GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Seminar Report

Distributed Cloud Native Benchmarking Tool for Scalable Databases

Kariem Ali

MatrNr: 19773201

Supervisor: Patrick Höhn

Georg-August-Universität Göttingen
Institute of Computer Science

March 31, 2026

Abstract

Selecting the most suitable database when optimizing for performance of your system under specific workloads and data access patterns is a difficult decision without benchmarking. This requires simulating realistic load at scale that closely resembles production scenarios and the expected workload patterns. Existing tools/frameworks such as YCSB, TSBS, and JMeter each address parts of this problem but have their limitations such as, generic key-value workloads, hardcoded schemas, or lack infrastructure automation. In this report, we present a distributed cloud-native benchmarking tool that can replicate different workload patterns with high throughput on OpenStack infrastructure, operated through a terminal interface where the entire benchmark is described in a single YAML file. The tool is evaluated by benchmarking TimescaleDB and InfluxDB under identical workload conditions and to demonstrate the results the tool creates with zero manual infrastructure setup.

Declaration on the use of ChatGPT and comparable tools in the context of examinations

In this work I have used ChatGPT or another AI as follows:

- ☐ Not at all
- ☑ During brainstorming
- ☐ When creating the outline
- ☐ To write individual passages, altogether to the extent of 0% of the entire text
- ☑ For the development of software source texts
- ☐ For optimizing or restructuring software source texts
- ☑ For proofreading or optimizing
- ☐ Further, namely: -

I hereby declare that I have stated all uses completely.
Missing or incorrect information will be considered as an attempt to cheat.

Contents

List of Tables

List of Figures

List of Listings

List of Abbreviations

GWDG Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen

IoT Internet of Things

SDK Software Development Kit

TUI Terminal User Interface

VM Virtual Machine

1 Introduction

Choosing the right database is one of the most fundamental and essential decisions in system design. There is no single database that performs best in every scenario. Different databases and database paradigms or types are optimized for different data models, access patterns, and operations [Cat11]. Once an application is built around a specific database, switching to another can be costly and time-consuming [Roy19], since it often involves rewriting application code, migrating data, and dealing with possible incompatibilities that appear because of different database-specific implementation and design details.

Database benchmarking is commonly used to evaluate different databases before committing to one and is essential to many application or system developers [Baj+20; Dif+13]. By running controlled stress load test that simulate realistic workloads, developers and system designers can compare throughput, latency, and resource usage across databases and make a more informed choice. The effectiveness of a benchmark, however, depends significantly on the capability of the benchmarking tool being used and how it was used. A good benchmarking tool should be extensible to allow for alternatives to be validated with ease and without too many conflicts or issues [Dif+13]. This can include generating data that conforms to a user-defined schema rather than a hardcoded one, simulating realistic access patterns with configurable read/write ratios, communicating with each database through its native optimized protocols rather than generic ones, and scaling load across multiple worker or stressor nodes to stress the database under production-like conditions.

There is a vast number of benchmarking tools that currently exist, but they tend to fall short in one or more of these areas, whether in schema flexibility, database coverage, or the ability to customize and describe workloads or database access patterns. Additionally, setting up a distributed benchmarking environment is a manual and repetitive process that demands significant effort, which compounds as the number of databases, configurations, workloads, and deployment scenarios grows. Modern databases alone expose a wide range of configuration parameters, and repeating the full setup for each combination quickly becomes costly and impractical. Ideally, environment provisioning or setup would be integrated directly into the benchmarking tool itself and the entirety of the infrastructure and benchmark lifecycle rather than leaving it as a separate manual task.

To address these limitations, this paper presents a distributed, cloud-native benchmarking tool that integrates the OpenStack Python SDK [Ope26a] for automated infrastructure provisioning and Locust [HHB26] for distributed workload generation. The tool creates virtual machines, deploys database and benchmark containers, monitors the test in real time, collects results from all nodes, and generates visualization charts, all from a single command. Each database is accessed through its native protocol via a pluggable client abstraction, and the entire benchmark is configured through a single YAML file that describes the database connection, table schema, workload parameters, and infrastructure setup. All experiments and benchmarks presented in this paper were conducted on an OpenStack instance provided by the GWDG cloud infrastructure.

The tool is evaluated through two experiments comparing TimescaleDB and InfluxDB. These experiments demonstrate how the tool is operated in practice and what kind of performance insights it produces.

2 Background

2.1 Database Benchmarking

Database benchmarking is the process of running a defined workload across systems and measuring their performance [Gra93]. Common metrics include response time, throughput, and reliability (measured as the ratio of successful requests) [HJZ17].

Several benchmarking tools exist, each with notable limitations. sysbench [26c] is tightly coupled to MySQL and PostgreSQL, making cross-database comparisons hard to do. pgbench [26b] is built exclusively for PostgreSQL and therefore also tightly coupled to it. Apache JMeter [26a], while more general-purpose, is GUI-centric and requires significant manual scripting for complex database interactions and it offers no native schema-aware data generation without custom scripting. None of these tools combine schema-flexible workloads, native database protocols, distributed load generation, and automated cloud infrastructure provisioning in a single solution.

2.2 OpenStack and Locust

OpenStack is a cloud infrastructure platform that manages pooled compute, storage, and networking resources across datacenters, that offers both administrative control and user-facing self-service provisioning through a web interface [Ope26b]. The OpenStack Python SDK [Ope26a] enables programmatic VM provisioning, and Cloud-init support allows custom setup scripts to be injected directly at boot time. These scripts can be customised per VM depending on its role (e.g. database, worker, or master), and further parameterised using Jinja2 templates [Pal], whose placeholders are populated during runtime by the benchmark tool.

Locust [HHB26] is an open-source, Python-native load testing tool that defines load tests as code, making them highly customisable and reproducible. This code-first approach means any native database driver can be imported directly into a test scenario. Distributed execution is supported out of the box through a master–worker architecture, in which a master node coordinates the test while worker nodes generate the actual load.

Together, these two tools serve as the infrastructure and load generation components of the benchmarking framework presented in this paper.

3 System Design

3.1 Architecture Overview

The tool is organized into six layers, as shown in Figure 1: the terminal interface, the driver layer, the orchestration layer, the result processing layer, the core module, and the remote infrastructure.

The **Terminal Interface** is the user's entry point. Before deployment, it presents an interactive menu for selecting OpenStack resources (network, keypair, image, flavor)then during execution, it renders a live monitoring view with spinners and status updates.

The **Driver layer** contains the components that communicate with the remote cloud environment. `SSHClient` is a lightweight client that handles remote command execution over SSH, `Monitor` parses the Locust master's container logs to extract live metrics such

as request counts and throughput then pass it through to the TUI, and `WorkloadConfig` loads and validates the YAML configuration that defines the benchmark.

