Scientific Design of a Runner Script for ns-3 Experiments

1 Introduction

In ns-3 projects (whether as a university assignment or a research-oriented project), writing the C++ simulation program is only "half the job." The other half consists of building a **stable workflow** that can repeatedly execute an experiment under different parameter combinations and ultimately produce data that are **clean, reproducible, traceable, and analyzable**. A runner script (typically written in Bash) implements this workflow in executable form; therefore, it is not merely a convenience tool, but rather an "experimental protocol" encoded as an executable process and considered part of the project deliverables.

From a scientific perspective, such scripts address three fundamental requirements:

- **Repeatability**: the same command must produce output in the same format, with explicit control over seeds, runs, and parameters.
- **Traceability**: every output must be attributable to a unique configuration, and logs must allow auditing of failures or unexpected behaviors.
- **Experimental hygiene**: old data must not silently mix with new data, and concurrent executions must not corrupt shared files.

The standard execution model can be described as a pipeline: optional cleanup of previous outputs, staging the scenario into `ns-3/scratch`, building (usually once), executing the sweep (serially or in parallel) with isolated outputs per run, merging single-run outputs, and, if necessary, removing temporary directories. This pipeline follows best practices for computational experiments and prevents common sources of generating invalid results.

2 Separation of Artifacts and Folder Structure

The foundation of experimental reliability is an orderly folder design that clearly separates *source artifacts* from *generated artifacts*. Sources (the `.cc` scenarios and the runner scripts themselves) are stable and version-controlled, whereas outputs (CSV files, logs, PCAP files, and FlowMonitor XML files) are generated artifacts and should not be mixed with code. Mixing them leads to repository clutter, accidental overwrites, and unintended submission of large output files.

A common and reliable pattern is to place each project's outputs under `results/pX/` and divide them into several standard subdirectories:

- `raw/` for analyzable data (primary experimental evidence such as time series, summary metrics, and traces),
- `logs/` for build logs and per-run execution logs (the experiment notebook and evidence of actual execution),
- `plots/` for derived outputs (even if plotting is performed later, a designated location is defined),
- `tmp_runs/` for storing isolated outputs of each run (the most critical component for correct parallelization).

3 Isolation and Parallel Execution

The fundamental problem in parallel execution is that if multiple runs write their outputs into the same directory, they are no longer independent. Files may be overwritten, multiple processes may append concurrently to the same CSV file, or mixed and non-deterministic outputs may be produced. This is not merely a technical bug; from a methodological standpoint, it destroys traceability.

Therefore, a standard runner must treat each run as a self-contained unit that writes its outputs to a dedicated `outDir`. Instead of writing all runs directly into `results/p8/raw/`, the script creates directories such as `results/p8/tmp_runs/p8_ON_be40_s1_r3/` and sets `outDir` accordingly. After all runs have completed, the required files are merged into the final datasets under `results/p8/raw/`. This follows the practical principle of "isolate first, then aggregate."

4 Scenario Staging

In many projects, the reference version of a scenario is stored in `scenarios/`, while ns-3 executes scenarios from `ns-3/scratch/`. Without staging, there is a risk of *version confusion*: a developer edits the scenario in `scenarios/`, but an outdated version in `scratch` is executed. By explicitly copying the scenario into `scratch` before building or running, the runner guarantees that the executed program corresponds exactly to the intended version.

5 Build Management

Building ns-3 is time-consuming; therefore, runner scripts typically build ns-3 once at the beginning of the pipeline, for example using:

```
./ns3 build --jobs=N
```

The build output is stored in `results/pX/logs/build.log`. If the build fails, it is common practice to display only the final portion of the log on the terminal, preserving readability while ensuring auditability.

6 Deterministic Experimental Sweeps

During execution, the runner constructs an *experimental matrix*: the set of parameter combinations defining the sweep (e.g., MODE × BE_RATES or NSTAS × transmission type). Each run corresponds to exactly one point in the parameter space, and these points are enumerated deterministically with a well-defined order.

Parameters are commonly provided via environment variables (e.g., `NSTAS="2,5,10"` or `BE_RATES="0 10 20 40 60"`), converted into arrays, and expanded using nested loops. This approach avoids hardcoding and allows both small test sweeps and full sweeps without modifying the script.

7 Controlled Parallelization

Parallelization improves throughput for large experimental matrices but must be controlled. Scripts typically define a concurrency limit such as `JOBS=6`. Up to this limit, runs are launched in the background; when the limit is reached, the script waits for at least one run to complete before scheduling another. This strategy prevents system overload while maintaining manageable execution time.

8 Progress Counters and Tags

Progress counters (e.g., `[3/10]`) improve observability and facilitate debugging, especially in long sweeps. Even when runs complete out of order, counters printed at the start and end of each run provide a stable reference.

Tags further enhance traceability. A tag encodes key parameters and forms the run directory and log names. For example, `p8_ON_be40_s1_r3` uniquely identifies a configuration with mode ON, BE rate 40, seed 1, and run 3.

9 Merging Outputs

While per-run isolation ensures correctness, analysis typically requires aggregated data. The runner therefore includes a merge stage that collects relevant files from `tmp_runs/` and produces final datasets under `results/pX/raw/`. For CSV files, headers must be handled carefully: the header is written once, and only data rows from each run are appended to avoid corruption.

10 Cleanup and Reproducibility

Initial cleanup is critical for result validity. Residual directories in `tmp_runs/` or leftover merged files can contaminate new results. Runner scripts therefore commonly provide options such as `CLEAN=true/false` and `KEEP_TMP=true/false` to support both clean final runs and debug executions that preserve intermediate artifacts.

11 Defensive Bash Practices

Runner scripts should be implemented defensively. Enabling strict modes such as:

```
set -euo pipefail
```

prevents silent failures by terminating execution on undefined variables or command errors. Logging standard output and error streams for each run is essential for traceability, and prerequisite checks improve robustness across environments.

12 Conclusion

This scripting pattern provides a compact yet scientifically rigorous framework for ns-3 experiments. By combining executable version control through staging, auditable builds, deterministic experimental matrices, controlled parallelization with isolated outputs, robust traceability mechanisms, and disciplined result merging, the runner script effectively serves as the *Methods* section of an experimental study, defining precisely how simulations are executed and how data are transformed into defensible and analyzable results.