possible values:
          - always: Colors on
          - auto:   Colors on
          - never:  Colors off

Display:
  -v, --verbosity...
          Set the minimum log level.

          -v      Errors
          -vv     Warnings
          -vvv    Info
          -vvvv   Debug
          -vvvvv  Traces (warning: very verbose!)

  -q, --quiet
          Silence all log output
```

reth test-vectors tables

Generates test vectors for specified tables. If no table is specified, generate for all

```
$ reth test-vectors tables --help

Usage: reth test-vectors tables [OPTIONS] [NAMES]...

Arguments:
  [NAMES]...
          List of table names. Case-sensitive
```

`reth config`

Write config to stdout

```
$ reth config --help

Usage: reth config [OPTIONS]

Options:
      --config <FILE>
          The path to the configuration file to use.

      --default
          Show the default config

      --chain <CHAIN_OR_PATH>
          The chain this node is running.

          Possible values are either a built-in chain or the path to a chain
specification file.

          Built-in chains:
          - mainnet
          - goerli
          - sepolia
          - holesky

          [default: mainnet]

      --instance <INSTANCE>
          Add a new instance of a node.

          Configures the ports of the node to avoid conflicts with the
defaults. This is useful for running multiple nodes on the same machine.

          Max number of instances is 200. It is chosen in a way so that it's
not possible to have port numbers that conflict with each other.

          Changes to the following port numbers: - DISCOVERY_PORT: default +
`instance` - 1 - AUTH_PORT: default + `instance` * 100 - 100 - HTTP_RPC_PORT:
default - `instance` + 1 - WS_RPC_PORT: default + `instance` * 2 - 2

          [default: 1]

  -h, --help
          Print help (see a summary with '-h')

Logging:
      --log.directory <PATH>
          The path to put log files in

          [default: /reth/logs]

      --log.max-size <SIZE>
          The maximum size (in MB) of log files

          [default: 200]

      --log.max-files <COUNT>
          The maximum amount of log files that will be stored. If set to 0,
background file logging is disabled

          [default: 5]

      --log.journald
          Log events to journald

      --log.filter <FILTER>
          The filter to use for logs written to the log file

          [default: error]

      --color <COLOR>
          Sets whether or not the formatter emits ANSI terminal escape codes
for colors and other text formatting

          [default: always]

          Possible values:
          - always: Colors on
          - auto:   Colors on
          - never:  Colors off

Display:
  -v, --verbosity...
          Set the minimum log level.

          -v      Errors
          -vv     Warnings
          -vvv    Info
          -vvvv   Debug
          -vvvvv  Traces (warning: very verbose!)

  -q, --quiet
          Silence all log output
```

`reth debug`

Various debug routines

```
$ reth debug --help

Usage: reth debug [OPTIONS] <COMMAND>

Commands:
  execution          Debug the roundtrip execution of blocks as well as the
generated data
  merkle             Debug the clean & incremental state root calculations
  in-memory-merkle   Debug in-memory state root calculation
  help               Print this message or the help of the given subcommand(s)

Options:
      --datadir <DATA_DIR>
          The path to the data dir for all reth files and subdirectories.

          Defaults to the OS-specific data directory:

          - Linux: `$XDG_DATA_HOME/reth/` or `$HOME/.local/share/reth/`
          - Windows: `{FOLDERID_RoamingAppData}/reth/`
          - macOS: `$HOME/Library/Application Support/reth/`

          [default: default]

      --chain <CHAIN_OR_PATH>
          The chain this node is running.

          Possible values are either a built-in chain or the path to a chain
specification file.

          Built-in chains:
          - mainnet
          - goerli
          - sepolia
          - holesky

          [default: mainnet]

      --instance <INSTANCE>
          Add a new instance of a node.

          Configures the ports of the node to avoid conflicts with the
defaults. This is useful for running multiple nodes on the same machine.

          Max number of instances is 200. It is chosen in a way so that it's
not possible to have port numbers that conflict with each other.