The **Orchestration layer** manages the entirety of the benchmark lifecycle. The `Orchestrator` manages provisioning, deployment, monitoring, and result collection across all nodes. Deployment scripts are generated from Jinja2 templates whose placeholders are injected during runtime by the tool to allow for customization and extensibility instead of having multiple static scripts. A shared base template handles common bootstrapping (Docker installation, image pulling etc.), and role-specific templates are appended as additions to the base template for each node type.

The **Result processing layer** handles the visualizations of results, after the benchmark has finished, using the raw CSV extracted from benchmark results. Once the run completes and CSV data has been collected from the cluster, the `Visualizer` reads the custom per-request logs, Locust's built-in aggregated statistics, and the database resource metrics to generate multiple visualizations covering throughput, latency distributions, percentile statistics, and resource usage.

The **Core module** runs inside Docker containers on the deployed VMs. It contains the locustfile with the execution logic, and the database-specific clients for each supported database. This module is packaged into a Docker image and pushed to a registry. During deployment, each VM pulls the image via cloud-init Jinja2 templates and launches the container with the correct startup arguments based on the node type (different startup arguments depending on whether the node is a master or a worker node in locust).

The **Remote infrastructure** is the OpenStack environment hosted on GWDG Cloud where the benchmark executes. It consists of a database node, a Locust master node, and one or more worker nodes, all provisioned as VMs with Docker installed via their corresponding cloud-init script. The master coordinates the distributed load test, workers generate traffic against the database, and the Docker registry provides the dockerized core module image that all nodes pull at boot time.

3.2 Configuration

A single YAML file describes the benchmark across four sections: **Database** (type, port, credentials, and any database-specific extras such as InfluxDB tokens), **Table** (column names, types, and value bounded used for data generation), **Benchmark** (concurrent users, spawn rate, duration, batch size, write weight, and weighted read queries), and **Infrastructure** (Docker image tag, SSH credentials, worker count). The database `host` field is left empty in the config file and is patched at deployment time with the database node's private IP once it is known. An example configuration is shown in Appendix A.

3.3 Database Client Abstraction

`BaseDBClient` declares four abstract methods that will be implemented by the database specific sub-classes with their native and optimized protocols: `connect()`, `disconnect()`, `write_batch(table, columns, data)`, and `query(query)`, see Figure 2. It also provides two function wrappers, `timed_write` and `timed_query`, that measure wall-clock time and return an `OperationResult` dataclass containing a success flag, elapsed milliseconds, the number of rows affected, and any exception/error that occurred.

The `TimescaleDBClient` uses `psycopg3` to talk over the PostgreSQL wire protocol. Batched writes are implemented using `executemany()`, however, other methods

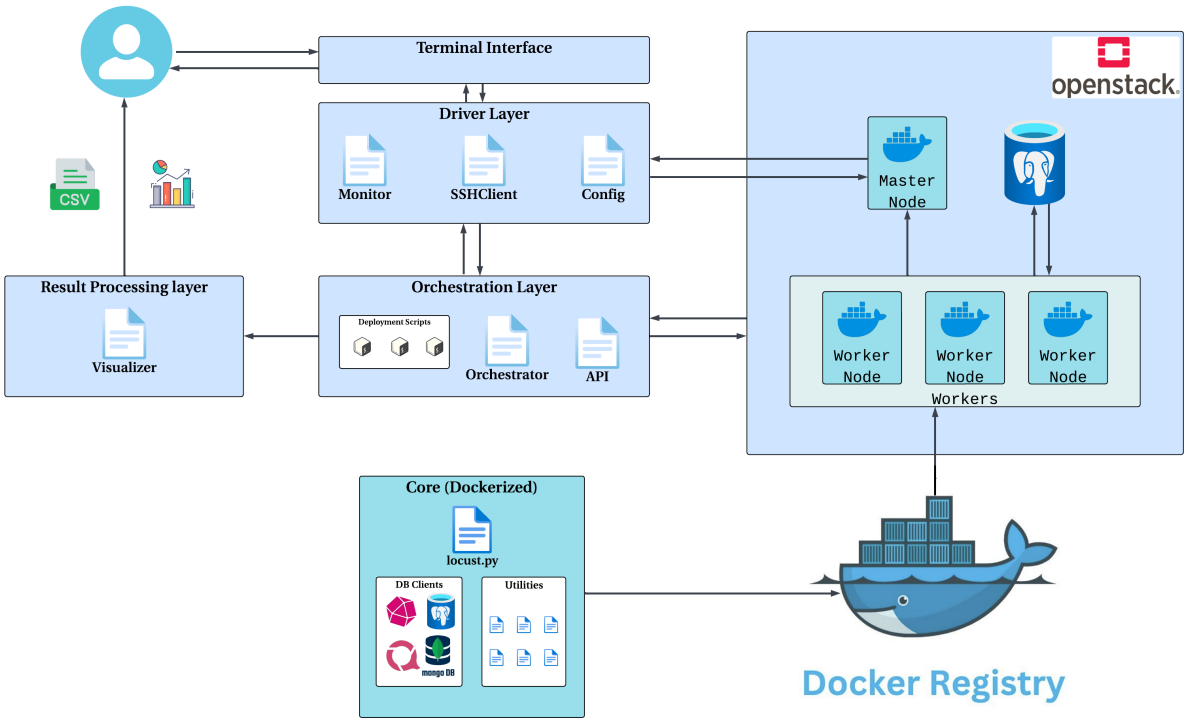

Figure 1: High-level architecture of the benchmarking tool

can be used since it depends which functions or specific client implementation details the user would like to test benchmark or test. The `InfluxDBv2Client` uses the official `influxdb-client` library over HTTP. Each row is converted to a `Point` object: string columns become tags, numeric values become fields, and the `time` column is set as the point's timestamp with millisecond precision.

Supporting a new database involves implementing a `BaseDBClient` subclass, registering it in the `create_client` factory function, updating deployment configurations (environment variables to be injected and how the DB init-script is going to be rendered) in the orchestrator, exporting the new client, and creating a corresponding benchmark configuration YAML.

3.4 The Locustfile

Each simulated user opens its own database connection and enters a loop with no delay between iterations. On each iteration it picks a random action from a weighted list constructed at startup from the YAML configuration: write operations appear according to `write_weight`, and each read query appears according to its own `weight`. If a connection drops, the user reconnects automatically; after five consecutive failures it backs off for one second before trying again.

Data generation is based on the table schema (Figure 3). Timestamps use the current wall clock, numeric columns are sampled uniformly within their configured bounds, and string columns either pick from a predefined list of choices or produce a random 10-character string. Rows are assembled into batches of size `batch_size` before being handed to the database client.

