Qanat

Experiment Tracking for HPC Users

Ammar Mian

November 11, 2025

1. Introduction
2. Core Workflow
3. Power Features
4. Demo Examples
5. Wrap-up

The Reproducibility Problem

Computational research faces a critical challenge:

- "Which parameters gave that result 3 months ago?"

Computational research faces a critical challenge:

- “Which parameters gave that result 3 months ago?”
- “What was the exact code version for that experiment?”

The Reproducibility Problem

Computational research faces a critical challenge:

- "Which parameters gave that result 3 months ago?"
- "What was the exact code version for that experiment?"
- "How do I reproduce this on the cluster?"

Computational research faces a critical challenge:

- "Which parameters gave that result 3 months ago?"
- "What was the exact code version for that experiment?"
- "How do I reproduce this on the cluster?"

The Challenge

Existing tools (MLflow, Weights & Biases) require GUIs and databases, but HPC users work in **headless terminal environments** with job submission systems.

What HPC Users Need:

- CLI-first workflow
- Works on remote servers
- HTCondor/Slurm integration
- Minimal dependencies
- Offline operation

What HPC Users Need:

- CLI-first workflow
- Works on remote servers
- HTCondor/Slurm integration
- Minimal dependencies
- Offline operation

What GUI Tools Require:

- Web browser access
- Database servers
- Network connectivity
- Complex setup
- Heavy dependencies

Why Existing Tools Fall Short for HPC

What HPC Users Need:

- CLI-first workflow
- Works on remote servers
- HTCondor/Slurm integration
- Minimal dependencies
- Offline operation

What GUI Tools Require:

- Web browser access
- Database servers
- Network connectivity
- Complex setup
- Heavy dependencies

Qanat's Solution

A minimal, CLI-based experiment tracking system designed specifically for computational research on HPC systems.

What is Qanat?

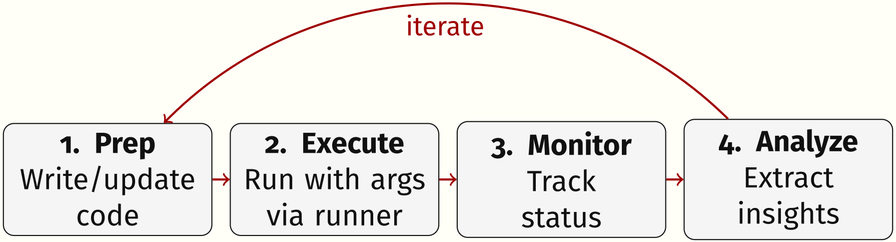
```
 _____                   _
|  _  |                 | |
| | | | __ _ _ __   __ _| |_
| | | |/ _` | '_ \ / _` | __|
\ \/' / (_| | | | | (_| | |_
 \_/\_\\__,_|_| |_|\__,_|\__|

Qanat, version 1.9.4
Minimal experience management system.
```

- A tool I built for my own research needs
- Used by some students, but nothing groundbreaking
- Everything it does *can* be done without it
- Power users could write it in 50 lines of bash

So why use it? It enforces a structured workflow and handles the boring parts automatically.

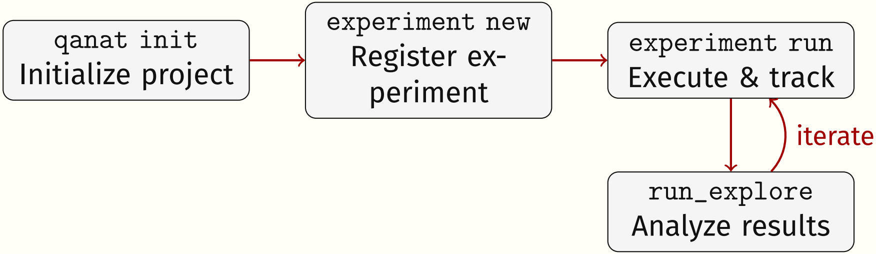
Overview

Standard steps in any computational experiment:



Qanat organizes these steps and forces you to work in a reproducible way.

The Experiment Lifecycle with Qanat



Automatic tracking: git commit, parameters, stdout/stderr, duration, status

Let's see qanat in action!

We'll demonstrate:

1. Initialize a project
2. Create a simple training script
3. Register an experiment
4. Run with parameters
5. Explore results interactively

Demo Note

Everything shown works the same on your laptop or on a 1000-node cluster.