          Changes to the following port numbers: - DISCOVERY_PORT: default +
`instance` - 1 - AUTH_PORT: default + `instance` * 100 - 100 - HTTP_RPC_PORT:
default - `instance` + 1 - WS_RPC_PORT: default + `instance` * 2 - 2

          [default: 1]

  -h, --help
          Print help (see a summary with '-h')

Logging:
      --log.directory <PATH>
          The path to put log files in

          [default: /reth/logs]

      --log.max-size <SIZE>
          The maximum size (in MB) of log files

          [default: 200]

      --log.max-files <COUNT>
          The maximum amount of log files that will be stored. If set to 0,
background file logging is disabled

          [default: 5]

      --log.journald
          Log events to journald

      --log.filter <FILTER>
          The filter to use for logs written to the log file

          [default: error]

      --color <COLOR>
          Sets whether or not the formatter emits ANSI terminal escape codes
for colors and other text formatting

          [default: always]

          Possible values:
          - always: Colors on
          - auto:   Colors on
          - never:  Colors off

Display:
  -v, --verbosity...
          Set the minimum log level.

          -v      Errors
          -vv     Warnings
          -vvv    Info
          -vvvv   Debug
          -vvvvv  Traces (warning: very verbose!)

  -q, --quiet
          Silence all log output
```

reth debug execution

Debug the roundtrip execution of blocks as well as the generated data

```
$ reth debug execution --help

Usage: reth debug execution [OPTIONS] --to <TO>
```

reth debug merkle

Debug the clean & incremental state root calculations

```
$ reth debug merkle --help

Usage: reth debug merkle [OPTIONS] --to <TO>
```

reth debug in-memory-merkle

Debug in-memory state root calculation

```
$ reth debug in-memory-merkle --help

Usage: reth debug in-memory-merkle [OPTIONS]
```

`reth recover`

Scripts for node recovery

```
$ reth recover --help

Usage: reth recover [OPTIONS] <COMMAND>

Commands:
  storage-tries  Recover the node by deleting dangling storage tries
  help           Print this message or the help of the given subcommand(s)

Options:
      --chain <CHAIN_OR_PATH>
          The chain this node is running.

          Possible values are either a built-in chain or the path to a chain
specification file.

          Built-in chains:
          - mainnet
          - goerli
          - sepolia
          - holesky

          [default: mainnet]

      --instance <INSTANCE>
          Add a new instance of a node.

          Configures the ports of the node to avoid conflicts with the
defaults. This is useful for running multiple nodes on the same machine.

          Max number of instances is 200. It is chosen in a way so that it's
not possible to have port numbers that conflict with each other.

          Changes to the following port numbers: - DISCOVERY_PORT: default +
`instance` - 1 - AUTH_PORT: default + `instance` * 100 - 100 - HTTP_RPC_PORT:
default - `instance` + 1 - WS_RPC_PORT: default + `instance` * 2 - 2

          [default: 1]

  -h, --help
          Print help (see a summary with '-h')

Logging:
      --log.directory <PATH>
          The path to put log files in

          [default: /reth/logs]

      --log.max-size <SIZE>
          The maximum size (in MB) of log files

          [default: 200]

      --log.max-files <COUNT>
          The maximum amount of log files that will be stored. If set to 0,
background file logging is disabled
```

```
          [default: 5]

      --log.journald
          Log events to journald

      --log.filter <FILTER>
          The filter to use for logs written to the log file

          [default: error]

      --color <COLOR>
          Sets whether or not the formatter emits ANSI terminal escape codes
for colors and other text formatting

          [default: always]

          Possible values:
          - always: Colors on
          - auto:   Colors on
          - never:  Colors off

Display:
  -v, --verbosity...
          Set the minimum log level.

          -v      Errors
          -vv     Warnings
          -vvv    Info
          -vvvv   Debug
          -vvvvv  Traces (warning: very verbose!)

  -q, --quiet
          Silence all log output
```

reth recover storage-tries

Recover the node by deleting dangling storage tries

```
$ reth recover storage-tries --help

Usage: reth recover storage-tries [OPTIONS]
```

Developers

Reth is composed of several crates that can be used in standalone projects. If you are interested in using one or more of the crates, you can get an overview of them in the developer docs, or take a look at the crate docs.

Contribute

Reth has docs specifically geared for developers and contributors, including documentation on the structure and architecture of reth, the general workflow we employ, and other useful tips.

You can find these docs here.

Check out our contributing guidelines here.