A locustfile is the Python file that defines the operations that Locust workers (or

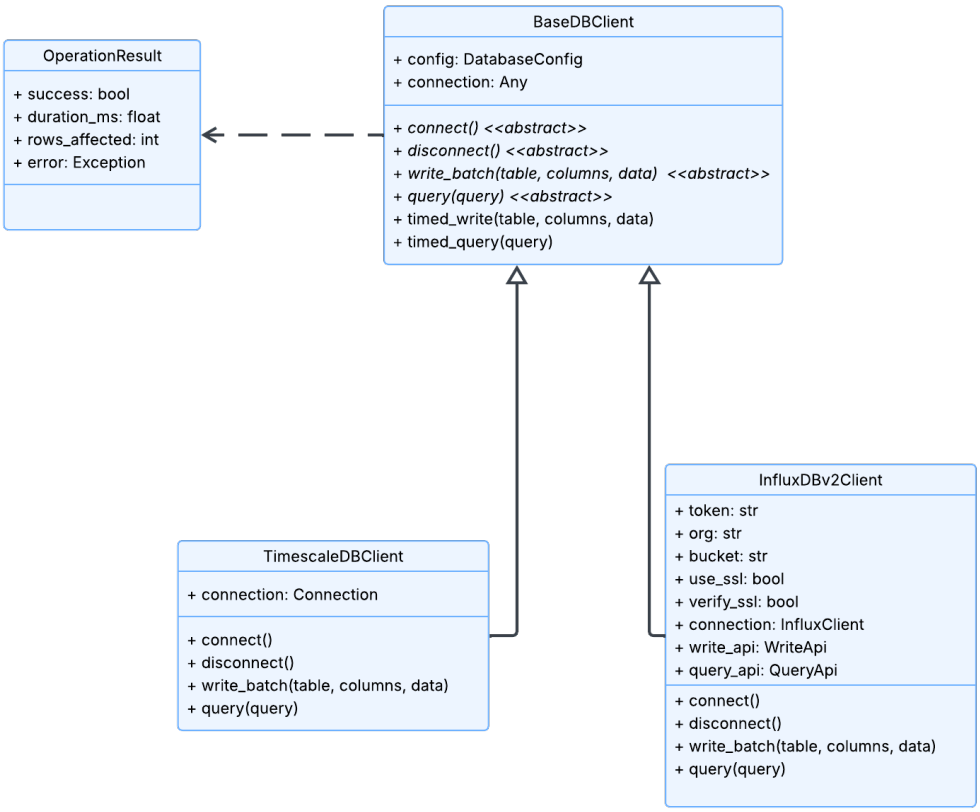

Figure 2: Database client class hierarchy.

spawned users) will execute. The `BenchmarkUser` class that extends Locust's `User` and uses the database clients described above instead of Locust's default HTTP client. Every operation, whether a write or a read, is timed and reported to Locust's event system under a custom `DB` request type and saved separately by each worker node locally. This lets Locust's built-in statistics engine track throughput and latency percentiles as if these were HTTP requests. Additionally, each worker writes a per-request CSV log (timestamp, worker ID, operation name, response time, success flag, and any error message). The orchestrator later merges these worker-level logs into a single file for visualization.

3.5 Orchestration Pipeline

The orchestrator handles provisioning, running, and collecting results from the benchmark so that no manual SSH access or intermediate steps are needed. The pipeline runs in four phases. Screenshots of each phase as displayed in the terminal UI are shown in Appendix B.

3.5.1 Phase 1: Interactive Resource Configuration

The orchestrator first connects to the OpenStack API and presents the user with an interactive menu: network, SSH keypair, OS image, and hardware flavor (the image and flavor are selected separately for database and worker/master nodes, since they may have different resource requirements). It then reserves three floating IPs: one for the database VM, one for the master (used for SSH monitoring and result collection), and one for the

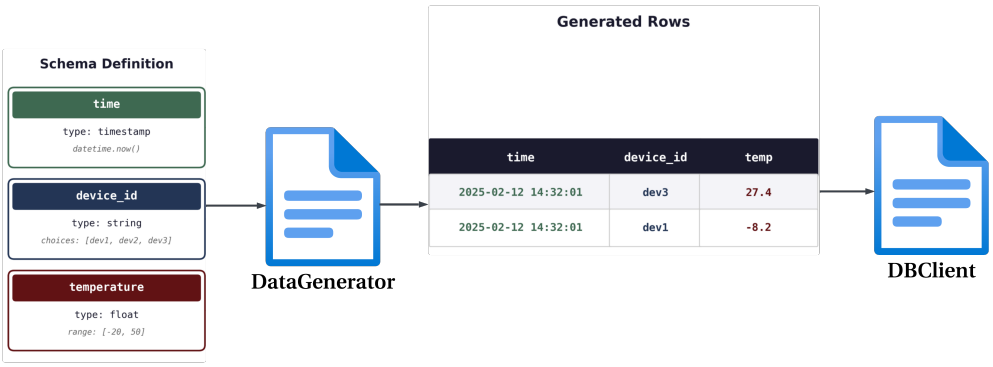

Figure 3: Schema-driven data generation.

first worker (used for debugging purposes if the benchmark produces zero requests).

3.5.2 Phase 2: VM Provisioning and Container Deployment

Each VM is bootstrapped via a cloud-init script assembled from Jinja2 templates. A shared base template (`base.sh.j2`) handles DNS configuration, Docker installation, daemon setup, image pulling, and writing the YAML config to disk. Figure 4 shows how a role-specific template is appended to the base to form the final script for each node type.

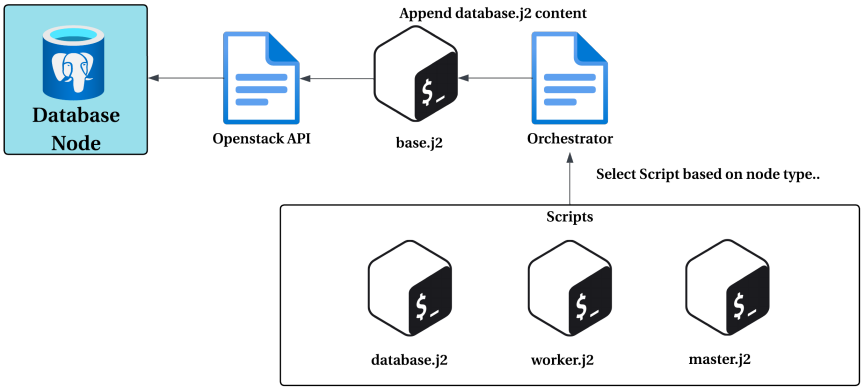

Figure 4: Cloud-init template components. The orchestrator picks a role-specific template and appends it to the shared base.

The three node roles are deployed using a thread pool with futures that are blocked when their dependencies (IP's) are not resolved yet(Figure 5):

1. **Database node**: submitted first. It launches the database container (TimescaleDB or InfluxDBv2) with the appropriate port mapping, environment variables, and for TimescaleDB, an SQL init script mounted into the container's entrypoint directory. Once the container is up, a background shell loop starts recording per-second resource metrics (CPU, memory, disk I/O, network I/O) from `docker stats`.

