Distributed Cloud Native Benchmarking Tool for Scalable Databases

Kariem Ali, Supervisor: Patrick Höhn

Agenda

Why Benchmark Databases?

- There is no universally optimal database: performance varies with workload, schema, and hardware.
- The same database under different configurations can yield drastically different throughput.
- Benchmarking before deployment avoids costly database migrations later.

Challenges

- **Workload Replication**: Specific workloads, delays, progressive increase, write and query patterns that generic benchmarks cannot replicate easily.
- **Distributed Setup Overhead**: Provisioning nodes, configuring networking, deploying containers, and coordinating workers.
- **Database Diversity**: Supporting multiple database types (relational, time-series, NoSQL) each with different protocols and drivers.

Existing Tools

- **YCSB**: Supports many databases but limited to generic key-value workloads. No infrastructure automation.
- **TSBS**: Restricted to hardcoded schemas.
- **JMeter**: GUI-centric, hard to replicate custom workloads.
- No existing tool combines specific workloads, native protocols, distributed load generation, and cloud automation in a single solution.

Ideal Tool

- **Single Configuration File**: Define workload, schema, DB type, infrastructure, and load parameters.
- **Multi-Database Support**: Native drivers for each database type.
- **Extensible**: Adding a new database requires minimal code changes.
- **Distributed Load Generation**: Scale across multiple nodes and workers.
- **Infrastructure Automation**: Provision, deploy, and destroy cloud resources automatically.
- **Live Monitoring & Result Visualisation**: Track state in real time, collect metrics, and generate charts.

Integrated Tools

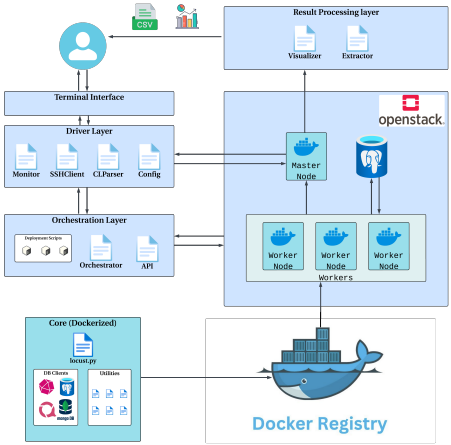
Locust

- Python-native: any native database driver can be imported directly.
- Built-in master/worker distribution.
- Load test as code: highly customisable.

OpenStack

- Python SDK for programmatic VM provisioning.
- Cloud-init: setup scripts injected at boot.

Architecture Overview



Benchmark Lifecycle

Configure → **Provision** → **Initialize** → **Run** → **Monitor** → **Collect** → **Visualize** → **Destroy**

1. Select network, keypair, image, flavor.
2. Provision VMs.
3. Cloud-init configures and starts containers.
4. Workers generate load against the database.
5. Live status displayed via TUI.
6. Result CSVs collected.
7. Charts generated from collected metrics.
8. All VMs destroyed, resources freed.

Configuration Overview

A single `config.yaml` file drives the entire benchmark.

1. **Database**: Type, port, credentials, and DB-specific extras.
2. **Deployment**: Docker image, SSH credentials, and optional init-SQL.
3. **Table Schema**: Column definitions that drive automatic data generation.
4. **Node Topology**: Hardware flavors and instance counts for database and worker nodes.
5. **Load Parameters**: Batch size, concurrent users, spawn rate, and test duration.

Configuration: Connection

- DB type, port, and credentials.
- `extras` holds DB-specific parameters.
- Switching databases requires changing only this section.

```yaml
connection:
  db_type: "timescaledb"
  host: "localhost"
  port: 5432
  db_name: "tsdb"
  username: "postgres"
  password: "password"
  extras: {}
```

Configuration: Deployment

- Docker image with the benchmark core.
- SSH credentials for VM access.
- Optional init SQL for table creation.

```
deployment:
  docker_image:  "kariemmdev/tool:latest"
  ssh_user:  "cloud"
  ssh_key:  "~/.ssh/id_rsa"
  init_sql_path:  "init.sql"
```