For every run, qanat stores:

```
results/my_experiment/run_42/
|-- info.yaml              # All parameters + metadata
|-- stdout.txt             # Complete output
|-- stderr.txt             # Errors & warnings
`-- [your results]         # Whatever your script saves
```

info.yaml contains:

- Exact git commit SHA
- All command-line arguments
- Execution date, duration, status
- Tags, description, dataset links

Overview

Feature 1: Parameter Grid Sweeps

The Problem: Running experiments over multiple parameter combinations

The Problem: Running experiments over multiple parameter combinations

Qanat's Solution: Groups

The Problem: Running experiments over multiple parameter combinations

Qanat's Solution: Groups

- `-g` flag: Define parameter groups

The Problem: Running experiments over multiple parameter combinations

Qanat's Solution: Groups

- `-g` flag: Define parameter groups
- `-r` flag: Define ranges with start/stop/step

The Problem: Running experiments over multiple parameter combinations

Qanat's Solution: Groups

- `-g` flag: Define parameter groups
- `-r` flag: Define ranges with start/stop/step
- Cartesian product between groups and ranges

Hyperparameter search across learning rates and seeds:

```
qanat experiment run train_model \
  --epochs 100 \
  -g "--lr 0.001" \
  -g "--lr 0.01" \
  -g "--lr 0.1" \
  -r "--seed 0 5 1"
```

Hyperparameter search across learning rates and seeds:

```
qanat experiment run train_model \
  --epochs 100 \
  -g "--lr 0.001" \
  -g "--lr 0.01" \
  -g "--lr 0.1" \
  -r "--seed 0 5 1"
```

This creates a **single run_id** with 15 groups (3 LRs × 5 seeds)

Hyperparameter search across learning rates and seeds:

```
qanat experiment run train_model \
  --epochs 100 \
  -g "--lr 0.001" \
  -g "--lr 0.01" \
  -g "--lr 0.1" \
  -r "--seed 0 5 1"
```

This creates a **single run_id** with 15 groups (3 LRs × 5 seeds)

```
results/train_model/run_17/
|-- group_0/  # lr=0.001, seed=0
|-- group_1/  # lr=0.001, seed=1
|-- ...
`-- group_14/ # lr=0.1, seed=4
```

Benefits of the groups concept:

- **Aggregation:** K-fold cross-validation, multiple random seeds
- **Organization:** Related experiments stay together
- **Efficiency:** Submit all variations in one command
- **Analysis:** Actions can operate on entire groups

Design Choice

Groups let you think about parameter *sweeps* rather than individual runs.

Runner Abstraction: Same command, different execution environments

Runner	Use Case	Backend
local	Laptop/workstation	subprocess
htcondor	HTCondor clusters	Python bindings
slurm	Slurm clusters	sbatch

Just add: `-runner htcondor`

Everything else stays the same!

HTCondor Note

Requires `htcondor` Python bindings `<11` (version 10.x recommended)

Submit to cluster:

```
qanat experiment run train_model \
  --runner htcondor \
  --submit_template gpu_template \
  -g "--lr 0.001" -g "--lr 0.01" -g "--lr 0.1" \
  -r "--seed 0 5 1"
```

Monitor progress:

```
qanat experiment status train_model --live
```

Qanat creates submit descriptions, tracks job status, and aggregates results automatically.

Reusable resource specifications:

Define once, use everywhere:

- CPU cores, memory requirements
- GPU requests (number, type)
- Docker/Singularity images
- Queue names, time limits

Example

`gpu_template`: Requests 1 GPU, 16GB RAM, cuda11.8 environment

Switch between small CPU tests and large GPU runs with just the template name.

Reproducible environments with Singularity/Apptainer:

- `-container path/to/image.sif`
- Automatic bind mounting of working directory
- Dataset paths mounted automatically
- GPU support via `-nv` flag
- Works with both local and HPC runners

Reproducibility++

Container + git commit = exact code *and* exact environment.

Container Example

```
# Run in container on cluster with GPU
qanat experiment run deep_learning \
  --runner htcondor \
  --container /shared/images/pytorch.sif \
  --gpu True \
  -g "--model resnet50" \
  -g "--model efficientnet" \
  -r "--batch_size 16 64 16"
```

Qanat handles:

- Bind mounting code and data
- GPU passthrough (`-nv`)
- Submit description generation
- Tracking container path in metadata

The most important feature: Qanat refuses to run uncommitted code!

Enforced Reproducibility

If your code isn't committed, your experiment isn't reproducible. Period.

Every run automatically tracks:

- Exact git commit SHA (e.g., `a3f4b2c`)
- Branch name
- Repository remote URL
- Working directory state

This is the foundation that makes everything else meaningful.

Qanat handles git checkout automatically!

Scenario 1: Exact reproduction

```
# Reruns with same commit + same parameters
qanat experiment rerun my_experiment 42
```

Scenario 2: Old code, new parameters

```
# Check which commit was used
qanat experiment run_explore my_experiment
# Shows: git_commit: a3f4b2c

# Run with that commit + different parameters
qanat experiment run my_experiment \
  --commit_sha a3f4b2c \
  --new_param value
```

Reproducing Experiments]Reproducing Experiments

No manual git checkout needed! Qanat does it for you.