2. **Master node**: its future blocks on the database future. Once the database's private IP is known, the orchestrator patches the YAML config with it and starts a headless Locust master. The deployer's SSH private key is also embedded on the master so that it can later SCP results from the workers and the database node.

3. **Worker nodes**: their futures block on the master future, then all workers deploy in parallel. Each one launches a Locust worker that connects to the master over the internal network. If a worker VM fails to provision, it is retried up to three times with a short backoff before it is marked as failed.

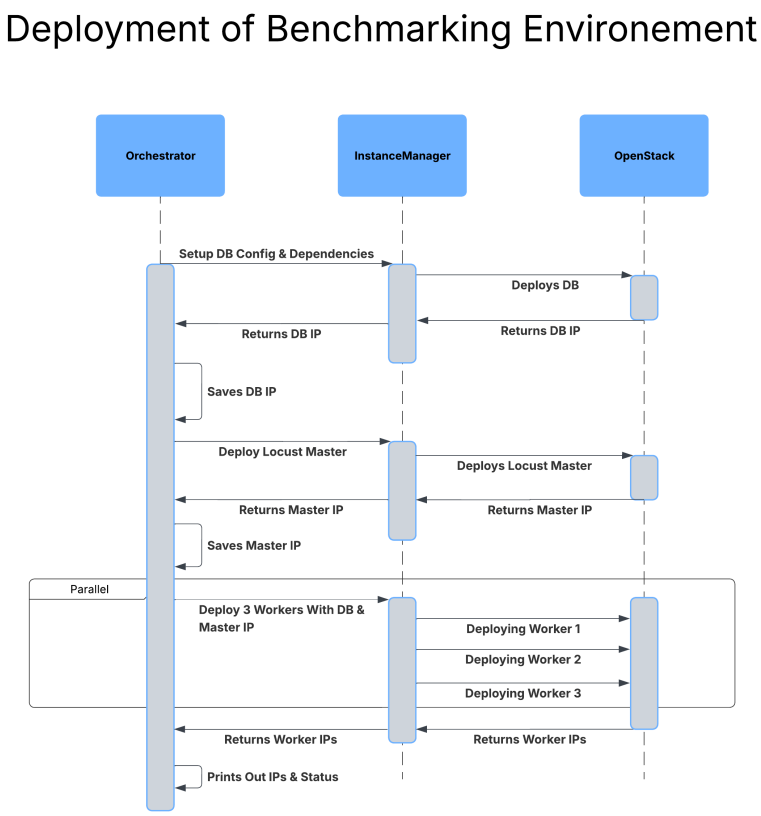

Figure 5: Deployment sequence. The database is provisioned first; the master blocks on the database IP; workers block on the master and deploy in parallel.

3.5.3 Phase 3: Live Monitoring

Once the master VM is deployed, the orchestrator waits for it to become reachable over SSH (polling for up to 60 seconds) and then waits for the Locust container to appear (up to another 120 seconds). After that, it enters a polling loop: every two seconds it pulls the last 300 lines of the container's logs via SSH and passes them to a log parser. The parser uses regular expressions to extract the current benchmark state (waiting for workers, spawning users, running, or stopped) along with aggregate request counts, throughput, and latency figures. These are rendered in the terminal UI, see in Appendix B.

3.5.4 Phase 4: Result Collection and Visualization

Results are collected in two steps. The master node holds an embedded copy of the user's SSH private key (the matching public key is already on all VMs via the OpenStack keypair), which it uses to copy each worker's per-request CSV and the database

node's resource metrics over the private network. It then merges the worker logs into a single sorted file. The local machine copies everything from the master in one step. The visualization module then reads the merged data and generates charts, which are saved alongside the raw CSVs.

3.6 Result Visualization

The visualizer consumes three data sources: the merged per-request log from all workers, Locust's aggregated statistics CSV, and the database resource metrics. From these it produces eight chart types:

- **Throughput over time:** requests per second, with a secondary axis showing the number of active users.
- **Latency by operation:** horizontal box plots comparing response time distributions across operation types, with median values annotated.
- **Per-request latency over time:** a scatter plot of every individual request, colored by operation. This is useful for identifying latency spikes or changes over time that aggregated metrics do not show.
- **Latency distribution:** per-operation histograms with vertical lines marking the p50, p95, and p99 percentiles.
- **Windowed percentiles:** rolling p50, p95, and p99 computed over two-second sliding windows, plotted per operation.
- **Worker comparison:** box plots of latency across workers, generated only when multiple workers were used.
- **Benchmark summary:** a two-panel chart showing request counts per operation and a grouped bar chart of median, average, and maximum latency.
- **Database resource usage:** CPU, memory, disk I/O, and network I/O time series derived from the `docker stats` data collected on the database node, aligned to the benchmark's start time.

All charts are saved as PNG files automatically once the benchmark finishes. Figure 6 shows an example.

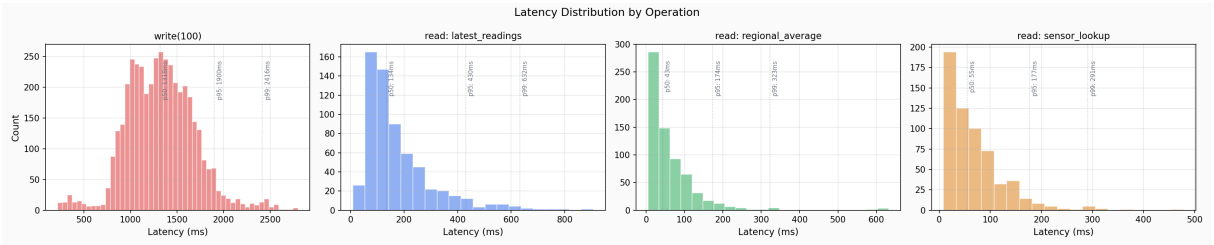

Figure 6: One example of the generated visualizations: latency distribution per operation with percentile markers.

4 Evaluation

This chapter uses the tool to compare TimescaleDB and InfluxDB under a hypothetical mixed IoT sensor workload. The goal is not to determine which database is universally better, but to demonstrate how the tool operates in practice, what kind of output it produces and possible insights that could be gained by using it.

4.1 Scenario

The experiment simulates an IoT system where sensors report telemetry data. The schema contains a timestamp, sensor ID (5 sensors), region (3 regions), temperature (-20 to 50°C), humidity (0 to 100%), and battery level (0 to 100). TimescaleDB's table is created via `init.sql` and converted into a hypertable partitioned by time. InfluxDB has a schema-on-write approach, automatically creating the measurement, tags, and fields the moment data is first written, so no need for an init.sql file for it.