Configuration: Table Schema

- Column name, type, and constraints.
- Types: `timestamp`, `int`, `float`, `string`.
- Drives automatic data generation.

```yaml
db_table: "sensor_data"

db_schema:
  - name: "time"
    type: "timestamp"

  - name: "device_id"
    type: "string"
    choices: ["dev1", "dev2", "dev3"]

  - name: "temperature"
    type: "float"
    min_value: -20
    max_value: 50
```

Configuration: Nodes & Load

- Hardware flavor and OS image per node role.
- Number of worker instances.
- Batch size, concurrent users, spawn rate, and test duration.

```yaml
database_node:
  flavor:  "m1.medium"
  image:  "Ubuntu 22.04"

worker_nodes:
  count:  2
  flavor:  "m1.small"
  image:  "Ubuntu 22.04"
```

```yaml
insert_batch_size:  10
max_stress_users:  20
stress_duration:  60
user_spawn_rate:  20
```

Configuration: Extras

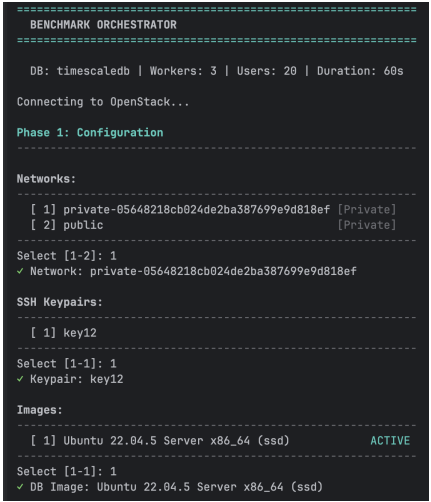
- DB-specific params (e.g. InfluxDB token, org).
- Define custom benchmark queries.
- Extensible per database type.

```yaml
connection:
  db_type:  "influxdb"
  extras:
      token:   "my-token-here"    # required
      org:  "my-org"         # required
    bucket:  "benchmark"
    use_ssl:  false

extras:
  benchmark_queries:
    - name:  "latest_readings"
      description:   "Last 10 readings"
      sql:  "SELECT * FROM sensor_readings
              WHERE device_id = 'sensor_01'
              ORDER BY timestamp DESC LIMIT 10"
```

Resource Selection

1. **Network**: OpenStack virtual network.
2. **SSH Keypair**: for VM access.
3. **OS Image**: for database and worker nodes.
4. **(Optional) Hardware Flavor**: CPU, RAM, disk per node type.

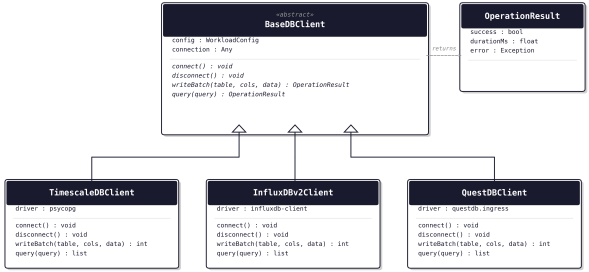


Terminal Interface & Driver Layer

- **CLIParser**: Parses commands to the Orchestrator method.
- **SSHClient**: Lightweight SSHClient to access the Master & DB node.
- **Monitor**: Parses Locust master logs into structured benchmark metrics using regex.
- **TUI**: Displays live benchmarking status (RPS, Errors, Status)

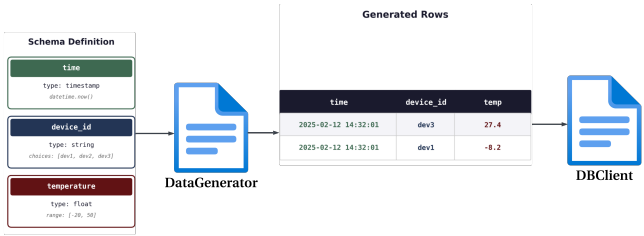
Core Layer: DB Clients

- Abstract base class handles timing and error wrapping.
- Subclasses implement four methods.
- Native protocol drivers for TimescaleDB, InfluxDBv2, QuestDB.