Feature 5: Actions for Result Analysis

Problem: Don’t want to rerun experiments just to change a plot.

Feature 5: Actions for Result Analysis

Problem: Don't want to rerun experiments just to change a plot.

Solution: Actions - Post-processing scripts that analyze stored results

Problem: Don't want to rerun experiments just to change a plot.

Solution: Actions - Post-processing scripts that analyze stored results

- Receive `-storage_path` automatically from qanat
- Access all files saved during the run
- Generate plots, compute metrics, export data
- Can be run from CLI or interactive menu

Feature 5: Actions for Result Analysis

Problem: Don't want to rerun experiments just to change a plot.

Solution: Actions - Post-processing scripts that analyze stored results

- Receive `-storage_path` automatically from qanat
- Access all files saved during the run
- Generate plots, compute metrics, export data
- Can be run from CLI or interactive menu

Benefit: Separate computation from visualization. Run experiment once, analyze many ways.

Define action in experiment YAML:

```yaml
actions:
  - plot_curves:
      executable: scripts/plot_learning_curves.py
      executable_command: python
      description: Plot training/validation curves
```

Define action in experiment YAML:

```yaml
actions:
  - plot_curves:
      executable: scripts/plot_learning_curves.py
      executable_command: python
      description: Plot training/validation curves
```

Action script structure:

```python
import argparse, numpy as np, matplotlib.pyplot as plt

parser = argparse.ArgumentParser()
parser.add_argument("--storage_path", required=True)
args = parser.parse_args()

# Load results from storage_path
results = np.load(f"{args.storage_path}/metrics.npz")
plt.plot(results['train_loss'], label='train')
plt.savefig(f"{args.storage_path}/learning_curves.png")
```

Define action in experiment YAML:

```yaml
actions:
  - plot_curves:
      executable: scripts/plot_learning_curves.py
      executable_command: python
      description: Plot training/validation curves
```

Action script structure:

```python
import argparse, numpy as np, matplotlib.pyplot as plt

parser = argparse.ArgumentParser()
parser.add_argument("--storage_path", required=True)
args = parser.parse_args()

# Load results from storage_path
results = np.load(f"{args.storage_path}/metrics.npz")
plt.plot(results['train_loss'], label='train')
plt.savefig(f"{args.storage_path}/learning_curves.png")
```

Execute: `qanat experiment action my_exp plot_curves 42`

Real-time experiment monitoring:

- **TQDM integration:** Training loops show progress bars
- **Count format:** `Progress: 42/100` for parallel jobs
- **Group aggregation:** See progress across all parameter combinations
- **Live dashboard:** `-live` flag for auto-refreshing status

Implementation

Your script writes progress to stdout in recognized formats. Qanat parses and displays it.

Simple C program with parallel processing:

```
# Run with different thread counts
qanat experiment run montecarlo_pi --n_trials 10000000 \
  -g "--num_threads 1" -g "--num_threads 2" -g "--num_threads 4"  -g "
      --num_threads 8"
```

Simple C program with parallel processing:

```
# Run with different thread counts
qanat experiment run montecarlo_pi --n_trials 10000000 \
  -g "--num_threads 1" -g "--num_threads 2" -g "--num_threads 4"  -g "
      --num_threads 8"
```

Qanat tracks:

- Progress: Each thread writes count to `progress.txt`
- Results: Pi estimate, error, execution time
- Parameters: n_trials, num_threads

Simple C program with parallel processing:

```
# Run with different thread counts
qanat experiment run montecarlo_pi --n_trials 10000000 \
  -g "--num_threads 1" -g "--num_threads 2" -g "--num_threads 4"  -g "
     --num_threads 8"
```

Qanat tracks:

- Progress: Each thread writes count to `progress.txt`
- Results: Pi estimate, error, execution time
- Parameters: n_trials, num_threads

Plot action (gnuplot):

```
qanat experiment action montecarlo_pi plot 42
```

Creates convergence plot + random sampling visualization

PyTorch training with TensorBoard:

```
# Compare CNN vs pretrained EfficientNet
# Each experiment has its own dedicated script
uv run qanat experiment run mnist_cnn \
  --epochs 10 --lr 0.001 \
  -r "--seed 0 3 1"  # 3 random seeds

uv run qanat experiment run mnist_efficientnet \
  --epochs 5 --lr 0.0001 \
  -r "--seed 0 3 1"
```