4.2 Infrastructure

Both runs provision 5 VMs (3 workers, 1 DB, 1 Master) on the OpenStack environment using the same `m1.small` flavor for each one (1 vCPU, 2048 MB RAM, 20 GB disk): 1 database, 1 master, and 3 workers. All nodes are destroyed and reprovisioned between each different database run.

Table 1: Benchmark configuration parameters.

Parameter	**Value**
Concurrent users	50
Spawn rate	10 users/sec
Duration	120 seconds
Batch size	100 rows per write
Write weight	7
Read queries	3 (weight 1 each)
Approx. read/write ratio	70% writes, 30% reads
Worker nodes	3

Three read queries are defined in each database's YAML benchmark configuration file: **latest_readings** (100 most recent rows), **regional_average** (mean temperature and humidity per region over the last 5 minutes), and **sensor_lookup** (50 recent readings for sensor S03). The full YAML configurations for both databases are provided in Appendix A.

4.3 Results

All charts are generated automatically by the `Visualizer` at the end of each benchmark run. Since each run is provisioned and torn down independently, the charts for TimescaleDB and InfluxDB are scaled independently and should not be compared visually side by side. When only considering the automatically generated charts, it should be kept in mind that each run most likely will have different scaling. For users who would

want to create their own visualizations independently and possibly with the same scaling, all the CV's extracted and used for the automatically made visuals are saved alongside them and can be used to create custom visuals depending on the user's needs.

4.3.1 Throughput

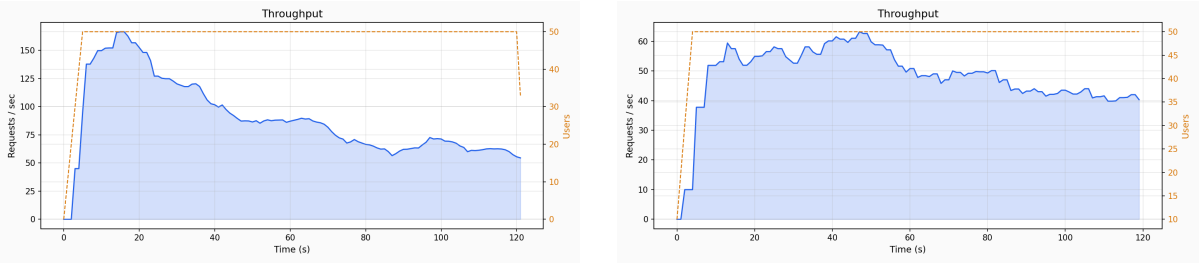

Figure 7: Throughput over time: TimescaleDB (left) and InfluxDB (right). Axes scaled independently per run.

TimescaleDB peaked at ∼175 req/s before declining to around 55–60 req/s; InfluxDB peaked at ∼65 req/s and settled around 40 req/s (Figure 7). Both databases show declining throughput over the course of the experiment, which is expected the more data that is saved the higher the cost of read operations are, particularly those involving aggregation over the accumulated data.

4.3.2 Latency by Operation

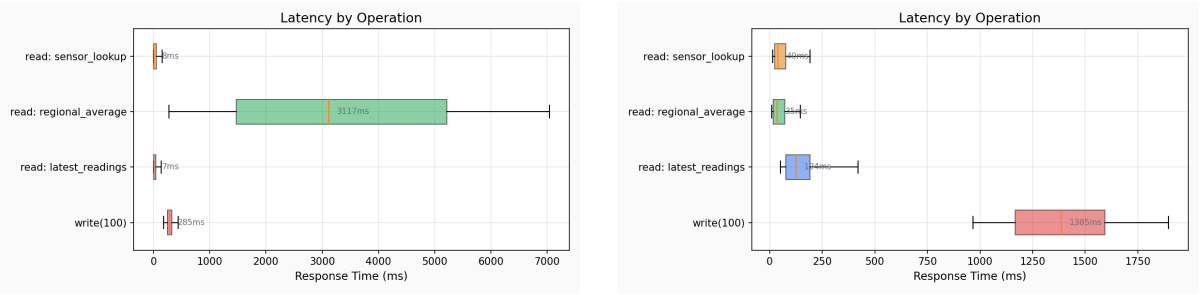

Figure 8: Latency by operation: TimescaleDB (left) and InfluxDB (right). Axes scaled independently per run.

TimescaleDB writes had a median of 285 ms compared to InfluxDB's 1,385 ms (Figure 8). For reads, the results were mixed: TimescaleDB's `latest_readings` median was 7 ms versus InfluxDB's 124 ms, and `sensor_lookup` was 8 ms versus 40 ms. However, InfluxDB's `regional_average` median was 35 ms compared to TimescaleDB's 3,117 ms.

4.3.3 Per-Request Latency Over Time

The scatter plots (Figure 9) reveal that on TimescaleDB, the `regional_average` query climbs from under 500 ms to almost 8,000 ms over the duration of the test, while writes and the other reads remain stable. On InfluxDB, writes drift upward from ∼500-800 ms to over 2,000 ms with periodic spikes, while reads remain mostly low over time.

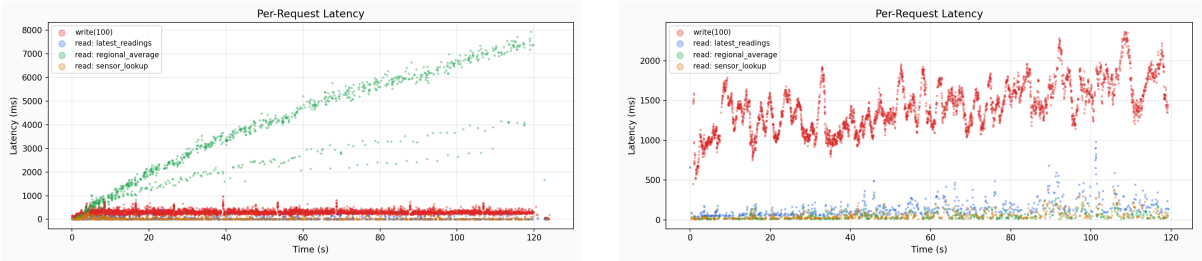

Figure 9: Per-request latency over time: TimescaleDB (left) and InfluxDB (right). Each dot is one request. Axes scaled independently per run.