Core Layer: Data Generation

- Reads the table schema from the configuration.
- Based on column type to produce appropriate values.
- Outputs batches of tuples passed directly to the DB client.

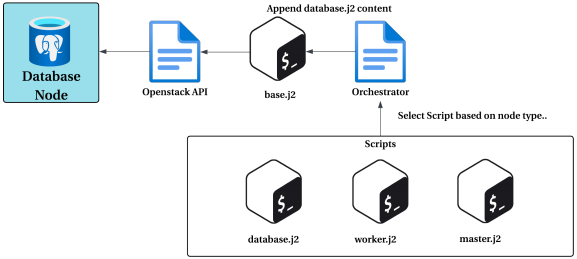


Core Layer: Locust File

- Each Locust user generates a batch, writes it via the native client, and reports the result to the master node.
- Every operation is tracked by Locust: latency, throughput, errors.
- The master node coordinates spawn rate and test duration.

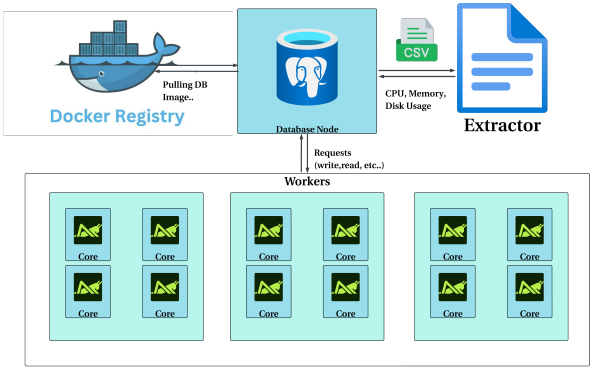
Orchestration: Cloud-Init Templates

- `base.sh.j2`: Shared setup for all nodes: DNS configuration, Docker installation, image pull.
- Role-specific templates fill a slot in the base template.
- The configuration YAML is embedded into each VM's filesystem at boot.



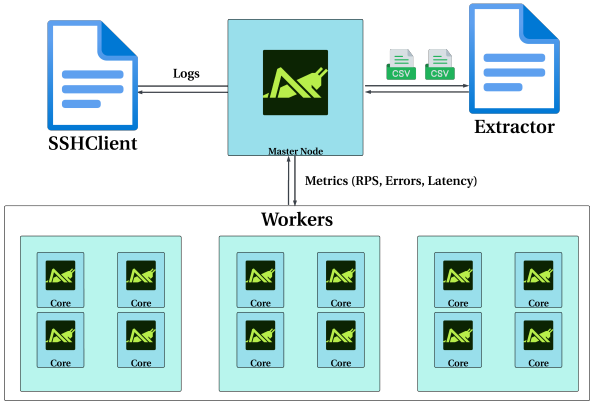
Orchestration: Database Node

- Deployed first: all other nodes depend on its IP address.
- Jinja Script starts the database container with port mapping, credentials, and optional init SQL.
- A background process collects CPU, memory, and I/O metrics every second.



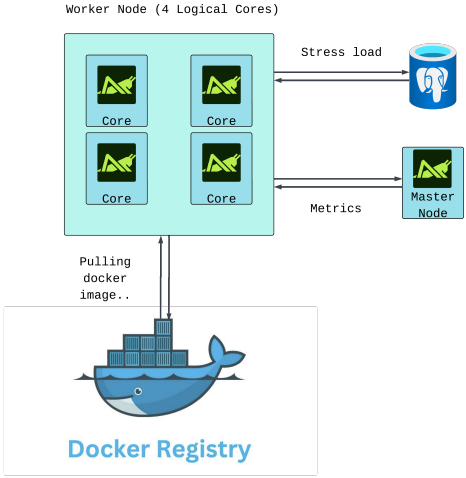
Orchestration: Master Node

- Deployed after the database node is ready.
- Runs Locust in master mode: coordinates workers, spawn rate and test duration.
- Benchmark results saved as CSV.