PyTorch training with TensorBoard:

```
# Compare CNN vs pretrained EfficientNet
# Each experiment has its own dedicated script
uv run qanat experiment run mnist_cnn \
  --epochs 10 --lr 0.001 \
  -r "--seed 0 3 1"  # 3 random seeds

uv run qanat experiment run mnist_efficientnet \
  --epochs 5 --lr 0.0001 \
  -r "--seed 0 3 1"
```

Automatic tracking:

- Progress: Updates after each epoch (count format)
- TensorBoard logs: Saved to `storage_path/tensorboard/`
- Metrics: Train/val/test accuracy and loss

PyTorch training with TensorBoard:

```
# Compare CNN vs pretrained EfficientNet
# Each experiment has its own dedicated script
uv run qanat experiment run mnist_cnn \
  --epochs 10 --lr 0.001 \
  -r "--seed 0 3 1"  # 3 random seeds

uv run qanat experiment run mnist_efficientnet \
  --epochs 5 --lr 0.0001 \
  -r "--seed 0 3 1"
```

Automatic tracking:

- Progress: Updates after each epoch (count format)
- TensorBoard logs: Saved to `storage_path/tensorboard/`
- Metrics: Train/val/test accuracy and loss

If you can do this in 50 lines of bash, why use qanat?

Why Bother Using It?

If you can do this in 50 lines of bash, why use qanat?

- **Enforces discipline:** Forces git commits, no shortcuts
- **Handles boilerplate:** Job submission, directory creation, logging
- **Consistency:** Everyone on your team uses the same structure
- **Lower barrier:** Students don't need to be bash experts
- **Documentation:** Forgot how something works? Check old runs

If you can do this in 50 lines of bash, why use qanat?

- **Enforces discipline:** Forces git commits, no shortcuts
- **Handles boilerplate:** Job submission, directory creation, logging
- **Consistency:** Everyone on your team uses the same structure
- **Lower barrier:** Students don't need to be bash experts
- **Documentation:** Forgot how something works? Check old runs

Reality

It's just organized file management and script wrappers. Nothing revolutionary, but it saves time.

Overview

What qanat is:

- A personal tool I made to stop losing my own experiments
- Used by a few students in my group
- Open source on GitHub (AGPLv3)
- Maintained when I need new features

What qanat is NOT:

- Not polished or well-documented
- Not feature-complete (Slurm support is half-baked)
- Not tested beyond my own use cases
- Not a replacement for proper experiment management systems

Reality

It works for what I need. Might work for you. Might not. Your mileage may vary.

Limitations & Future Work

Known limitations:

- Slurm support still in development
- Interactive menu requires Linux/macOS (no Windows support)
- SQLite database limits concurrent writes
- No built-in visualization dashboard

Potential improvements:

- Better dataset versioning
- Experiment comparison tools
- Integration with Jupyter notebooks
- Enhanced action system

Key Takeaways

1. **Git commit tracking is essential:** Without it, nothing else matters

Key Takeaways

1. **Git commit tracking is essential:** Without it, nothing else matters
2. **Forced discipline saves you later:** No uncommitted code = no regrets

Key Takeaways

1. **Git commit tracking is essential:** Without it, nothing else matters
2. **Forced discipline saves you later:** No uncommitted code = no regrets
3. **Checkout old commits to reproduce:** Time-travel debugging actually works

Key Takeaways

1. **Git commit tracking is essential:** Without it, nothing else matters
2. **Forced discipline saves you later:** No uncommitted code = no regrets
3. **Checkout old commits to reproduce:** Time-travel debugging actually works
4. **HPC job submission is just boilerplate:** Qanat handles the boring parts

Key Takeaways

1. **Git commit tracking is essential:** Without it, nothing else matters
2. **Forced discipline saves you later:** No uncommitted code = no regrets
3. **Checkout old commits to reproduce:** Time-travel debugging actually works
4. **HPC job submission is just boilerplate:** Qanat handles the boring parts
5. **Nothing revolutionary:** Just organized wrappers, but that's often enough

Key Takeaways

1. **Git commit tracking is essential:** Without it, nothing else matters
2. **Forced discipline saves you later:** No uncommitted code = no regrets
3. **Checkout old commits to reproduce:** Time-travel debugging actually works
4. **HPC job submission is just boilerplate:** Qanat handles the boring parts
5. **Nothing revolutionary:** Just organized wrappers, but that's often enough

Bottom Line

If you're already disciplined with git and bash scripts, you don't need this. If not, it might help.

Documentation & Code:

- **Documentation:** `https://ammarmian.github.io/qanat/`
- **GitHub:** `https://github.com/ammarmian/qanat`
- **Installation:** `pip install git+https://github.com/ammarmian/qanat`

Questions?

Feedback and contributions welcome!

Thank You!

Happy experimenting on your HPC clusters

ammar.mian@univ-smb.fr