4.3.4 Database Resource Usage

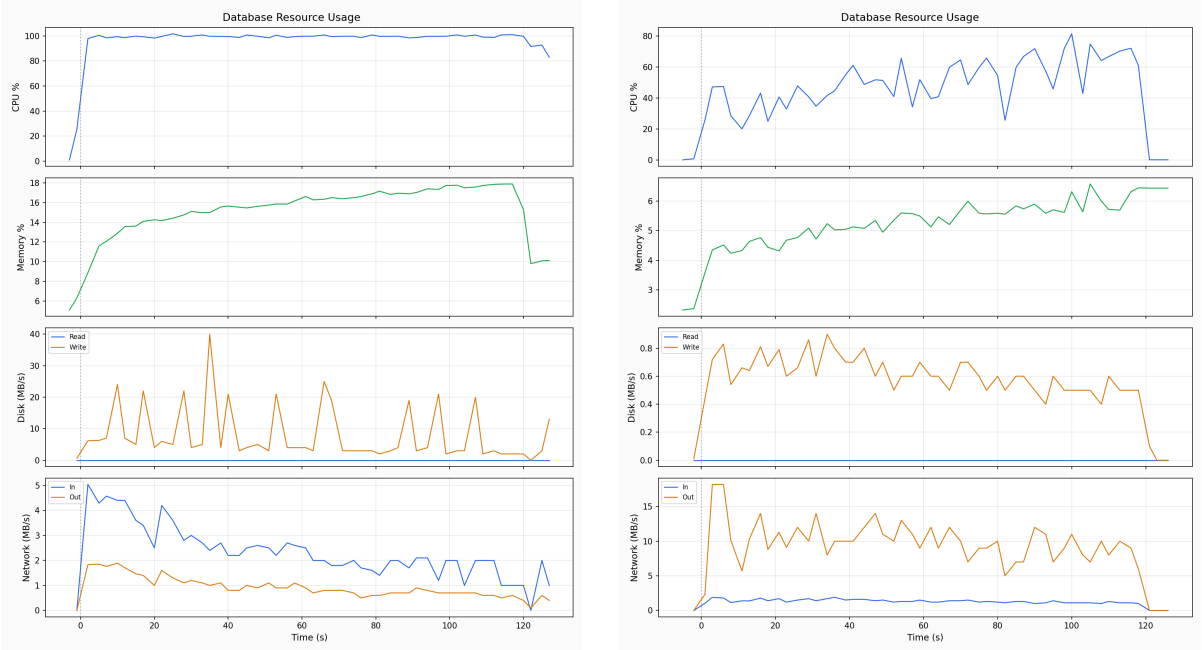

Figure 10: Database resource usage: TimescaleDB (left) and InfluxDB (right). Top to bottom: CPU, memory, disk I/O, network I/O. Axes scaled independently per run.

TimescaleDB saturated the vCPU at ∼100% throughout, while InfluxDB operated at around 40–80% (Figure 10). Memory rose from ∼5% to ∼18% on TimescaleDB and from ∼2.5% to ∼6.5% on InfluxDB. Disk I/O patterns differed: TimescaleDB showed bursty write spikes of 20–40 MB/s, while InfluxDB wrote steadily at 0.5–0.8 MB/s. Disk reads were mostly minimal for both, meaning the data sent during the benchmark fits in main memory without needing to push some to the disk. Network I/O also differed, InfluxDB's inbound traffic peaked at ∼18 MB/s versus TimescaleDB's 2-5 MB/s, which is possibly due to the overhead difference between InfluxDB's HTTP-based protocol and PostgreSQL's more compact binary wire protocol.

4.3.5 Latency per Worker

Load was almost evenly distributed across all three workers in both runs, which shows the operations that are done are relatively well distributed across workers, which is expected given locust's distributed mode (Figure 11).

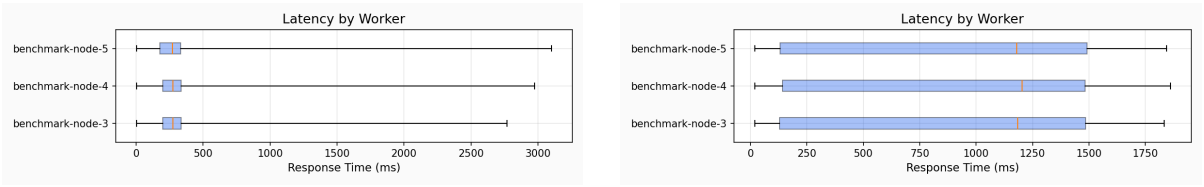

Figure 11: Latency by worker: TimescaleDB (left) and InfluxDB (right).

4.4 Discussion

The results show clear differences between the two databases, but it is important to note that the observed performance is shaped by multiple factors beyond the database engine itself. The database-specific client driver implementation, network protocol (PostgreSQL binary protocol vs. HTTP), serialization format, and connection handling all contribute to the measured latencies and throughput. For example, the write latency gap may be partly explained by `psycopg`'s `executemany` sending a full batch in one round trip, while the InfluxDB client constructs individual `Point` objects over HTTP. Similarly, the network I/O difference (18 MB/s for InfluxDB vs. 2–5 MB/s for TimescaleDB) is consistent with the overhead differences between HTTP and a binary wire protocol, though other factors may also contribute. InfluxDB's periodic write latency spikes could be related to background maintenance tasks such as compaction or WAL flushing, which is consistent with the bursty disk I/O visible in the resource charts (Figure 10.

For a more rigorous evaluation, each configuration would ideally be run multiple times to account for variance from different factors such as scheduling, network congestion, or garbage collection cycles. The results presented here are from single runs and should be interpreted as a demonstration of the tool's capabilities rather than a definitive performance comparison.

5 Conclusion

5.1 Limitations

The tool has a few practical limitations worth mentioning:

First, there is a small difference between Locust's built-in statistics summary and the per-request logs written by the custom CSV logger. The custom logger consistently records slightly more operations than Locust's summary reports. This is most likely because some operations complete after the master has already sent its shutdown signal, so the custom logger captures them while Locust's internal accounting does not.

Second, because the tool is written in Python and the CSV logger flushes after every row (`buffering=1`), the generated throughput is not so optimal. This is a trade-off since unbuffered writes ensure that minimal data is lost if a worker crashes mid-run, but they add overhead compared to batched I/O and can hinder each worker's throughput.

Third, if a worker VM's CPU becomes saturated, it can introduces delays into the measurements. The benchmark results can then reflect some worker-side bottlenecks rather than database performance. In practice this is resolved by assigning more resources (a larger flavor) to each worker node and not adding too many users and large batch size write/complicated queries.

Finally, cloud-init bootstrapping is only tested on Ubuntu images and the tool restricts all other OS's. Other Linux distributions may require changes to the template scripts. This was to reduce the project's scope rather than a technical constraint.

5.2 Future Work

Several features could extend the tool's usefulness including:

1. **Additional database support:** The client abstraction layer is designed to be extensible, but only TimescaleDB and InfluxDB are implemented so far. Adding support for databases such as QuestDB, MySQL or MongoDB would make the tool more broadly applicable for comparisons, especially regarding non time-series based databases.

2. **Multi-database benchmarks:** Currently each run targets a single database. Supporting benchmark runs against multiple databases within the same environment could help replicate even more complicated distributed environments.