Orchestration: Worker Nodes

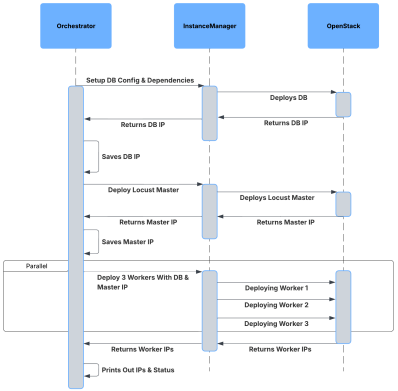
- Deployed after the master node is ready.
- Each worker connects to the master via its private IP.
- Workers execute the load: generating data and writing to the database.



Orchestration: Deployment Pipeline

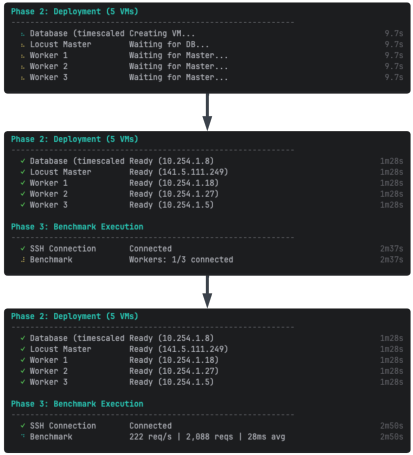
- VMs are provisioned concurrently.
- Database → Master → Workers.

Deployment of Benchmarking Environement



Orchestration: Monitoring

- Master node polled via SSH every 2 s.
- Logs parsed into metrics: state, RPS, latency, worker count.
- State machine: starting → waiting → configuring → running → stopped.
- Auto-detection of errors.



Networking

- All nodes deployed on the same OpenStack network (must have internet access).
- Three floating IPs allocated:
  - **Database node**: For server-side metrics extraction.
  - **Master node**: For monitoring and result extraction.
  - **One worker node**: For debugging purposes.

Docker Strategy

- Benchmark core packaged into a single Docker image.
- Pushed to a registry once, pulled by every non-DB VM at boot.
- Configuration injected via volume mount: same image, different behaviour per node role.

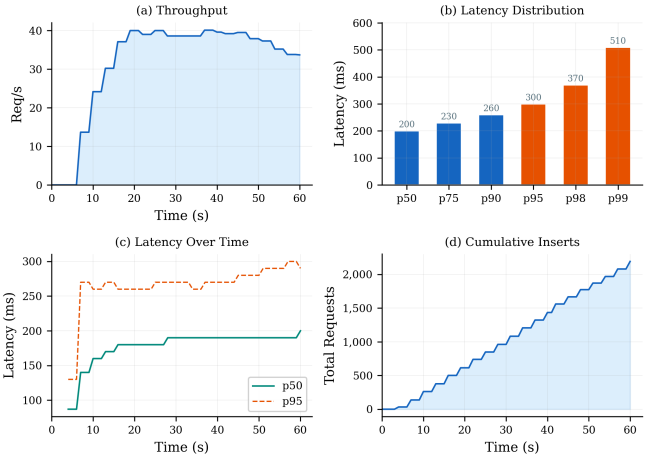
Result Processing

- **Request metrics** (Locust CSVs): Requests per second, latency percentiles, failure rates over time.
- **DB Server-side metrics** (`docker stats` CSV): CPU utilization, disk, memory usage.

Example Benchmark Run: TimescaleDB

Configuration

Database	TimescaleDB
OS	Ubuntu
Flavor	m1.small
Batch size	1 000 rows
Users	10 concurrent
Spawn rate	2 users/s
Duration	60 s
Failures	0 / 2 227



Summary

- Single YAML describes the entire benchmark.
- Solves key challenges: custom workloads, distributed load generation, and infrastructure automation.
- Pluggable native-protocol clients with schema-driven data generation.
- Easily extensible: one class and one factory branch per new database.
- Report will include database comparisons and tuning techniques.