3. **Broader OS support:** Extending the cloud-init templates to work with other distributions (e.g., Debian, CentOS, NixOS).

4. **Web-based interface.** Replacing the terminal UI with a web application would make the tool accessible to users who are not comfortable with the command line and non-developers to also use it.

References

[26a] *Apache JMeter*. https://jmeter.apache.org/. 2026.

[26b] *pgbench: PostgreSQL Benchmark Tool*. https://pgbench.github.io/. 2026.

[26c] *sysbench: Scriptable Database and System Performance Benchmark*. https://github.com/akopytov/sysbench. 2026.

[Baj+20] Fuad Bajaber et al. "Benchmarking big data systems: A survey". In: *Computer Communications* 149 (2020), pp. 241–251.

[Cat11] Rick Cattell. "Scalable SQL and NoSQL data stores". In: *Acm Sigmod Record* 39.4 (2011), pp. 12–27.

[Dif+13] Djellel Eddine Difallah et al. "Oltp-bench: An extensible testbed for benchmarking relational databases". In: *Proceedings of the VLDB Endowment* 7.4 (2013), pp. 277–288.

[Gra93] Jim Gray. *Database and transaction processing performance handbook*. 1993.

[HHB26] Jonatan Heyman, Lars Holmberg, and Andrew Baldwin. *Locust: An Open Source Load Testing Tool*. https://locust.io/. GitHub: https://github.com/locustio/locust. Accessed: 2026. 2026.

[HJZ17] Rui Han, Lizy Kurian John, and Jianfeng Zhan. "Benchmarking big data systems: A review". In: *IEEE Transactions on Services Computing* 11.3 (2017), pp. 580–597.

[Ope26a] OpenStack Contributors. *OpenStack SDK: A Client Library for Building Applications to Work with OpenStack Clouds*. https://docs.openstack.org/openstacksdk/latest/. GitHub: https://github.com/openstack/openstacksdk. Accessed: 2026. 2026.

[Ope26b] OpenStack Foundation. *OpenStack: Open Source Cloud Computing Software*. https://www.openstack.org/. 2026.

[Pal] Pallets. *Jinja*. Accessed: 2026-03-31. URL: https://jinja.palletsprojects.com/en/stable/.

[Roy19] Noa Roy-Hubara. "The Quest for a Database Selection and Design Method." In: *CAiSE (Doctoral Consortium)*. 2019, pp. 69–77.

A Benchmark Configuration Files

Both experimental benchmark runs share an identical schema, workload, and infrastructure configuration. Only the `database` and `queries` blocks differ. The Docker image reference `placeholder/tool:latest` is a placeholder and should be replaced with the user's own image when deploying the tool.

```yaml
table:
  name: "sensor_data"
  columns:
    - { name: "time",          type: "timestamp" }
    - { name: "sensor_id",     type: "string",
        choices: ["sensor_01","sensor_02","sensor_03","sensor_04","sensor_05"] }
    - { name: "region",        type: "string",
        choices: ["eu-west","us-east","asia-pacific"] }
    - { name: "temperature",   type: "float",  min: -20.0, max: 50.0  }
    - { name: "humidity",      type: "float",  min: 0.0,   max: 100.0 }
    - { name: "battery_level", type: "int",    min: 0,     max: 100   }

benchmark:
  users: 50
  spawn_rate: 10
  duration: 120
  batch_size: 100
  write_weight: 7

infrastructure:
  docker_image: "placeholder/tool:latest"
  ssh: { user: "cloud", key: "~/.ssh/id_rsa" }
  workers: 3
```

Listing 1: Shared table, benchmark, and infrastructure configuration (identical in both runs).

```yaml
database:
  type: "timescaledb"
  port: 5432
  name: "benchmark"
  username: "admin"
  password: "password123"
  init_sql_path: "db_init_scripts/init.sql"

queries:
  - name: "latest_readings"
    weight: 1
    query: "SELECT * FROM sensor_data ORDER BY time DESC LIMIT 100"
  - name: "regional_average"
    weight: 1
    query: >
      SELECT region, AVG(temperature) AS avg_temp,
             AVG(humidity) AS avg_hum
      FROM sensor_data
      WHERE time > NOW() - INTERVAL '5 minutes'
      GROUP BY region
  - name: "sensor_lookup"
    weight: 1
    query: >
      SELECT * FROM sensor_data
      WHERE sensor_id = 'sensor_03'
      ORDER BY time DESC LIMIT 50
```

Listing 2: TimescaleDB YAML config.

```sql
CREATE TABLE IF NOT EXISTS sensor_data (
    time          TIMESTAMPTZ      NOT NULL,
    sensor_id     VARCHAR(50),
    region        VARCHAR(50),
    temperature   DOUBLE PRECISION,
    humidity      DOUBLE PRECISION,
    battery_level INTEGER
);
SELECT create_hypertable('sensor_data','time', if_not_exists => TRUE);
```

Listing 3: TimescaleDB hypertable initialisation script (`init.sql`).

```yaml
database:
  type: "influxdb"
  port: 8086
  name: "benchmark"
  username: "admin"
  password: "password123"
  extras:
    token: "my-benchmark-token"
    org: "benchmark-org"
    bucket: "benchmark"
    use_ssl: false
    verify_ssl: false

queries:
  - name: "latest_readings"
    weight: 1
    query: |
      from(bucket: "benchmark")
        |> range(start: -1h)
        |> filter(fn: (r) => r._measurement == "sensor_data")
        |> sort(columns: ["_time"], desc: true)
        |> limit(n: 100)
  - name: "regional_average"
    weight: 1
    query: |
      from(bucket: "benchmark")
        |> range(start: -5m)
        |> filter(fn: (r) => r._measurement == "sensor_data")
        |> filter(fn: (r) =>
            r._field == "temperature" or r._field == "humidity")
        |> group(columns: ["region", "_field"])
        |> mean()
  - name: "sensor_lookup"
    weight: 1
    query: |
      from(bucket: "benchmark")
        |> range(start: -1h)
        |> filter(fn: (r) => r._measurement == "sensor_data")
        |> filter(fn: (r) => r.sensor_id == "sensor_03")
        |> sort(columns: ["_time"], desc: true)
        |> limit(n: 50)
```

Listing 4: InfluxDBv2 YAML config.

B TUI and Workflow Screenshots

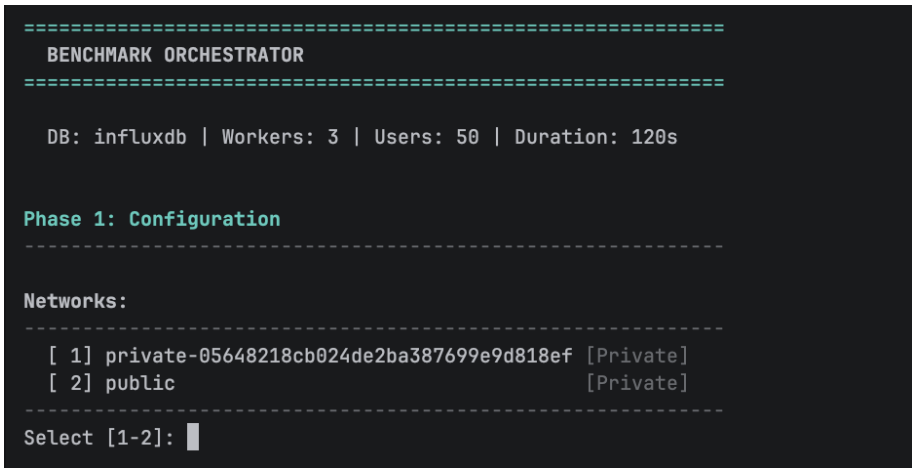

Figure 12: Phase 1 – Configuration: network selection.

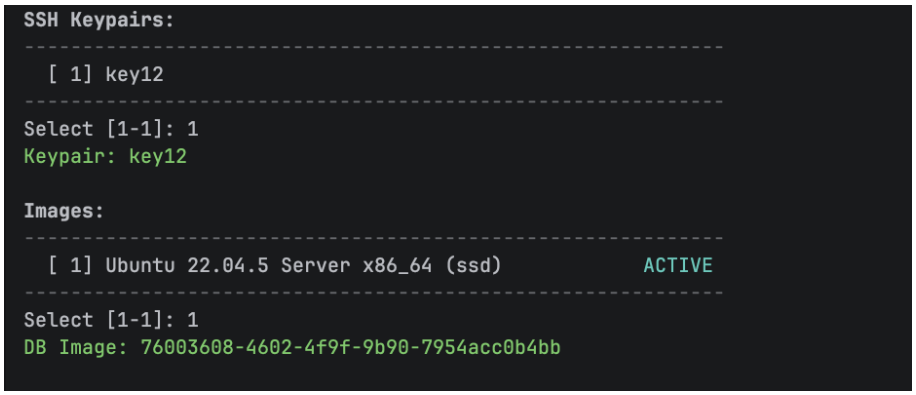

Figure 13: Phase 1 – Configuration: SSH keypair and base image selection.

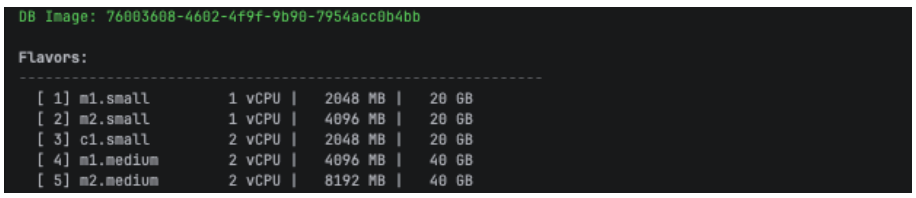

Figure 14: Phase 1 – Configuration: VM flavour selection.

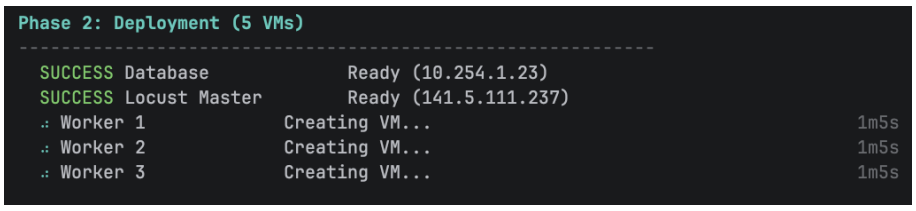

Figure 15: Phase 2 – Deployment: database and master ready, workers provisioning.

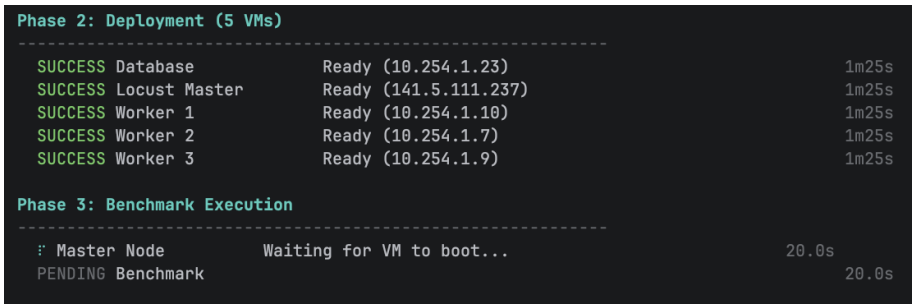

Figure 16: Phase 2 – Deployment complete: all five VMs ready.

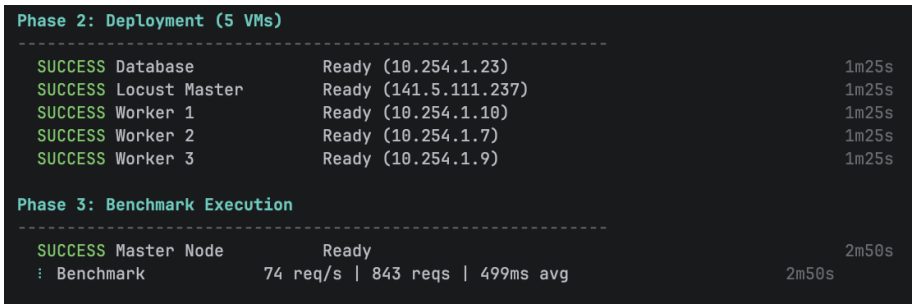

Figure 17: Phase 3 – Benchmark Execution: live throughput statistics.

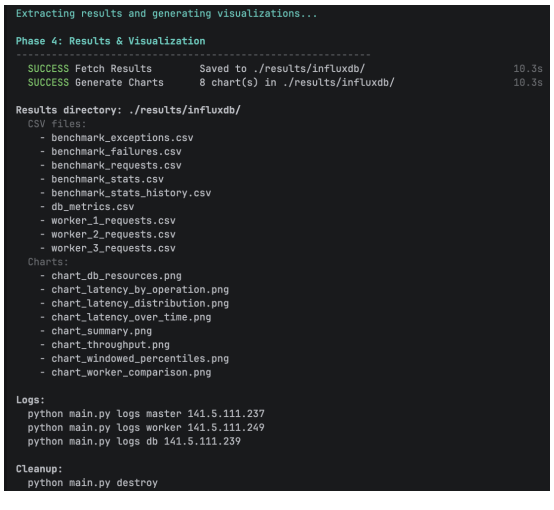

Figure 18: Phase 4 – Results & Visualisation: output files and charts generated.